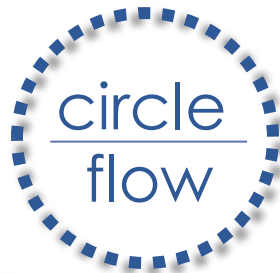

architecture and integration



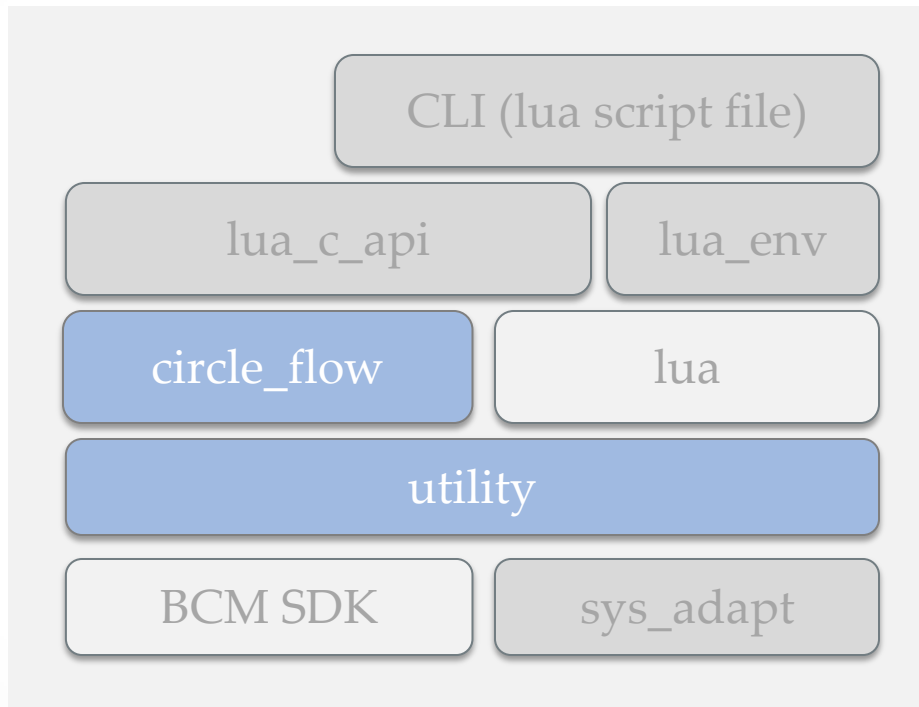
overview




- whole picture
- design principle
- integration guide
- implementation detail

note:

it is necessary to have a basic knowledge on BCM switch chip and SDK before starting.

whole picture

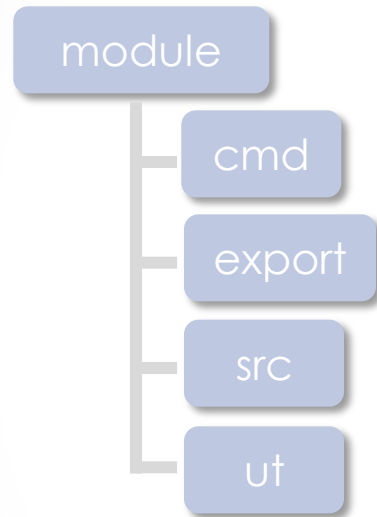


-  : kernel module
-  : system/application module (optional)
-  : third party module

design rule

- resource optimization first
- support multiple switch chip(same model) on board
- open platform: API + lua script
- error handling via exception
- multiple thread safe
- adapt both C++ 98 and C++1x compiler

directory & makefile



#example of makefile

```
SRCS = $(shell /bin/ls -1 src/*.cpp)
SRCS += $(shell /bin/ls -1 src/xxx/*.cpp)
...

OBJS = $(SRCS:.cpp=.o)

#for SDK header file including
FLAG += -DINCLUDE_L3 -DBCM_ESW_SUPPORT
        -DSDK_DIR/include

#for circle_flow hal implementation
FLAG += -DCF_BCM_56334
```

