

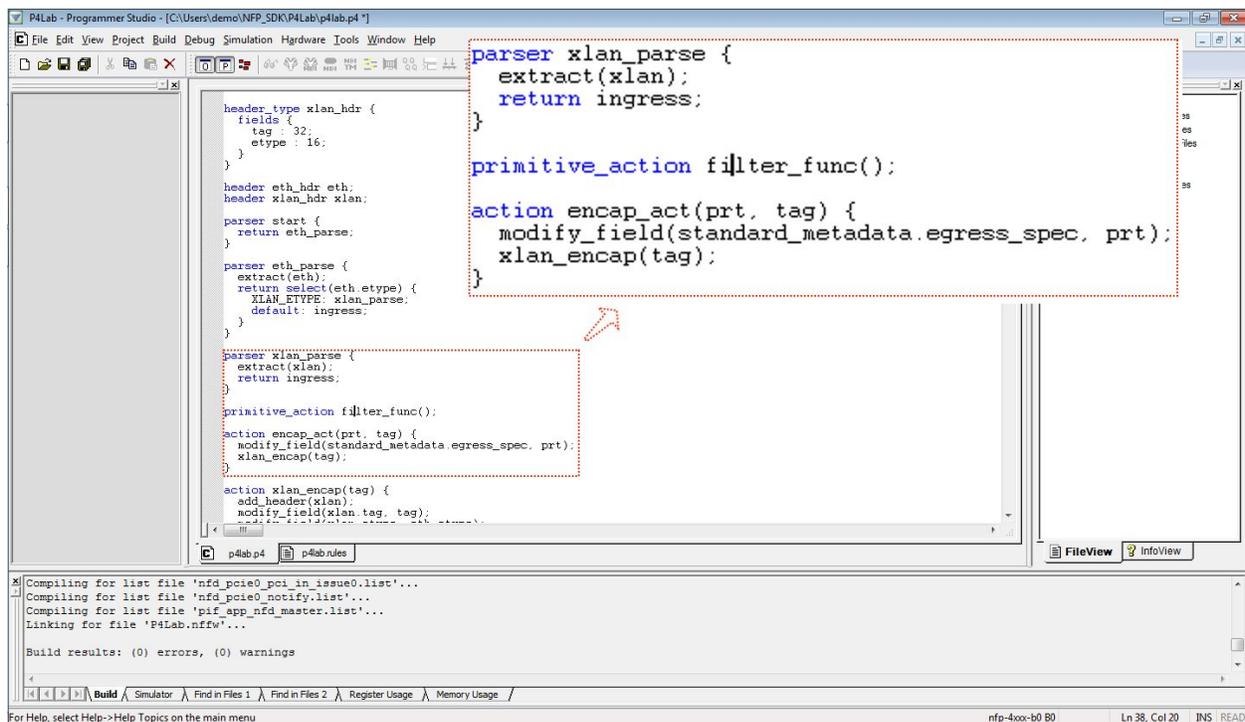
Building a Custom Action with a C Sandbox in P4

In this lab you will be defining a C sandbox function to perform custom action processing on packets matching the same rule set utilized in previous labs. Note, completing lab 1 and lab 2 is a prerequisite for this tutorial, as the instructions in this lab build on the work done previously.

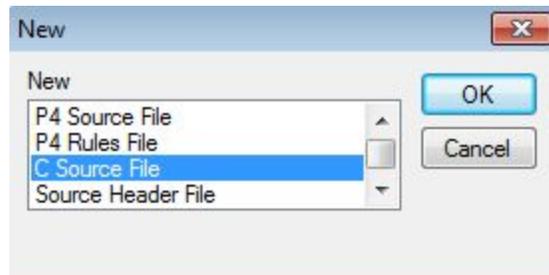
1. Define an entry point for your filter function as a primitive action in your P4 program. Add the following code into your P4 source file after the `xlan_parse` definition block:

```
primitive_action filter_func();
```

