**Selected Topics of Embedded Software Development 2**

**WS-2021/22**

**Prof. Dr. Martin Schramm**

# Testing and generating Prime Numbers and Safe Primes using CryptoCore

**Group 2 – Team 4**
**Rashed Al-Lahaseh – 00821573**
**Vikas Gunti - 12100861**

**Supriya Kajla – 12100592**

**Srijith Krishnan – 22107597**

**Wannakuwa Nadeesh – 22109097**

# CryptoCore Implementation

## cryptocore_ioctl_header.h

The header file with the IOCtl (an abbreviation of input/output control) definitions where the declarations here have to be in a header file, because they need to be known both the kernel module in *_driver.c and the application *_app.c

```
// Add CryptoCore Struct Declarations here...
typedef struct MRT_prime_params_t{
    __u32 prec;         // Precision
    __u32 s;
    __u32 d[128];
    __u32 sec_calc;     // Secure calculation bit
    __u32 n[128];       // User input
    __u32 k;            // Number of iterations (sensitivity parameter)
    __u32 probably_prime;     // Result as boolean flag (1|0)
} MRT_prime_params_t ;
```

*Figure 1: CryptoCore Struct Declarations*

```
// Define further IOCTL commands here...
#define     IOCTL_MWMAC_MRT_Prime           _IOWR(IOCTL_BASE, 20, MRT_prime_params_t)
```

*Figure 2: Define IOCTL command*

## cryptocore_driver.c

We start by defining our function

```c
// Add further Function Prototypes here...
static void MWMAC_MRT_Prime(MRT_prime_params_t *MRT_prime_params_ptr);
```

*Figure 3: MRT function protocol*

Then define the structure inside IOCtl driver

```c
static long cryptocore_driver_ioctl( struct file *instance, unsigned int cmd, unsigned long arg)
{
  // Add CryptoCore Structs here and allocate kernel memory...
  MRT_prime_params_t *MRT_prime_params_ptr = kmalloc(sizeof(MRT_prime_params_t), GFP_DMA);
```

*Figure 4: Allocate MRT to kernel memory*

```c
  // Add further CryptoCore commands here
  case IOCTL_MWMAC_MRT_Prime:
    rc = copy_from_user(MRT_prime_params_ptr, (void *)arg, sizeof(MRT_prime_params_t));
    MWMAC_MRT_Prime(MRT_prime_params_ptr);
    rc = copy_to_user((void *)arg, MRT_prime_params_ptr, sizeof(MRT_prime_params_t));
    break;
```

*Figure 5: Define command*

Then free the kernel memory

```c
  default:
    printk("unknown IOCTL 0x%x\n", cmd);

    // Free allocated kernel memory for defined CryptoCore Structs here...
    kfree(MRT_prime_params_ptr);
```

*Figure 6: Free kernel memory*

```c
  // Free allocated kernel memory for defined CryptoCore Structs here...
  kfree(MRT_prime_params_ptr);

  return 0;
```

*Figure 7: Free kernel memory*

Then we are going to write the functionality of our MRT function

Starting by initializing variables

```
// Add further CryptoCore Functions here...
static void MWMAC_MRT_Prime(MRT_prime_params_t *MRT_prime_params_ptr)
{
    // Temporary counter
    u32 i;
    // User input (number to test)
    u32 n[128];
    // Precision
    u32 prec = MRT_prime_params_ptr->prec;
    // Word count
    u32 word_count = prec/32;
    // Index of last word (least significant word)
    u32 last_i = word_count-1;
    // D parameter
    u32 d[128];
    // S parameter
    u32 s = 0;
    // Flag to check if least sig block is equal to 2
    u32 last_i_is_2;
    // Flag to check if other blocks are equal to 0
    u32 blocks;

    // Clear RAM
    Clear_MWMAC_RAM();

    // Copy input number into n and d
    for(i=0; i<word_count; i++) {
        n[i] = MRT_prime_params_ptr->n[i];
        d[i] = MRT_prime_params_ptr->n[i];
    }
```

Figure 8: MRT Function

Then we check the input number if it is equal to 2 return prime or if it is even to return composite

```
// ---------------------------------------------------------------------------
// Preliminary tests on the number [(n == 2) then prime or (n % 2 == 0) then composite]
// ---------------------------------------------------------------------------

// If number is equal to 2 then it is a prime (check least sig block to be 2)
last_i_is_2 = 0;
if(n[last_i]==2) {
   last_i_is_2 = 1;
} else {
   last_i_is_2 = 0;
}

// Check other blocks to be 0
blocks = 1;
for(i=0; i<last_i; i++) {
   if(n[i]!=0) {
      blocks = 0;
      break;
   }
}

// Make sure that least significant bit is 1 (final check)
if(last_i_is_2==1 && blocks==1) {
   MRT_prime_params_ptr->probably_prime = 1;
   return;
}

// If number is even then it is composite
if(!(n[last_i] & 0x00000001)) {
   MRT_prime_params_ptr->probably_prime = 0;
   return;
}
```

*Figure 9: MRT Function*

Then we calculate the value of s and d

```
// ---------------------------------------------------------------------------
// Calculate value of d and s
// ---------------------------------------------------------------------------

// Do minus 1 operation:
//  Since we know received random number is odd we can confidently only AND it
//  with 0xFFFFFFFE to make the right most digit 0 which means number-1
d[last_i] &= 0xFFFFFFFE;

// Shift operation:
// While(least significant block %2==0)
while(!(d[last_i] & 0x00000001)) {
    s++; // Counting the number of shifting
    d[last_i] >>= 1; // Divide by 2 (shift least significant block)
    // Shift other blocks starting from one block after the least significant block
    // [B0 B1 B2 B3 <- B4]   [most sig, ..., least sig]
    for(i=1; i<word_count; i++) {
        // First check if the right most bit of current block is 1.
        // If yes, before shifting, we insert a 1 into the previous (less significant) block from its left side
        if(d[last_i-i] & 0x00000001) // If(d%2==1)   (if right most bit of current part is 1)
            d[last_i-(i-1)] |= 0x80000000; // Sets most left bit of previous block to 1
        d[last_i-i] >>= 1; // Divides current block by 2
    }

    // After finding the value of d & s we are copying them to the structure
    for(i=0; i<word_count; i++) {
        MRT_prime_params_ptr->d[i] = d[i];
    }
    // value of s and d is obtained [ (2^s) * d = (n-1) ]
    MRT_prime_params_ptr->s = s;
}
```

*Figure 10: MRT Function+*