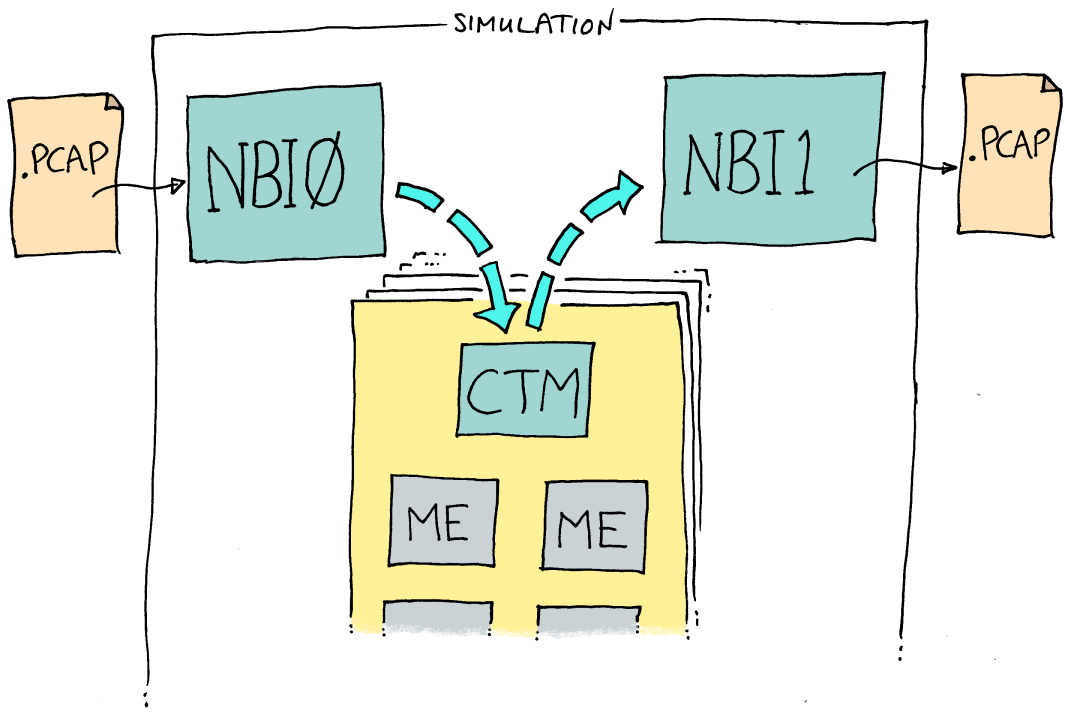
# The Joy of Micro-C Filter Annex

This document explains how to set up and run a simple packet filter program on the NFP-6xxx simulator in the Programmer Studio SDK. Note that the document you are reading is not self contained. It assumes that you have read and understood *The Joy of Micro-C*. For that reason, we will call the next part chapter 17:

## 17 A simple packet filter

The code up to this point in *The Joy of Micro-C* has focused on the Micro-C language. We have seen small programs which illustrate one aspect or another of the language, but they have not really done anything useful. Sooner or later, we need a "real example", and since the NFP-6xxx is a network processor, that example clearly has to involve receiving and transmitting packets. So here it is: a very simple example, but one which could easily be extended to do something more interesting, using what you already know.




Here we have a simple 100G Ethernet "bump-in-the-wire" packet filter with two network interfaces. Looking at traffic going in one direction, frames are taken from a PCAP file, and fed into the one of the simulated Network Block Interfaces, `NBI0`. The simulated chip is configured by start-up scripts so that the Network Block Interfaces deliver the first part of each packet to the CTMs of certain islands, where MEs are waiting to work on them. As each packet

arrives at a CTM, a worker thread wakes up to deal with it. (If all available worker threads are busy, new packet headers queue up in the CTM. The NBI knows how much capacity is left in each CTM and balances delivery between CTMs in different islands.) Along with the start of its packet, each ME worker thread also gets some metadata about that packet, previously extracted by the NBI.

Short packets will fit completely in the space available in a CTM; if needed, the tail of larger packets is written separately by the ingress NBI to an IMEM. (This is not shown on the diagram.) The ME worker thread can examine the packet headers, the metadata, and if it needs to, the packet body. It can consult tables to decide what to do with the packet. It can increment counters. It can rewrite fields in the packet. In our simple filter example, the worker just checks the IP source address in each packet and filters on that.

Finally, each worker thread passes the packet on to the appropriate egress NBI and then makes itself available to handle another packet from its CTM. Since our example is a "bump-in-the-wire" packet filter, the illustration above shows packets which arrived on `NBI0` and are sent out of `NBI1`. In real life, these would then travel over a physical network, but our simulator is configured to write them to another PCAP file. (Traffic going the other way is handled similarly, but for simplicity we don't show that in the illustration.)