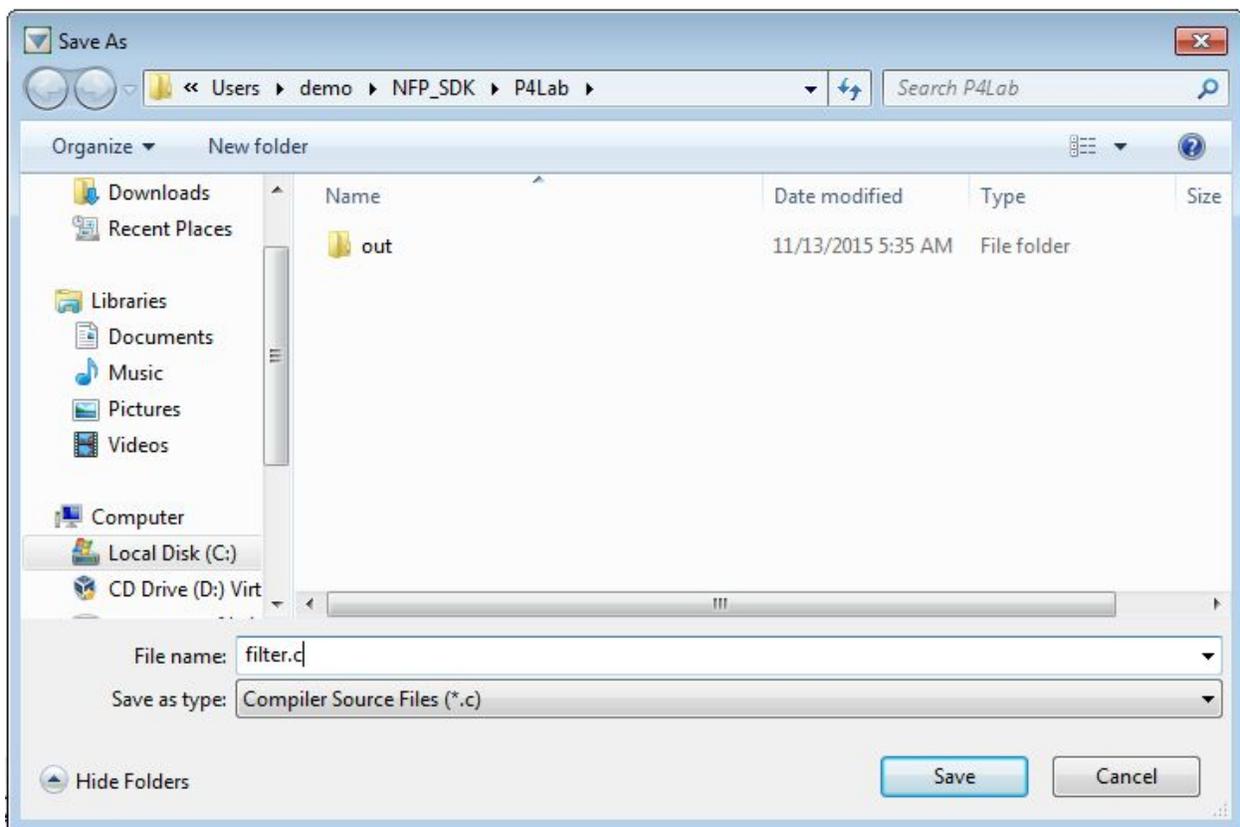
The program should appear as follows:



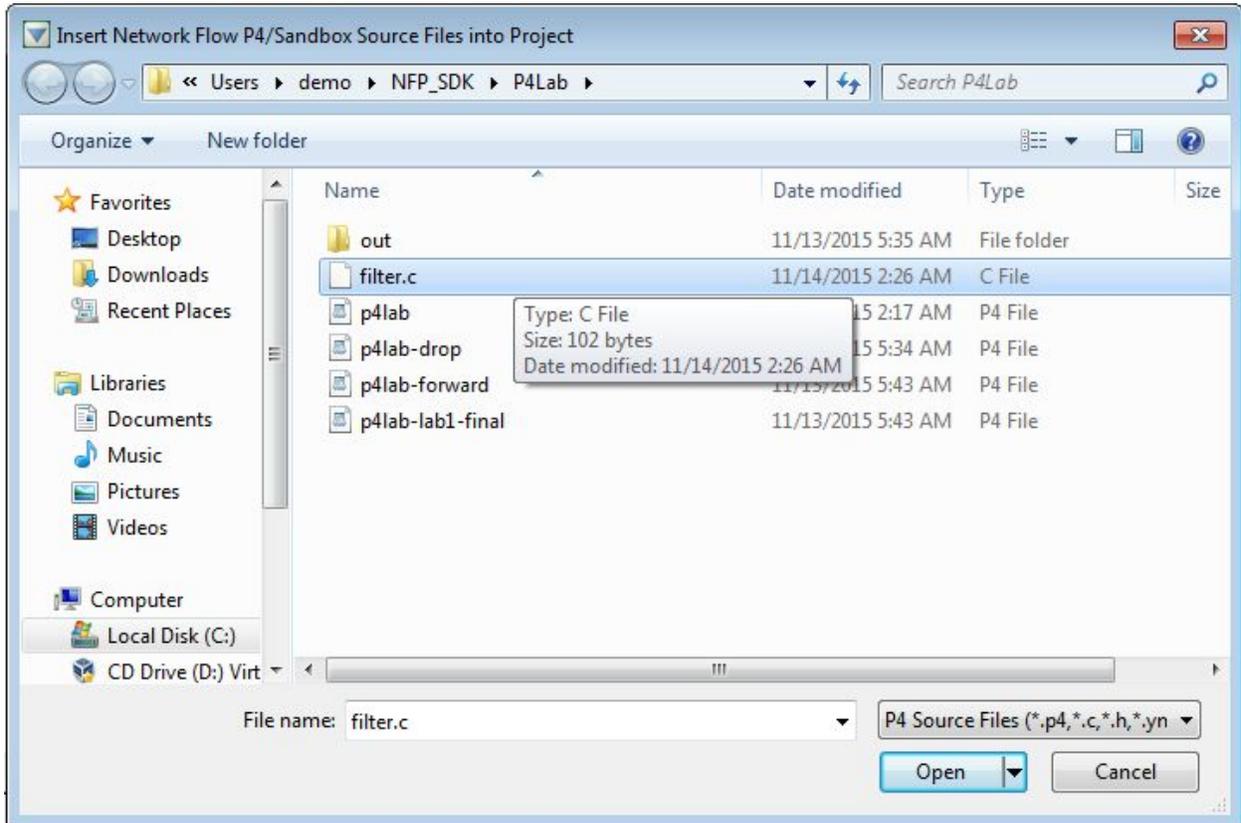
2. Create a new source file in which to place the C implementation for filter_func() by selecting File -> New -> C Source File from the Programmer Studio main menu:



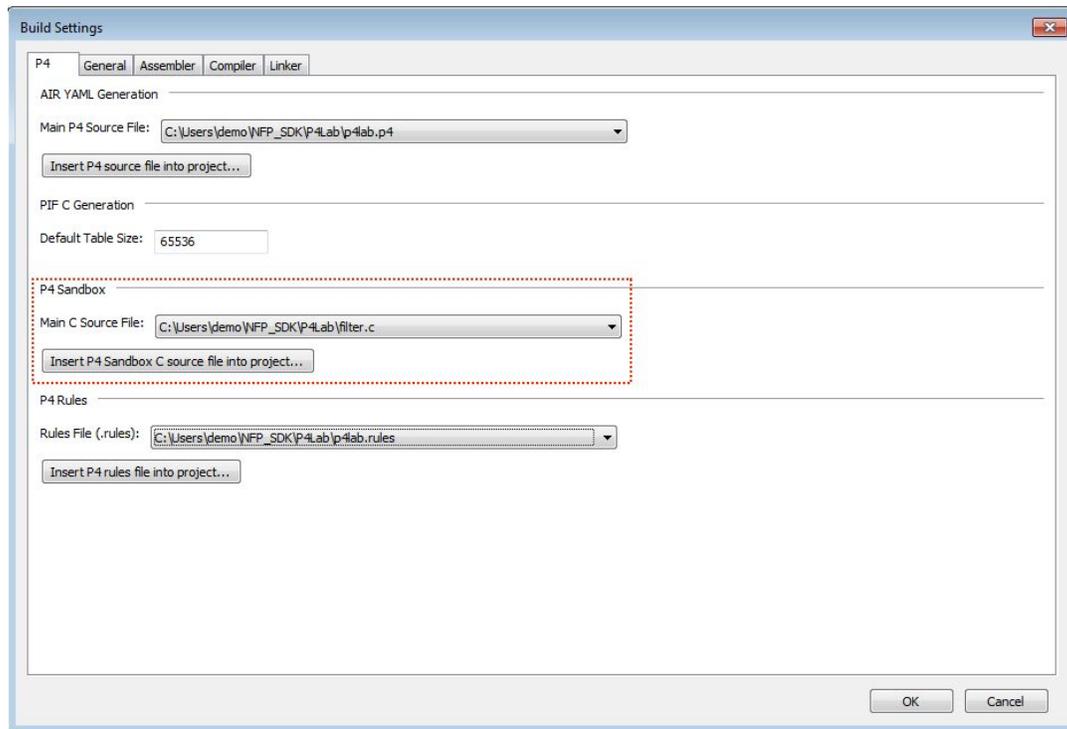
3. Save this newly created C file under a new name by selecting File -> Save As... from the menu and call it filter.c:



4. Next, insert the newly created C source file into the project definition by navigating to the Build -> Settings dialog and click on the “Insert P4 Sandbox C source file into project” button. This should pop up the following file dialog where you can select the filter.c file that was saved in the previous step:



5. Also assign filter.c as the main P4 Sandbox C file for the project in the same Build->Settings window:



6. Insert a basic drop filter into the C file and save it by pressing Ctrl-S:

```
#include <pif_plugin.h>

int pif_plugin_filter_func(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *data)
{
    return PIF_PLUGIN_RETURN_DROP;
}
```

This trivial example filter function will simply drop all packets that invoke it via primitive action in P4. The *headers* argument provides access to the parsed representation of the packet headers and the packet metadata written by your P4 code, while the *data* argument provides access to the action parameters specified by the P4 rule that matched the packet.

