Journal of Advanced Computer Science and Technology, 1 (3) (2012) 116-133
©Science Publishing Corporation
www.sciencepubco.com/index.php/JACST

Implementing Stack E6 via OS Linux Sockets

Zaitsev D.A., Kharsun M.A.

Dr.Sci., Professor
Senior Member of the IEEE, Member of ACM, SIAM
International Humanitarian University
E-mail: zsoftua@yahoo.com
Bachelor of Computer Science
International Humanitarian University
E-mail: mikefromsky@gmail.com

Abstract

New software implementation of Ukrainian national stack of networking protocols E6 was presented. Within operating system Linux kernel, families of E6 protocols and addresses were created as well as functions of their processing, in the form of a loadable module. Application of socket technology gives a series of considerable advantages among which unified with other stacks application interface, reuse of the kernel resources facilitating the further development and enterprise implementation of stack E6, engineering corresponding networking devices.

Keywords: *stack of networking protocols, E6, socket, kernel, Linux*

1 Introduction

Packet switching networks dominate in modern communications world. There is a tendency of network devices transformation into specialized computers with predominance of software that implements high layers of open systems interconnection basic reference model before data-link encoding and direct transfer of signals within a media.

Acquirement of specialized software development methods for the implementing network protocol stacks, packet switching and routing algorithms is crucial for providing ways not only to exploit devices that are developed in other countries, but also to produce the own ones.

Recently Ukraine turns into a country, using network technologies developed in other countries, due to the importing network devices and passive equipment. Not only development, but the production of high-tech equipment is situated outside the country.

Information about using programming code in the shape of virus-worm for destabilization of uranium enrichment centrifuges in Iran appeared in ACM news. That's why aspects of exploited software source code openness become more important. Complexities in examination of open specifications stimulate own development of hardware (processors) by many other countries. Well known examples of using software and hardware methods of communicating equipment in realization of military and political sanctions against Third World countries promote the initiation of national programs of production computing and communications machinery.

As the production of equipment requires significant investment, the software development is a more dynamic sphere. Its possibility of expansion is stipulated by personnel's level of qualification and organization. Using operating environments compatible with freeware OS Linux in network devices, allows an experimental development on home and office personal computers.

In Ukraine, a national stack of networking protocols E6 [1,2] was developed and provided its experimental realization via additional OS Linux kernel system call [3,4]. In spite of some advantages, said way has significant number of shortcomings, which prevent further development of national stack E6 and its industrial implementation. The main flaw is the need of OS kernel recompilation for the new system calls addition and substantial modification of the API; besides there are no possibilities of the reuse kernel methods involved in the implementation of other stacks, TCP/IP for example.

Therefore, the independent realization of E6 datagram style on the basis of standard OS Linux socket interface (the same as vast majority of other well known protocols) was implemented in International Humanitarian University (www.mgu.com.ua). This development is unique, because it requires a detailed study of OS Linux kernel environment and significant integration with its data structures and functions.

The goal of the present work is the rendering the software implementation of E6 stack via OS Linux socket interface. It may be also considered as a case study of new protocols software implementation in Unix-like operating environments.

2 Creating E6 sockets within OS Linux

OS Linux socket is a generalized inter-process communication channel, which is represented as a file descriptor. As a rule, sockets are used for communication between processes running on different computers. To provide communication between two sockets, they have to support the same style and

protocol of interaction. There are three basic styles supported: datagram (`DGRAM`), stream (`STREAM`) and raw (`RAW`).

For the external indication of socket, its address is used, so the main features of socket classes are address family `(AF)` and the appropriate protocol family `(PF)`. Among the known address families, `AF_LOCAL` – local format of operating system and `AF_INET` – IP address family should be noted. During the realization, the new E6 address family was created.

The main advantage of using sockets is a standard and consistent API for each protocol, implemented via this interface. Basic functions of sockets are contained in `libc` library, the main header file is `sys/socket.h`; besides for specific address/protocol families, own header files are added, `netinet/in.h` in TCP/IP, for example. For stack E6, the file `e6.h` was created.

Socket is created by the function

```
int socket(int protocol_family, int style, int protocol)
```

as result, the integer descriptor of associated file is returned; the set of available protocols is determined by the selected family, for example, `UDP_PROTO`, `TCP_PROTO` for TCP/IP family. In most cases, the protocol is uniquely determined by the chosen style and can be omitted.

E6 socket for the datagram communication mode, that is similar to UDP protocol, is created by the command

```
sock = socket(PF_E6, SOCK_DGRAM, IPPROTO_UDP);
```

Binding socket with a particular address is performed by the function

```
int bind (int socket, struct sockaddr *addr, socklen_t length)
```

where `sockaddr` structure is a generic data type, which allows using different address systems, so the third parameter is an actual length of the structure. For E6 address family the following description from `e6.h` header file is used

```
#define E6_ADDR_LEN     6
#define PF_E6           33
#define AF_E6           PF_E6
typedef uint8_t e6_addr_t[E6_ADDR_LEN];
struct e6_addr
{
      e6_addr_t s_addr;
};
typedef uint16_t e6_port_t;
struct sockaddr_e6
{
      __SOCKADDR_COMMON (se6_);
      e6_port_t se6_port;           /* E6 port number.  */
      struct e6_addr se6_addr;      /* E6 address.  */
      /* Pad to size of `struct sockaddr'.  */
      unsigned char se6_zero[sizeof (struct sockaddr) -
      __SOCKADDR_COMMON_SIZE - sizeof (e6_port_t) - sizeof (struct
e6_addr)];
};
```

Thus E6 socket address `sockaddr_e6` consists of E6 host address of 6 octets and E6 port number of 2 octets. System of addresses and ports is

independent of other protocol families; particularly, it is independent of TCP/IP (fig. 1).