## 18 A message from our sponsor

There are some aspects of this example which have not yet been explained in *The Joy of Micro-C*, for example the CTM "work queues" used to rendezvous between available worker threads and the packets that they work on. This annex concentrates on setting up and running this "real example" program, rather than giving all the background detail. That background detail will be explained more fully in some subsequent expansion of *The Joy of Micro-C*.

Library functions are used to encapsulate the detailed code necessary to get packets in and out of ME worker threads. Unfortunately, the necessary libraries are still in flux as I write this document, and do not yet ship with the SDK. Thus we currently need to indulge in a bit of "plumbing" to use these other libraries. In future this "plumbing" will not be necessary.

To function as we want, the NBIs need to be configured. We use some script files to do this when the simulator starts and before ME code runs. For now, we are going to treat these start-up scripts as opaque code, without explaining what they are doing.

So … let's get started!

# 19 Setting up the packet filter

For this example, we use the files supplied in the **filter** folder.

Steps

1. Create a new project, as explained in *The Joy of Micro-C* chapter 2, with one difference: in the New Project dialog box, select **nfp-6xxx-a0** from the Chip Family/Variant list. (*The Joy of Micro-C* uses version nfp-6xxx-b0, but the start-up configuration scripts that we will use in here are for version nfp-6xxx-a0 only.)

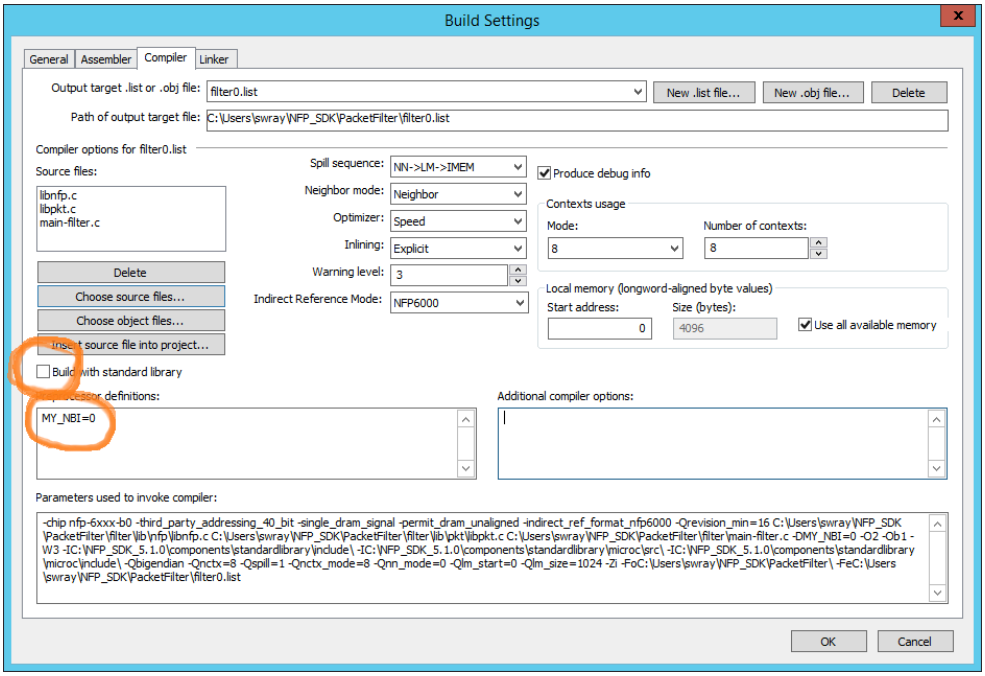
2. Using file explorer, copy the **filter** folder into your project folder.

3. Using **Project > Insert Network Flow C Compiler Source Files**, insert the following source files into the project:
   - **filter\main-filter.c**
     (We will come back later and explain this main program.)
   - **filter\lib\nfp\libnfp.c**
   - **filter\lib\pkt\libpkt.c**

4. Using **Build > Settings …** go to the **Compiler** tab and create a new .list file called **filter0.list**. (The reason for this choice of name will become apparent soon.)

   - Using the **Choose Source Files …** button, select the three source files **main-filter.c**, **libnfp.c** and **libpkt.c**.

