

Linux kernel cryptographic functions: CRC calculation

J.-M Friedt, January 13, 2026

We wish to exploit the Cyclic Redundancy Check (CRC) capability provided by the Linux kernel. CRC is ubiquitous in most data storage and exchange to assess to integrity of the transmitted data. CRC calculation involves binary division of the input bytes with a set of coefficients representing a binary polynomial: the remainder of the division is representative of the input bit sequence.

1. How would you compare CRC calculation and parity bit added at the end of an RS232 communication stream?
2. Based on your knowledge of the SPI local communication protocol and IP local and world area networking (LAN/WAN) protocols, what are the degrees of freedom in ordering the information between two remote digital systems?
3. Provide a function able to flip the bit order within a byte, *i.e.* converting a byte provided as least significant bit first to a most significant bit first format.

The Linux kernel implements some common CRC calculation involved for example in Ethernet frame retrieval. The CRC result is available from userspace through a socket using the same interface than would be used for network communication. However, the socket structure is not initialized with an IP address but with a structure including

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type   = "hash",
    .salg_name   = "crc32" // CRC definition
};
```

following the definition of the socket as using a communication domain to access the kernel cryptographic API AF_ALG and communicating through a reliable, sequence interface SOCK_SEQPACKET:

```
f=socket(AF_ALG, SOCK_SEQPACKET, 0);
bind(f, (struct sockaddr*)&sa, sizeof(sa));
g=accept(fd, NULL, 0);
```

Once the file descriptor `g` has been initialized, communication with the kernel crypto layer is bidirectional, sending an array of bytes and retrieving the 32-bit resulting checksum.

One particular verification sequence is using a string including the ASCII characters representing digits 1 to 9. This 9-character array is provided to the kernel crypto API resulting in the checksum value.

4. How can you check if the cryptographic functions are included in the running kernel of the Red Pitaya, and especially the CRC32 algorithm?
5. Demonstrate how you communicate from the GNU/Linux userspace through the socket to send the data array and retrieve the CRC. What is the result?
6. Find in the kernel source tree the definition of the CRC and most importantly the binary value defining its polynomial. Where did you find the information? How did you find it? What is the polynomial defining `crc32`?
7. The online web site crccalc.com provides the means for calculating various types of CRC involving various polynomial lengths N (8, 16 or 32 bits), whether the output is XORed with $2^N - 1$ or not (last column) and the initial value of the CRC accumulator. The “Check” column is used to assess the CRC output with the ASCII string representing digits from 1 to 9. How does your output compare?
8. Based on the second question, can you find a data order that matches one of the outputs of the web site?

Rather than accessing the Linux cryptographic layer from userspace, we wish to perform the calculation from within the kernel.

9. Which header file do you include so the kernel module is aware of the CRC functions provided by the kernel?
10. Demonstrate how you provide a kernel module able to receive through a character device pseudo-file the sequence of values to be encoded.
11. when writing the string “123456789” from userspace to kernel space from the shell through a character device entry in `/dev`, how many bytes are actually transferred? Interpret this result and take care to use this information for the next question.
12. display the CRC calculation performed from kernel space when fed the ASCII string with digits 1 to 9: how does the result compare with the output from userspace?
13. how does the kernel cryptographic API implementation of the CRC compare with the userspace? Are both implementations providing the same level of flexibility?
14. the Cryptodev kernel module found at <https://github.com/cryptodev-linux/cryptodev-linux> allows for configuring through Input/Output Control system calls the cryptographic layer. Demonstrate how to cross-compile this module to the Red Pitaya using the Buildroot framework provided in room 235B. How do you make sure the resulting kernel module is targeted for the ARM architecture?
15. Demonstrate that after loading this kernel module, a new entry is created in `/dev/` to access the cryptographic system of the Linux kernel: demonstrate how a userspace program can access the newly created features. Which system call do you use to access the Cryptodev features? How do you know which argument to provide to this system call?

Answers

1. Bit parity cannot detect even number of errors. CRC will be more robust since analyzing all bits through a binary operation (XOR or polynomial division)
2. SPI can be MSB or LSB, so bit order can be reversed. Networking requires tuning the endianness so ordering the bytes amongst the larger integer structures (short, long).
3. a one-line implementation could be