An example of binding E6 server socket that allows calls from arbitrary E6 addresses to port 25, has the following form

```
#define E6ADDR_ANY ((struct
6_addr){.s_addr={0x00,0x00,0x00,0x00,0x00,0x00}})
server.se6_family = AF_E6;
server.se6_port = htons (25);
server.se6_addr = E6ADDR_ANY;
bind( sock, (struct sockaddr*) &server, sizeof(server))
```

Note, that operation of conversion `(struct sockaddr*)` results in the transforming a pointer of specific socket address to a generalized type, that is required by function `bind;` zero address `E6ADDR_ANY` can receive datagrams from arbitrary E6 addresses.

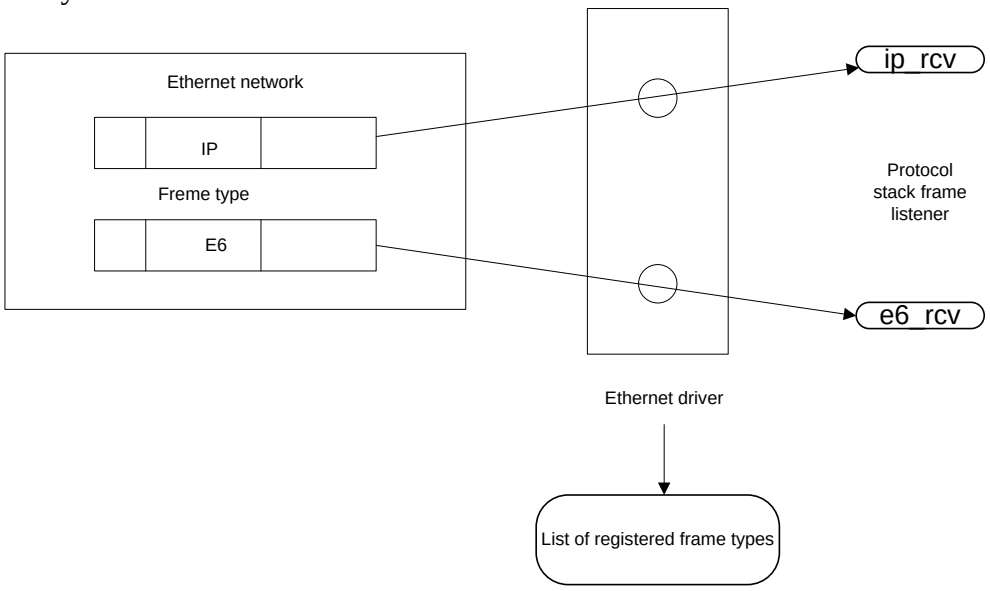


Fig. 1: Demultiplexing frames of different types by Ethernet driver

After binding, the exchange of data is performed by `recvfrom` and `sendto` functions, which provide an opportunity of remote socket address receiving. In addition, it is permissible to establish a connection with `connect` function and then the further exchange is performed by functions `send` and `recv`, which do not require address of the remote socket; this style of exchange is more characteristic for the stream mode.

Functions' headers look as

```
int recvfrom (int socket, void *buffer, size_t size, int flags,
                  struct sockaddr *addr, socklen_t *length-ptr);
int sendto (int socket, void *buffer, size_t size, int flags,
                  struct sockaddr *addr, socklen_t length);
```

where `buffer` variable points to the datagram address buffer, and `size` is its length in bytes, `flags` variable sets the mode of exchange and in the simplest case equals to zero. Functions return the actual number of sent/received bytes; negative

values correspond to an error. Function `recvfrom` creates socket address `addr` and length values using `length-ptr` pointer in accordance with the actual sender of the datagram; `sendto` function requires the receiver socket address `addr` and its length `length`.

An example of the server work cycle body has the form

```
len = recvfrom( sock, request, 1500, 0, (struct sockaddr *)&client,
&size );
      <обработка запроса request, формирование ответа reply>
len = sendto ( sock, reply, len, 0, (struct sockaddr *)&client,
size );
```

The socket address mapping into domain names is also provided by `gethostbyname` function and its modifications as well as socket listening `listen` and receiving connection requests `accept`. These methods are not used in the current implementation of E6 stack.

3 Interface of the loadable kernel module E6-socket

Almost each OS Linux protocol stack is implemented as a loadable kernel module. On the one hand it provides some autonomy for independent development, and on the other hand the direct use of the kernel environment in the form of special data structures and functions. Some modules, such as, for example, TCP/IP stack, is loaded together with the OS loading that creates an external effect of its presence in the static part of kernel - binary file `linuz`.

Modules that are not included in Linux startup scripts are loaded, unloaded and maintained by special commands:

`insmod <module name> <parameters>` – load module;

`rmmod <module name>` – unload module;

`modinfo <module name>` – get information about loaded module;

`modprobe <keys> <module name> <parameters>` – load system module.

Parameters are formed by special macros `module_param` during the module development and have the following key format of instruction `<parameter name>=<parameter value>`.

Command `insmod` loads an arbitrary specified file, whereas command `modprobe` works with modules located in `/lib/modules` directory and registered within system with `depmod` command; advantage of `modprobe` command is the ability to specify additional keys.