   - Un-check the **Build with standard library** check-box:



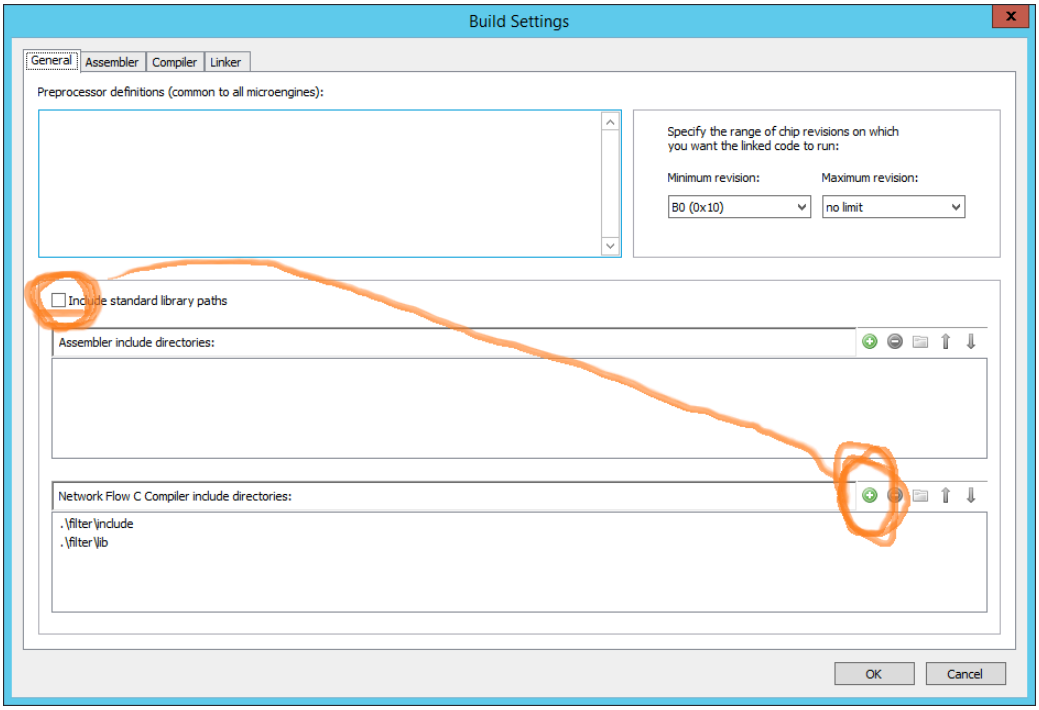
If you again click on the **Choose Source Files …** button, you will see that the files **<SDK> intrinsic.c**, **<SDK> libc.c** and **<SDK> rtl.c** have now disappeared. (It's not possible to get rid of them from inside the **Compiler Sources** dialog box. You have to uncheck the **Build with standard library** check-box.)

- In the **Preprocessor definitions** text entry box, type **`MY_NBI=0`**.

  The main program expects to have this symbol defined to tell it which network interface is giving it packets, and when it decides to forward a packet, it will send that packet out of the other interface. In this case we set **`MY_NBI`** to zero to indicate that packets will arrive from `NBI0` (and therefore will be forwarded out of `NBI1`). If you look closely at the compiler command-line arguments shown at the bottom of the **Build Settings** dialog box, you will see that it now includes `-DMY_NBI=0`.

5. Still in the Build Settings dialog box, go to the **General** tab.

   - Uncheck the **Include standard library paths** check-box:



