Safety RAM: Protecting memory section with checksum

This particular blog post is not going to be about Linux memory managements and its safety or how to write a safety critical software but will touch on the topic that all safety critical software must address properly. This topic is Freedom From Interference (FFI). Before we dive in, let's define some terms and what do they mean in reality:

Term	Meaning	What does it mean really?
ISO26262	Automotive standard for functional safety	This gives you some guidance what you shall be doing to make safety critical software. At least in automotive. If you are care mainly about software then Part 6 of this standard is the most important.
QM	Quality Managed software	Not a safety critical software. It can (and will) go nuts and start writing to your memory or halt the CPU.
ASIL A-D	Automotive Software Integrity Level	Safety critical software for automotive. This software can also go nuts. But the probability of this happening is limited. How much limited depends on the integrity level where ASIL D is the highest integrity.

The Part 6 of ISO26262 defines the freedom from interference as absence of cascading failures between software components. In another words how to ensure that safety critical component runs as designed without giving a chance to others (less critical component) to impact the execution. There are in general 3 types of of interference: spatial (memory), temporal (scheduling) and communication (corruption of input and output data). And as the title suggests here we'll deal in detail only with one particular hardware independent mechanism of memory protection.

But in general there are several more mechanisms one can think of when it comes to spatial freedom of interference:

1. protecting a memory region with a checksum
2. protecting each variable by dual storage of its complement
3. memory protection mechanisms implemented in operating system, base software or hardware layer

As said, this post is mainly about (1) - protecting a given memory section of a process with checksum. This will be illustrated on an example for Linux operating system but essentially (as you will see later) it is more suitable for more bare-metal applications. In all environments it shall be assumed that any other process with lower ASIL (or QM) than desired integrity level of your component can (and will) execute wrongly and will (if it has a chance) change your memory. Often we cannot stop this from happening but we can at least ensure detecting it in timely manner and take corrective actions.

But without further ado let's go through the example. All the source code shown here (and more) can be found on GitHub here.

This example has a particular goal of protecting a dedicated memory section of a process with a checksum. This shall allow us to detect any intrusions and reset the process in case the memory corruption happens. There is however one important assumption here: no preemption can happen in the safety critical section of the code and no other code can execute in parallel (single core CPU). This section is represented by this function:

th_safram_crc.c - Safety critical section

```
void periodic_task(char **argv, uint8_t count)
{
        /* Start of safety critical section */
        uint8_t crc;
        if (safram_crc_check()) {
                restart(argv, count);
        }
        testarray[0]++;
        crc = safram_crc_protect();
        /* End of safety critical section */

        printf("testarray[0]=%u, checksum=%u\n", testarray[0], crc);
        sleep(1);
}
```

Also please note that the stack (or any other memory section besides ".data_safram" in fact) is not protected. So if this were running on a bare-metal with simple preemptive scheduling then no spatial freedom of interference would be achieved as any other process could alter the memory during execution of safety critical section (e.g. imagine other process changing the memory just after the checksum is checked) and no mechanism would detect that (actually we would even store a new checksum at the end of the run calculated over wrong data and keep going).

But let's back up a little - the first step is to define the memory section in a linker script . In our example we'll call this section ".data_safram":

data_safram.ld - Linker script - define custom memory section

```
SECTIONS
{
    .data_safram : {
        _DATA_SAFRAM_START = .;
        *(.data_safram);
        . = . + 1;
        _DATA_SAFRAM_END = .;
    }
}
INSERT AFTER .data
```

Note that the ". = . +1" added one byte at the end of this section. This is where we'll be storing our checksum. This linker script then can be then passed to linker using "-T" option (if using gcc):

Makefile - Linker option

```
gcc --std=c99 -Wall safram_crc.o crc8.o -o th_safram_crc th_safram_crc.c -Wl,-v -T data_safram.ld
```

Note that this way the default linker script is used and only change is a newly added memory section. Then we can start putting variables to this section:

th_safram_crc.c - Safety critical variables

```
__attribute__((section(".data_safram"))) uint8_t testarray[] = {0x33, 0x22, 0x55, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE,
0xFF};
```

