Switch Software Development Know-hows

Revision History

Revision	Author	Contact Info	Description	Date
0.1	Chen Liting		Initial revision	2023-10-20
0.2	Chen Liting		Completed section 5 “BDCOM File Operation Routines”	2023-10-25
0.3	Chen Liting		1. Completed section 4 “CLI Commands Development” 2. Added section 5 “Add a New Module to the Software System”	2023-10-27
0.4	Chen Liting		1. Added something about using union type in packet structure 2. Added section 7 “Protocol Entity Design”	2023-11-9
0.5	Chen Liting		1. Added instructions for printing routines for commands 2. Corrected some inaccuracies in textual expression	2023-11-15

1. Data Structures and Encoding/decoding of Network Protocol Packets

To strictly conform to the format of protocol packets transferred on network, definition of data structures, encoding and decoding of packets data in the implementation of protocol entities should follow the following rules:

- The order of fields in the data structure must be consistent with the packet format defined in the protocol standard.
- The length of each field must be consistent with the definition in the protocol standard. To ensure this, the type of a field must be a type that can explicitly determine a consistent length across different CPU architectures or compilers. It is recommended to use basic types such as uint8, uint16 and uint32 that have been defined by the software system and can explicitly indicate length, rather than using types such as int and enum. In BDCOM switch software system, types like uint16 are defined in include\sys_global\global.h. In addition, since the size of union type depends on its largest member, placing it in the structure of packet can bring uncertainty to the offset of the next field. Therefore, do not use union type in the packet structure to attempt to unify different packet types, unless union is only used for different expressions of a fixed length field for easy access.
- To prevent the compiler from actually expanding the field length in order to align the address boundary of the field, the data structure of the packets must use packed attribute like the following.

```
typedef struct pkt_a_header_
{
    uint16 type;
    uint8 field_a;
    uint8 filed_b;
    uint16 field_c;
    uint16 field_d;
    uint32 field_e;
} __attribute__ ((packed)) pkt_a_header_t; /* Place the type name after the attribute */
```

Or:

```
typedef struct pkt_a_
{
    uint16 type;
    uint8 field_a;
    uint8 filed_b;
    uint16 field_c;
    uint16 field_d;
    uint32 field_e;
    uint8 payload[0]; /* Zero length array indicating the payload portion with variable
length. */
} __attribute__ ((packed)) pkt_a_t;
```

- Byte order: For multi-byte fields, they are always converted into network order before being sent to the network; After receiving from the network, they are always converted into host order before processing. Use htons() (for 2-byte) and htonl() (for 4-byte) for converting host order to network order, and conversely use ntohs() and ntohl(). However, there is an exception for this rule, that is, when we process IP addresses locally, we often use their network order in many cases, so we may not need to convert the byte order for the IP address fields in the received packets. Of course, it depends on the situation, for example, if you want to compare two IP addresses to see which one is larger, you still need to convert it to the host order.

Some programming techniques:

- If the length of the packet to be sent is fixed, directly use the struct variable to send:

```
pkt_b_t pktb;
......
sendto(..., &pktb, sizeof(pktb), ...);
```

- If the length of the packet to be sent is variable, for example, it has a variable payload or some variable TLVs, use a byte buffer of sufficient length, and then use the struct pointer variable of the packet to fill in the values of those fixed fields, without having to fill in the byte buffer by calculating the offsets.

```
char buf[600];
uint8 a, b;
uint16 c, d;
uint32 e;
pkt_a_t *p_pkta = (pkt_a_t *)buf;
......
/* Write (encode) the fields of the packet */
p_pkta->type = htons(PKT_TYPE_A);
p_pkta->field_a = a;
p_pkta->field_b = b;
p_pkta->field_c = htons(c); /* h2n converting */
p_pkta->field_d = htons(d);
p_pkta->field_e = htonl(e);
memcpy(p_pkta->payload, &payload, 512); /* for example lengh of 512*/
sendto(..., buf, sizeof(*p_pkta) + 512, ...);
```