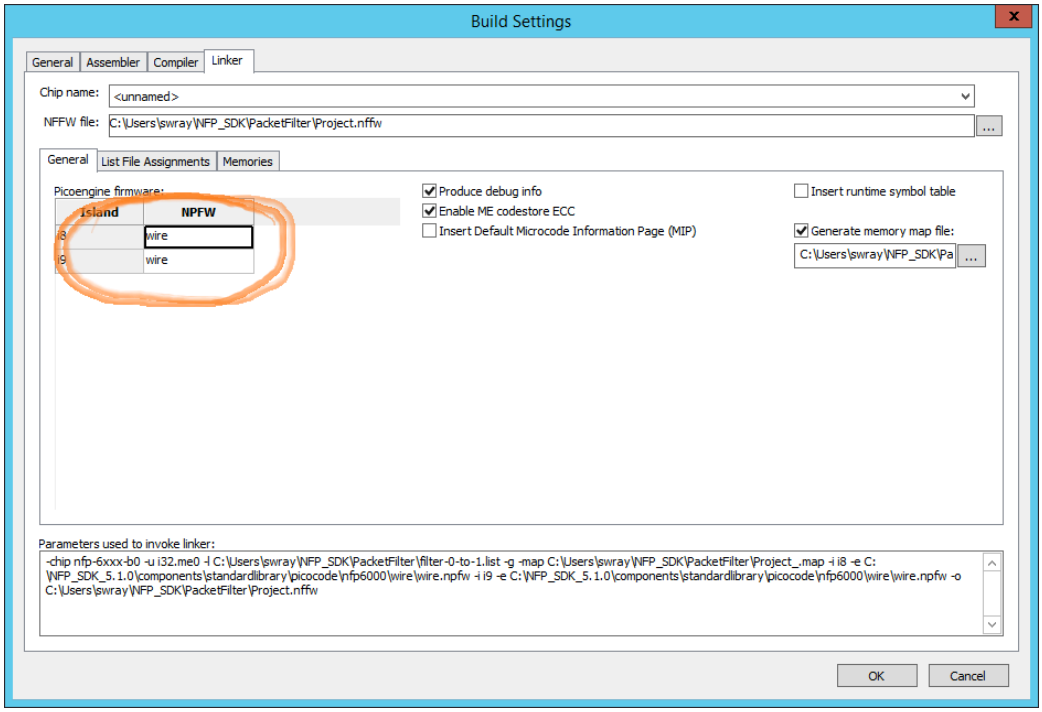

   - Using the '**+**' button in the **Network Flow C Compiler include directories** pane, add the folder **filter\include**.

   - Using the '**+**' button again, add the folder **filter\lib**.

6. Still in the Build Settings dialog box, go to the **Linker** tab.

    - In the sub-tab **ListFileAssignments**, place **`filter0.list`** on island 32 ME 0, island 33 ME 0 and island 34 ME 0. (The start-up scripts arrange for packets from `NBI0` to be delivered to these three islands, and in practice we would want to use all the MEs in these islands to service that traffic, but it's a little tedious to set that up using the GUI, and we are mainly just trying to demonstrate the concept here.)

    - In the sub-tab **General**, click on the boxes underneath the NFPW heading. From the selection box, choose '**wire'**. Do this for both island 8 and 9. (This chooses the entirely separate firmware loaded into the classifier in `NBI0` and `NBI1` respectively. This 'Pico Engine' firmware generates the meta-data which arrives at the ME along with each packet.)




7. Close the Build Settings dialog box and use **Build > Build** to check that everything so far is fine.

8. Now repeat the above steps for a new .list file called **filter1.list**. This is mostly the same, but with the following differences:

    - In the **Compiler** tab, in the **Preprocessor definitions** text entry box, set **`MY_NBI=1`**.

    - In the **Linker** tab, in sub-tab **ListFileAssignments**, place **`filter1.list`** on island 36 ME 0, island 37 ME 0 and island 38 ME 0. (The start-up scripts configure `NBI1` so that packets which arrive there get sent to those islands.)

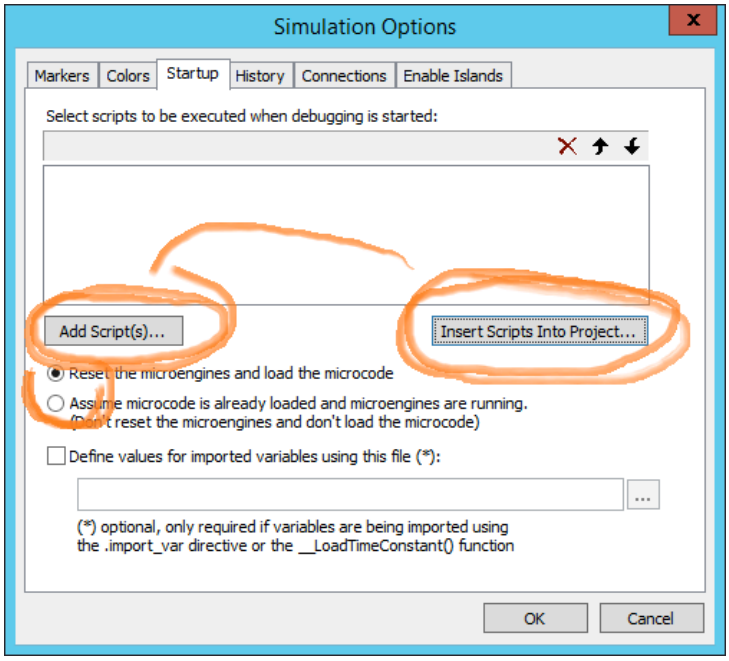
9. Close the Build Settings dialog box and use **Build > Build** to check again that everything so far is fine.

Next we are going to configure the start-up scripts.

10. From the **Simulation** menu, select **Options …** A dialog box appears.

11. In the Simulations Options dialog box:,

    - Click the **Startup** tab.
    - Check that the **Reset the microengines and load the microcode** radio-button is selected.