To check if our memory section is correctly defined, we can run:

readelf - Memory sections

```
> readelf -a -W th_safram_crc
...
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000400238 000238 00001c 00   A  0   0  1
  ...
  [14] .text             PROGBITS        00000000004005a0 0005a0 000402 00  AX  0   0 16
  [15] .fini             PROGBITS        00000000004009a4 0009a4 000009 00  AX  0   0  4
  [16] .rodata           PROGBITS        00000000004009c0 0009c0 0001f4 00   A  0   0 32
  [17] .eh_frame_hdr     PROGBITS        0000000000400bb4 000bb4 00005c 00   A  0   0  4
  [18] .eh_frame         PROGBITS        0000000000400c10 000c10 000190 00   A  0   0  8
  [19] .init_array       INIT_ARRAY      0000000000600e00 000e00 000008 08  WA  0   0  8
  [20] .fini_array       FINI_ARRAY      0000000000600e08 000e08 000008 08  WA  0   0  8
  [21] .dynamic          DYNAMIC         0000000000600e10 000e10 0001e0 10  WA  7   0  8
  [22] .got              PROGBITS        0000000000600ff0 000ff0 000010 08  WA  0   0  8
  [23] .got.plt          PROGBITS        0000000000601000 001000 000048 08  WA  0   0  8
  [24] .data             PROGBITS        0000000000601048 001048 000010 00  WA  0   0  8
  [25] .data_safram      PROGBITS        0000000000601058 001058 00000a 00  WA  0   0  8
  [26] .bss              NOBITS          0000000000601062 001062 000006 00  WA  0   0  1
  ...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

...
```

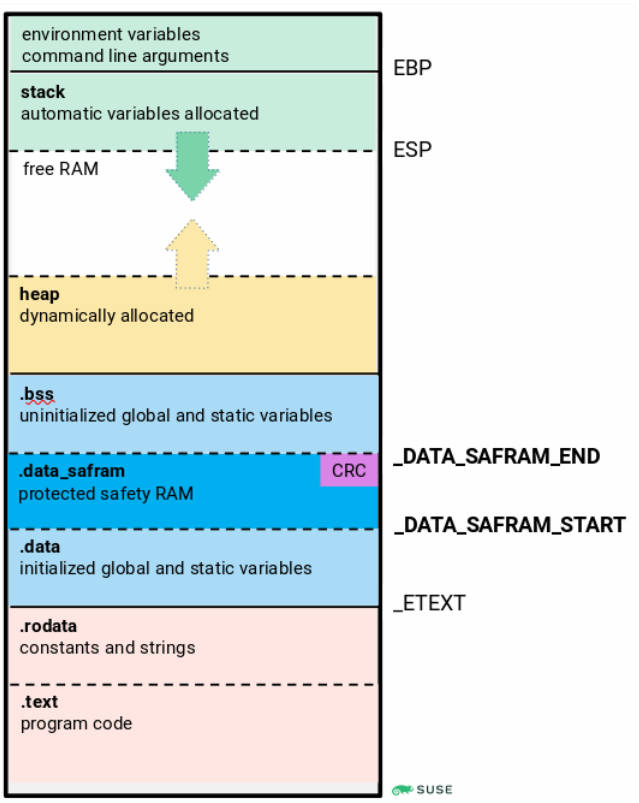
Linker will supply addresses we can use to know where our ".data_safram" memory section starts and ends. In the code these are following "extern" variables:

safram_crc.h - Start and end address of the safety RAM

```
extern uint32_t _DATA_SAFRAM_START;
extern uint32_t _DATA_SAFRAM_END;

#define PTR_UINT8_SAFRAM_START (uint8_t*)&_DATA_SAFRAM_START
#define PTR_UINT8_SAFRAM_END   (uint8_t*)&_DATA_SAFRAM_END
```

To illustrate, the memory layout of our application looks like as shown on the following figure. This picture is inspired by this.



And with all that in place, we can finally implement the functions to store (protect) and check the checksum. The checksum used is the J1850 CRC-8 as described in this AUTOSAR specification of CRC routines.
The CRC approach is fine here as this intended against random faults, not a malicious attacks against our software. For large datasets one may however want to replace CRC-8 by CRC-16 or CRC-32.

safram_crc.c - Function to protect and check the safety RAM

```
uint8_t safram_crc_protect(void)
{
        uint8_t *safram_start = PTR_UINT8_SAFRAM_START;
        uint8_t *safram_end = PTR_UINT8_SAFRAM_END;
        uint8_t crc;

    /* calculate CRC - excluding checksum itself */
        crc = crc8(safram_start,(uint32_t) (safram_end - safram_start - 1) , 0xFF);
        crc = crc ^ 0xFF;

        /* last byte in .data_safram memory section is the checksum */
        *safram_end = crc;
        return crc;

}

uint8_t safram_crc_check(void)
{
    uint8_t *safram_start = PTR_UINT8_SAFRAM_START;
        uint8_t *safram_end = PTR_UINT8_SAFRAM_END;
        uint8_t crc;

        crc = crc8(safram_start,(uint32_t) (safram_end - safram_start - 1) , 0xFF);
        crc = crc ^ 0xFF;

        if (*safram_end != crc)
        {
                /* Memory is corrupted */
                return 1;
        }
        else
        {
                /* All good, checksum matches */
                return 0;
        }
}
```

These can be then used at the beginning and end of our safety critical section - please see above. Then comes a question what to do in case a memory corruption is detected. In simple hardware this would be a reset, in our example we will just restart the process hoping that everything will come back to order:

th_safram_crc.c - Restart process in Linux

```
void restart(char **argv, uint8_t count)
{
        char buffer[32] = {};
    char *nargv[] = {argv[0], buffer, 0};

    snprintf(buffer, sizeof(buffer), "%d",count + 1);
    execv("/proc/self/exe", nargv);
    printf("This should never be reached!");
}
```

This piece of code is inspired by this answer.

