

# Programming NFP with P4 and C

---

THE NFP FAMILY OF  
FLOW PROCESSORS  
ARE SOPHISTICATED  
PROCESSORS  
SPECIALIZED TOWARDS  
HIGH-PERFORMANCE  
FLOW PROCESSING.

---

## CONTENTS

INTRODUCTION .....	1
PROGRAMMING THE NFP WITH P4.....	1
PROGRAMMING THE NFP WITH C .....	5
CONCLUSION .....	9

## INTRODUCTION

This whitepaper describes the programming options for the Netronome Network Flow Processor (NFP) used on the Agilio® SmartNICs. The Agilio SmartNIC is supplied with Agilio Software, which has a comprehensive set of features mainly based on Open vSwitch offloads. In the case that there is a requirement for customization of the NFP data path by users, the NFP can also be programmed for the custom packet/flow processing using P4 and C languages.

The NFP family of flow processors are sophisticated processors specialized towards high-performance flow processing. The NFP programming environment comes with a set of libraries, and common packet processing functions. The NFP has multiple PCIe Gen-3 interfaces for high-speed data/packet transfer between the host and the NFP. Software running on general purpose CPUs can also control the behavior of the data plane running on the NFP through the API calls. The NFP software typically implements the data plane of a networking application with the control plane (and additional data plane code) running on the host.

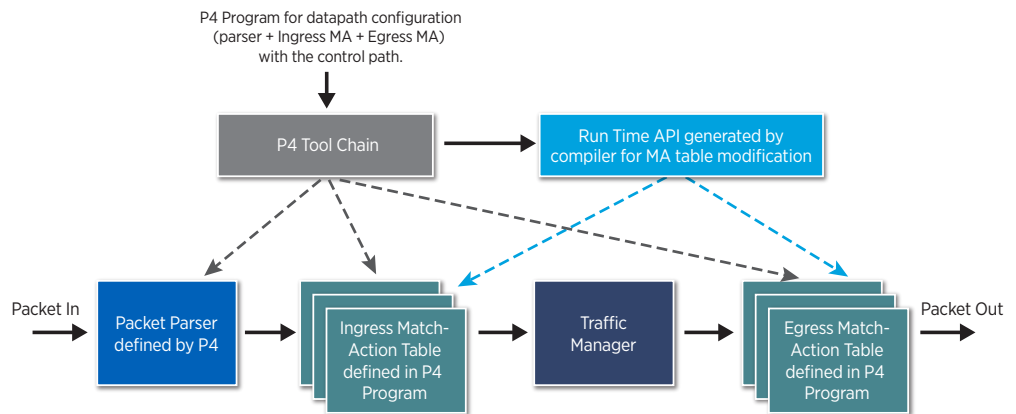
The NFP comes with a Software Development Kit (SDK), which has a compiler, linker and cycle accurate simulator in an Integrated Development Environment running as a graphical user interface (GUI) on windows platform. The SDK also comes with the command line interface versions of the compiler, linker and simulator necessary for running and debugging the code on an Agilio SmartNIC. In this paper, we provide an introduction of NFP programming using the P4 and C languages. The SDK provide complete software development and debug environment for packet/flow processing programs written in P4 and C languages.

