Webinar
Chapter one

Date : Jan, 18th 2022

Configuration Files and
Linux Threading in the OAI RAN Code

Configuration

- Two ways
  - Command line
  - Config file
    - All config file parameters can also be overwritten on command line
- OAI documentation
  - Wiki
  - Git server, with source code
    - Doc directory
    - Each module source directory
    - Markdown : nice view in gitlab

Command line

- Add a parameter

addaparam.md 4.4 KB

Edit

To add a new parameter in an existing section you insert an item in a `paramdef_t` array, which describes the parameter `config_get` call. You also need to increment the numparams argument.

existing code:

```
unsigned int varopt1;
paramdef_t someoptions[] = {
/*--------------------------------------------------------------------------------*/
/*                  configuration parameters for some module                      */
/* optname     helpstr     paramflags    XXXptr      defXXXval      type    numelt */
/*--------------------------------------------------------------------------------*/
   {"opt1",  "<help opt1>",   0,      uptr:&varopt1, defuintval:0, TYPE_UINT,  0 },
};

config_get( someoptions,sizeof(someoptions)/sizeof(paramdef_t),"somesection");
```

Command line

```
#define CMDLINE_PARAMS_DESC {  \
   {"rf-config-file",       CONFIG_HLP_RFCFGF,    0,   strptr:(char **)&RF_CONFIG_FILE,  defstrval:NULL,
TYPE_STRING,sizeof(RF_CONFIG_FILE)},\
   {"split73",             CONFIG_HLP_SPLIT73,     0,           strptr:(char **)&SPLIT73,          defstrval:NULL,       TYPE_STRING, sizeof(SPLIT73)},      \
   {"thread-pool",         CONFIG_HLP_TPOOL,       0,           strptr:(char **)&TP_CONFIG,        defstrval:"n",    TYPE_STRING, sizeof(TP_CONFIG)},
\
   {"phy-test",            CONFIG_HLP_PHYTST,      PARAMFLAG_BOOL, iptr:&PHY_TEST,                  defintval:0,          TYPE_INT,    0},                \
   {"do-ra",               CONFIG_HLP_DORA,        PARAMFLAG_BOOL, iptr:&DO_RA,                     defintval:0,          TYPE_INT,    0},                \
   {"sa",                  CONFIG_HLP_SA,          PARAMFLAG_BOOL, iptr:&SA,                        defintval:0,          TYPE_INT,    0},                \
   {"usim-test",           CONFIG_HLP_USIM,        PARAMFLAG_BOOL, u8ptr:&USIM_TEST,                defintval:0,          TYPE_UINT8,  0},
\
   {"clock-source",        CONFIG_HLP_CLK,         0,           uptr:&CLOCK_SOURCE,                 defintval:0,          TYPE_UINT,   0},                \
```

Command line

- Direct usage by simple variable
- Usage with structured data containers
-

```
int load_channellist(uint8_t nb_tx, uint8_t nb_rx, double sampling_rate, double channel_bandwidth) {
  paramdef_t achannel_params[] = CHANNELMOD_MODEL_PARAMS_DESC;
  paramlist_def_t channel_list;
  memset(&channel_list,0,sizeof(paramlist_def_t));
  memcpy(channel_list.listname,modellist_name,sizeof(channel_list.listname)-1);
  int numparams = sizeof(achannel_params)/sizeof(paramdef_t);
  config_getlist( &channel_list,achannel_params,numparams, CHANNELMOD_SECTION);
  AssertFatal(channel_list.numelt>0, "List %s.%s not found in config file\n",CHANNELMOD_SECTION,channel_list.listname);
  int pindex_NAME = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_NAME_PNAME);
  int pindex_DT = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_DT_PNAME );
  int pindex_FF = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_FF_PNAME );
  int pindex_CO = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_CO_PNAME );
  int pindex_PL = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_PL_PNAME );
  int pindex_NP = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_NP_PNAME );
  int pindex_TYPE = config_paramidx_fromname(achannel_params,numparams, CHANNELMOD_MODEL_TYPE_PNAME);

  for (int i=0; i<channel_list.numelt; i++) {
    int modid = modelid_fromstrtype( *(channel_list.paramarray[i][pindex_TYPE].strptr) );

    if (modid <0) {
```

Config module

- A very special parameter on the command line « -O »
  - Loads the libparams_<configsource>.so
    - today only «Linux « libconfig library : libparams_libconfig.so shared library
    - Looks for config_<config source>_init symbol and calls it , passing it an array of string corresponding to the « : » separated strings used in the –O option
    - Looks for config_<config source>_get, config_<config source>_getlist and config_<config source>_end symbols which are the three functions a configuration library should implement. Get and getlist are mandatory, end is optional.
    - Stores all the necessary information in a configmodule_interface_t structure, which is of no use for caller as long as we only use one configuration source.
    - if the bit CONFIG_ENABLECMDLINEONLY is set in initflags then the module allows parameters to be set only via the command line. This is used for the oai UE.