- external API declaration located under “export”
- internal implementation located under “src”
makefile need to be added, refer the example above
- unit test source code located under “ut”
- command line implementation located under “cmd”

unit test

- **test framework**
opmock (0.9x) + gtest
the reason of chosen opmock is mainly because of its un-intrusive mock method which avoids the ut code pollution.
there's few bugs in opmock, for example, in case of the function return type is constant, the auto-generated mock code have syntax error. we have put a shell script in makefile to fix this automatically.
few place in code needs different implementation for ut environment, which is done by compile flag "ENV_UT", mostly located in lua_c_api module.
- **setup**
linux + gcc(g++)
cd ut, make, to run ut for all
or go into specific sub dir, and make, to run ut locally
- **fit for**
ut is valuable for most logical or algorithm part of code, but not for hardware configuration stuff.
for the part of code which needs ut coverage, divide it into separated file, and write corresponded ut case for it.

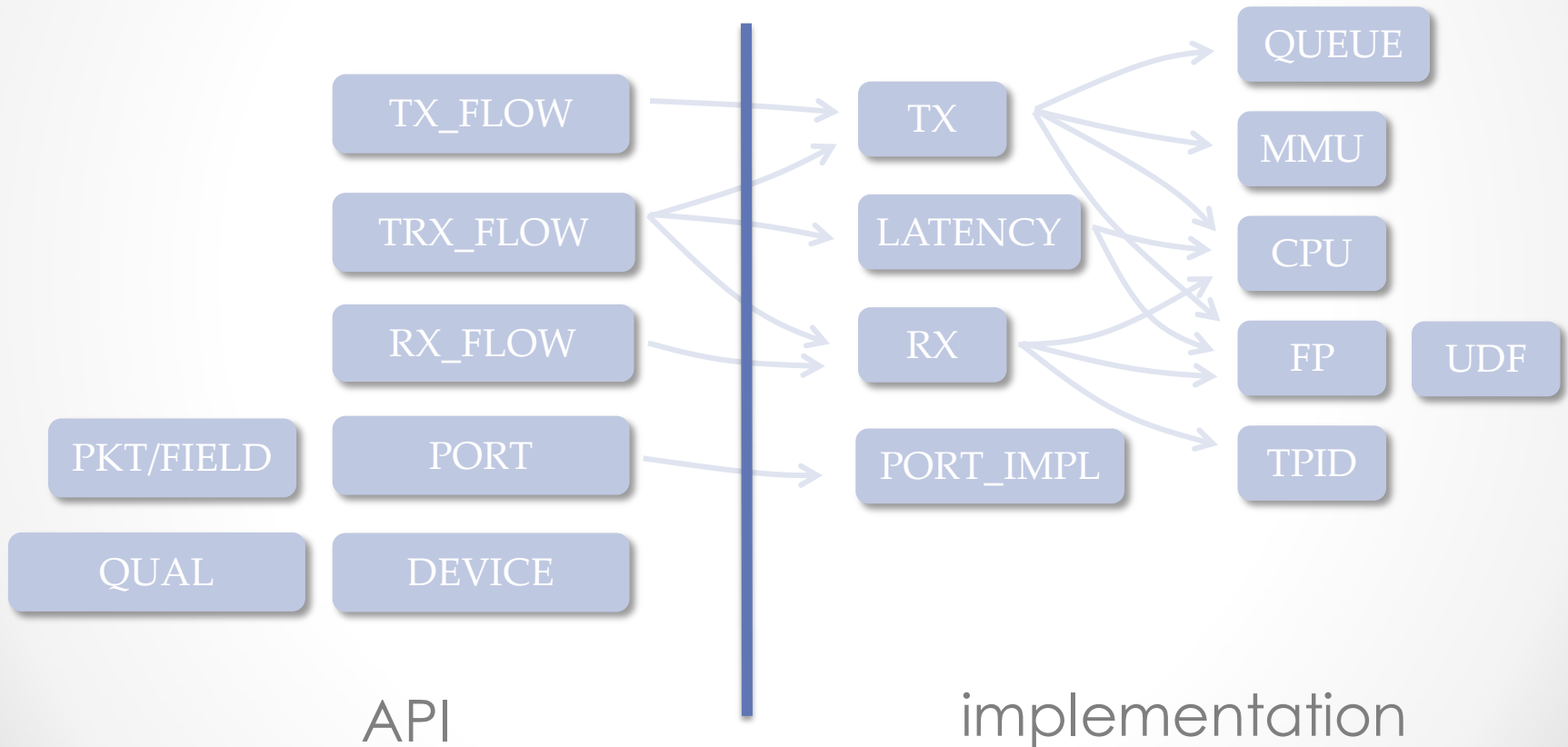
dependency

- **switch chip related**
some part of implementation depends on hardware specs, for example, resource dimension, or need direct touching on hw register.
all of them are guarded by compile flag "CF_BCM_XXX"
pls search "CF_BCM_56334", make accordingly change, while integrating.
- **BCM SDK implementation related**
there's only one place depends on SDK internal implementation to get the index of FP policer which combined to one FP entry :
(policer_hw_index()).
which is verified on v5.6.5 ~ v6.3.4, pls make sure before paired with other version.