- When receiving a packet, since we cannot predict the type of it, we usually also use a byte buffer of sufficient length to receive it. Then, first identify its type and use the corresponding struct pointer variable to read the values of other fields based on the type.

```
char buf[600];
uint32 type;
uint8 a, b;
uint16 c, d;
uint32 e;
```

```
char *payload;
pkt_a_t *p_pkta;
so_recvfrom(..., buf, sizeof(buf), ...);
type = ntohs(*(uint16 *)buf); /* or we could use a common header struct to structure the
beginning part of the packet */
switch(type)
{
        case PKT_TYPE_A:
                p_pkta = (pkt_a_t *)buf;
                /* Read (decode) the fields of the packet */
                a = p_pkta->field_a;
                b = p_pkta->field_b;
                c = ntons(p_pkta->field_c); /* n2h converting */
                d = ntons(p_pkta->field_d);
                e = ntonl(p_pkta->field_e);
                payload = p_pkta->payload;
                /* process the packet here according to the protocol procedure */
                ......
                break;
        case PKT_TYPE_B:
                /* process packet type B here */
                break;
        default:
                break;
}
```

2. MESSAGE FORMAT

As a specification in the BDCOM switch software system, the message format used for message queue is four words of type uint32. The first word is always used for the message type, and the remaining three words are left for modules to define themselves except for some special occasions where there are special conventions. Message variables are usually straightforwardly as follows,

```
uint32 msg_buf[4];
```

Other formats for message body are also allowed, as long as they are compatible with the above conventions.

3. BDCOM SOCKET ROUTINES

3.1 BDCOM Socket Routines

BDCOM socket routines are all prefixed with so_ like so_socket(), so_bind(), so_sendto() etc, which differs in form from standard socket routines. The routines are provided by the network stack we apply in our switch software system, rather than by VxWorks. However, in terms of usage, they are very similar to standard socket routines, except for the significant difference in return values, which requires special attention in use. See include\ip\socket.h for their declarations.

3.2 BDCOM Socket Extension for Message Notification

In order to promptly notify the task through messages whenever data arrives on the socket or some events occur with the socket, we have extended the socket layer functionality. This is achieved by the routine socket_register(), which associates a socket with a message queue. The declaration of this routine in include\ip\socket.h is as follows,

int socket_register __P((int s, ULONG qid, int arg));

The argument s is the file descriptor of the socket, qid the identifier of the message queue and arg will be the value of the fourth word of the message.

socket_register() is called after the message queue and the socket are successfully created usually in module's initialization function. After binding the socket to the message queue, the task of the module can now be notified promptly of the arrival of packets on the socket or the occurrence of other events. For a UDP socket (i.e. SOCK_DGRAM socket), the module task generally only concerns SOCKET_ DATARCVD message. Upon receiving this message, the task immediately receives packet from the socket. For TCP sockets, the module task also needs to be concerned with messages related to TCP connection events, such as SOCKET_ CONNECTED message, etc. For details, refer to the definition of socket messages in include\ip\msg.h. The following is the example code of using a UDP socket in a module.

```
MSG_Q_ID msgq_id;
int my_sock = -1; /* Note that 0 is a valid value of socket fd */
int demo_init()
{
    int rv;

    msgq_id = sys_msgq_create(DEMO_MSGQ_LEN, 0);
    if (msgq_id == NULL)
        syslog(.....);

    my_sock = so_socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (my_sock < 0)
```

Deleted[自在]: 0

```
            syslog(.....);

        rv = socket_register(my_sock, (ULONG)msgq_id, 0); /* Bind the socket to the message
        queue */
        if (rv != 0)
            syslog(.....);
        .....
}

void module_main_process()
{
        uint32 msg_buf[4];
        char pkt_buf[MAX_PKT_SIZE];
        int rv;
        .....
        while(1)
        {
            rv = sys_msgq_receive(msgq_id, (unsigned long *) msg_buf, SYS_WAIT_FOREVER);
            if (rv != SYS_OK)
            {
                syslog(.....);
                continue;
            }

            switch(msg_buf[0])
            {
                case MODULE_MSG_A: /* Other messages of the module */
                    .....
                    break;
                case SOCKET_DATARCVD: /* Notification message for the arrival of a packet on
            socket, sent by the socket layer of network stack */
                    rv = so_recvfrom(my_sock, pkt_buf, .....);
                    .....
                    break;
                default:
                    break;
            }
        }
    }
```