E6-socket module has name `e6.ko`; type `.ko` is standard and formed from the reduction of words “kernel object”. To build a kernel module, a special way of compilation and linking with `make` command is used to access the current kernel symbol tables; the required by command `make` files `makefile` and `KBuild` are the following:

```
Makefile::
      obj-m  := e6.o
      KERNELDIR := /lib/modules/2.6.31.5-desktop-1mnb/build
      all:: $(MAKE) -C $(KERNELDIR) M=`pwd` modules
```

```
KBuild::
      obj-m  := e6.o
```

E6-socket module parameters are: name of E6 device in OS Linux devices' name format – `E6_devname`, E6 device address (put in place of the factory MAC-address) – `E6_devaddr`. Loading command example is the following:

```
insmod e6.ko E6_devname=eth0 E6_devaddr=000000000001
```

In case of conflict when loading the module into the unknown kernel, using the `modprobe` command with the forced loading keys is recommended.

After E6-socket module is loaded, it is integrated into the system of known protocol stacks and Ethernet frames by registering new protocol family (`E6_PF=33`) and new frame type (`ETH_P_E6=0xE600`). As a result, the standard socket environment sets pointer to E6 create socket program `e6_create`, and the Ethernet driver receives a pointer to E6 frames processing program `e6_rcv`.

This provides the use of standard user interface to work with sockets' library `libc`, as described in the previous section, based on the functions `socket`, `bind`, `recvfrom`, `sendto`, and others. Necessary supplements are associated with a specific E6 address format, which determines the appropriate E6 socket format. Recall that, in comparison with the IP address, the E6 address has a length of 6 bytes, the hierarchical structure and is used both as a network and the data-link address at the same time [1,2] - instead of the IP and MAC addresses.

Data structures for describing the E6-address, E6-socket, and macros for standard E6 addresses, such as the loopback address, arbitrary address, and broadcast address are collected in the customer header file `e6.h`. Said header is required to specify in the include directive in the beginning of an application that uses the E6-socket module facilities:

```
#include "e6.h"
```

For testing the E6-socket module, the simplified remote shell application was used. It is recommended for self-study training to modify and recompile the standard TCP/IP stack applications, for example, TFTP to work via E6 stack.

4 The interaction of E6-socket module with the kernel

The interaction of a loadable module with the kernel is divided into four main classes:

- call to the module initialization function;
- call to the module deinitialization function;
- calls of the kernel to the module standard functions for working with E6 sockets;
- calls of the module programs to the supporting functions of the kernel.

The overall structure of the E6-socket module is shown in fig.2. After the module is loaded into memory, the `insmod` command starts automatically the module initialization function that is registered with the system macro

```
module_init(e6_init);
```

the header of the initialization function has the following form

```
static int __init e6_init(void)
```

Within the initialization function, the following procedures are executed: E6 protocol family registering, E6-device search and its initialization and the announcement of a new type of E6 frame. Thus the loadable module methods are integrated into OS Linux kernel environment. To register a family of protocols, the function

```
proto_register(&e6_proto, 0);
```

is called which receives the `e6_proto` structure address indicating the name, owner, and the length of E6 protocol family socket. Structure `e6_proto` has the following form

```
static struct proto e6_proto = {
      .name       = "E6",
      .owner      = THIS_MODULE,
      .obj_size   = sizeof(struct e6_sock),
};
```

Then the E6-device initialization takes place by the internal E6-socket module function `e6_init_dev` call

```
e6_dev=e6_init_dev(e6_devname,e6_devaddr,&e6_myaddr);
```

During executing this function, an attempt of the network device initialization is done; in case of success E6 address is assigned to the network device and the system is notified regarding the readiness of the device to the further work. Then the virtual loopback interface is initiated

```
lo_dev = dev_get_by_name(&init_net, lo_devname);
```

The next command

```
sock_register(&e6_family_ops);
```

implements the E6 socket registration: it specifies the protocol family, address of the sockets' creating function and their owner. Structure `e6_family_ops` has the following form:

```
static struct net_proto_family e6_family_ops = {
      .family = PF_E6,
      .create = e6_create,
      .owner  = THIS_MODULE,
};
```

It contains E6 protocol family name, function `e6_create` to create a socket of this type and its belonging. Function `e6_create` has the header

```
static int e6_create(struct net *net, struct socket *sock, int
protocol)
```

and describes the process of creating E6 socket.

The following call of function

```
dev_add_pack(&e6_packet_type);
```

registers new type of Ethernet frame: this function adds specified frame type handler, using `e6_packet_type` structure and includes E6 frame type into the kernel database

```
static struct packet_type e6_packet_type = {
      .type =     __constant_htons(ETH_P_E6),
      .func =     e6_rcv,
};
```