7. Modify the P4 `encap_act()` action definition to include a call to your newly defined filter function as follows:

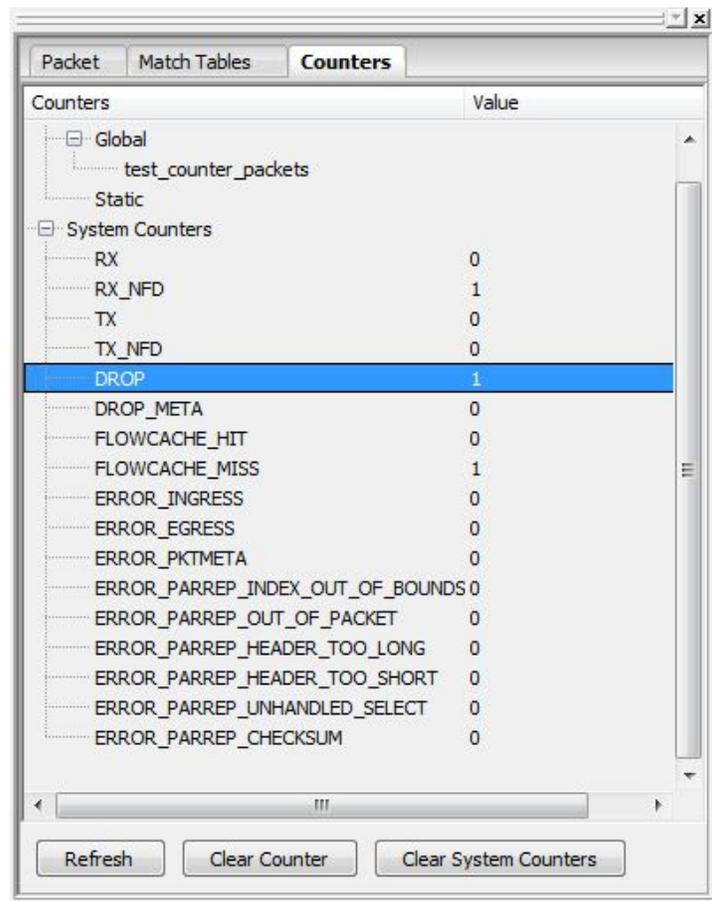
```
action encap_act(prt, tag) {  
    filter_func();  
    modify_field(standard_metadata.egress_spec, prt);  
    xlan_encap(tag);  
}
```

8. You should now be able to rebuild and reload the project in the debugger to verify that all packets that hit the `encap_act()` action are indeed being dropped.

First, send a new packet into the system:

```
tcpreplay -i p4p1 udp.pcap  
sending out p4p1  
processing file: udp.pcap  
Actual: 1 packets (58 bytes) sent in 0.02 seconds.           Rated:  
2900.0 bps, 0.02 Mbps, 50.00 pps  
Statistics for network device: p4p1  
    Attempted packets:           1  
    Successful packets:          1  
    Failed packets:              0  
    Retried packets (ENOBUFS):   0  
    Retried packets (EAGAIN):    0
```

Then, verify that the drop counter was incremented by clicking “Refresh” in the Counters view:



9. Now alter the C filter function in filter.c to forward packets instead of dropping them:

```
int pif_plugin_filter_func(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *data)
{
    return PIF_PLUGIN_RETURN_FORWARD;
}
```

10. Rebuild by pressing F7 and run by pressing F12 and F5 to confirm that packets are able to successfully pass through the system before moving on:

```
tcpdump -i vf0
listening on vf0, link-type EN10MB (Ethernet), capture size 65535 bytes

16:50:38.436411 11:22:33:44:55:66 (oui Unknown) > 22:33:44:55:66:77 (oui
Unknown), ethertype Unknown (0x9999), length 64:
    0x0000:  0000 0082 0800 4500 002c 0001 0000 4011  .....E.,....@.
    0x0010:  665c 0a00 0001 0a00 0064 0bb8 0fa0 0018  f\.....d.....
    0x0020:  97c1 0001 0203 0405 0607 0809 0a0b 0c0d  .....
    0x0030:  0e0f
```

11. These examples demonstrate a trivial sandbox filter function in order to show how to integrate C functions into a P4 project. Next, you will build a somewhat more interesting filter function and for this purpose you will require access to various IPv4 fields of a packet. Add the following header definition to the P4 parser source, directly below the eth_header block:

```
#define IPV4_ETYPE 0x0800

header_type ipv4_hdr {
    fields {
        ver : 4;
        ihl : 4;
        tos : 8;
        len : 16;
        id : 16;
        frag : 16;
        ttl : 8;
        proto : 8;
        csum : 16;
        src : 32;
        dst : 32;
    }
}
```

12. Instantiate the newly defined IPv4 header. For neatness, this can be done amongst the header instantiations for `eth_header` and `xlan_header`, as follows:

```
header eth_hdr eth;
header ipv4_hdr ipv4;
header xlan_hdr xlan;
```