- **system related**
encapsulate by "sys_adapt" module, detail elaborates later.
- **link lib**
with C++1xcompiler, besides std lib, nothing else
with C++98compiler, besides tsd lib, boost lib also required

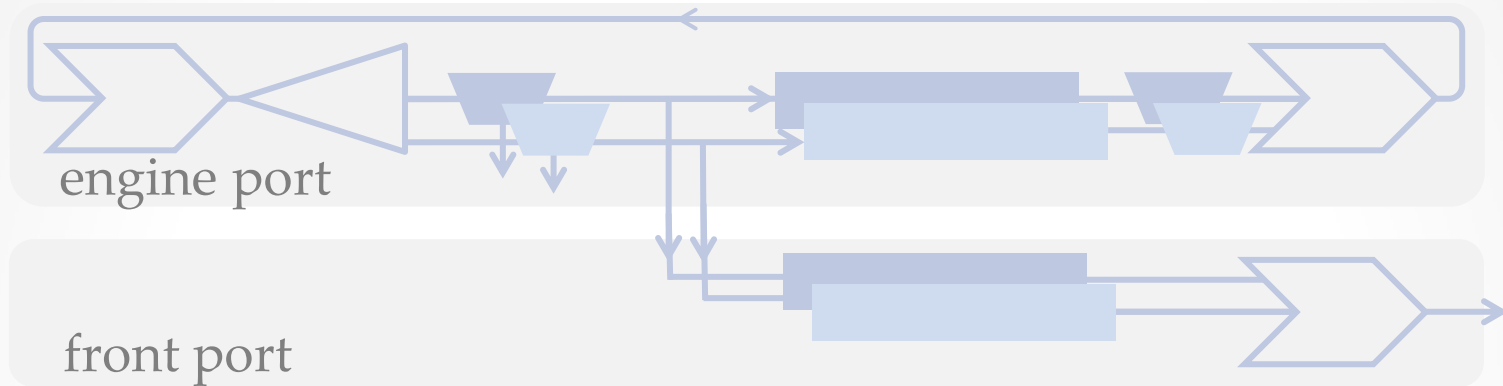
integration guide

- **circle_flow**
copy "circle_flow" module, review related parts guarded by "CF_BCM_56334"
flag, adjust it if necessary.
add your own makefile
- **utility, sys_adapt**
detail elaborated later
C++1x compiler is preferred, which help to save porting efforts.
- **lua script**
download and integrate lua official source code, comment out "main"
function, adjust or rewrite makefile
copy "lua_env" and "lua_c_api" module, make own makefile
copy lua scripts which under "lua_script/cflow" to the root directory of your
target running device (not compiling device).
- **ut framework**
download official source code of "[gtest](#)", "[swig](#)" and "[opmock](#)", intergrate to
the path of "ut_framework".
cd "opmock/support", make opmock.o

circle flow insight



port pair



- **one function unit is build by a pair of ports**
engine port is put to loopback and generate the traffic
front port connect and pump the traffic to the external device
- **more choice for physical interface**
it is possible to bring both ports to the front panel, with different physical type, for example, one is RJ45, the other is SFP.
when one interface works as front port, the other works as engine port. this gives more choice for connectivity.

TX flow

- **functionality**

instance of TX flow provides traffic rate control including burst, statistics.

the content of traffic could be the repeat of single pkt, or a sequence of variant pkts.