## PROGRAMMING THE NFP WITH P4

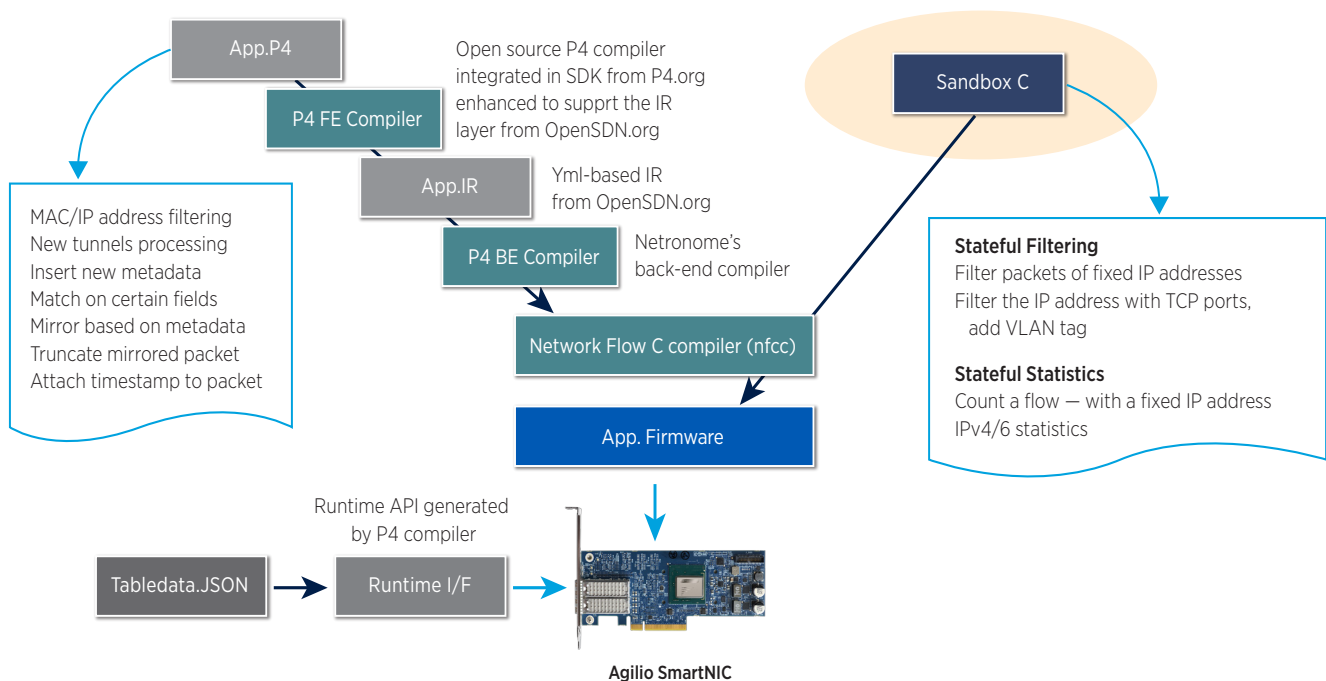
P4 is a target independent network programming language where users can write the forwarding behavior of the network devices (ASIC/NPUs/FPGAs) using the standard forwarding model defined in the P4 architecture. P4 allows the user to create their own headers and

protocols along with their processing behavior in a networking device.

The packet-processing model proposed by the P4 language is shown in figure below. The user writes the datapath of a network device in P4 language without any knowledge of the target hardware device (ASIC, FPGA or NPU). The tool chain (compiler and linker), developed by the device vendor converts the P4 program into the device specific firmware. The P4 tool chain also generates a run time API (similar to the OpenFlow model) to allow the match action table modification.



The SDK supports P4 syntax as defined on the P4 specifications published on P4 consortium website ([www.p4.org](http://www.p4.org)). Netronome has integrated the open source P4 compiler in the SDK to generate an intermediate representation (App.IR) of the P4 program in the yaml format, which is further compiled by the P4 back-end compiler to generate a C program for the data path on the NFP. P4 programming support on the SDK is shown in the figure below.



Since P4 is meant for hardware independent programming (flow processing), the user does



not need to be aware of any NFP specific data structures. The P4 compiler and linker automatically maps the different part of the P4 program into the NFP internal resources in an efficient manner.

The figure below shows an example of a very simple P4 program.

```
header_type eth_hdr {
    fields {
        dst : 48;
        src : 48;
        etype : 16;
    }
}

header eth_hdr eth;

parser start {
    return eth_parse;
}

parser eth_parse {
    extract(eth);
    return ingress;
}

action drop_act() {
    drop();
}

table in_tbl {
    actions {
        drop_act;
    }
}
```

The above program has:

1. Header definition
2. Parser with packet field extraction
3. Action table
4. Control flow for ingress packets

The NFP software development kit (SDK) has an inbuilt editor for editing and compiling the P4 programs and generating the firmware for loading on the Agilio SmartNIC. When the P4 program is compiled using the SDK, parse graph, ingress/egress packet processing flow graphs are generated. The P4 compilation also generates the packet processing pipeline code in yaml language based intermediate representation (IR) format.