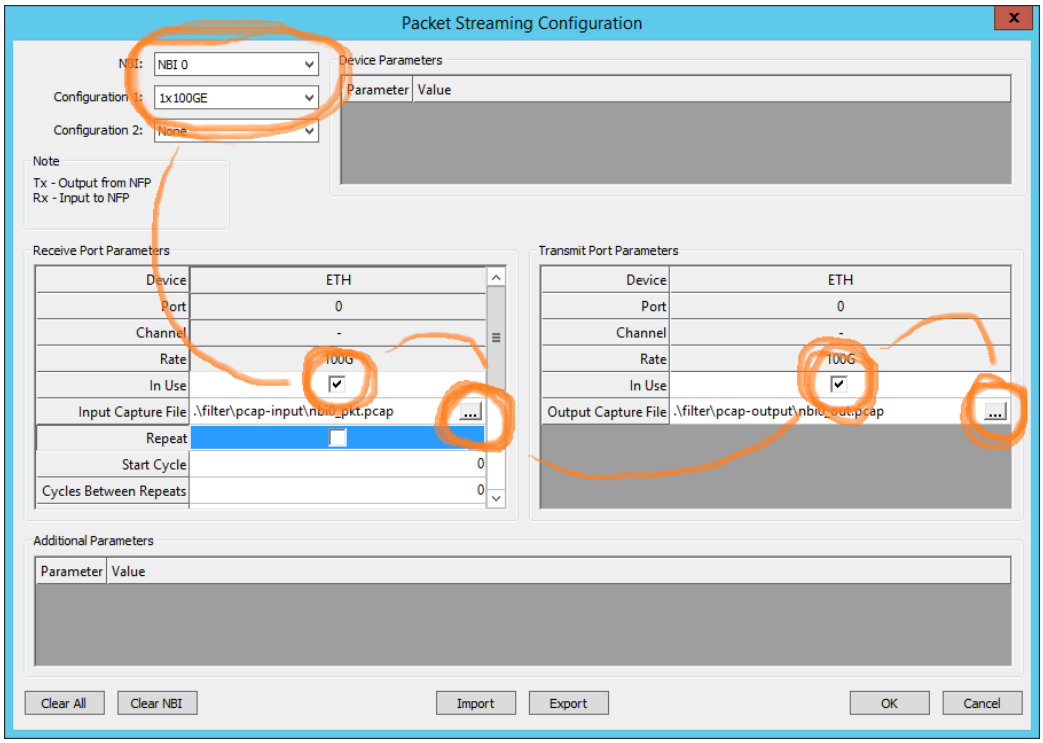

    - Click on the **Insert Scripts into Project** button.
    - A file selection box opens. Choose **filter\scripts\clksetup_cs.c**
    - Click on the **Add Scripts(s) …** button.
    - A dialog box opens. Select **clksetup_cs.c**

    - In a similar way, insert and add **filter\scripts\mem_init_cs.c**
    - Insert and add **filter\scripts\nbidma_setup_cs.c**
    - Insert and add **filter\scripts\nbitm_setup_cs.c**

    What is that all doing? These “script” files are written in C, but run in a C *interpreter* in the simulator at start-up. The scripts set appropriate values in various configuration registers in the simulated hardware. This is how `NBI0` knows to deliver packets to islands 32-34, and `NBI1` to deliver packets to islands 36-38. It’s probably best to regard these script files as Deep Voodoo and not look too closely.

Next we will associate particular PCAP files with each NBI.

12. From the **Simulation** menu, select **Packet Streaming Configuration**…




13. In the Packet Streaming Configuration dialog box:
    - In the **NBI** selection box at the top, choose **NBI 0**.
    - In the **Configuration 1** selection box, choose **1x100GE**.
    - In **Receive Port Parameters**, make sure that the **In Use** check-box is checked.
    - In **Receive Port Parameters**, in **Input Capture File**, click on the '**…**' button to choose the input PCAP file using a file browser. (The file `filter\pcap-input\nbi0_pkt.pcap` is provided for you to use.)
    - In **Transmit Port Parameters**, make sure that the **In Use** check-box is checked.
    - In **Transmit Port Parameters**, in **Output Capture File**, click on the '**…**' button to choose the output PCAP file using a file browser. (Set this to `filter\pcap-output\nbi0_out`.)