4. CLI COMMANDS DEVELOPMENT

4.1 Fundamentals of CLI Commands

All working parameters that can be configured by commands have their default values. For example, the SSH server is disabled by default, the default response time for IGMP snooping is 15 seconds, etc. The default values are assigned during module initialization and do not require commands to explicitly configure them. Only when the current value of a working parameter is not the default value, will it be saved in the startup configuration file when saving the configurations, otherwise it does not need to be saved. If you want to change the default value of a working parameter, you need to use the corresponding command, such as the following.

Switch_config#ip igmp-snooping timer response-time 30

The “no” form of a command does not mean to eliminate the working parameter, but to reset its value to the default one.

When saving all the configurations of the entire software system through the command “wrtie” to the startup configuration file, it is collecting configurations from all modules. These configurations are actually stored in global variables, i.e. working parameters, of the modules, and the process of collection is to convert the values of the working parameters that are different from the default ones into corresponding command strings and write them to the startup configuration file. The process of “show running-config” is actually similar, that is, the process of collecting configurations from all modules is reused, except that it does not save the results to the configuration file. Therefore, as long as the configurations of my module can be collected correctly by “show running-config”, saving the configurations will also be done correctly.

We need to distinguish between command keywords and command parameters. For example, in the command “ip igmp snooping timer response time {*time*}”, only {*time*} is a parameter, and the rest are keywords.

4.2 Commands of a Module

The commands are implemented in BDCOM switch software as a command tree. Generally speaking, each command keyword occupies one and only one node in this tree, where the node refers to the data structure that implements the keyword sequence at that level, including the corresponding method function. And this tree specifies the relationship between all keywords. For example, if a module has already registered the keyword sequence “aaa bbb”, and another module to be initialized later needs to register the command “aaa bbb fff”, it can only register “fff” under the keyword sequence “aaa bbb” in the form of sub-command registration.

The following sections roughly describe how we develop commands for a module in principle. For detailed logic, please refer to the sample code, such as mqttc-cmd.c or command implementation source files of other modules.

4.2.1 Register A Top-level Keyword

If the first keyword of a module's command(s), which is called top-level keyword, has not been registered, it needs to be registered using the following routine,

int registercmd(struct topcmds *cmdentry);

where the argument cmdentry is the pointer to the data for this keyword of type of struct topcmds. See the header file include\libcmd\cmdparse.h for the detail of the type struct topcmds.

Usually, top-level keywords are registered in the initialization function of the module. According to convention, the first keywords of a module's configuration commands is the same, so usually only one top-level keyword is registered for a module, if necessary .

4.2.2 Sub-commands

Any keyword following an existing keyword sequence is called a sub-command. There are two ways to implement sub-commands. If the parent command of the sub-command is registered in another earlier initialized module, we register the sub-command(s) of this module through the following routines,

int register_subcmd (char *, unsigned long, unsigned long, unsigned long long, struct cmds *);
int register_subcmd_tab (char *, unsigned long, unsigned long, unsigned long long, struct cmds *, int);

The difference between register_subcmd_tab() and register_subcmd() is that the former registers a set of sub-commands at the same level, while the latter only registers a single one. The argument “struct cmds *” is the pointer to the data for this keyword as a sub-command of type of struct cmds. See the header file include\libcmd\cmdparse.h for the detail of the type struct cmds.

An example for the above situation is the “show” commands of a module, as the top-level keyword “show” has already been registered in the earlier initialized system module.

Usually, sub-commands are also registered in the initialization function of the module.

If the parent command is defined and registered by this module itself, the sub-commands are just executed by calling the following routine in the method function of the parent command,

int subcmd(struct cmds tab[], unsigned long *mskbits, int argc, char *argv[], struct user *u);

4.2.3 Method Function of a Command

To distinguish it from general functions, we refer to the implementation function of a specific command as a method function, or method.