- **resource**

a dedicated queue to control transmitting rate

enough MMU to buffer pkts pending in queue

FP entry to control burst, replication and queue assigning

once traffic stops (manually or link down), free all resource above

RX flow

- **functionality**

RX flow instance is used for statistics and snooping pkts on qualified traffic flow

- **qualify rule**

predefined layer 2/3/4 fields

user defined fields: offset + data/mask

it is need to know, FP(TCAM) works in the way of sequenced lookup. assume we have 2 instance of RX flow, with different qualify rule. one incoming pkt is able to match both of them. the result will be, only the instance located prior will be matched and increasing counter.

- **resource**

FP entry, UDF window

CPU RX

global TPID

user define field

- **functionality**

UDF allows user to define customer qualify field, which is used for RX flow and the flow track field of TRX flow.
it is able to cover the range of offset 0~127 bytes in pkt.

- **mechanism**

according to BCM SDK, UDF object must be created before FP group object. this limitation can not meet the requirement of dynamic creation of UDF object. we have to pre-create UDF object, and directly touch HW register to manage its offset windows resource.

- **resource**

compatible to BCM XGS line product, with just little tuning on UDF dimension specs (block and chunk, capsuled by UDF_HW_SPECS)
UDF resource is very limited, it is possible to have failure due to out of resource.

TRX flow

- **functionality**

bind TX flow and RX flow together, and provide latency measurement additionally.