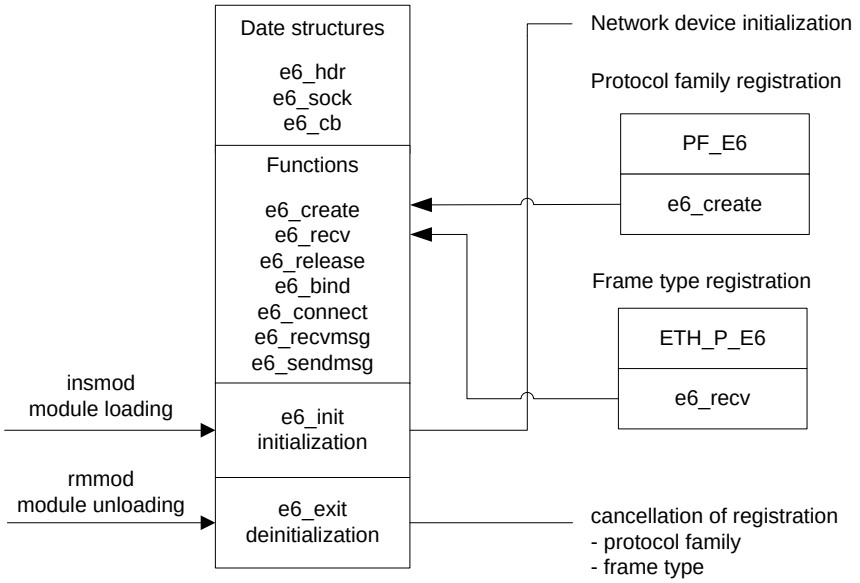


Fig. 2: E6 loadable module structure

As a result of executing the loadable module initialization function, a set of data structures, constants, macros and functions joins OS Linux kernel, to implement the datagram mode of transmission on the basis of the E6 stack. There are two basic module entry points that are imported into OS Linux database: `e6_create` to create E6 socket with `socket` user-end function at protocol family `PF_E6` specification; `e6_rcv` for the listening received frames of `ETH_P_E6` type. Thus, module interfaces of application and data-link layers are created.

Access to the functions of E6 module, implementing the standard socket interface `(bind, connect, release, recvfrom, sendto)` is provided by the table `e6_dgram_ops` of pointers to the standard socket entry points

```
const struct proto_ops e6_dgram_ops = {
      .family              = PF_E6,
      .owner               = THIS_MODULE,
      .release             = e6_release,
      .bind                = e6_bind,
      .connect             = e6_connect,
      .socketpair          = sock_no_socketpair,
      .accept              = sock_no_accept,
      .getname             = sock_no_getname,
      .poll                = datagram_poll,
      .ioctl               = sock_no_ioctl,
      .listen              = sock_no_listen,
      .shutdown            = sock_no_shutdown,
      .setsockopt          = sock_common_setsockopt,
      .getsockopt          = sock_common_getsockopt,
      .sendmsg             = e6_sendmsg,
```

```
      .recvmsg             = e6_recvmsg,
      .mmap                = sock_no_mmap,
      .sendpage            = sock_no_sendpage,
};
```

Note that only the user functions `e6_release`, `e6_bind`, `e6_connect`, `e6_sendmsg`, `e6_recvmsg` are implemented. For the rest of functions, system stubs are used instead. The following is a snippet of `e6_create` function code that creates E6 socket

```
sk = sk_alloc(net, PF_E6, GFP_KERNEL, &e6_proto);
sock->ops            = &e6_dgram_ops;
sk->sk_family        = PF_E6;
sk->sk_protocol      = protocol;
e6_insert_socket(&e6_sklist, sk);
```

After the memory allocation by `sk_alloc` kernel function, address of table of pointers to the standard functions `sock->ops` is installed. So each socket contains a pointer to the table of its functions, which provides its autonomy. Finally, created socket is included into general E6 socket list by `e6_insert_socket` function.

In the case of input `rmmod` command from the Linux console, the process of loadable module deinitialization begins. Before releasing memory occupied by the module, the entry point to the completion of the module is automatically invoked, given by the system macro

```
module_exit(e6_exit);
```

The header of the completion function of the module has the form

```
static void __exit e6_exit(void)
```

The main purpose of this function is cancelling the registration of the protocols family and types of packets (frames). The function call

```
dev_remove_pack(&e6_packet_type);
```

removes records about E6 frame type in the kernel database. Then functions

```
sock_unregister(PF_E6);
proto_unregister(&e6_proto);
```

perform removal of E6 protocol and addresses family from the kernel database.

Thus, the two entry points to the module are explicitly registered and called by the kernel in the following cases: `e6_create` to create a customer E6 socket with the `socket` function; `e6_rcv` by Ethernet driver when receiving E6 frames. Other functions correspond to the user-end functions to work with sockets; kernel determines their addresses from `e6_dgram_ops` entry points table for the specified socket (fig.3).

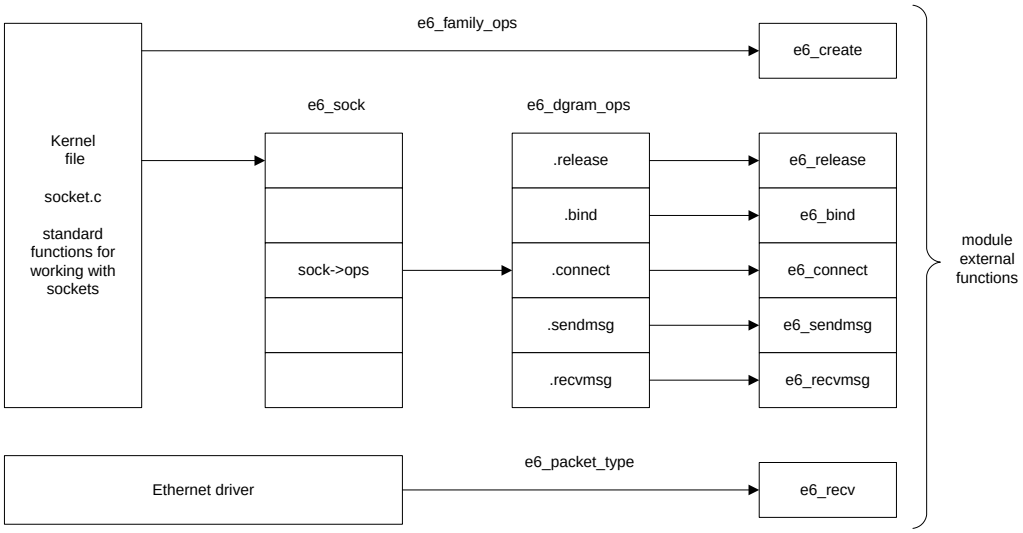


Fig. 3: Access to the E6-socket module functions

More subtle issues of interaction, which reveal the main stages of the transfer of control and data from the application process and the network device driver by the module functions, are discussed in Section 6.