Usually, each level of a command's keyword sequence must have a corresponding method function, even if this level has not yet reached the end of a complete command. For example, for the command keyword sequence “mqtt client”, at the level of “client”, the command is not yet complete, but there must also be a method function corresponding to it.

In struct topcmds and struct cmds, the syntax of method functions is the same as follows,

int (*func)(int argc, char *argv[], struct user *u);

When the method function is called, the meanings of arguments argc and argv are the same as usual command execution conventions, where argc is the number of remaining keywords and parameters of the command a user input,counting in the current keyword, and argv is the strings of corresponding keywords and parameters. For example, if the command string entered by the user is "mqtt client server-port 4800", and do_mqtt_client (int argc, char * * argv, struct user * u) is the method function corresponding to the keyword sequence "mqtt client", when it is called, the value of argc should be 3, and the strings in argv be “client”, “server-port” and “4800”.

4.2.4 Parameter Parsing

Use the following routine to parse the parameters of the command.

int getparameter(int argc, char *argv[], struct user *u, struct parameter *param);

4.3 debug commands

TBD

4.4 show running-config

“show running-config” is a system command. As mentioned above, it needs to collect the configurations of all modules, so each module needs to write a callback function that collects the its configurations and register it in the system. Register the configuration collection callback function of this module through the following routine,

Deleted[自在]: of itself

int interface_set_showrunning_service(MODULE_TYPE module_type, INT32 (*func)(DEVICE_ID));

where the argument func is the pointer of the callback function. If the value of the argument “DEVICE_ID” is zero when the callback function is called, it indicates that the system is collecting global configurations; otherwise, it is collecting the configurations under a given interface. Note that we should use vty_printf() instead of other printing routines to print the command lines in this callback function, so that they can be saved into the startup configuration file when the command “wrtie” is executed.

Usually, the configuration collection callback function is registered in the initialization function of the module.

Refer to section 4.1 to infer how to write configuration collection callback function.

4.5 show version

“show version” is a system command. It is required that the version information of all modules or a given module should be shown in the commands “show version all” or “show version module {*module-name*}”. This can be achieved by registering the version information of modules. Use the following routine to register the version information of a module.

int register_module_version(struct version_list *);

See the header file include\libsys\verctl.h for the declaration of this routine and the type struct version_list.

Usually, the version information of a module is registered in the initialization function of the module.

4.6 Printing routines in commands

For printing statements in commands that show module information (not “show version” or “show running-config”), or prompt message printing statements for configuration commands (such as illegal parameter prompts, configuration failures, etc.), vty_output() is recommended as the printing routine, so that it’s guaranteed that the information is printed on the CLI whether the vty session is console, telnet or SSH.

Other more advanced printing routines used in vty sessions, such as routines that support pause feature, are discussed in other documents.

5. BDCOM File Operation Routines

5.1 Mutual Exclusion Mechanism for File System Access

In VxWorks based switch software, access to the file system requires a mutual exclusion mechanism. In short, no matter what operation you want to perform on a file, you must first obtain access to the file system using the following routine,

UINT32 enter_filesys(UINT32 mode);

where the argument mode can be OPEN_ WRITE (for write, create and delete) or OPEN_ READ (for read only). The return value of zero indicates that the permission is successfully obtained and the desired operation can be performed, otherwise it is not allowed.

Multiple simultaneous read operations are allowed, but neither multiple simultaneous write operations nor simultaneous read and write operations are allowed.

After completing the file operation, the following routine needs to be called to return the permission. Note that the file must be closed before calling this routine. In other words, it is impossible to continue operating on the file while maintaining its current status, such as read/write cursor, after calling this routine.