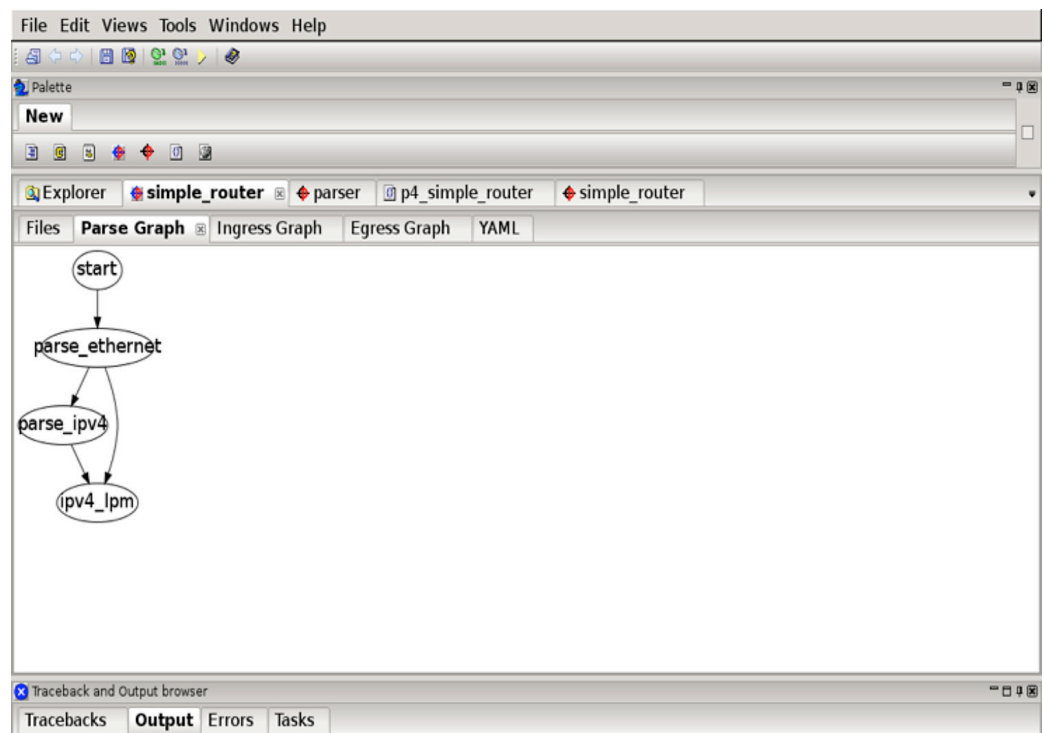
FOR BOTH P4  
AND C LANGUAGE  
PROGRAMMING,  
NETRONOME PROVIDES  
THE COMPREHENSIVE  
LIBRARIES AND  
LOW-LEVEL ACCESS  
FUNCTIONS FOR  
STANDARD PACKET  
PROCESSING.

```

File Edit Views Tools Windows Help

Palette
New

Explorer simple_router p4_simple_router simple_router
Files Parse Graph Ingress Graph Egress Graph YAML
100 src_lineno : 25
101
102 parse_ipv4 :
103   type : parse_state
104   extracts :
105     - ipv4
106   src_filename : /home/dgeorge/work/scratch/p4demo_2015_06_26/p4demo/p4src/includes/parser.p4
107   src_lineno : 62
108
109 #####
110 # Parser
111 #####
112
113 parser :
114   type : parser
115   format : dot
116   start_state : start
117   implementation : >-
118     digraph {
119       start -> parse_ethernet [value="default", mask="none"]
120       parse_ethernet -> parse_ipv4 [value="0x800", mask="none"]
121       ... ethernet -> just too fast to be written
122     }
123
Traceback and Output browser
Tracebacks Output Errors Tasks
Generating AIR output for /home/dgeorge/work/scratch/p4demo_2015_06_26/p4demo/p4src/simple_router.p4
parsing successful
  
```



The P4 back-end compiler compiles the yaml program into the C program, which can be compiled and linked to generate the NFP firmware using the network flow C compiler (NFCC). The firmware generated by the P4 code is loaded on multiple processing engines (referred as flow processing cores in the NFP), each of which can independently process packets according



to the packet processing code written as a P4 program. An engine idles in a loop waiting for a packet to arrive and start processing when a new packet arrives. Management logic in the processor provides the execution guarantees required by P4 program. A P4 program has to be developed assuming it is running in switch architecture as specified in P4 specification.