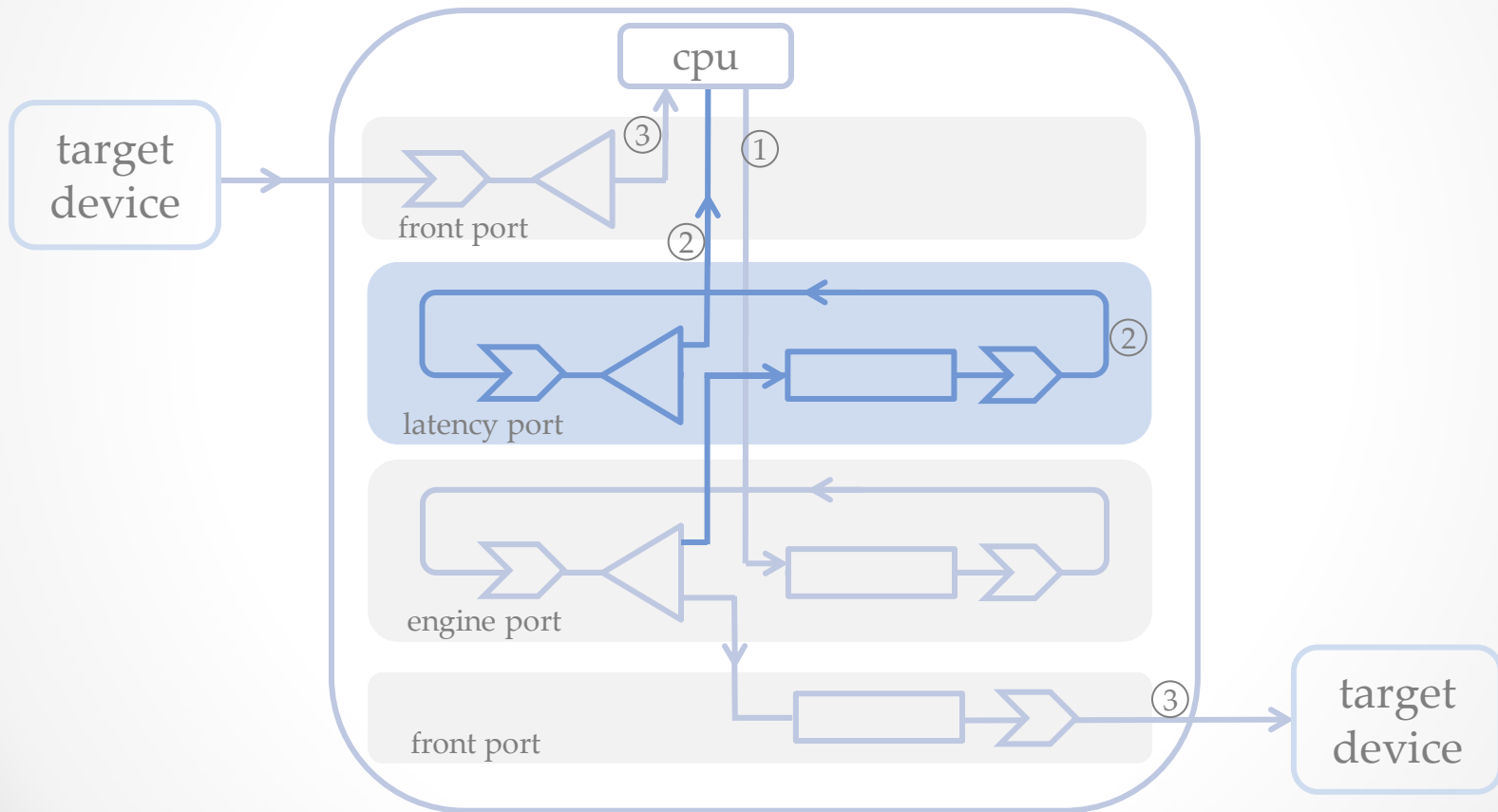
- **mechanism**

with the help of a dedicated flow track field located in transmitted pkt, it is able to identify this flow at RX side by qualify this specific field, no need of other qualification rule anymore. it is using UDF method to do the qualification.

- **latency measurement**

based on flow track and the sequence number within it, it is able to identify one pkt (not just one flow). then it is possible to measure the time between transmit and receive. details on next pages.