UINT32 exit_filesys(UINT32 mode);

5.2 File Operation Routines

The declarations of the file operation routines are all in the header file include\libfile\file_sys.h. In addition to the two routines mentioned above, the following are the main routines that are frequently used:

FCB_POINT *file_open(UINT8 *filename, UINT8 *mode, struct user *u);
UINT32 file_close(FCB_POINT *fpoint);
UINT32 file_write(FCB_POINT *fpoint, INT8 *buf, UINT32 length);
UINT32 file_read(FCB_POINT *fpoint, INT8 *buf, UINT32 length);
UINT32 file_seek(FCB_POINT *fpoint, INT32 length, INT32 base);
UINT32 file_eof(FCB_POINT *fpoint);
UINT32 create_dir(UINT8 *dir_name, struct user *u);
int file_remove_directory(char *dir_name, struct user *u);
UINT32 file_del(UINT8 *filename, struct user *u);

The mode for file_open() can be “r”, “r+”, “w” or “w+”. If the file is operated on in a command implementation function, the argument u is the user of this command, otherwise it can be NULL.

6. Add a New Module to the Software System

Besides the necessary additions related to the makefile structure mentioned in document “How to Build Switch Software Image”, when a new module is added to the software system, the things to be added in the source code of some other modules include:

(1) The macro definition of the module type, which uniquely identifies this module in the software system, needs to be added in the header file include\libdev\modules.h. This module type is not only used for the registration of debug commands, show running-config callback function and module version information, but also for redundancy management in stacking or distributed systems.
(2) Some logic needs to be added in the function interface_omnivorous_callback_showrunning() in the file sys\libdm\interface.c for configurations collection by command “show running-config”.
(3) Add a call to the initialization function of this module in the module init. Note that adding the call to the initialization function in root.c is an informal alternative, which may not ensure that the module is initialized in an appropriate order.

Due to source code permissions, in actual development, you can request your team leader to help you with these actions.

7. Protocol Entity Design

7.1 Protocol Entity

A protocol entity refers to an implementation of hardware, software, or a combination of software and hardware that conforms to the semantics of one end of a protocol. The semantics of the entities of some protocols in the network are the same, for example, the semantics of IP, UDP and TCP protocol entities of different end stations are almost identical. However, the semantics of protocol entities in the majority of application layer protocols differ because they play different roles on two ends of the protocol, such as the client and server sides of TFTP.

When we say we want to design and implement the functionality of a protocol, we always need to clarify which protocol entity we will work on, such as client or server side.

7.2 Procedural Design

Procedural design should be able to be directly converted into code, rather than just providing an abstract conceptual model from a human perspective. Converting procedural design into code usually only involves refining the processing details represented by the flowchart or pseudo-code. If you find it's difficult to implement the procedure when trying to convert the design into code, or the processing structure in the design differs significantly from the feasible structure in code, you may not have designed it from a code perspective.

A procedure refers to a relatively independent and continuous behavior completed within the same unit of work (e.g. task, Linux process, thread etc.) of the same protocol entity (e.g. server side of TFTP). Therefore, the design of a procedure cannot be in the following situations:

(1) Spanning multiple protocol entities, such as including both the client and server sides. This is not procedural design, but protocol interaction diagram from a human perspective.
(2) Operations across multiple units of work within a single protocol entity, such as operations across two tasks. Maybe a behavior needs to be completed by two tasks in collaboration, however, as a procedural design, it needs to be done from the perspective of a single unit of work. In this case, you can design two procedures.
(3) Operations that are not continuous in time, although within a single unit of work. For example, the operations of a behavior of a protocol entity is like this: receiving a packet from peer, responding with an ACK, and then waiting for the next packet. From a human perspective, this seems to be a continuous procedure, however, from a code perspective, it is not. After I send an ACK, the next packet may not arrive immediately, and I cannot accurately predict when it will arrive. Therefore, the code is not suitable to wait for the arrival of the packet at this point all the time, but it can turn to handling other events, such as timeout event, command configuration event, etc. Therefore, if we sequentially put the above operations, i.e., receiving packet, acknowledging and receiving packet again, from a human perspective, into a procedure, it is inconsistent with the code perspective and cannot be directly converted into code. For this case, we need to design the procedure as an event driven processing structure, rather than the single line packet receiving

structure mentioned above.