```
#define bitswap(x) (((x&1)<<7)|((x&2)<<5)|((x&4)<<3)|((x&8)<<1)|((x&16)>>1)|((x&32)>>3)|((x&64)>>5)|((x&128)>>7))
```

but other solutions are found in the Linux kernel source tree in `linux/bitrev.h`

4. `CONFIG_CRYPT=y` in `output/build/linux-xilinx-v2024.1/.config`, or make `linux-menuconfig/` in the Buildroot directory and check that Cryptographic API→CRCs (cyclic redundancy checks)→CRC32 and CRCT10DIF are * or M. The socket call requires that Cryptographic API→User interface→Hash algorithms is enabled¹, otherwise the `accept()` call returns `-1`.
5. Using the socket calls:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/socket.h>
#include <linux/if_alg.h>

int main(void)
{int fd, od;
  struct sockaddr_alg sa=
  { .salg_family=AF_ALG,
    .salg_type  ="hash",
    .salg_name  ="crc32" // cat /proc/crypto | grep crc
  };

  //unsigned char datain[9]="123456789";
  unsigned char datain[9]={0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39};
  uint32_t crc;
  fd=socket(AF_ALG, SOCK_SEQPACKET, 0);
  bind(fd,(struct sockaddr*)&sa, sizeof(sa));
  od=accept(fd,NULL,0);
  write(od,datain,9);
  read(od,&crc,sizeof(crc));
  printf("CRC32: 0x%08x 0x%08x\n", crc, crc^0xffffffff);
  return 0;
}
```

outputs CRC32: 0x2dfd2d88 0xd202d277

6. the kernel source tree includes in `include/linux` a file `crc32.h` which defines the functions `crc32_le()` or `crc32_be`. These functions require the definition of the generator polynomials found in `include/linux/crc32poly.h` where we are told that `#define CRC32_POLY_LE 0xedb88320` and `#define CRC32_POLY_BE 0x04c11db7` as identified in the headers included by drivers implementing CRC32, e.g. `drivers/crypto/stm32/stm32-crc32.c` identified by searching for the call to `crc32_le()` in the kernel source tree.
7. We find on the online web site the solutions for the generator polynomials we have identified before

CRC-32/ISCSI	0xE3069283	0x1EDC6F41	0xFFFFFFFF	true	true	0xFFFFFFFF
CRC-32/BZIP2	0xFC891918	0x04C11DB7	0xFFFFFFFF	false	false	0xFFFFFFFF
CRC-32/CKSUM	0x765E7680	0x04C11DB7	0x00000000	false	false	0xFFFFFFFF
CRC-32/ISO-HDLC	0xCBf43926	0x04C11DB7	0xFFFFFFFF	true	true	0xFFFFFFFF
CRC-32/JAMCRC	0x340BC6D9	0x04C11DB7	0xFFFFFFFF	true	true	0x00000000
CRC-32/MPEG-2	0x0376E6E7	0x04C11DB7	0xFFFFFFFF	false	false	0x00000000

¹<https://xilinx-wiki.atlassian.net/wiki/plugins/viewsource/viewpagesrc.action?pageId=18841654>

with the first line matching `crc32c` and the last 6 lines matching `crc32`, with the initial accumulator setting of `0xFFFFFFFF` and XORing the output with `0xFFFFFFFF`. None of the solutions match the output of our program provided above, not even when XORing the output with `0xFFFFFFFF`.

8. We must try the various bitswap and byteswap (endiannesses) to check which one matches the online calculator: based on the above program

```

unsigned char data[9];
unsigned char *c;
unsigned char cc[4];
uint32_t *l;
[...]
for (int i=0;i<9;i++) data[i]=bitswp(datain[i]);
write(od,data,9); read(od,&crc,sizeof(crc));
printf("CRC32 data (kernel): 0x%08x 0x%08x -> ", crc,crc^0xffffffff);
c=(unsigned char*)&crc;
for (int i=0;i<4;i++) cc[i]=bitswp(c[3-i]);
l=(uint32_t*)(cc);
printf("byte/bitflip 0x%08X after XOR(0xffffffff)\n", (*l)^0xffffffff);
close(od); close(fd);

```

outputs, in addition to the previous answer:

```
CRC32 data (kernel): 0xfe918591 0x016e7a6e -> byte/bitflip 0x765E7680 after XOR(0xffffffff)
```

and we conclude that the entry CRC-32/CKSUM 0x765E7680 matches the final output where the bits had to be flipped on the input and the output as well as byte ordering. Swapping the bits of each byte is achieved through two pointers, from 32-bit integer to bytes and back from bytes to 32-bit integer.