5 Algorithms of E6-socket module basic functions

To understand the algorithms of the module, should pay attention to the basic data structures at first. In addition to described in the previous section structures `e6_proto`, `e6_family_ops`, `e6_packet_type`, providing the registration of a new stack in the kernel, as well as structures `sockaddr_e6`, `e6_addr` and the type `e6_port_t` used to specify the E6 addresses and described in Section 2, the module contains a description of a number of internal data structures.

Moreover, the module uses the standard kernel data structures

`struct socket` – generalized socket;

`struct sock` – representation of a socket at the network level;

`struct sk_buff` – buffer for the transmission of the Ethernet frame;

`struct net_device` – network device descriptor.

Generalized socket is designed to provide a uniform standard handling of different types of sockets in the kernel; structure `socket` contains a pointer to the associated structure `sock`; frame buffers `sk_buff` are created by the module during message transmission and by driver during receiving frame from network; net device descriptor `net_device` contains its features. To understand the interaction between data structures, it is essential to the study formed linked lists. All E6 sockets are collected in a list `e6_sklist`, in addition, all the sockets of all

types are included in the general list. To each socket `sk`, a queue of received messages of `sk_buff` type is formed. To network device `net_device`, a single queue of sent packets (frames) of type `sk_buff` is formed. Note, that the key of socket search in `e6_sklist` list is E6 port.

Thus, the message transmission function comes to the formation of `sk_buff` and inserting it to the queue of network device with `net_device` header; the receiving message function comes to the extracting a frame from the appropriate socket queue `sk`, depending on specified flags in case of the frame absence in the queue, either the blocking of the process is executed for frame entrance waiting or an error code returning. Frames `sk_buff` get into the socket queue `sk` via the call by Ethernet driver of registered during module initialization function `e6_rcv` after receiving E6 frame from network.

Module contains definitions of the following internal data structures. E6 frame header (data-link and network) contains pares of E6 addresses of receiver `de6a` and sender `se6a`, and pares of sender `de6p` and receiver `se6p` ports

```
struct e6_hdr {
      struct e6_addr          de6a;
      struct e6_addr          se6a;
      __be16                  type;
      e6_port_t               de6p;
      e6_port_t               se6p;
};
```

E6 socket contains standard socket `sk` and specific fields to specify the E6 addresses and port numbers of the sender `saddr, sport` and receiver `daddr, dport` and flags as well.

```
struct e6_sock {
      struct sock sk;
      struct e6_addr          saddr;
      e6_port_t               sport;
      struct e6_addr          daddr;
      e6_port_t               dport;
      __u16                   cmsg_flags;
};
```

The following static variables are created

`static struct hlist_head e6_sklist;` – header of E6 sockets list;

`struct e6_addr e6_myaddr;` – own E6 address;

`static struct net_device *e6_dev;` – address of E6 device descriptor.

Let us classify the functions of E6-socket module:

- initialization and deinitialization functions `e6_init` and `e6_exit;`
- socket user API implementation functions: `e6_create, e6_release, e6_bind, e6_connect, e6_sendmsg, e6_recvmsg`;
- function-handler of received E6 packets (frames) `e6_rcv;`
- supporting functions: socket removing `e6_remove_socket`, inserting socket to the list `e6_insert_socket`, searching socket with specified port `e6_find_port`, searching free port `e6_bind_port`, searching socket with

specified address and port `e6_find_socket`, device initialization `e6_init_dev`.

All module functions and data structures have prefix `e6_`. Note that for the implementation of several user functions, only one module function can be used; for example to implement functions `sendmsg, sendto, send` in the datagram style, the module function `e6_sendmsg` is used, and to implement `recvmsg, recvfrom, recv` functions, `e6_recvmsg` is used. Formation of various call parameters is executed by intermediate kernel programs that are located in `socket.c` kernel file.

E6 socket creating function code snippets were considered in Section 4. Let us study the functions' algorithms implementing sending and receiving message operations. The sending message function has the form

```
static int e6_sendmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg, size_t len)
```

where `iocb` is the block of input/output control, `msg` is a message header, `len` is the message length. Initially, memory is allocated for the packet (frame) buffer

```
skb = sock_alloc_send_skb(sk, len + sizeof(struct e6_hdr),msg-
>msg_flags & MSG_DONTWAIT, &err);
```

Copying information from the user's buffer to `skb` block is performed by the `memcpy_fromiovec` command using the message header `msg` parameters

```
skb_reserve(skb, sizeof(struct e6_hdr));
err = memcpy_fromiovec(skb_put(skb,len), msg->msg_iov, len);
```

initial part of buffer is reserved by `skb_reserve` command for subsequent placement of `e6_hdr` header. Recall that, `e6_hdr` is a combined network and data-link layers header. Then the packet (frame) header is formed

```
hdr = (struct e6_hdr *)skb_mac_header(skb);
hdr->de6a = daddr;
hdr->se6a = addr;
hdr->type = htons(ETH_P_E6);
hdr->de6p = dport;
hdr->se6p = port;
```

The output device and the priority of the operation is specified

```
skb->dev = e6_dev;
skb->priority = sk->sk_priority;
```

The actual launch of the operation on the device is performed by the sequence of commands

```
dev_queue_xmit(skb);
dev_put(dev);
```

Command `dev_queue_xmit` inserts the frame buffer `skb` to the device queue; command `dev_put` forces the device driver to get started in case of absence of active operation and does not involve any changes otherwise; the driver will check the queue at the end of the active operation.