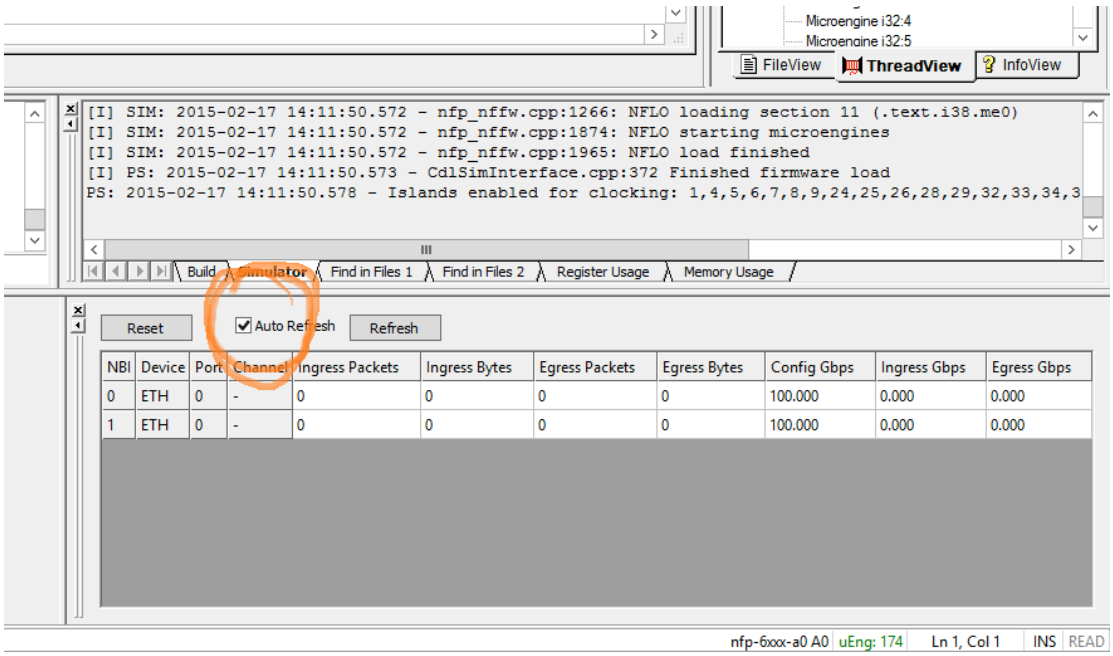
14. Still in the Packet Streaming Configuration dialog, do the same again for **NBI 1**, by selecting it in the **NBI** selection box at the top. However, this time take the input from `filter\pcap-input\nbi1_pkt.pcap` and send the output to to `filter\pcap-output\nbi0_out`.)

15. Click **OK**.

16. Using **Build > Build**, compile and link the project.

17. Using **Debug > Start Debugging**, run the simulator. It may take a little while to run the start-up scripts. (It will say 'Executing startup scripts' in the status bar at the bottom left of the SDK window. You might think it has died. Just give it a while.)

18. Using **View > Debug > Packet Streaming Statistics**, bring up the debug window which shows the network traffic status:




19. Make sure that the **Auto Refresh** check-box is checked.

20. Using **Debug > Run Control > Go** (or **F5**) run the simulation. You will see the Ingress Bytes and Ingress Packets counters tick upwards. After a while you will also see Egress Bytes and Egress Packets tick upwards. (This will happen before the uEng clock, at the bottom right of the status bar, reaches 1000 cycles.)

21. Using a file browser, have a look in your folder `filter\pcap-output`. You should see the two output files. To see what has happened, you will have to look at the input and the output files using a PCAP file inspection program, for example *Wireshark*, available from *www.wireshark.org*.

22. Stop the simulator.

23. Save your project.

Let's next take a look at the program `filter.c` for an outline of what is happening. (The detail is hidden in library functions.)

```
 1  #include <nfp.h>
 2  #include <nfp/me.h>
 3  #include <nfp6000/nfp_me.h>
 4  #include <pkt/pkt.h>
 5
 6  #define PKT_NBI_OFFSET 32
 7  #define OUT_TXQ 0
 8
 9  int
10  is_allowed(__addr40 char *pbuf)
11  {
12      __addr40 char *pkt = pbuf + PKT_NBI_OFFSET;
13      // We make the rather bold assumption that this is
14      // an IP packet in an ethernet frame with no VLAN tag.
15      // If so, IP source address is in pkt[26] to pkt[29].
16      // Allow if least significant byte is odd:
17      int lsbyte = pkt[29];
18      return lsbyte & 0x01 == 1;
19  }
20
21  void
22  main(void)
23  {
24      __xread struct nbi_meta_null nbi_meta;
25      __xread struct nbi_meta_pkt_info *pi = &nbi_meta.pkt_info;
26      __addr40 char *pbuf;
27      __gpr struct pkt_ms_info msi;
28
29      while (1) {
30          pkt_nbi_recv(&nbi_meta, sizeof(nbi_meta));
31          pbuf = pkt_ctm_ptr40(pi->isl, pi->pnum, 0);
32          msi = pkt_msd_noop_write(pbuf, PKT_NBI_OFFSET);
33          if (is_allowed(pbuf)) {
34              pkt_nbi_send(0, pi->pnum, &msi, pi->len,
35                           (MY_NBI == 0 ? 1 : 0), OUT_TXQ,
36                           nbi_meta.seqr, nbi_meta.seq);
37          } else {
38              pkt_nbi_drop_seq(0, pi->pnum, &msi, pi->len,
39                           (MY_NBI == 0 ? 1 : 0), OUT_TXQ,
40                           nbi_meta.seqr, nbi_meta.seq);
41          }
42      }
43  }
```