Config module

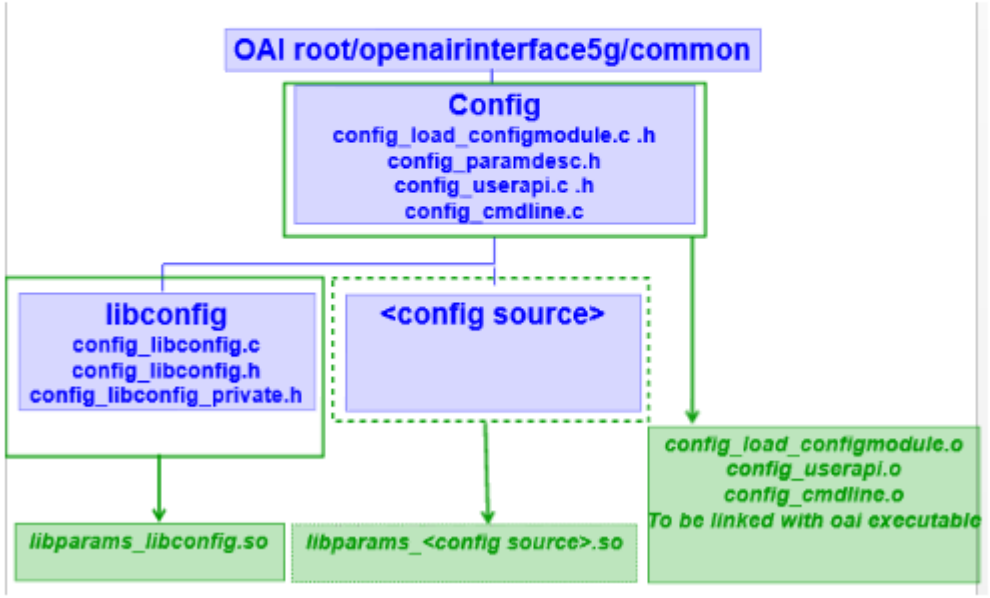
- We are back to command line « config_get() »
  - Also used to read config file
- All parameters can be overwritten on command line
  - $ ./lte-softmodem -O <config> --eNBs.[0].component_carriers.[0].N_RB_DL 100
  - This example shows how to follow the config file tree

Configuration module SW architecture

- Arch.md

arch.md 288 Bytes

config module source files



Example : log configuration

- int logInit (void);
  - A component encapsulate it's configuration reading
- Documentation
-

global parameters

name	type	default	description
`global_log_level`	`pre-defined string of char`	`info`	Allows printing of messages up to the specified level. Available levels, from lower to higher are `error` , `warn` , `info` , `debug` , `trace`
`global_log_online`	`boolean`	1 (=true)	If false, all console messages are discarded, whatever their level
`global_log_options`	`list of pre-defined string of char`		3 options can be specified to trigger the information added in the header of the message: `nocolor` , disable color usage in log messages, usefull when redirecting logs to a file, where escape sequences used for color selection can be annoying, `level` , add a one letter level id in the message header (T,D,I,W,E for trace, debug, info, warning, error), `thread` , add the thread name in the message header, `function` , adds the function name, `line_num` , adds the line number, `time` adds the time since process starts

Example log configuration

- We can find quickly in the cource code

```
#define LOG_GLOBALPARAMS_DESC { \
    {LOG_CONFIG_STRING_GLOBAL_LOG_LEVEL,   "Default log level for all componemts\n",
     0, strptr:(char **)&gloglevel, defstrval:log_level_names[2].name, TYPE_STRING,     0}, \
    {LOG_CONFIG_STRING_GLOBAL_LOG_ONLINE,  "Default console output option, for all
components\n", 0, iptr:&(consolelog),        defintval:1,                    TYPE_INT,        0}, \
    {LOG_CONFIG_STRING_GLOBAL_LOG_OPTIONS, LOG_CONFIG_HELP_OPTIONS,
            0, strlistptr:NULL,          defstrlistval:NULL,             TYPE_STRINGLIST, 0} \
  }
```

- This example match the general explaination

Configuration, log example

- xxx_log_level
  - We can’t find it in any .h file
  - common/utils/LOG/log.c creates the table
    - log_getconfig() builds the table
      - paramdef_t logparams_level[MAX_LOG_PREDEF_COMPONENTS];
    - Then it calls
      - config_get(logparams_level,…
    - The parameters table is a regular C array, it can be built dynamically
- xxx_log_verbosity
  - Is not in the code, so these parameters doesn’t exist even if we see it in most of *.conf files
  - But, log_options exists
    - You can set "nocolor", "level", "thread", "line_num", "function", "time"