Message receiving function has the header

```
static int e6_recvmsg(struct kiocb *iocb, struct socket *sock,
struct msghdr *msg, size_t len, int flags)
```

where the assignment of parameters corresponds to the previously described function `e6_sendmsg` reversing the transmission direction at the data

interpretation. Getting datagram in `skb` block from queue to the specified socket `sock` is performed by the command

```
skb=skb_recv_datagram(sk,flags,flags&MSG_DONTWAIT,&err);
```

which in case of the datagram absence blocks the current process at indicating `MSG_WAIT` flag and returns an error at `MSG_DONTWAIT` flag. Then the copying of received data from `skb` block to user buffer is done, which `msg->msg_iov` address is contained in `msg` header block

```
err = memcpy_toiovec(msg->msg_iov, skb->data+4, copied);
```

Then from `skb` block the socket address is copied to the to message header block `msg` for the following return to the application process

```
msg->msg_namelen = sizeof(struct sockaddr_e6);
memcpy(msg->msg_name, skb->cb, msg->msg_namelen);
```

Using a simple copy command `memcpy` is due to the fact, that copying the socket address is performed inside the kernel address space as opposed to copying data from kernel space to user space (input/output block) by command `memcpy_toiovec`. The processing is completed via releasing datagram memory by the command

```
skb_free_datagram(sk, skb);
```

Thus, the receiving function is related to network device driver asynchronously through a datagram queue `skb` to a socket, which is formed by `e6_rcv` function considered later.

Function of handling E6 frames received by driver has the header

```
static int e6_rcv(struct sk_buff *skb, struct net_device *dev,
struct packet_type *pt, struct net_device *orig_dev)
```

where `skb` is a received frame buffer, `dev` is a network device, `pt` is a frame type descriptor, `orig_dev` is a source device, which may not match the `dev` in the case of redirection.

Function `e6_rcv is` called by Ethernet driver after receiving frame of type `ETH_P_E6` in accordance with registered during initialization packet type `e6_packet_type`, address of the corresponding structure `pt` is specified additionally in call parameters.

The driver performs a preliminary check of the packet destination address, the results of which can be used to ignore packets addressed to other hosts, because E6 address is used as well as the MAC address and set on the NIC during the module initialization

```
if (skb->pkt_type == PACKET_OTHERHOST) goto drop;
```

Label `drop` designates action of the packet ignoring, reduced to releasing block `skb` and return of the error code `NET_RX_DROP`

```
drop: kfree_skb(skb);
      return NET_RX_DROP;
```

In the module, a reliable version of the test is used in the form of an explicit comparison of E6 destination address with own `e6_myaddr` and loopback `lo_e6_addr` addresses`

```
hdr = (struct e6_hdr *)(skb->data-E6_TRANSPORT_HEADER_OFFSET);
if((memcmp(hdr->de6a.s_addr, &e6_myaddr, E6ADDRLEN) != 0) &&
```

```
   (memcmp(hdr->de6a.s_addr, &lo_e6_addr, E6ADDRLEN) != 0) ) goto
drop;
```

Then the search for E6 socket with specified in the datagram port is run

```
sk = e6_find_socket(hdr->de6p, hdr->de6a);
if (!sk) goto drop;
```

Then the socket address of the received frame is stored in the supporting block `cb` for the further use in the receiving packet function `e6_recvmsg`

```
e6cb=(struct e6_cb *)skb->cb;
e6cb->se6.se6_family=AF_E6;
e6cb->se6.se6_addr=hdr->se6a;
e6cb->se6.se6_port=hdr->se6p;
```

The processing is completed with inserting `skb` block to the queue of received datagrams for found socket `sk`

```
if (sock_queue_rcv_skb(sk, skb)) goto drop;
```

Nonzero return code of `sock_queue_rcv_skb` function corresponds to an error and leads to ignoring the packet. Interaction of Ethernet driver, `e6_rcv` function and `e6_recvmsg` function are illustrated in fig.4. Note that running of functions `e6_rcv`, `e6_recvmsg` is asynchronous: `e6_rcv` is called by driver and `e6_recvmsg` runs while handling the corresponding system call of the application process.

