json app design

limestone

October 6, 2021

1 Introduction

jsonapp is written in C. It parses a json file with specific formatting and makes updates to uci config files based on the values taken from the input json file.

To parse the json file, it uses libjson-c library. To interface to the UCI subsystem, it uses the libuci library.

2 High Level Design

jsonapp is divided into two main parts. The first part is the main module that abstracts the high level operations of the jsonapp. The second part is the backend driver that does the actual work specific to its responsibility.

3 Main Module

The main module of jsonapp provides a framework that enables backend drivers to register themselves to the main module. Aside from this functionality, it also provides API function calls that are generic to all backend drivers. For example, the main module provides APIs which enable backend drivers to process JSON arrays or to create/add new options in uci config files.

The main module collects all registered backend modules and iterates through them so that they can do their processing on the input json file. For example, two backend engines are registered to the main module - a wireless backend module and a hotspot backend module. The main module first selects the wireless backend module and calls it so that it can process all ”wireless” related information in the input json file and transfer them all to /etc/config/wireless.

When the wireless backend module is finished. It gives back control to the main module, that subsequently will choose the next backend module in line which is the hotspot module. The hotspot module will then parse the same input json file but this time, will only process all data related to hotspot.

3.1 Main Module objects

3.1.1 Parse Backend

Parse backends are the processing centers that do the real work of parsing and generating UCI configurations. There are currently two parsing backends in `jsonapp`. The main module delegates all processing to the parse engines and remains oblivious to what each engine does.

```
struct jsonapp_parse_backend {
        struct jsonapp_parse_backend *next;
        struct jsonapp_parse_ctx *(*init)(struct jsonapp_parse_ctx *jctx);
        int (*process_json)(struct jsonapp_parse_ctx *jctx);
        void (*exit)(struct jsonapp_parse_ctx *jctx);
};
```

- `next` - link to the next parse backend.
- `init` - function pointer to the initialization function for this backend.
- `process_json` - the input json file processing function for this backend.
- `exit` - the cleanup/de-initialization function for this backend.

3.1.2 Parse Context

Parse context holds global settings and other shared variables being shared by all engines. It also holds the UCI context for the specific parse engine.

```
struct jsonapp_parse_ctx {
        struct jsonapp_parse_backend *backend;
        struct json_object *root;
        struct uci_context *uci_ctx;
};
```

- `backend` - pointer to the current active backend being used.
- `root` - the main module calls the libjson library and saves a reference to the input JSON file in this member. all parsing done on a JSON file starts from the root.
- `uci_ctx` - the backend is incharge of opening a reference to the UCI config and stores this in the `uci_ctx` member.

3.2 Main Module APIs

3.2.1 __jsonapp_init__

__json_app_init__ is an annotation used to mark a function as an "entry" point for a parse backend engine. when a function is annotated with __jsonapp_init__ it is automatically run by GNU libc before the jsonapp main() function is given control. This provides a backend engine the opportunity to ”register” itself to the jsonapp application.

3.2.2 jsonapp_die()

```
void jsonapp_die(const char *fmt, ...);
```

`jsonapp_die()` is called when a gross error occured on the engine or the entire application cannot continue any further execution. For example, if the program encountered a memory allocation error, this is treated as a gross error and the program will print an informatory message and consequently exit prematurely. This function accepts printf() style formatting.

3.2.3 jsonapp_register_backend()

```
jsonapp_register_backend(struct jsonapp\_parse\_backend *backend)
```

`jsonapp\_register\_backend()` is called when a parse backend engine wants to register its services to the main jsonapp module.

3.2.4 jsonapp_object_get_object_by_name()

```
struct json_object
*jsonapp_object_get_object_by_name(struct json_object *parent,
                                   char *name,
                                   enum json_type expected_json_type);
```

`jsonapp\_object\_get\_object\_by\_name()` is used when an engine wants to get a reference to a json object thru it's name. there are many examples of this API being used in wireless_engine.c and hotspot_engine.c as well.

3.2.5 jsonapp_process_array

```
void
jsonapp_process_array(struct jsonapp_parse_ctx *jctx,
                      struct json_object *obj_arr,
                      void *user_data,
                      void (*process_member)(struct jsonapp_parse_ctx *jctx,
                                             struct json_object *obj,
                                             void *user_data));
```

`jsonapp\_process\_array()` is called when the same set of processing is needed to do on objects of the same kind. the process_member() function is called repeatedly for all the objects found in a json_object that is of type json_type_array.

4 Wireless Backend Module

4.1 Wireless backend objects

4.2 Wireless backend APIs

5 Hotspot Backend Module

5.1 Wireless backend objects

5.2 Hotspot backend APIs