latency measurement

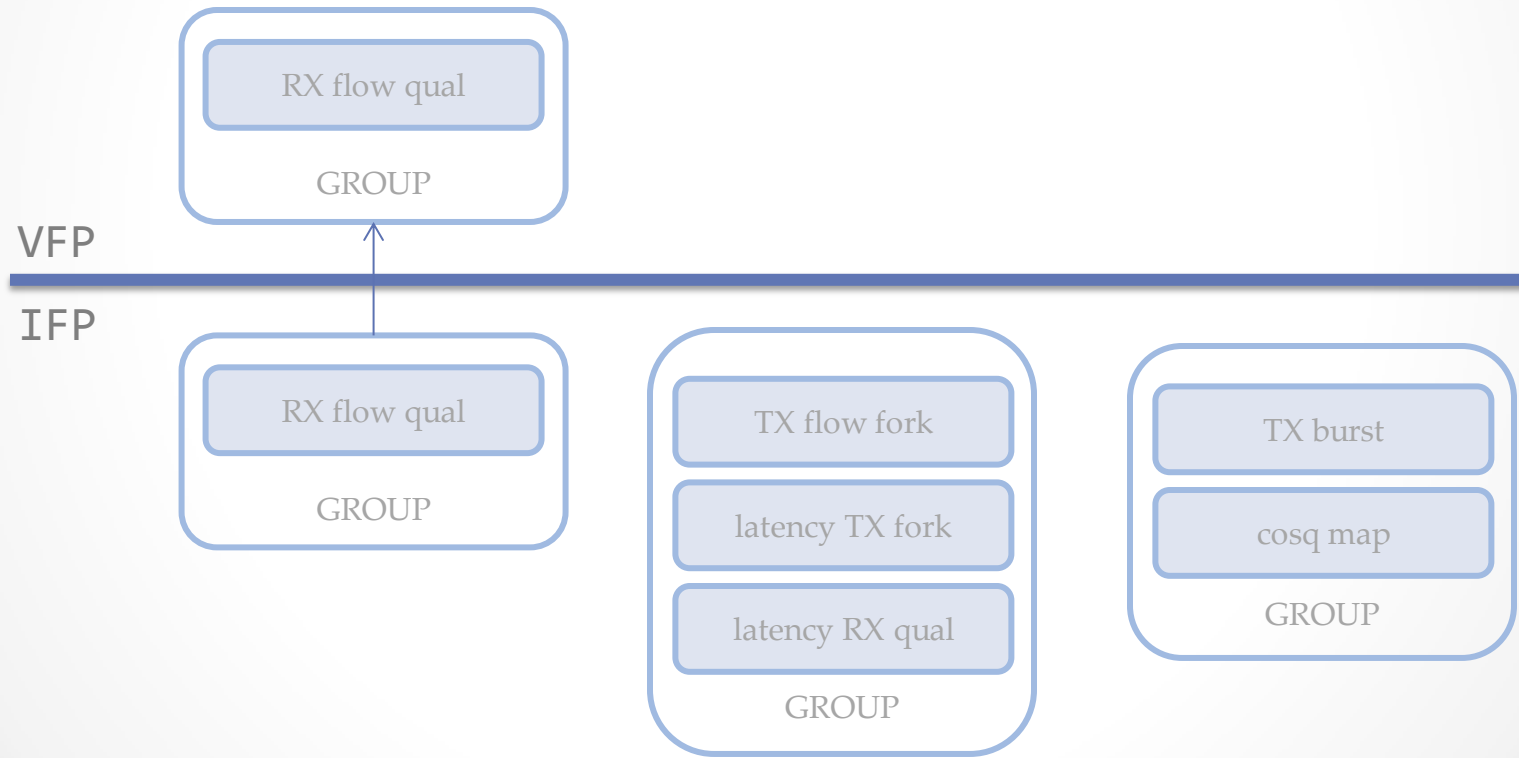


- ① first copy of pkt from CPU
- ② second copy of pkt destined to CPU
- ③ third copy of pkt destined to CPU

latency measurement

- due to limitation that FP action cannot have both “replication” and “timestamp”, we introduce the latency port to provide the copy of pkt with the timestamp of transmitting timing.
- engine port replicate 2 copy separately for front port and latency port. the latter one is in loopback, and finally sent the copy to CPU with the timestamp of T1.
- the other copy which was sent out from front port, reached target device, finally came back to one of front ports, and was captured to CPU with timestamp of T2, the received timing.
- compare T1 and T2, get the latency by calculation.

FP resource



TPID

- **background**

BCM switch chip is designed as that, few TPID value are configurable globally, each port is allowed to pick any of them, used for pkt parse.
normally, BCM have 4 global TPID value available.

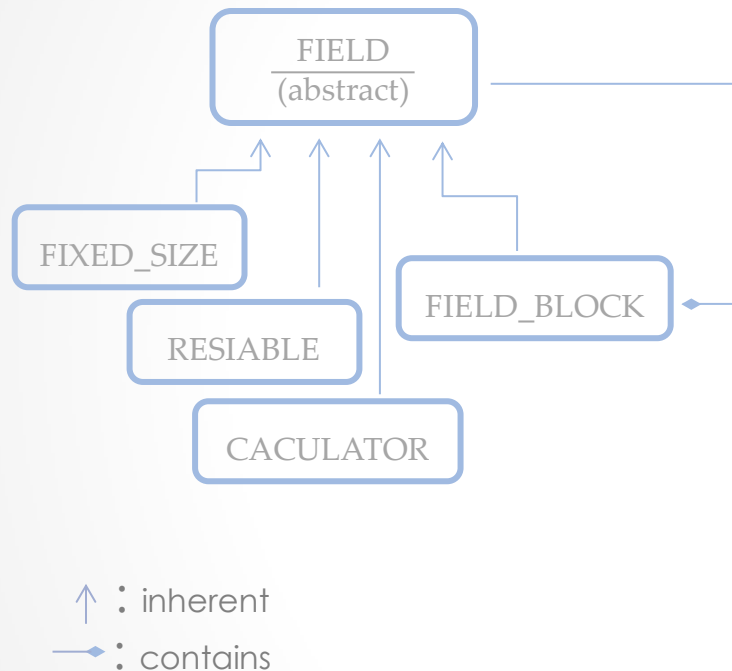
- **functionality**

as one of the qualification field of RX flow, it is necessary to manage these limited global TPID resource, and port selection properly as well as.

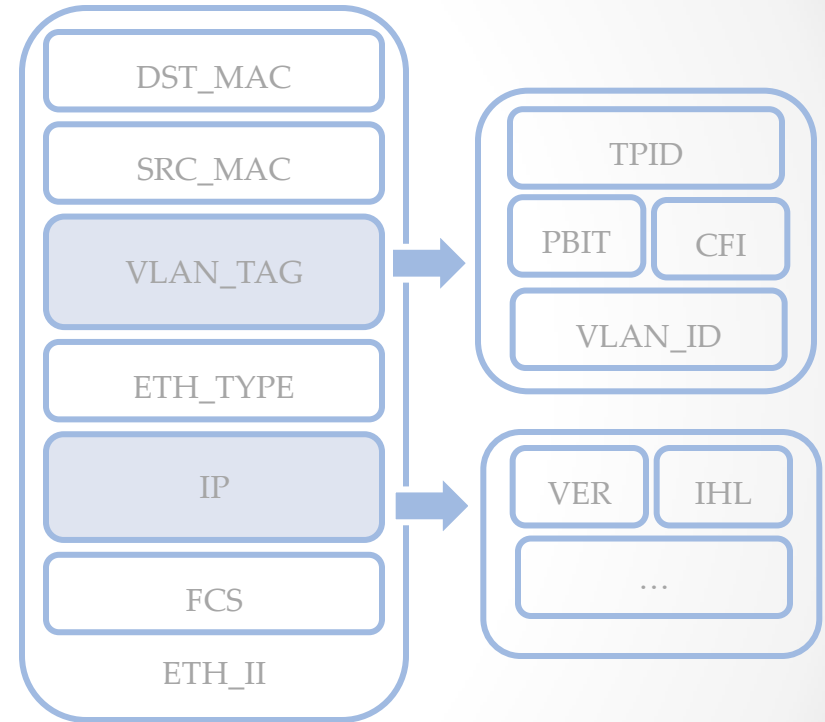
- **resource**

due to limited TPID value dimension, it is possible to have failure once resource exhausted.

field/field block/packet



FIELD_BLOCK inherits from FIELD,
and contains FIELD instances inside.



typical packet structure:
IP over ethernet with vlan tag

field/field block/packet

- **value assignment**
it is able to assign string or BYTES (vector<UINT8>) to field instance directly, and convert field instance to string or BYTES directly.
- **string parse**
it is able to define customized parser object for field instance, to convert between string and BYTES.
by default, there are 2 pre-defined parser, separately for dec and hex value string.
- **content**
seed object which defines the variable content of value: increasing, decreasing and random, is able to directly assign to field instance.
in similar, pattern object is used for repeated value for big size of field instance.
- **field block**
field block is defined as the container of field instance. it is able to manage the structure of field elements inside of it, like append, insert, delete.
packet is the alias of BYTES.
one field block instance could convert to one or more packet, depends if there's variable content field inside.

packet dispatch

- **background**

pkt would reach CPU when RX flow enables snooping. to identify which RX flow instance it belongs to, ideally we could put a match id in FP actions, but unfortunately, this action can not work together with action of timestamp.
the solution turns to be, by combination of source port + cpu queue, to indicates which RX flow.

- **resource**

BCM XGS line product have up to 48 queues for CPU port. so, each front port has up to 48 RX flows with snooping enabling.

as for TRX flow, latency measurement requires an extra CPU queue for pkt dispatch.

utility

the abstraction and encapsulation of utility class
which help to simplify coding and mask out target gap

ERROR/TRACE/LOG	debug facility
CMD	abstraction of command line
MUTEX	RAII based mutex
shared_ptr/weak_ptr	alias of c++11/boost smart pointer
TIMER/TIME	async/sync timer
THREAD	RAII based thread
WAIT	wait with existing condition
RANDOM	optimized random value generator
UNIQUE	index alloc/free management

utility integration

- **C++98 compiler**

require custom implementation for several system dependent interface(detail refer dependency.h)

integrate boost(download boost code, copy the header file sub-dir “boost”, no other dir required, no compiling required)

- **C++11 compiler (recommended)**

no more custom implementation, most part of implementation is build on std lib
enable the compile flag “ENV_C11” in makefile

- **CMD (which depends on target system)**

need custom implementation (refer dependency.h)

utility : output

error print	in case of error happened, output debug info (file, line, call stack, timestamp ...) by default, use “printf”, and could be customized.
trace print	output run time context info to help debugging. by default, use “printf”, and could be customized.
cli print	command result output

error clean

- **exception based error handling**
separate error handle from business handle path, thus there's only happy path remains.
the purpose of function return now is back to as result again. this helps much to make code clean.
- **exception safe programming**
RAII based class ensured resource free during stack wind back.
no more explicit error handling, thus happy path only.
- **process error at the boundary**
process error at the boundary of API:
re-encapsulate exception and re-throw,
or terminate exception, returns log ID by which get the error info.

smart trace

- **object based trace output**

trace object constructed by variable parameters, for example:

`TRACE("format", a, b, ...);`

with constructor/destructor, it is able to automatically print the life time of any code block.

when trace turn off, the construction of trace object just simply copy the constructor parameters, no more other running cost.