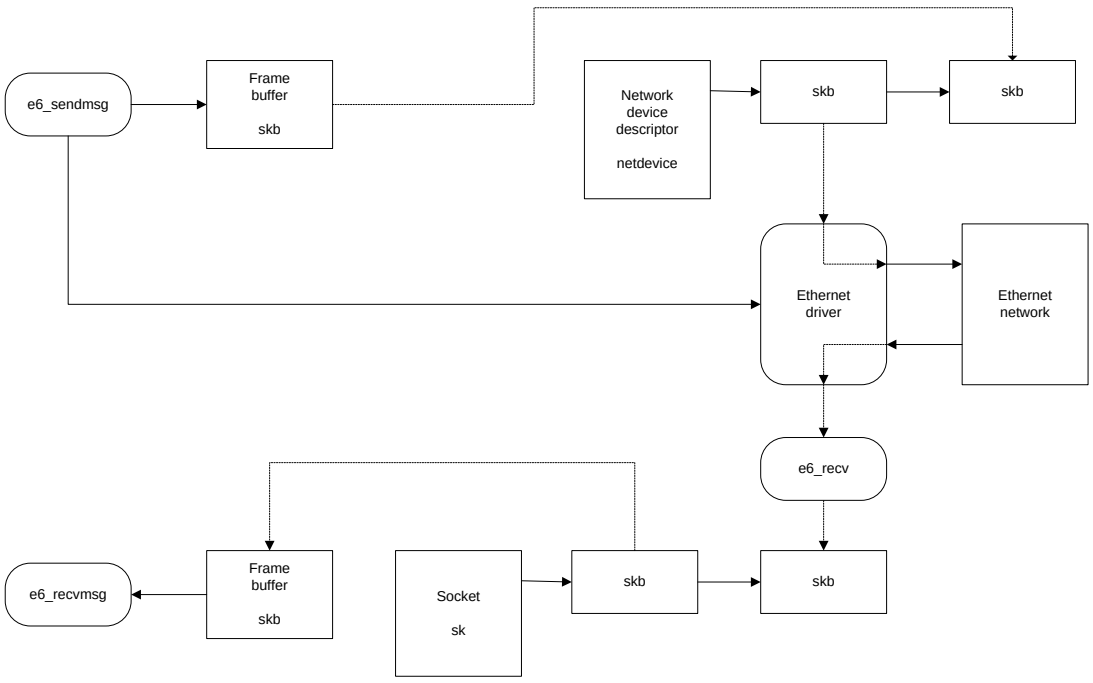


Fig. 4: Implementation of the datagram send/receive functions

6 Tracing E6 packets transfer processes

Tracing the sequence of actions implementing certain user commands allows us to understand peculiarities of the interaction among:

- application source code in C language written by the user;
- the object code of the library `libc` linked to the application;
- static part of the Linux kernel;
- loadable module.

Let us consider the tracing process on the example of user-defined function of the receiving message by a server application

```
len = recvfrom( sock, request, 1500, 0, (struct sockaddr *)&client,
&size );
```

Initially, control inside application code is passed into `recvfrom` function of `libc` that performs a preliminary check of parameters and generates a block of input/output parameters presented with an array of type `long`.

Then it calls the `syscall` function of `libc` with the number of system call `SYS_SOCKETCALL=102` and the number of subfunction `call=SYS_RECVFROM=12`, which execution leads to the application processor interruption number `0x80`, intended to implement the kernel system calls in Linux.

From the interrupt vector `0x80` in operating memory a new processor status word is retrieved, which contains the system call handler address `system_call`, and performs switching operation of the processor from the user to the system mode (fig. 5).

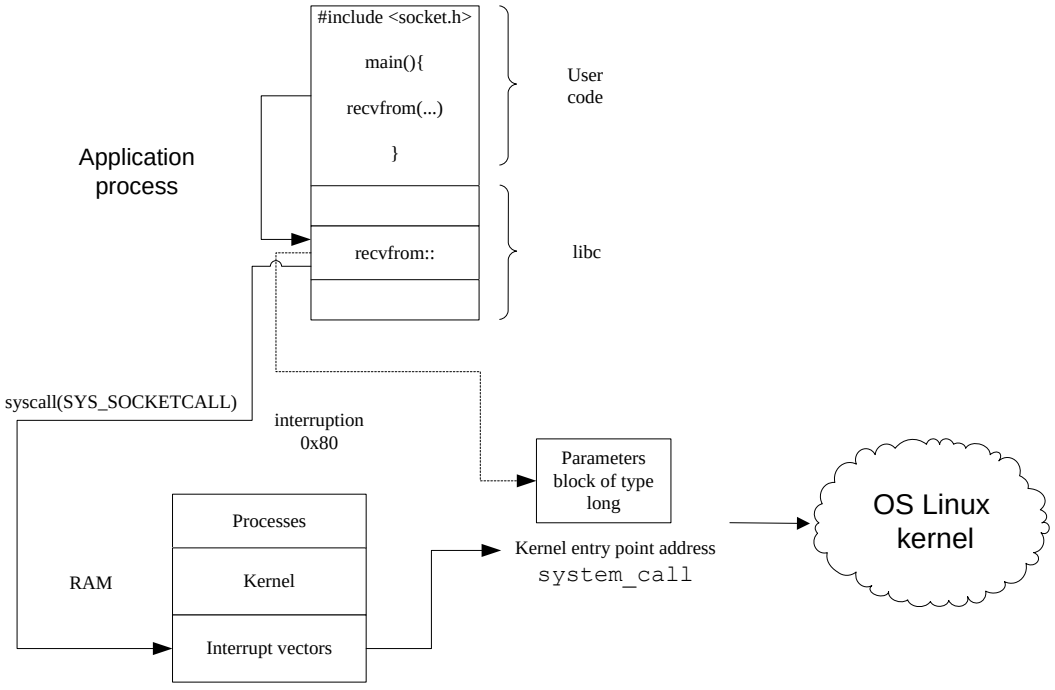


Fig. 5: Sequence of entry into OS Linux kernel

The program system_call placed in the kernel file arch/x86/kernel/entry_64.S works as a manager (switch): it finds the record having number SYS_SOCKETCALL=102

```
.long sys_socketcall
```

in system call table sys_call_table of kernel file syscall_table_64.S that contains the address of the function sys_socketcall which is a handler of all system calls regarding sockets.

Program sys_socketcall is a manager of socket system call subfunctions, which retrieves the parameters from the parameters block and passes control to the appropriate subprogram.