9. `#include <linux/crc32.h>`

10. a basic character device kernel module implementing the CRC32 algorithm, where only the `write` (from userspace to kernel space) is implemented:

```

#include <linux/module.h>           /* Needed by all modules */
#include <linux/kernel.h>          /* Needed for KERN_INFO */
#include <linux/init.h>            /* Needed for the macros */
#include <linux/fs.h>              // define fops
#include <linux/uaccess.h>         // get_user, put_user
#include <linux/miscdevice.h>     // allocation dynamique de /dev
#include <linux/crc32.h>

static ssize_t crc_write(struct file *, const char *, size_t, loff_t *);

int crc_start(void);
void crc_end(void);

static struct file_operations fops={.write=crc_write,};

int crc_start() {int t=register_chrdev(91,"jmf",&fops);return 0;}
void crc_end() {unregister_chrdev(91,"jmf");}

static ssize_t crc_write(struct file *fil,const char *buff,size_t len,loff_t *off)
{int dummy,mylen;
 char buf[15];
 int crc;
 int seed=0; // 0xffffffff;
 printk(KERN_ALERT "write %d",(int)len);
 if (len>14) mylen=14; else mylen=len;
 dummy=copy_from_user(buf,buff,mylen);
 buf[mylen]=0;                printk(KERN_ALERT "%s",buf);
 crc = crc32_le(seed,buf, len-1); printk(KERN_ALERT "le %08x",crc);
 crc = crc32_be(seed,buf, len-1); printk(KERN_ALERT "be %08x",crc);
 crc = crc32(seed,buf, len-1);  printk(KERN_ALERT "no %08x",crc);
 return mylen;
}

module_init(crc_start);

```

```
module_exit(crc_end);
MODULE_LICENSE("GPL");
```

11. The string sent from userspace to the kernel includes the trailing line feed (ASCII 0x0A). This trailing character must be removed to match the string with ASCII characters representing digits from 1 to 9 only, otherwise the input string is erroneous to compare with the online calculator Check function. This is seen by displaying the `len` argument of the `write` system call implementation.

12. the kernel module output is

```
[10389558.435460] 123456789
[10389558.435491] 1e 2dfd2d88
[10389558.435495] be 89a1897f
[10389558.435508] no 2dfd2d88
```

so the `1e` implementation matches the userspace implementation when initialized with seed 0x0.

13. Linux kernel CRC calculation allows for defining the initial state of the CRC accumulator, which the userspace interface does not.

14. edit the Makefile and update the kernel source tree directory (argument of `-C`: `KERNEL_DIR ?= /home/jmfriedt/buildroot-2025.05.1_redpit/output/build/linux-xilinx-v2024.1/`), target architecture et cross-compilation prefix

```
KERNEL_MAKE_OPTS := ARCH=arm CROSS_COMPILE=arm-buildroot-linux-gnueabihf- -C $(KERNEL_DIR)
M=$(CURDIR)
```

Make sure that `/home/jmfriedt/buildroot-2025.05.1_redpit/output/host/usr/bin` is added to the `PATH` when compiling. The result of `make` is `cryptodev.ko` and we check the target architecture with

```
% $ file cryptodev.ko
% cryptodev.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV)
```

Make sure to have `AEAD` enabled in the Buildroot kernel to avoid errors at the linking step such as `crypto_alloc_aead` missing (option `CRYPTO_AEAD=y`)

15. the `ioctl` system call is found in `ioctl.c` and is the only one implemented in the driver (no read, no write). The list of possible options in `cryptodev_ioctl()` as defined in the file operation list `static const struct file_operations cryptodev_fops` where `.unlocked_ioctl = cryptodev_ioctl` so that the calls are `switch (cmd) : CIOCASYMFEAT, CRIOGET, ...`. Calling an `IOCTL` from userspace is achieved with

```
int main()
{int fe, fd=open("/dev/crypto",O_RDWR);
  ioctl(fd, CRIOGET, &fe);
  ioctl(fe, CIOCGSESSION);
  ...
}
```