You will notice a few obvious differences from the programs which you have seen earlier. (For example, more include files.)

A cosmetic difference can be seen on lines 10, 12 and 24-27, where there are type annotations such as `__gpr`, which is defined to be an abbreviation for `__declspec(gp_reg)`. This makes programs slightly less cluttered and more readable.

These abbreviations are `#define`-d in `include/nfp.h` in the `filter` folder.

The core of this program is the infinite loop on lines 29-42 which accepts packets and forwards or drops them depending on the results returned by our function `is_allowed()`.

On line 30, the function `pkt_nbi_recv()` waits for work to arrive in its CTM. When a packet is available, the metadata for that packet will be placed in the `nbi_meta` structure, which is allocated in the thread's read registers. (When work arrives, this metadata is Push-ed to the waiting thread by the CTM.)

> The library functions used in the main-loop, such as `pkt_nbi_recv()` are all declared in `lib/pkt/pkt.h` and defined in `lib/pkt/libpkt.c`.

The metadata received by the thread also specifies where its packet has been placed. However, this is encoded as an island number along with a packet number. So, on line 32 we turn this into a 40-bit pointer using the function `pkt_ctm_ptr40()`.

On line 33 we use `pkt_msd_noop_write()` to arrange for no modifications to be performed on the packet at egress. (It is possible to arrange for bytes to be removed from or added to the start of the packet, and for header values and checksums to be changed *by the egress NBI*, without the ME thread handling the packet having to move data around or do any further calculations. In this case though, we just ask the egress NBI to do nothing.)

On line 34 we call our own function `is_allowed()` to decide whether to forward or to drop the packet. This function is straightforward (even simple-minded). The only subtlety is on line 12 of that function where we make a new pointer `pkt` to the start of the packet data. The parameter `pbuf` holds a pointer to the start of the *buffer* in the CTM, but the start of the *packet* within that buffer is a further `PKT_NBI_OFFSET` bytes further along.

Depending on the value returned by `is_allowed()`, we call either `pkt_nbi_send()` on line 34, to forward the packet, or `pkt_nbi_drop_seq()` on line 38, to drop the packet. (It is necessary to tell the egress NBI that a packet is being dropped, otherwise there will be problems with packet re-ordering and buffer recirculation.)

In either case, whether the packet is forwarded or dropped, the thread chooses the egress NBI based on the value of the symbol `MY_NBI`. You will remember that we earlier set this value using a preprocessor definition in the **Compiler** tab of the Build Settings dialog box. Code running on the MEs in islands 32, 33 and 34 has `MY_NBI=0`, indicating that packets come from `NBI0`. Code running on the MEs in islands 36, 37 and 38 has `MY_NBI=1`, indicating that packets come from `NBI1`. Therefore, when we call `pkt_nbi_send()` or `pkt_nbi_drop_seq()`, we invert our value of `MY_NBI` so as to send packets out of the opposite network interface to the one where they entered.

## 20 Random and subtle points

We wrap up this Annex with a few second-order points, in no particular order:

- The example input PCAP files contain UDP packets with sequentially increasing IP source address. Our filter should let through all the ones with odd numbered source addresses, and it does that successfully. However, if you look in the output PCAP files you will note that the packets are not in exactly the right order. This is a general problem for systems with a pool of workers, such as the NFP-6xxx: when workers start handling packets at similar times, sometimes one finishes faster than another and packets can get swapped around.

  Because of this problem, the NFP-6xxx contains hardware in the ingress NBI to add sequence-numbers to packets as they arrive, and there is more hardware in the egress NBI to buffer and re-order packets as they leave, so that they will go out in the same order they arrived, even in the face of the "pool of workers" problem. (And that's why we need to tell the egress NBI that we are dropping a packet, so that it doesn't wait around for a long time for that packet to arrive from what it imagines must be a very, very slow worker.)