Optionally, a P4 processing can be mixed with the C processing as C provides architecture aware stateful processing. This is termed as a P4 data path with C sandbox. For inclusion of the C sandbox into the P4 code, users need to define the action as a “primitive\_action” which is a P4 construct.

The following example illustrates the use of the C sandbox with the P4 code.

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

```
primitive_action filter_func();
```

If the compiler encounters a “primitive\_action” in a P4 program, it inserts the C function call for that action which is specified in a separate C file in the SDK project. Below is an example of the plugin C sandbox function.

```
#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;
}
```

## PROGRAMMING THE NFP WITH C

The C programming language is a most efficient way of programming the Agilio SmartNIC as it can take advantage of NFP architecture specific data structures. Agilio software features are also implemented as C programs. The C programming on the NFP is slightly different from the host-based generic C programming, as the NFP data structures and memories are specific to the NFP architecture, so it is similar to any custom embedded programming.

The C programming language for the NFP is supported by a highly optimizing NFCC. The

---

**THE C PROGRAMMING LANGUAGE IS A MOST EFFICIENT WAY OF PROGRAMMING THE AGILIO INTELLIGENT SERVER ADAPTER AS IT CAN TAKE ADVANTAGE OF NFP ARCHITECTURE SPECIFIC DATA STRUCTURES.**

---

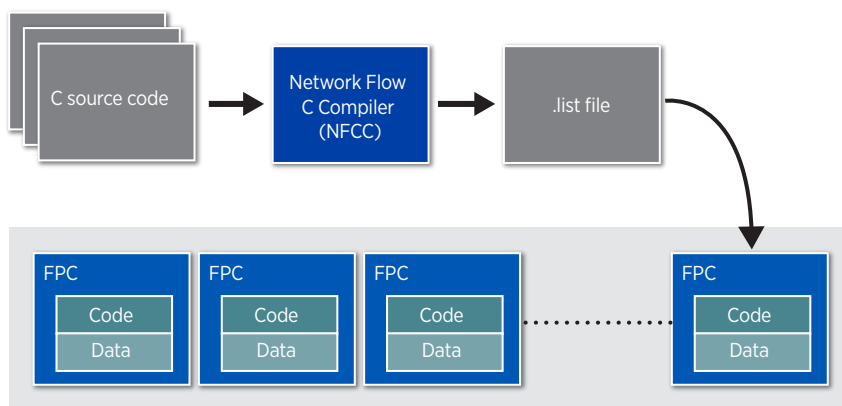
NFCC compiler offers several “extensions” to the C programming language, mostly through annotations, which allow a programmer to have better control over the generated code. This ultimately imposes a small number of restrictions on the programmer, which are rooted in the specifics of the Netronome flow processing cores architecture.

The Flow Processing Cores (FPCs) are fairly standard, RISC based, multi-threaded cores, which can be programmed in a variant of C. What distinguishes the NFP from general purpose CPUs is that the FPCs are connected to a number of functional units, implementing specialized functionality aimed at accelerating different aspects of packet processing.

The FPCs are distributed across the NFP in island architecture and each FPC island has multiple flow processing cores. Each FPC core has an ALU with its own code and data memory.

Each FPC has 8 hardware contexts or threads. These threads share the same ALU and only one of them is actively running at a given time. The threads in a FPC are non-preemptive and the thread scheduling is done explicitly and cooperative: A context must explicitly release control (yield) for other contexts to run. This non-preemptive nature significantly simplifies synchronization within a FPC,

Figure below shows the C programming methodology of FPCs.