13. Incorporate the new IPv4 header into the parser tree by replacing the parser descriptions for `eth_header` and `xlan_header` with the following (also add the new IPv4 parser block):

```
parser eth_parse {
  extract(eth);
  return select(eth.etype) {
    XLAN_ETYPE: xlan_parse;
    IPV4_ETYPE: ipv4_parse;
    default: ingress;
  }
}

parser xlan_parse {
  extract(xlan);
  return select(xlan.etype) {
    IPV4_ETYPE: ipv4_parse;
    default: ingress;
  }
}

parser ipv4_parse {
  extract(ipv4);
  return ingress;
}
```

14. At this stage, the P4 grammar understands IPv4 packet headers. Next, you will see how to access these newly defined header fields from the C sandbox function. Replace your existing filter function with the following code. This function will drop all non-IPv4 traffic and packets destined for 10.0.0.100:

```
#include <pif_plugin.h>

#define IP_ADDR(a, b, c, d) ((a << 24) | (b << 16) | (c << 8) | d)

int pif_plugin_filter_func(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *data)
{
    PIF_PLUGIN_ipv4_T *ipv4;

    if (! pif_plugin_hdr_ipv4_present(headers)) {
        return PIF_PLUGIN_RETURN_DROP;
    }

    ipv4 = pif_plugin_hdr_get_ipv4(headers);

    if (ipv4->dst == IP_ADDR(10,0,0,100)) {
        return PIF_PLUGIN_RETURN_DROP;
    }

    return PIF_PLUGIN_RETURN_FORWARD;
}
```

15. As an exercise, verify the behavior of this function by rebuilding the project, reloading it, sending various packets into the system and monitoring the DROP and TX counters.

16. Finally, as a more interesting example, modify the filter function to no longer drop packets to 10.0.0.100, but instead maintain a histogram of IP protocols for all IPv4 traffic destined for that host:

```
#include <pif_plugin.h>

#define IP_ADDR(a, b, c, d) ((a << 24) | (b << 16) | (c << 8) | d)

__shared uint32_t proto_counters[256];

int pif_plugin_filter_func(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *data)
{
    PIF_PLUGIN_ipv4_T *ipv4;

    if (! pif_plugin_hdr_ipv4_present(headers)) {
        return PIF_PLUGIN_RETURN_DROP;
    }

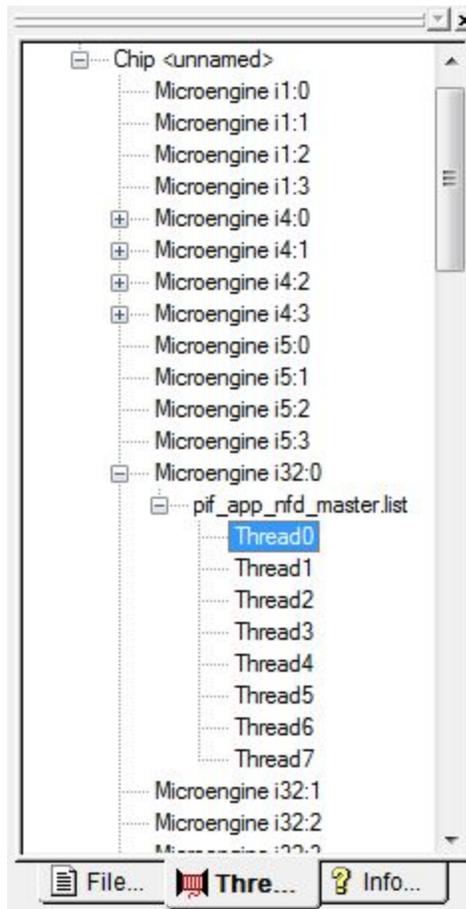
    ipv4 = pif_plugin_hdr_get_ipv4(headers);

    if (ipv4->dst == IP_ADDR(10,0,0,100)) {
        proto_counters[ipv4->proto]++;
    }

    return PIF_PLUGIN_RETURN_FORWARD;
}
```

