SFO17-303: A Modular Framework for Componentization

Yi He
Staff Software Engineer, Arm
(yi.he@linaro.org, yi.he@arm.com)
Agenda

- Motivation
- Benefits
- How to Use
- Next Steps
Motivation

- Extensible and modularize implementations (drivers) of packet I/O interface, allow build and run time selection.
- Derived from ODP Device and Driver Framework

While the DDF will deal with a very diverse type of hardware (NICs, SoC ports, crypto HW, compression HW...) the packetio structure will remain the central component to deal with dataplane aspects. So the DDF needs to update the packetio list so that applications continue to see a list of packetios.
Extend the Scope

- This approach can extend to wider ODP APIs subsets.
- Has been introduced into ODP project and experiment applying to multiple software components: cloud-dev
Possibilities

- ODP APIs can be partitioned into multiple subsets (called subsystems)
  - Encourage API polishing, abstraction and programming model design improvement per subsystem

- Each API subset can have multiple impls (called modules)
  - Encourage variant impls to be hosted in ODP project repository in either binary or source form

- ODP APIs can be extended by adding new subsystems
  - Encourage differentiation, convert ODP into an extensible framework for unifying programming APIs
Modular Programming Framework

- Decouple implementations from software interface
- Write each implementation as a module

- Use cases:
  - Select implementation modules in build time (linking)
  - Select implementation modules in run time (dynamic loading)
  - Select one implementation and replace the API stubs to eliminate API route overhead

- Advantage of using modular framework:
  - No source code changes for implementation modules to support all above use cases

- Demo: https://github.com/heyi-linaro/pure-interface
Software Interface Declaration

```c
/* Subsystems and APIs declaration */
extern SUBSYSTEM(scheduler);

SUBSYSTEM_API(scheduler, int, api_one, void);
SUBSYSTEM_API(scheduler, const char *, api_two, int);

typedef MODULE_CLASS(scheduler)
    api_proto(scheduler, api_one) api_one;
    api_proto(scheduler, api_two) api_two;
} scheduler_module_t;
```
Implementation as Module

```c
31    scheduler_module_t default_scheduler = {
32              .name = "default scheduler",
33              .init = default_scheduler_init,
34              .term = default_scheduler_term,
35              .api_one = default_scheduler_api_one,
36              .api_two = default_scheduler_api_two,
37          };
38
39    MODULE_CONSTRUCTOR(default_scheduler)
40    {
41           module_constructor(&default_scheduler);
42           ...
43           subsystem_register_module(scheduler, &default_scheduler);
44    }
```
Software Interface Stub - 1

- Route the API call to the default (active) module

```c
13   int SUBSYSTEM_API_STUB(scheduler, api_one)(void)
14   {
15       scheduler_module_t *module = NULL;
16
17       /* API stub routes the API call to the default(active) * implementation module. */
18       module = subsystem_active_module(scheduler, module);
19
20       if (module != NULL && module->api_one != NULL)
21           result = module->api_one();
22
23   }...
```
Software Interface Stub - 2

- Iterate the API call through all modules

```c
const char *SUBSYSTEM_API_STUB(scheduler, api_two)(int input)
{
    scheduler_module_t *module = NULL;

    /* API stub iterates the API call through all
     * implementation module. */
    subsystem_foreach_module(scheduler, module) {
        if (module->api_two != NULL)
            result = module->api_two(input);
    }
    ...
}
```
Select Modules in Build Time

- Such as built-in drivers which are always enabled in build time, e.g. loopback packet I/O

- **Static Linking:**
  - `-WI,--whole-archive <modules-list>` -`WI,--no-whole-archive`

- **Dynamic Linking:**
  - `-WI,--no-as-needed <modules-list>` -`WI,--as-needed`

- **Demo**, make targets:
  - main-static
  - main-dynamic
Select Modules in Run Time

- Build modules as Dynamic Shared Objects (DSOs)
- Write a module loader to load modules dynamically, according to config file, CLI options or platform probing
- **Demo**, make target:
  - main-plugin
Dynamic Module Loader

void dynamic_module_loader(const char *modules[])
{
    module_loader_start();
    ...
    for (name = &modules[0]; *name != NULL; name++) {
        handler = dlopen(*name, RTLD_NOW | RTLD_GLOBAL);
        if (handler != NULL)
            /* Install the DSO handler */
            module_install(handler, false /* Select as default or not */);
        else
            module_abandon();
    }
    module_loader_end();
}
API Stubs Override

- When one implementation only, need to eliminate the API stub overhead
- Prepare a header file to declare the API stub override and include it in Makefile to change module building rule

```
default-scheduler.o: default-scheduler.c default-scheduler-override.h
  $(CC) $(CFLAGS) -include "default-scheduler-override.h" -c -o $@
```

- **Demo**, make target:
  - main-static-override
  - main-preload (with override DSOs)
API Stubs Override

/* The header file serves to override the subsystem API symbols
 * directly with module implementations in static linked or pre-
 * load DSO builds.
 */

extern int SUBSYSTEM_API_OVERRIDE(
    scheduler, api_one, default_scheduler_api_one)(void);

extern const char *SUBSYSTEM_API_OVERRIDE(
    scheduler, api_two, default_scheduler_api_two)(int);
Next Steps

- Adopt Modular Framework in ODP
- Code auto-generation with Modular Framework
Thank You, Q & A

Contact: yi.he@linaro.org, yi.he@arm.com

#SFO17
SFO17 keynotes and videos on: connect.linaro.org
For further information: www.linaro.org