So let's test all that. For that we will also need a way to corrupt the data. Luckily in Linux this can be done rather easily by altering the "/proc/$pid/mem" with the right privileges:

th_safram_corruption.c - Accessing other process memory in Linux

```
uint8_t *data = malloc(length);

lseek(fd_proc_mem, start_address, SEEK_SET);
read(fd_proc_mem, data, length);

printf("Data at 0x%x in process %d is:", start_address, pid);
for (uint16_t i = 0; i < length; i++ ) {
        if (!(i % 10)) {
            printf("\n");
        }
        printf("  %X\n", data[i]);
}

printf("Adding one to the first byte!\n");
data[0]++;

lseek(fd_proc_mem, start_address, SEEK_SET);
if (write (fd_proc_mem, data, length) == -1) {
        printf("Error while writing\n");
        exit(1);
}

free(data);
```

This piece of code is heavily inspired by this.

Here you can see the results - the "..." lines are just an attempt to show the time alignment.

Process 1 - ASIL	Process 2 - QM

th_safram_crc

```
> ./th_safram_crc
This is an example of protecting RAM area with checksum! Run
#0!
pid=22808, safram start_address=0x601058, length=9,
checksum=203
ptrace restrictions are effectively disabled
testarray[0]=34, checksum=237
testarray[0]=35, checksum=136
testarray[0]=36, checksum=39
testarray[0]=37, checksum=66
testarray[0]=38, checksum=107
testarray[0]=39, checksum=14
testarray[0]=3A, checksum=161
This is an example of protecting RAM area with checksum! Run
#1!
pid=22808, safram start_address=0x601058, length=9,
checksum=203
testarray[0]=34, checksum=237
testarray[0]=35, checksum=136
testarray[0]=36, checksum=39
...
...
...
testarray[0]=37, checksum=66
testarray[0]=38, checksum=107
This is an example of protecting RAM area with checksum! Run
#2!
pid=22808, safram start_address=0x601058, length=9,
checksum=203
testarray[0]=52, checksum=237
testarray[0]=53, checksum=136
testarray[0]=54, checksum=39
testarray[0]=55, checksum=66
testarray[0]=56, checksum=107
testarray[0]=57, checksum=14
```

th_safram_corruption

```
...
...
...
...
...
...
...
...
...
> ./th_safram_corruption 22808 0x601058 4
Opening /proc/22808/mem, address is
0x601058
Data at 0x601058 in process 22808 is:
  3A
  22
  55
  AA
Adding one to the first byte!
...
...
> ./th_safram_corruption 22808 0x601058 4
Opening /proc/22808/mem, address is
0x601058
Data at 0x601058 in process 22808 is:
  38
  22
  55
  AA
Adding one to the first byte!
```

You can see that the process 1 is restarted whenever we corrupt the memory from process 2. If you don't then you might be prevented from writing to the other process memory by the way your Linux distribution is configured. But that's a topic for the future postings. This example shall however run fine on openSUSE Leap 15.

So it works but to be clear the process 1 here is not ASIL at all. Firstly I have not followed any coding practices required by ISO26262 but even if I did the assumption about preemption is not met here anyways. So please take this as an example that wrong (not suitable) technical concepts do not lead to safe software even if the software is otherwise completely ISO26262 compliant. On top of that we are running on non-safety Linux operating system so our process reset mechanism cannot be relied on and nothing ensures that our checking gets even executed. So no ASIL here nowhere near, sorry.

To sum up this approach comes handy if we need to protect global static variables in between of execution of function runs (e.g. states) and we can ensure that safety critical sections will not be preempted by less safety critical software components. The advantages of this approach is that it is mainly about linker and the other code needs to be altered very little and it also doesn't need much additional memory or computing power. The disadvantage is that often we want to or must allow the preemption and parallelism. In this case we must use other approaches. Please stay tuned if you wonder about dual-storage of a complement or what protection mechanism are implemented in Linux's memory management.

And as said, all the code is here: https://github.com/llansky3/safram in the “crc” folder.

Update (2022/04/12): There is now follow up blog post: Safety RAM: Protecting variables with one’s complement. And especially the section “Memory corruption problems” is a direct follow up on this blog post.