17. Add a data watch for the proto_counters[256] table as follows:

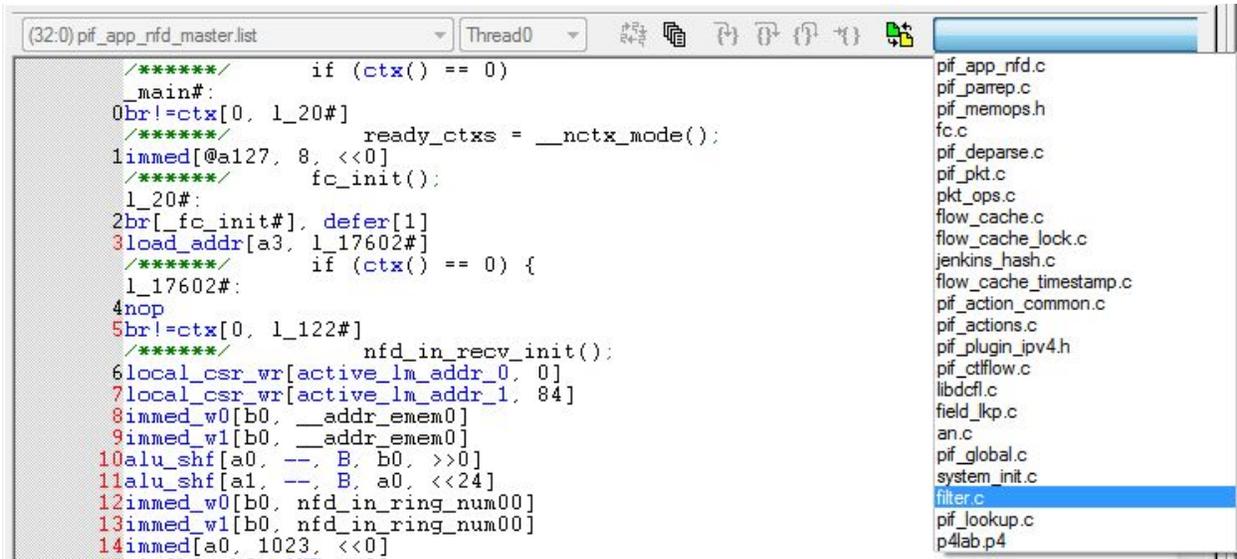
- a. In the debugger, select the active thread to watch (for the purposes of simplifying this workshop, you will find that the P4 generated code is only running on microengine 0 of island 32):