- **controlled, classified trace output**

trace filter object is introduced to control the output

to construct a filter object, only one unique string is required.

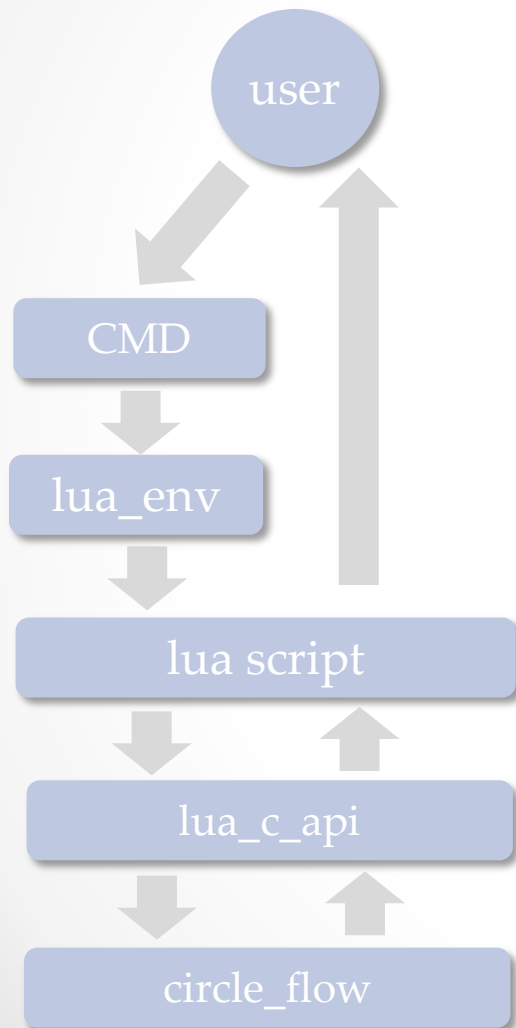
- **auto trace dump triggered by error (exception)**

in case of exception thrown and stack rewinding, destructor of trace object will trigger dumping of all trace objects within same thread. this would help to collect context info without using core dump which maybe not appropriate in some scenario .

sys_adapt

- **implementation of system dependent interface**
implementation required by the interface defined in
utility/export/_dependency.h
- **configuration on target device**
port configuration(port pair, default speed ...)
example pls refer device.cpp
- **init root of target device**
provides the root entrance of initialization on target
device by encapsulating above initializations.
example pls refer sys_adapt/src/init.cpp

CLI + lua script

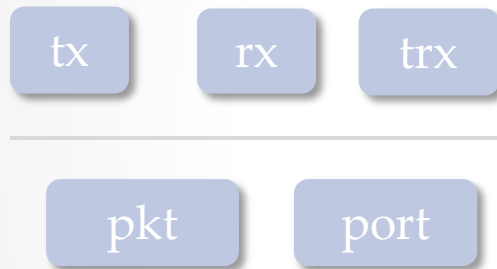


script based cmd implementation,
no compiling and loading.

with powerful string process
functionality from lua, cmd
interpreting is simple.

opened C_API,
it is able for user to have
customized command by self.

example of cmd



```
#define packet
> pkt create ipv4 prototype=eth_ipv4 length=64
> pkt set ipv4 dst_mac=00:00:00:00:00:11
> pkt set . src_mac=22
> pkt show .
```

```
#create tx flow
> tx create tx_1
> tx set . port=ge00 rate=100mbps pkt=ipv4
> tx start .
> tx show
> tx cnt
```

```
#create rx flow
> rx create rx_1
> rx set . port=ge01 dst_mac=11 src_mac=22
> rx start .
> rx show
> rx cnt
```

note: the period(.) is stand for the object name of last valid inputting.

_____ find more _____
at
www.circleflow.net
or
github.com/circleflow