  Why then are packets in the output PCAP files out of order? For the egress NBI to re-order packets, the re-order facility has to be switched on. In this example, the configuration scripts don't do that. (However, this has given us an opportunity to discuss the problem and its solution.)

- Another deficiency of our example, compared with a properly useful filter, is that when packets arrive at an ingress NBI, buffers must be allocated in CTM and IMEM. When an ME thread has processed a packet, it passes on those buffers to an egress NBI, which takes responsibility for them. When the egress NBI is finished with those buffers, they need to be recirculated back to the ingress NBI (or at least to some part of the system which will fill them). This functionality is usually provided by "buffer-list manager" code running on a dedicated ME. This is standard code, but for simplicity our example system does not use it. so after a while it will run out of buffers.

- If you watch the packet statistics, you will see packets arriving faster than they are leaving. (Even taking into account that half of them are being dropped.) Clearly we don't have enough MEs working on the task: to save configuration effort, we only dedicated one ME in each island. If you run the appropriate .list programs on more MEs in each island, the forwarding rate will go up proportionally.

- Those of you who have taken the time to read through some of the library source code, particularly libpkt.c and libnfp.c, may be puzzled as

to how the intrinsic functions defined in this code ever get to be inlined in main-filter.c. Surely these are separate compilation units?

What happens here is that as you might expect, all three of these files are pre-processes and parsed as usual. However, and rather unusually, the compiler then *merges* these at a parse-tree level and then does in-lining and partial-evaluation before finally going on to perform code-generation and dead-code removal on the whole, combined program at once.

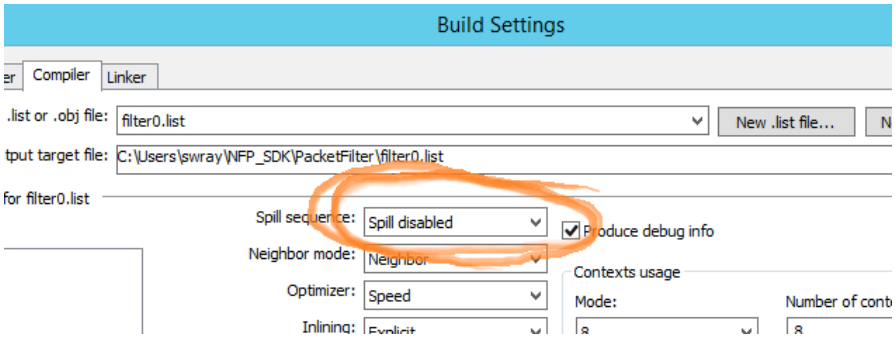
- This last point concerns programming style rather than packet forwarding, per se. On line 17, we have the statement

```
int lsbyte = pkt[29];
```

This is perfectly fine, and the compiler generates code which gets the required data from the CTM using a CPP read command. This value gets put in `lsbyte`, whose location is unspecified, but which is almost certainly going to be a general purpose register.

Now, it's nice to be able to be able to just write normal C like this, but this statement does rather hide the fact that something costly (a CPP transaction) is happening during this assignment. And by not specifying that `lsbyte` *must* be in a register, we risk being surprised later if this value was spilled to some other memory and then later we needed to wait for a CPP transaction to bring it back again.

Thus, as previously noted, it is considered to be good style to be completely explicit about the location of all variables. And furthermore, to completely prevent variables being spilled from registers to other locations, by setting the **Spill sequence** to 'Spill disabled' in the **Compiler** tab of the Build Settings dialog-box:




Also, it is considered clearer to use library functions such as `mem_read64()` (see `lib/nfp/mem_bulk.h`) to move data to and from CTMs and other memories. By using a function like `mem_read64()` you indicate to other programmers that something expensive is happening, rather than hiding this behind a cheap-looking assignment. (At the end of the day, the code generated will be about the same.)

## 21 Over to you

What next? Perhaps you would like to modify the example code to …

- Count packets transmitted / dropped. You might use the atomic add which we created in chapter 15. Or you might use some of the atomic functions provided in the libraries. (Have a look.)

- Parse packet headers more properly, and extract fields. Some useful `structs` are defined in the libraries.

- Look up fields in tables stored in CLS or IMEM to decide what to forward or drop. Table organization would be straightforward for compact tables, but sparse tables would need to use the usual tree or hash-table data structures. Now, there are Memory Engines which can help to speed-up operations using these data structures, by reducing the number of round-trips needed to get an answer. But this is another large topic. Clearly a topic for another Annex …

`stuart.wray@netronome.com`

END