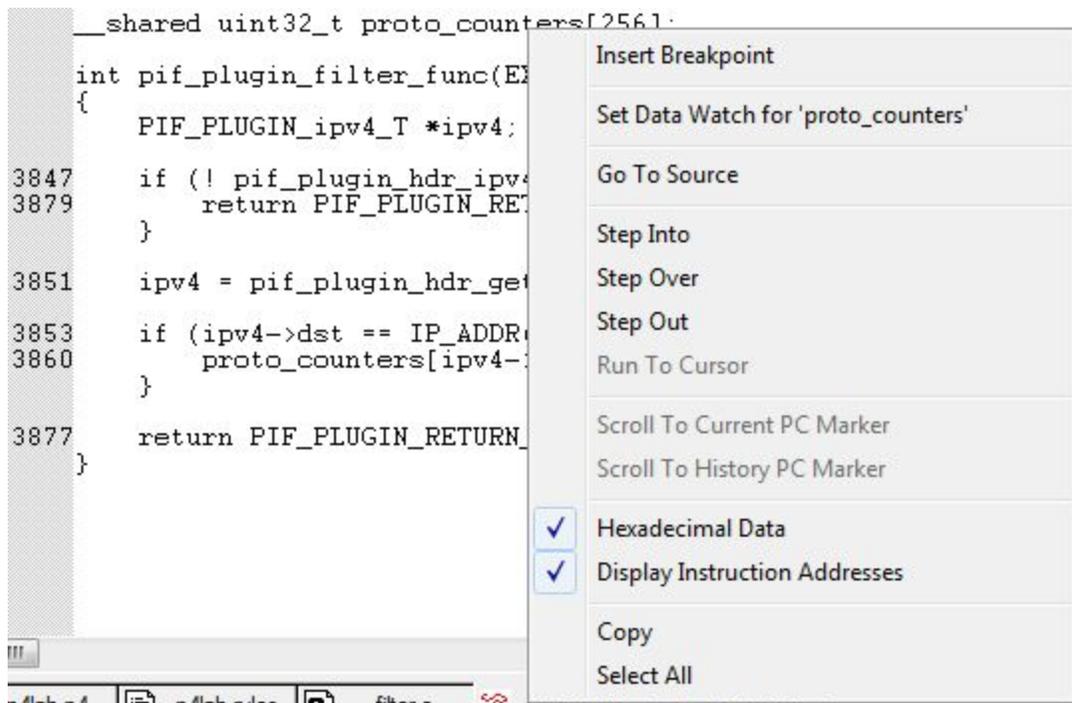
- b. Ensure that the microengines are in a stopped state by pressing the red light button in the debugger controls toolbar (this is necessary because the histogram is stored in local memory which is not accessible to the debugger while the microengines are running):



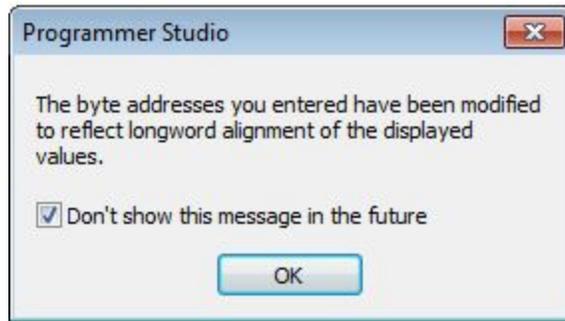
- c. Select the source file containing the filter function implementation in the main debugger code window:



- d. Right click the variable definition in the C source code and select the “Set Data Watch for ‘proto_counters’” option from the context menu:



- e. Select the “Don’t show this message in the future” checkbox to ignore the alignment notices that pop up and press “OK” to continue:



- f. Experiment by sending various packets of different IP protocols through the system to the matched destination and other addresses. The histogram data structure can now be viewed in the data watch window at the bottom of Programmer Studio to see the counts of each protocol type received by the selected destination (remember to take care to restart the microengines to process new packets and stop them again before inspecting the histogram):

Name	Value	Description
proto_counters[4]	0x00000000	unsigned int
proto_counters[5]	0x00000000	unsigned int
proto_counters[6]	0x00000003	unsigned int
proto_counters[7]	0x00000000	unsigned int
proto_counters[8]	0x00000000	unsigned int
proto_counters[9]	0x00000000	unsigned int
proto_counters[10]	0x00000000	unsigned int
proto_counters[11]	0x00000000	unsigned int
proto_counters[12]	0x00000000	unsigned int
proto_counters[13]	0x00000000	unsigned int
proto_counters[14]	0x00000000	unsigned int
proto_counters[15]	0x00000000	unsigned int
proto_counters[16]	0x00000000	unsigned int
proto_counters[17]	0x00000006	unsigned int

18. Freestyle:

```
#include <pif_plugin.h>

int pif_plugin_filter_func(EXTRACTED_HEADERS_T *headers, MATCH_DATA_T *data)
{
    // TODO: your code goes here

    // Idea 1: Stateful flow tracking
    // Idea 2: High speed web server running on iNIC
    // Idea 3: Caching reverse web proxy
}
```

THE INFORMATION IN THIS DOCUMENT IS AT THIS TIME NOT A CONTRIBUTION TO P4.ORG.