A C program is compiled from a number of C source-code files (and supporting header .h files) which are linked together into a .list file. Each .list file represents one complete program, and copies of this program to be loaded onto one or more specified FPCs. When we want different FPCs to run different programs, the compiler must produce different .list files, each compiled from particular C source code files, and to specify which FPC is to be loaded with which .list file. (Of course, the user can share C source-code and header files across several programs; in that case the .list file for each program would include the shared code.)

Along with the code store and data store for each FPC there are four other kinds of memories, which are accessible by the FPCs (C programs) through the keywords and constructs defined in Netronome Compiler User Guide:

- Cluster Local Scratch (CLS)
- Cluster Target Memory (CTM)
- Internal Memory (IMEM)
- External Memory (EMEM)



The above memories can work as packet header and data storage, and they have different densities and latencies. All of CLS, CTM, IMEM and EMEM contain multiple functional units or “Memory Engines” which do many more operations than simple read and write. Netronome provides the command and libraries to access those memories.

Below is an example of the simple C program for array reversal in the CTM memory, notice that the arrays are declared in one of the memories (Cluster Target Memory or CTM) described above. In the case no specific memory is assigned the compiler assigns it by itself but for the efficient flow processing, it is important to explicitly understand and declare the storage for the data structures.

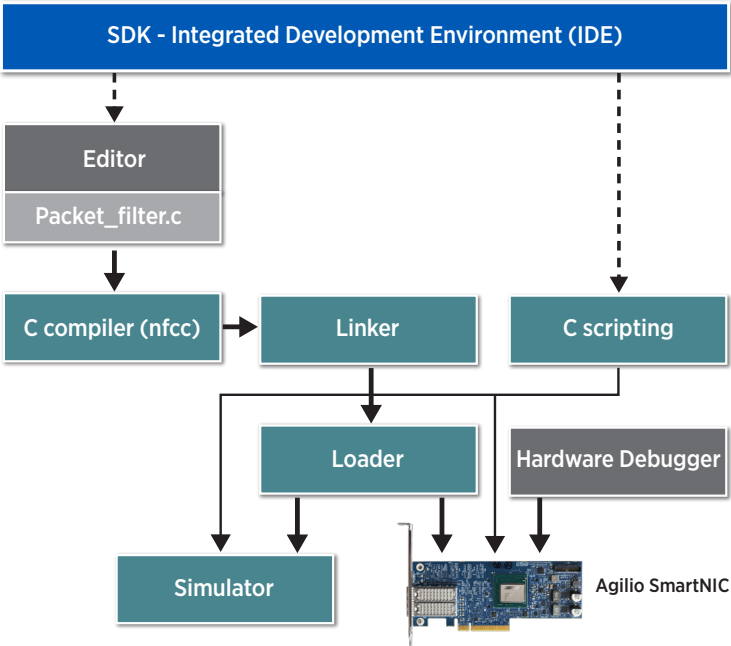
```
#include <nfp.h>
__declspec(ctm export scope(island))
int old[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
__declspec(ctm export scope(island))
int new[sizeof(old)/sizeof(int)];

int
main(void)
{
    if (__ctx() == 0) {
        int i, size;
        size = sizeof(old)/sizeof(int);
        for (i = 0; i < size; i++) {
            new[i] = old[size - i - 1];
        }
    }
    return 0;
}
```