For SYS_RECVFROM subfunction, subprogram sys_recvfrom is called

```
err = sys_recvfrom(a0, (void __user *)a1, a[2], a[3],
(struct sockaddr __user *)a[4], (int __user *)a[5]);
```

which receives the first six parameters from the parameters block in accordance with previously described format of the user-end function recvfrom; for addresses specified by the user, user address space indicator __ user is written explicitly.

The header of sys_recvfrom program is formed by the system macro

```
SYSCALL_DEFINE6(recvfrom, int, fd, void __user *, ubuf, size_t,
size,       unsigned, flags, struct sockaddr __user *, addr, int
__user *, addr_len)
```

which contains sequentially the types and names of parameters after the function name. First of all, the address of socket in memory is determined on the file descriptor, which the operation is performed for

```
sock = sockfd_lookup_light(fd, &err, &fput_needed);
```

Then the call parameters are copied into the fields of the message header block msg of type msghdr and an input/output control unit (vector) iovec of type iov

```
msg.msg_control = NULL;
msg.msg_controllen = 0;
msg.msg_iovlen = 1;
msg.msg_iov = &iov;
iov.iov_len = size;
iov.iov_base = ubuf;
msg.msg_name = (struct sockaddr *)&address;
msg.msg_namelen = sizeof(address);
```

Then the program sock_recvmsg is called

```
err = sock_recvmsg(sock, &msg, size, flags);
```

The program sock_recvmsg after operations of safety audit is completed by the calling

```
sock->ops->recvmsg(iocb, sock, msg, size, flags);
```

thus, the address specified by the description line

```
.recvmsg              = e6_recvmsg,
```

is retrieved from e6_dgram_ops structure, which leads to the actual e6_recvmsg function call of loadable module E6-socket (fig. 6), which algorithm was studied in the previous section.

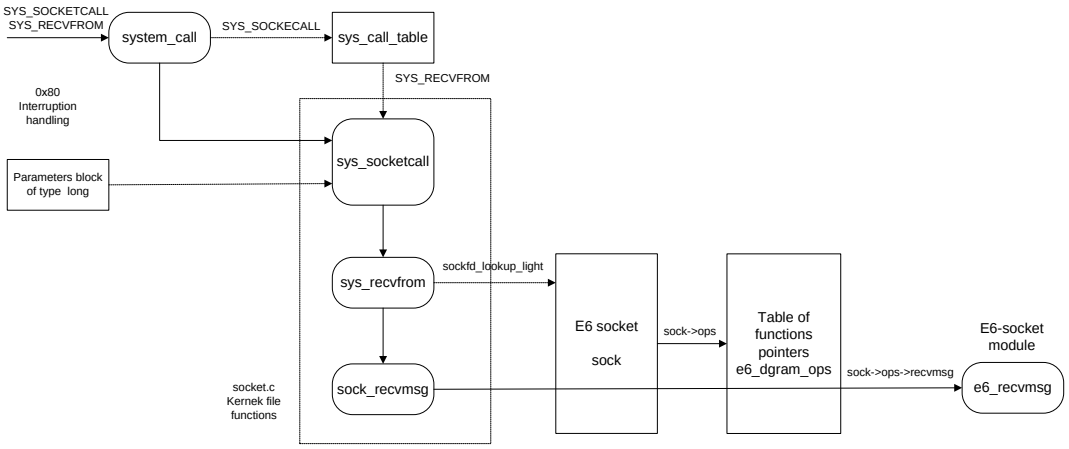


Fig. 6: The sequence of the module function calls within the kernel

Note that after the system call completion the reverse sequence of actions in not obligatory. Some of calls result in blocking the current process, for example, waiting for a message from the network. During this time, the kernel starts to execute other ready-to-execute processes, or a special process-sloth `idle` in the absence of ready processes.

In this case the change of process status will be caused by NIC interrupt and after its completion the driver starts the program `e6_rcv`, which in turn changes the state of the process while inserting the frame into the queue to the appropriate socket.

7 Conclusion

The technology of software implementation of Ukrainian national protocol stack E6 (the datagram mode) via OS Linux kernel sockets was presented which allows minimizing modification of user interfaces and provides the kernel code reuse that significantly reduces the complexity of development and creates a prototype for further industrial implementations of stack E6. The implementation was done in the kernel version 2.6.31.5 of the operating system Linux Mandriva 2010 and then moved to Linux Ubuntu, Debian, Fedora.

The material can also be considered as a case study of the software implementation technology of new protocol stacks in Unix-like operating environments.

The source files of the kernel module and test applications with brief instructions on how to compile, install and run them are put on the website http://daze.ho.ua and can be used as prototypes for the programmers to develop their own protocol stacks.

References

[1] Zaitsev D.A., Bolshakov S.I. “E6 Addressing Scheme and Network Architecture” *Journal of Advanced Computer Science and Technology*, No. 1, (2012), pp.18-31.
[2] Vorobiyenko P.P., Zaitsev D.A., Nechiporuk O.L. “World-wide network Ethernet?” *Zviazok (Communications)*, No. 5, (2007), pp.14-19. In Russ.
[3] Zaitsev D.A., Guliaiev K.D. “Stack E6 and its Implementation within Linux Kernel” *Journal of Software Engineering and Applications*, No. 4, (2011), pp.379-387.
[4] Guliaiev K.D., Zaitsev D.A. “Experimental Implementation of Networking Protocols Stack E6 into OS Linux Kernel” *Artificial Intelligence*, No. 2, (2009), pp.105-116. In Russ.
[5] Herrin G. *Linux IP Networking*, TR-0004, (2000).
[6] Bovert D., Cesati M. *Understanding the Linux Kernel*, O’Reilly, (2000).