Inter-thread communication

- High layers : ITTI
  - Documentation, same organization as we saw for parameters
  - A full framework to implement a infinite loop on events
- Thread pool
  - To use generic threads, typically mapped of physical cores
  - It saves the thread create/delete cost when we run in parallel several works, with typically a return software barrier
- AdHoc, direct usage of posix thread API
  - Low L1, 5G RLC and PDCP

ITTI

- Static organization
  - Queues are listed in
    - common/utils/ocp_itti/intertask_interface.h
  - Dynamic queue creation is now available, but not used yet
- Full Thread main loop features
  - Add any Linux socket handler in the wait message call
  - Add timers in the wait message call
- Cost
  - One system call (epoll()) for each message, even if it is thread to thread

ITTI example

```
Void * aLayer_main(void * context) {
    // initialize the layer (Thread) data
    // and register in itti the external sockets (like S1AP, GTP)
    aLayer_init();
    while (1) {
    MessageDef * msg
        itti_receive_msg(TASK_aLayer, &msg);
        // itti receive released, so we have incoming data
        if ( msg !=  NULL) {
            switch(ITTI_MSG_ID(msg)) {
                case …:
                    processThisMsgType1(msg);
                    break;
                case …:

            }
            Itti_free(msg);
       }
        // or data/event on external entries (sockets)
        Struct eppol_events* events;
        int nbEvents=itti_get_events(TASK_aLayer, &events);
        if ( nbEvents > 0)
          processLayerEvents(events, nbEvents);
    }
}
```

Thread pool

- OAI version of openMP(), C++ async/future, …
- OAI API, internal implemntation can evolve
- Real time
  - Map threads to HW cores
  - Faster non real time async calls like openMP
  - Use today posix thread
    - Isolate Posix pthread API from OAI main code
    - Better implementation if needed : futex(), gcc instrinsics, …
  - Pass-though is possible (synchronous calls)

OPEN AIR INTERFACE

Thread pool low layer : multi thread queues

- Simpler than ITTI
  - No system call (today : posix thread conditional vars)
  - OAI API
    - Implementation can change
    - Simpler, more robust than posix thread api
  - Dynamic fast queues creation
- As usual calls : push, pull, poll
  - Poll usage is discouraged in most of designs
- This can be used alone, and it is the thread pool basement

Thread pool

- One or several thread pools can be created
  - Simple and good usage is to make one thread pool per Linux process
    - Open to specialize cores for specific features
      - big.little architecture processors in future oai
  - Runtime threads positioning on physical cores
    - --thread-pool 1,3,5
    - The operator knowledge is required : NUMA bi-xeon is for Intel experts
    - --thread-pool 100,100,100
      - Fallback to floating threads is core 100 doesn’t exist
    - --thread-pool « n »
      - All calls are synchronous
        - Usage : to debug race conditions

Thread-pool example

- Somewhere in process initialization
  - initTpool(get_softmodem_params()->threadPoolConfig, threadPool, get_softmodem_params()→PoolMeasurEnable);
- Anywhere we have several works to do
  - More than 10 µsec CPU each
  - For (int i=0 ; i<nbJobs ; i++)
    - notifiedFIFO_elt_t *req=newNotifiedFIFO_elt(sizeof(turboEncode_t), id.p, proc->respEncode, TPencode);
    - turboEncode_t * rdata=(turboEncode_t *) NotifiedFifoData(req);
    - Fill rdata→xxx
    - pushTpool(proc→threadPool,req);
  - For (int i=0 ; i<nbJobs ; i++)
    - notifiedFIFO_elt_t *resp=pullTpool(proc->respEncode, proc→threadPool));
    - Process resp if needed
    - delNotifiedFIFO_elt(resp) ;
- new/del NotifiedFIFO_elt are not mandatory
  - Data can be any memory zone starting by type notifiedFIFO_elt_t C struct
  - Race conditions to be managed by designer
  - Think of 10µsec CPU, a pair malloc/free is maybe not the highest cost

Other threads

- OAI softmodem involve permanent threads
  - Each has it's own design
    - All are using posix threads API
      - A very few legacy exception use lfds (lock free data structures library)
        - Don't use it for new code
  - Major gNB threads
    - Ru_thread()
      - This is the MAIN loop to trigger everythings in gNB
        - Blocks on RF board acquisition : trx_read_func()
    - Complex multithread L1
    - RLC and PDCP layers have each a asynchronous msg thread