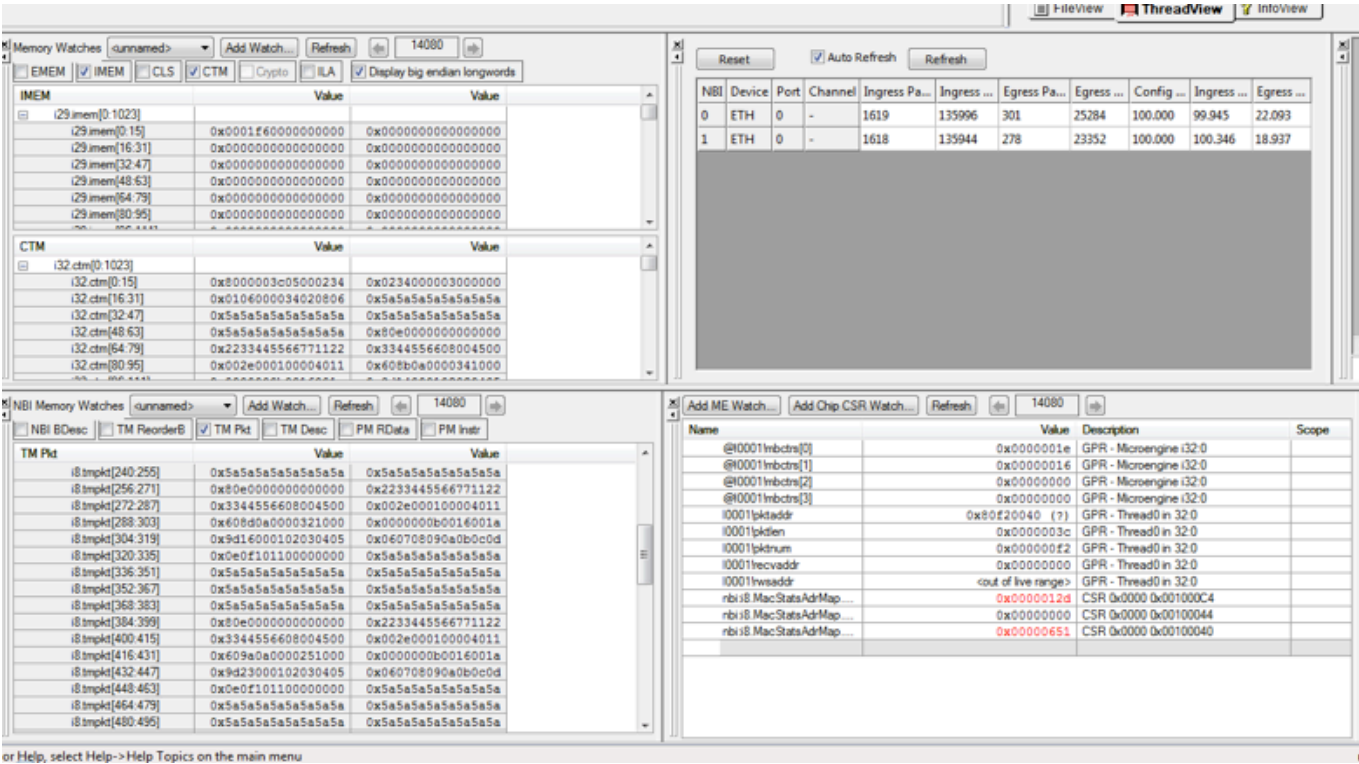
The C programs can be compiled and linked to generate the NFP firmware using the SDK. The SDK runs on Windows platform as a graphical user interface. The SDK has integrated simulator with a complete view of the NFP memory contents, C program variables and thread execution history which provide simplified debug and development environment. Using the SDK, the programmer can insert break points into the programs and can run the C programs step by step on each of the FPC threads.

C programs can also be compiled and linked using the command line tool chain running in standard Linux environments.

The figure below represents the compilation of the C programs for NFP using the SDK.



The SDK debug and watch window for a C program is shown in the figure below. As the figure shows the SDK has different memory watch windows and also allows the visibility of variables declared in the C program.



The SDK also has a hardware debugger, which runs on the host with the Agilio SmartNIC and interacts with the NFP through the host PCIe interface. The hardware debugger communicates with the SDK through a TCP connection. Using the hardware debugger, the C programs





can be downloaded debugged on Agilio SmartNIC in real time. The advantage of using the hardware debugger is that the program execution and debugging can be performed at hardware speed.

## CONCLUSION

As described throughout this paper the NFP comes with full P4 and C languages programming support. Though most of the Agilio SmartNIC features are already implemented in the Agilio software, such as OVS offload, tunnel encapsulation and de-capsulation, load balancing etc., the P4 and C language programming allows the user to implement the customization of the packet processing datapath.

For both P4 and C language programming, Netronome provides comprehensive libraries and low-level functions for standard packet processing. These libraries and function calls which include packet read-write from NFP memories, key lookup, hash and checksum calculations (and more) help a user to focus on programming at a higher level without getting into the lower level architectural details of the NFP.