## PseudoArp Exercise

This exercise will introduce the use of one of the provided DSL clients. A mapping will be created that maps IP address in a shelf to MAC address. We are doing this so that the next exercise has a database to work with.

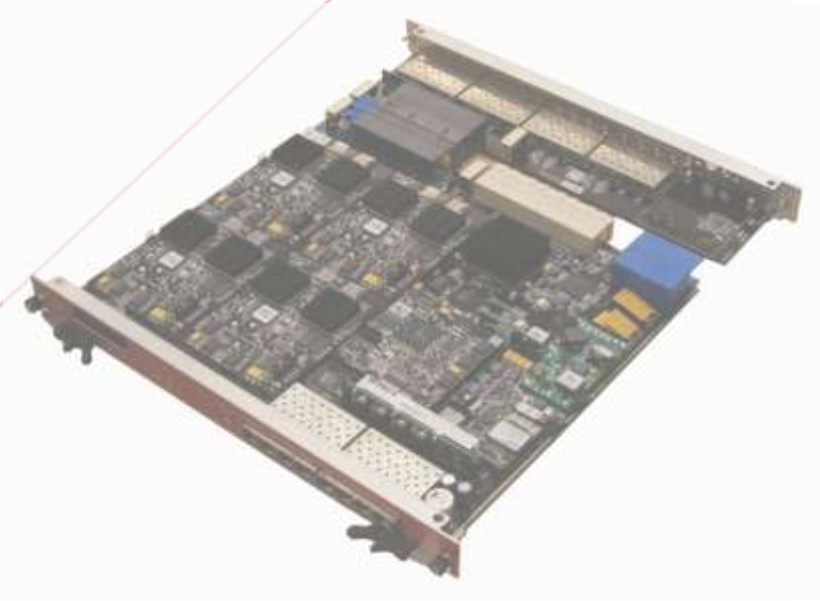
Objectives:

- Construct a C++ class and compile it rather than a set of C routines as in the Hello World example.

- Introduce a DSL client.

- Introduce the concept of the BSI

- Construct a Pseudo ARP table for use in a later exercise





SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

The Address Resolution Protocol (ARP) is used to resolve network layer addresses (IP address) to link layer addresses (MAC addresses). This exercise creates a "Pseudo ARP" database for use with the next exercise (Pseudo UDP). The methodology is to do the following:

- Loop over all RCEs in the crate by slot, bay and RCE (2 slots * 4 bays * 2 RCEs == 16)
- Ask each RCE for its IP and Ethernet information via the DSL client class `service::atca::Client`.
- Store this information in a small key-value-table.

**1** In your downloaded copy of `workshop_examples` (expanded in the previous exercise), change directory to `arp_example` and take a look at the `PseudoArp.hh` include file.

```
bash> cd workshop_examples/arp_example
bash> less PseudoArp.hh
```

**2** Here's the class definition for `PseudoArp`

```
#include <inttypes.h>
#include "kvt/Kvt.h"

namespace examples {
  class PseudoARP {
  public:
    PseudoARP();
    ~PseudoARP();
  public:
    uint64_t lookup(uint32_t);
  public:
    int refresh();
  private:
    KvTable _table;
  };
}
```

**Notes and Comments**

- The `lookup()` function takes an IPV4 address (in network byte order) as an argument and returns the MAC address associated with it.
- `refresh()` is the function which constructs the mapping between IPV4 and MAC.
- `KvTable` is a very lightweight key-value table provided with the RPT core. `KvTable` objects can only store 32 bit values, so they tend to have a pretty low memory footprint.

**3** Now, open `PseudoArp.cc`. The lookup function reads:

```
uint64_t PseudoARP::lookup(uint32_t ip)
{
  uint32_t reduced;
  KvtKey key = Hash64_32(0, ip);
  if ((reduced=(uint32_t)KvtLookup(key,_table))
      == 0) {
    // not found
    return 0;
  }
  return _unreduce(reduced);
}
```

**Notes**

- If the underlying lookup returns zero, this means that the IP address isn't in the table.
- `KvTable` can only store 32 bit objects. Since MAC addresses are 48 bits long, we must do something to make the address that length.
- `PseudoArp` uses knowledge that the MAC address space for RCEs is allocated with the first two bytes as `08:00`. `_unreduce()` adds these two bytes back.

**4** The next important function is refresh. The first part of the function fetches the shelf name from the BSI:

```
Bsi bsi = LookupBsi();
if (!bsi) {   // error state, deal with it.
  return rc;
}
uint32_t addr;
char
buffer[BSI_GROUP_NAME_SIZE*sizeof(unsigned)];
shelf = BsiReadGroup(bsi, buffer);
```



SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

**5**

The last part of refresh calls `_populate_loop()`. This function loops over slots, bays and elements and calls a `lookup` on each element in the shelf to get its IP info. Here we show the first part of the loop.

```
service::atca::Client client;
service::atca::Address addr(shelf,
     slot+1, bay, rce);
service::dsl::Location* loc =
     client.lookup(addr);
if (!loc) continue;
```

**Notes**

- `service::atca::Client` is tbe DSL client used by `atca_dump` and `atca_ip` to resolve the shelf/slot/bay/element space to IP space.
- When `service::atca::Client ::lookup()` is called, the class sends a broadcast out asking for a match If a response is received, that information is returned. If not, it retries (up to 5 times) and then times out, and lookup returns `NULL`.

**6**

The second part of the loop extracts the MAC address and IP address from the info returned by `client.lookup()`, chops off the known bytes from the MAC address and saves the info to the table.

```
uint32_t ipaddr  = loc->layer3.addr();
uint64_t mac     = loc->layer2;
KvtKey key = Hash64_32(0, ipaddr);
KvtValue value = (KvtValue)_reduce(mac);
// If the insert goes bad, return error
if (0 == KvtInsert(key, value, table))
      return -1;
++count;
```

**Notes**

- `count` is used here as an error check. If nothing is found by the end of `_populate_loop()`, then an error is declared.

**7**

The last part of `PseudoArp.cc` that is important is at the end, by `lnk_prelude`. (`PRINT` statements are removed for brevity.)

```
examples::PseudoARP*
     common_pseudoarp_instance = 0;
// "Install" this object
extern "C" const int lnk_options = LNK_INSTALL;

extern "C" int lnk_prelude(void *arg, void *elf)
```

**Notes**

- A common `PseudoArp` instance is created in `lnk_prelude`. This is used by code which uses the library.
- Notice the `lnk_options` constant. When set to `LNK_INSTALL`, the library is not simply loaded into memory, it is installed



SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

```
{
  common_pseudoarp_instance =
        new examples::PseudoARP();
  return 0;
}
```

into a load table.

- Do this either when multiple pieces of code will use the library, or when a task is going to be run over and over again and the library should only be loaded once (per boot cycle)

**8**

In `arp_task.cc`, `Task_Start()` looks up the MAC address of this node from the table, and prints the result.

**9**

Build the code with `build.sh`. Then run it with:

```
[/] run examples:arp_test.exe
```

Check the output via `syslog` and notice that your RCE's IP address is printed.

**Notes**

- If one or more of the RCEs in the shelf are absent, the task will take seconds (or minutes) to run. If all are there, the task will finish quickly



SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT