浙江大学

基于米勒拉宾算法的 RSA 密码系统

课程名称：	微控制器原理、接口与应用
组　　号：	第八组
姓　　名：	曹逸芃
学　　院：	信息与电子工程学院
系：	信息与电子工程
专　　业：	微电子科学与工程
学　　号：	3180102927
指导教师：	沈海斌 罗小华

2020 年 12 月 27 日

一、微控制器概述

1. 微控制器选择

经过筛选与调整，我们最终选择了 NXP 推出的 LPC2114 型微处理器（如图 1.1），主要原因在于该微处理器较为简洁，并且可以满足我们 RSA 加密的需要，使用起来比较方便。此外，该系列微处理器可以在网上找到较多的样例代码资源，对于我们的编程实践有很大帮助。

2. LPC2114 简介

LPC2114 基于一个支持实时仿真和跟踪的 32 位 ARM7TDMI-S 微处理器，并带有 128kB 嵌入的高速 Flash 存储器。128 位宽度的存储器接口和独特的加速结构使 32 位代码能够在最大时钟速率下运行。对代码规模有严格控制的应用可使用 16 位 Thumb 模式将代码规模降低超过 30%，而性能的损失却很小。由于 LPC2114 非常小的 64 脚封装、极低的功耗、多个 32 位定时器、4 路 10 位 ADC、PWM 输出、46 个 GPIO 以及多达 9 个外部中断使它们特别适用于工业控制、医疗系统、访问控制和电子收款机(POS)。由于内置了宽范围的串行通信接口，它们也非常适合于通信网关、协议转换器、嵌入式软件调制解调器以及其它各种类型的应用。

LPC2114 特性：

*32 位 ARM7TDMI-S 核，超小 LQFP64 封装；

*16kB 片内 SRAM；

*128/256kB 片内 Flash 程序存储器，128 位宽度接口/加速器可实现高达 60*MHz 工作频率；

*通过片内 boot 装载程序实现在系统编程(ISP)和在应用编程(IAP)。

*嵌入式跟踪宏单元(ETM)支持对执行代码进行无干扰的高速实时跟踪；

*4 路 10 位 A/D 转换器，转换时间低至 2.44μs；

*2 个 32 位定时器(带 4 路捕获和 4 路比较通道)、PWM 单元(6 路输出)、实时时钟和看门狗；

*多个串行接口，包括 2 个 16C550 工业标准 UART、高速 I2C 接口(400kHz)和 2 个 SPI 接口；

*通过片内 PLL 可实现最大为 60MHz 的*CPU 操作频率；

*向量中断控制器。可配置优先级和向量地址；

*多达 46 个通用 I/O 口(可承受 5V 电压),9 个边沿或电平触发的外部中断引脚；

*片内晶振频率范围：10~25MHz；

*2 个低功耗模式：空闲和掉电；

*通过外部中断将处理器从掉电模式中唤醒；

*双电源

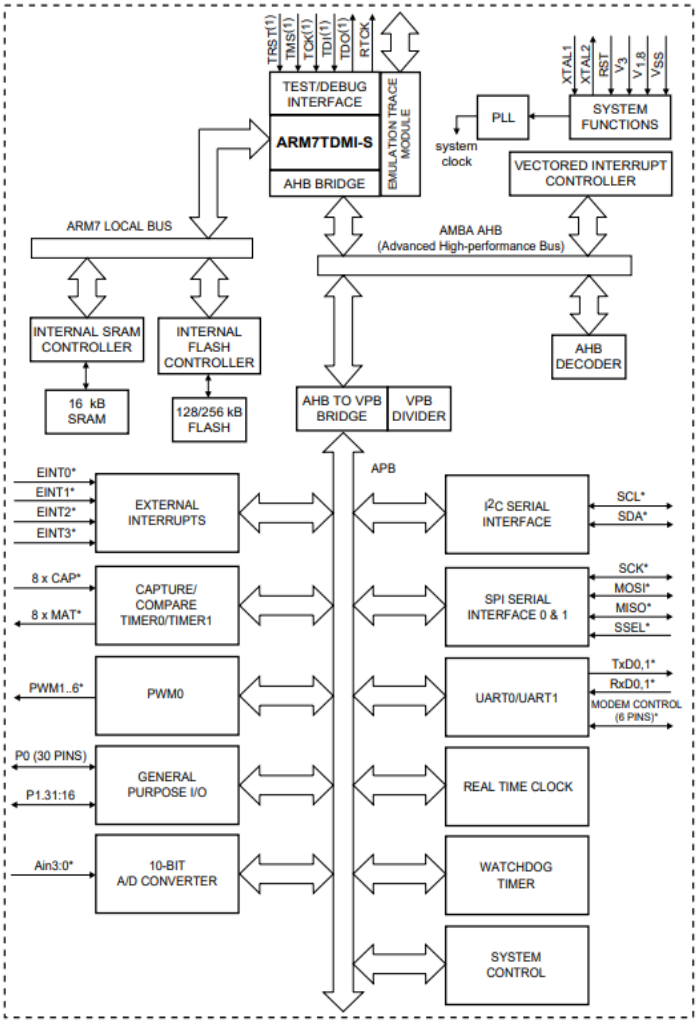

图 1.1 LPC2114 示意图

二、开发平台

1. 平台选择

1）Keil5

Keil C51（如图 2.1）是美国 Keil Software 公司出品的 51 系列兼容单片

机 C 语言软件开发系统，与汇编相比，C 语言在功能上、结构性、可读性、可维护性上有明显的优势，因而易学易用。Keil 提供了包括 C 编译器、宏汇编、链接器、库管理和一个功能强大的仿真调试器等在内的完整开发方案，通过一个集成开发环境（uVision）将这些部分组合在一起。

根据我们了解，Keil5 是现在 ARM 开发最为常用的平台之一，也是一种相对先进的平台。安装了 MDK-ARM 包后，Keil 可以支持很多厂商的芯片，支持 ARM7，ARM9，Cortex-M4/M3/M1，Cortex-RO/R3/R4 等 ARM 微控制器内核。非常适用于微控制器开发以及 MCU 开发。我们选择 Keil5 作为软件开发平台主要因为 Keil5 有如下一些优点：

*支持大量的器件包，可以根据需要自主选择安装。

*包含丰富的例程代码还有最重要的启动文件。

*界面简单，使用方便。

*有大量的教程资源供初学者参考。

*支持 C 文件的编译。

*更加适用于 MCU 应用开发。

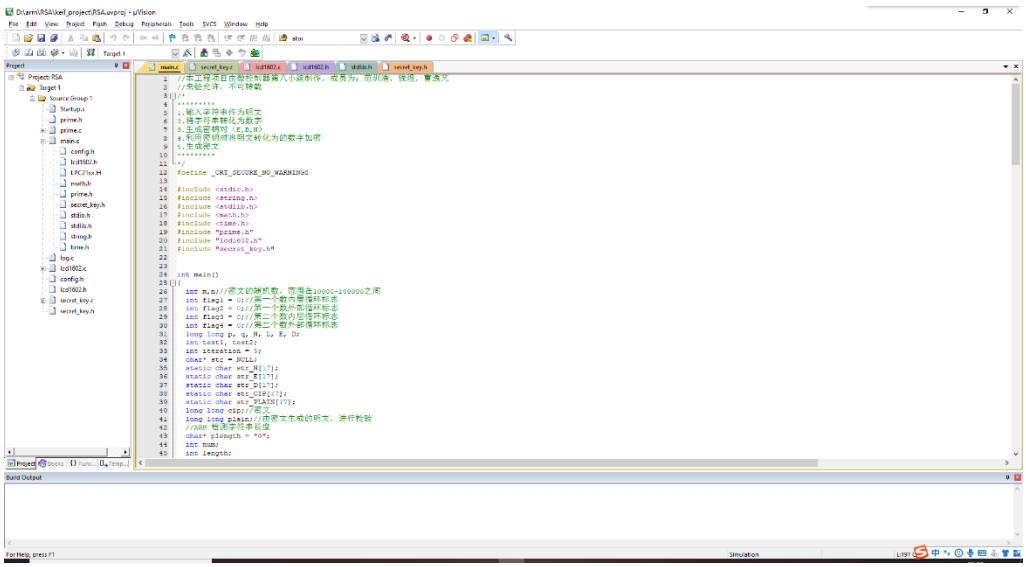

图 2.1 Keil C51 界面

2）Proteus8

在我们的项目中，除了进行软件开发和基于 Keil5 的调试仿真之外，还要进行微控制器与外部器件的协同仿真，因此除了 Keil5 平台，我们还需要 Proteus8 EDA 工具（如图 2.2）。

Proteus 是英国 Lab Center Electronics 公司的 EDA 工具(仿真软件)，从原理图布图、代码调试到单片机与外围电路协同仿真，一键切换到 PCB 设计，真正实现了从概念到产品的完整设计。是世界上唯一将电路仿真软件、PCB 设计软件和虚拟模型仿真软件三合一的设计平台，其处理器模型支持 8051、HC11、PIC10/12/16/18/24/30/DSPIC33、AVR、ARM、8086 和 MSP430 等，2010 年又增加了 Cortex 和 DSP 系列处理器，并持续增加其他系列处理器模型。在编译方面，它也支持 IAR、Keil 和 MATLAB 等多种编译器。Proteus8 可以很好的与 Keil5 进行联合调试，在我们进行 RSA 密码系统开发时，可以起到非常好的效果。

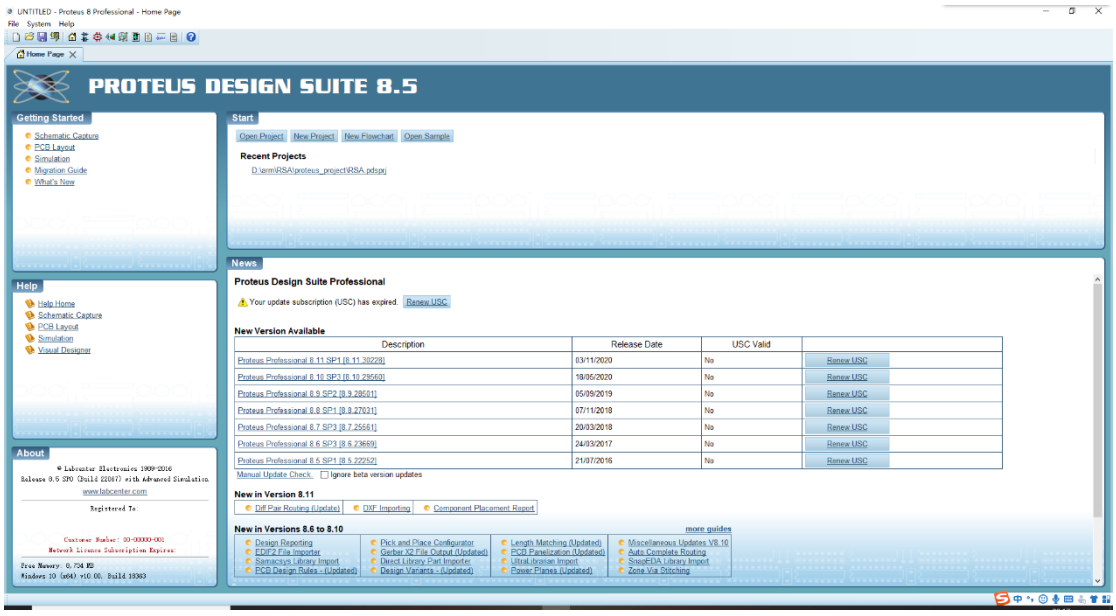

图 2.2 Proteus8 界面

2. 其他开发平台对比

除了 Keil5 之外，我们还比较了 DS-5 开发平台。DS-5 是一款支持开发所有 ARM 内核芯片的集成开发环境。提供具有跟踪、系统范围性能分析器、实时系统模拟器和编译器的应用程序和内核空间调试器。这些功能包括在定制、功能强大且用户友好的基于 Eclipse 的 IDE 中。借助于该工具套件，可以很轻松地为 ARM 支持的系统开发和优化基于 Linux 的系统，缩短开发和测试周期，并且可帮助工程师创建资源利用效率高的软件。

DS-5 主要有以下特点：

*定制的 Eclipse IDE，与第三方插件兼容

*功能强大的 C/C++ 编辑器和项目管理器

*为 ARM Linux 进行了验证的 GNU 编译工具

*集成的生产力实用工具，例如远程系统浏览器、SSH 和 Telnet 终端等

*在主机上的 Linux 应用程序调试

*预先与 ARM 嵌入式 Linux 一起加载的 Cortex-A8 系统模型模拟器

*高于 250 MHz 的典型模拟速度

但是 DS-5 相比于 Keil5，并不特别适用于我们的 MCU 开发。而且，DS-5 的使用比较复杂，而且能找到的教程有限，并不适合初学者使用。而 Keil5 与 Proteus8 的联合调试则更加适用于我们的项目开发，因此我们最终选择了 Keil5 与 Proteus8。

三、算法原理

1. RSA 加密

在公开密钥密码体制中，加密密钥（即公开密钥）PK 是公开信息，而解密密钥（即秘密密钥）SK 是需要保密的。加密算法 E 和解密算法 D 也都是公开的。虽然解密密钥 SK 是由公开密钥 PK 决定的，但却不能根据 PK 计算出 SK。

RSA 算法是先生成一对 RSA 密钥，其中之一是保密密钥，由用户保存；另一个为公开密钥，可对外公开，甚至可在网络服务器中注册。为提高保密强度，RSA 密钥至少为 500 位长，一般推荐使用 1024 位。这就使加密的计算量很大。为减少计算量，在传送信息时，常采用传统加密方法与公开密钥加密方法相结合的方式，即信息采用改进的 DES 或 IDEA 对话密钥加密，然后使用 RSA 密钥加密对话密钥和信息摘要。对方收到信息后，用不同的密钥解密并可核对信息摘要。

RSA 公开密钥密码体制的原理是：根据数论，寻求两个大素数比较简单，而将它们的乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥。

RSA 算法的具体实现方式可以描述为：

①任意选取两个不同的大素数 p 和 q 计算乘积：

$$n = pq，\ \varphi(n) = (p-1)(q-1)$$

②任意选取一个大整数 e，满足：

$$\gcd\left(e，\ \varphi(\mathrm{n})\right) = 1$$

整数 e 用作加密钥。

③确定解密钥 d，满足：

$$(de) mod \varphi(n) = 1$$

④公开整数 n 和 e，秘密保存 d。

⑤将明文 m 加密成密文 c，加密算法为：

$$c = E(m) = m^e mod n$$

⑥将密文 c 解密成明文 m，解密算法为：

$$m = D(c) = c^d mod n$$

出于复杂度与效果展示的考虑，我们缩小了素数的大小以及公钥私钥的长度，但是 RSA 密码系统的原理完全相同，并且可以很好地做到加密与解密。

2. 米勒拉宾算法

RSA 算法的实现离不开素数的生成与判断，在我们的算法中，我们采用随机数生成的方法来生成一个数，之后再判断其是否为素数。素数判断的常见方法是枚举法，即让 n 依次除以 2 到 sqrt(n)以内的整数，如果有除尽的情况，则不是素数。但是这种算法在对大素数进行判断时耗时较长，并不是一种理想的算法，因此我们采用米勒拉宾算法来进行素数判断。

米勒拉宾算法是一种基于费马小定理的算法，它对于素数的判断并非 100%准确，但是其准确率对于我们的 RSA 算法而言完全足够，而且米勒拉宾算法可以大大简化素数判断的复杂度和资源消耗，是一种非常有效的算法。

米勒拉宾算法的具体原理如下：

假设 n 是一个奇素数，且 n 大于 2。故 n-1 是一个偶数，可以被表示成$2^s \times d$的形式，s 和 d 都是正整数且 d 是奇数。对任意在$(Z/nZ)^*$范围内的 a 和$0 \le r \le s-1$，必然满足以下两种形式中的一种：

$$a^d \equiv 1(mod n)$$

$$a^{2^r d} \equiv -1(mod n)$$

由于费马小定理，对于一个素数 n，有：

$$a^{n-1} \equiv 1(mod n)$$

如果我们能找到一个这样的 a，使得对任意$0 \le r \le s-1$以下两个式子均满

足：

$$a^d \not\equiv 1(mod n)$$

$$a^{2^r d} \not\equiv -1(mod n)$$

那么 n 就不是一个素数。这样的 a 称为 n 是合数的一个凭证。否则 a 可能是一个证明 n 是素数的强伪证，即当 n 确实是一个合数，但是对当前选取的 a 来说上述两个式子均不满足，这时我们认为 n 是基于 a 的大概率素数。

米勒拉宾素数判断法的算法复杂度约为$O(\log^2(n))$，而枚举法的算法复杂度约为$O(\sqrt{n})$。米勒拉宾素数判断法在进行大素数判断时明显更优。

四、设计与调试技巧

1. 前期设计技巧

项目前期，考虑到阅读的方便性、使用的熟练度和编译器的成熟度，我们小组主要利用 Visual Studio 作为开发平台，编写基于 C 语言的项目工程。在该项目设计过程中，我们主要利用了以下技巧。

1）“自下而上”设计模式

在设计初期，由于对于代码量和具体功能覆盖程度难以做到完全预测，我们小组联系《硬件描述语言原理与应用》中所学到的知识，利用 “自下而上”的

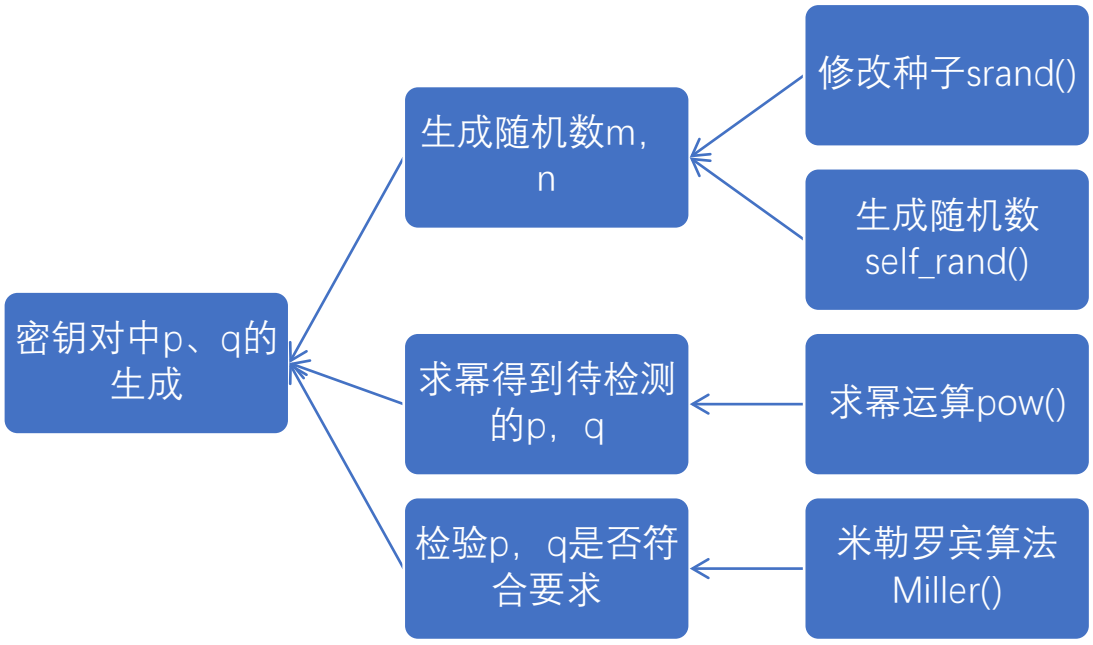

图 4.1 自下而上设计实例

设计模式采取以下思路进行设计：

如图 4.1，对于 RSA 加密过程中的 p、q 生成模块部分，由于其条理思路相对较为清晰，我们根据现有资料查找，能够从最终实现的功能入手，逐步推理分析出所需要的底层的各个功能，设计出对应的函数。通过从底层开始设计，一步步合并底层内容，最终实现总的目标。

我们小组选择这种方法的理由是：

a. 从底层开始实现，能够更有效地对模块的可实现性有所把控。如果某个部分需要大量代码，我们能更快发现，并针对该情况作出有效调整；

b. 便于分工。在制定了统一的代码编写规范（见后续）后，小组成员能够并行编写代码，独立调试和优化，提高了工作效率。

2）统一的代码规范和项目文件系统分配模式

代码规范方面，我们小组主要参考了Huawei C&C++ Secure Coding Standard v3.1 版本。例如所有函数都采用驼峰法命名、尽量使用const（这样就可以利用编译器进行类型检查，将代码的权限降到更低）等。

在项目文件系统分配上，将各个功能模块整合成各自的.c 文件存放函数，并创建对应的头文件进行函数声明，增强了项目工程代码的可读性（如图 4.2）。

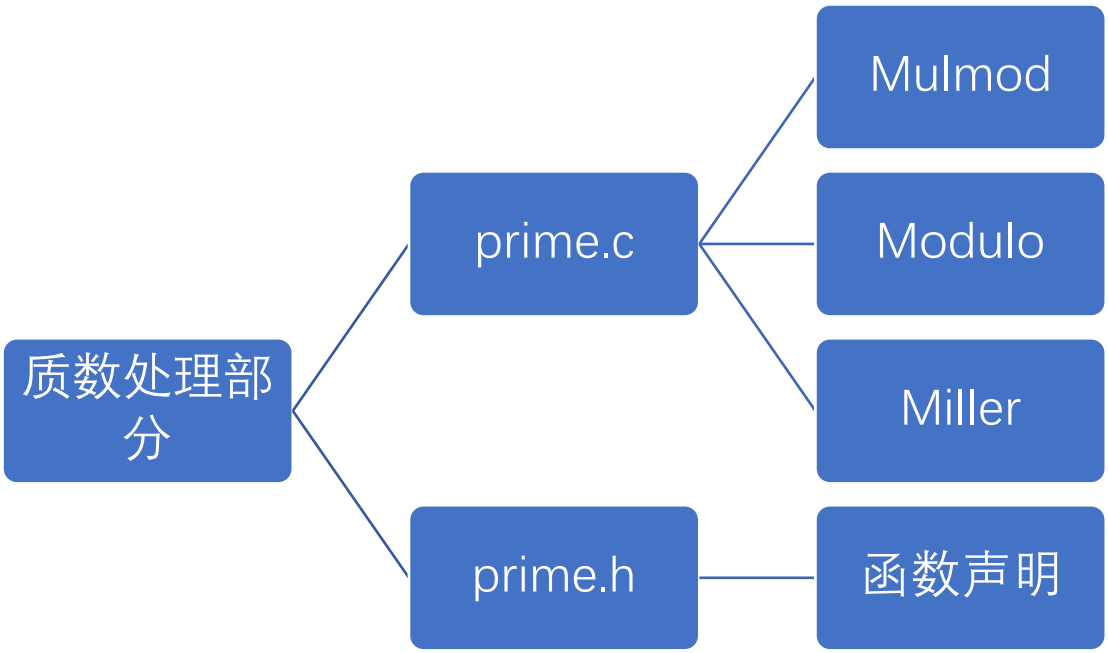

图 4.2 项目代码文件分配举例

3)工程项目版权、日志的管理

我们小组在每个代码文件的开头都添加了版权声明（如图 4.3）。

```
//本工程项目由微控制器第八小组制作
//作者：范圳浩，钱煜，曹逸芃
//联系方式：3180100808@zju.edu.cn，3180103948@zju.edu.cn，3180102927@zzju.edu.cn
//指导老师：沈海斌，罗小华
//隶属机构：浙江大学
//未经允许，不可转载
```

图 4.3 版权和作者信息声明

同时，我们也在每次修改完成后，会更新项目里的 log.c 文件（如图 4.4），重命名项目名称为日期+版本号的形式（如图 4.5、图 4.6），便于其他成员审阅和修改。

```
10
11  11.7 钱煜
12  N创建完成
13
14  ********************************************************************************
15
16  11.8 钱煜
17  创建了prime.c和prime.h，将“判断质数”这一行为用Rabin-Miller算法描述
18
19  ********************************************************************************
20
21  11.9 曹逸芃
22  创建了L,E,D密钥对
23  实现了欧几里得算法，并通过这些算法生成公钥对和私钥对
24
25  ********************************************************************************
26
27  11.11 范圳浩
28  完成了输入数进行加密
29
30  ********************************************************************************
31
32  11.13 范圳浩
33
34  完成了解密过程，对加密和解密算法进行了优化，修改了产生大质数的方法
35
36  ********************************************************************************
37
38  11.14 钱煜
39
40  在大数的幂次求余的问题上，撰写了新的思路：通过数组解决了超大数求余的问题，但余下问题是：如何将大数的幂次表示为数码？
```

图 4.4 log.c 部分截图

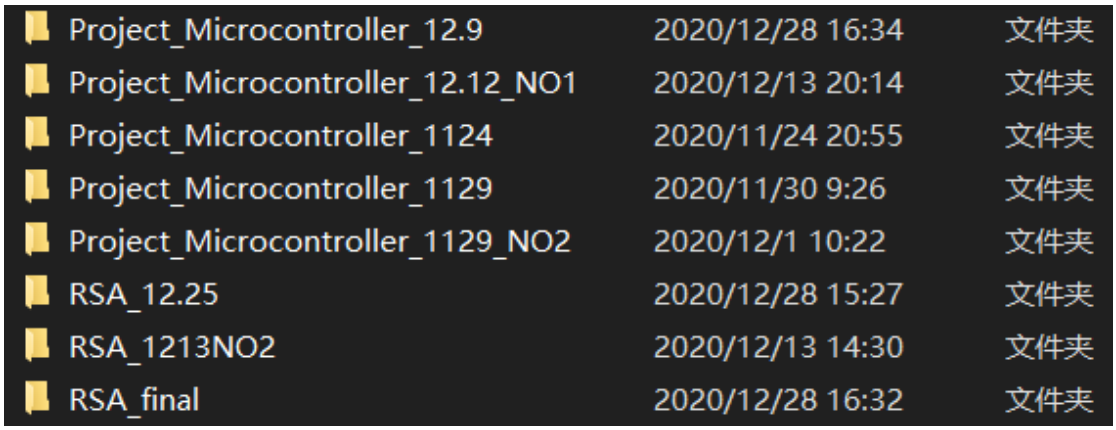

图 4.5 项目文件夹命名格式

图 4.6 钉钉群文件记录部分截图

2. 中后期调试技巧

项目中期，即 C 语言基本完成后的调试技巧，包括 C 代码的调试和 ARM 汇编语言的调试。

1）设置断点和中间变量可视化

在测试 C 代码时，我们小组通过在代码中添加断点，并将某些变量用“监视

器”或者控制台进行输出来监管运行过程是否正确（如图 4.7）。

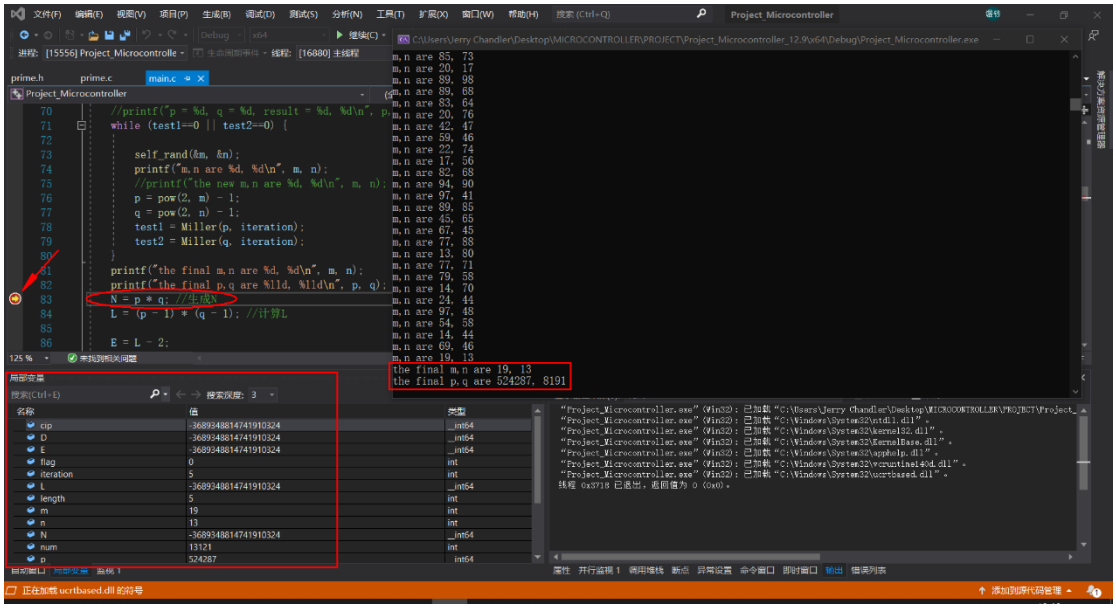

图 4.7 断点和中间变量可视化实例

2)基于 ARM 汇编的 debug 模式调试

在测试 ARM 汇编时，我们将平台迁移至 Keil5，注意此时选择仿真的 debug 选项中选择“use simulator”。此时，可以选择插入断点、逐步运行、查看寄存器内容、查看栈等多种操作（如图 4.8）。

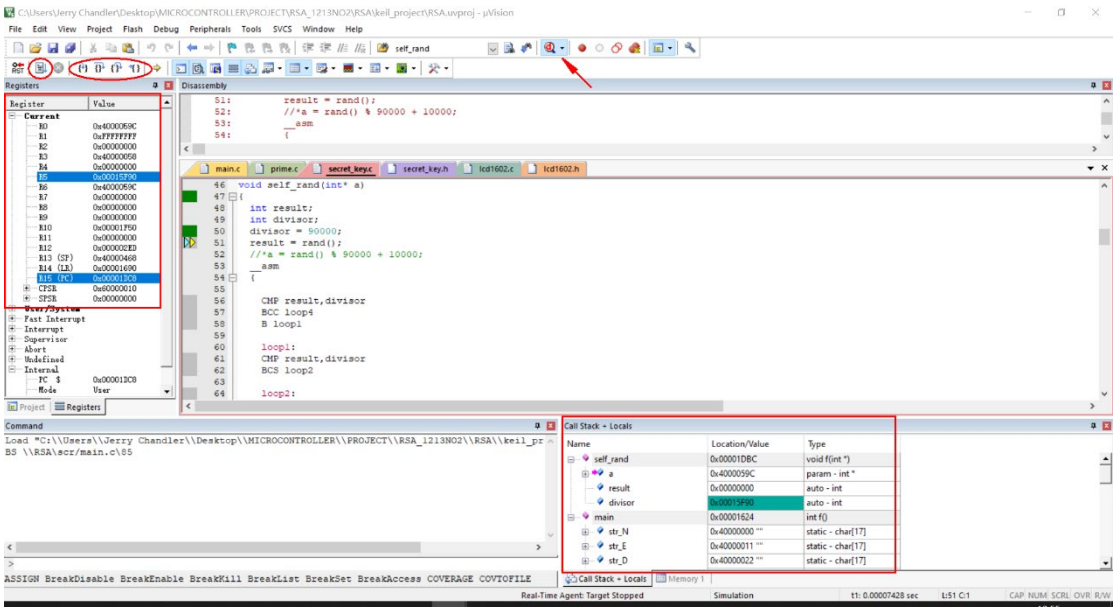

图 4.8 Keil5 调试 ARM 汇编实例

值得一提的是，我们小组专门编写了基于 ARM 的调试函数 My_Strlen，在 Keil 上也顺利得以运行。

3）自行编写不支持的库函数

在从 VS 平台跳到 Keil 平台后，部分函数不支持编译或原来状态下的运行，如 srand、itoa 等。为此，我们小组自行编写了部分函数（如 itoa），对于某些函数（如读取系统时间作为伪随机种子）则采用了常数作为参数的简约化处理。

3. 后期调试技巧

工程项目基本完成后，我们小组利用 Keil 产生的.hex 文件（如图 4.9）进行 Proteus 的仿真。虽然理论上可以进行 Proteus 和 Keil 的联调，但我们并未实现。当 Proteus 出现问题时，我们会回到 Keil 进行调试。

此时需注意将有关外显的代码都注释，否则 Keil 无法编译。

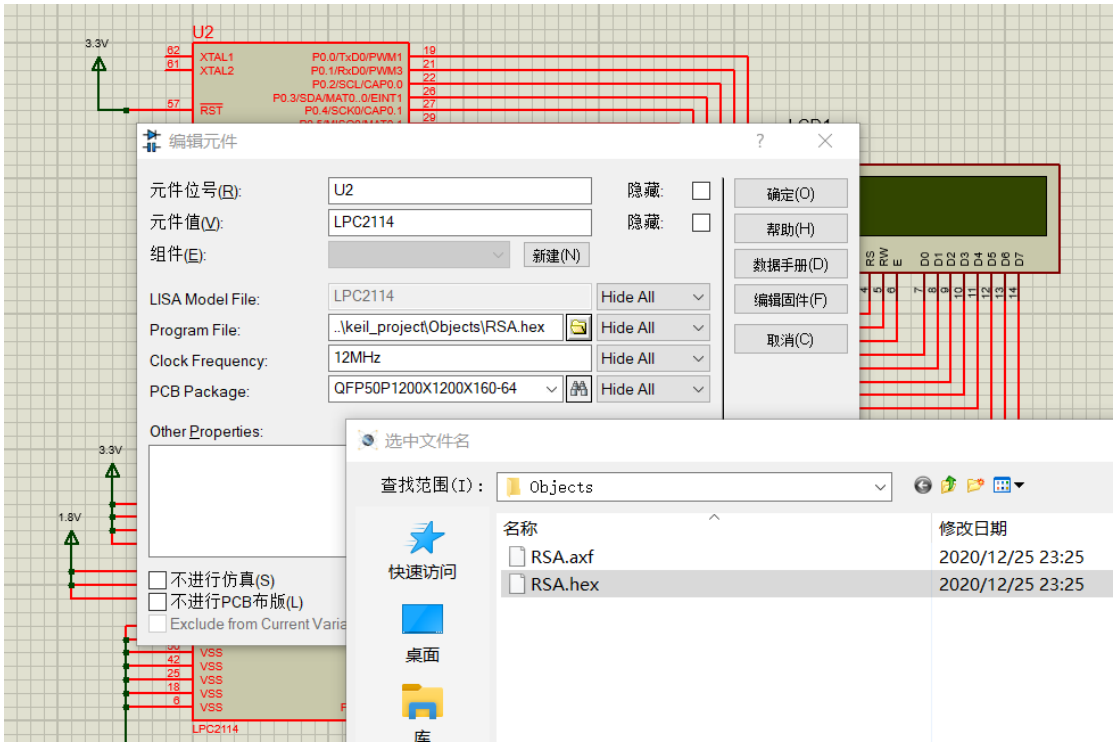

图 4.9 Proteus 利用.hex 运作实例

五、程序实现结果与性能

在该项，我们将通过程序结果展示、C 与汇编程序性能比较及原因分析、与最先进程序对比、代码行数四个方面来阐述。

1. 程序结果展示

我们的程序采用 LCD1602 来显示我们的运行成果，首先依次显示明文、根据算法产生的公钥、私钥，然后显示根据公钥加密后的密文，最后显示利用密钥将密文反解密后的明文，如果反解密后的明文和最初的明文相同则说明我们的 RSA 密码系统的各个过程都是正确的。图 5.1-5.5 是图片展示，程序结果在视频中也有展示。

① 显示明文

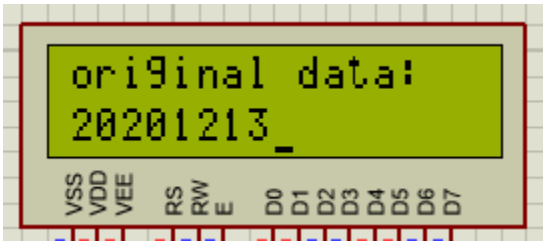

图 5.1 显示明文

②显示产生的公钥

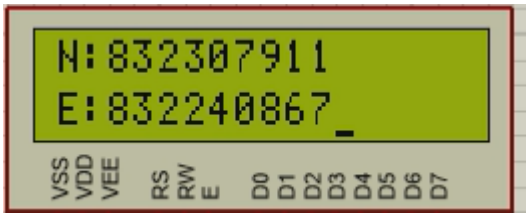

图 5.2 显示公钥

③显示产生的私钥

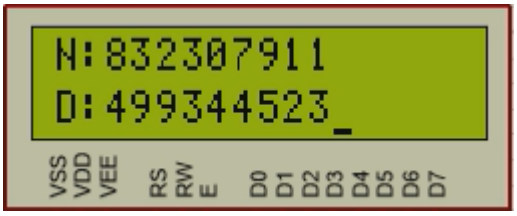

图 5.3 显示私钥

④显示加密后的密文

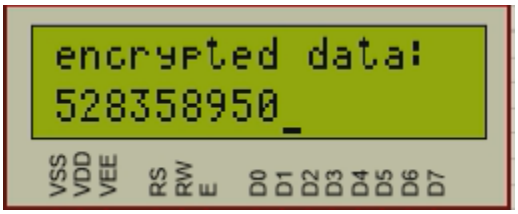

图 5.4 显示密文

⑤显示再将密文解密后的明文

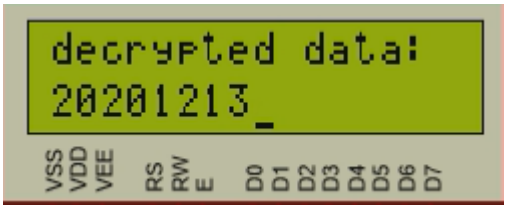

图 5.5 显示解密后明文

由以上 5 个过程可以看出，我们成功的产生了公钥对 N:“832307911”、E:“832240867”和私钥对 N:“832307911”、D:“499344523”，并且将明文“20201213”加密成为“528358950”，最后将加密后的密文反解密为“20201213”，与明文相同，说明我们 RSA 算法系统设计成功。

2. C 与汇编程序性能比较

在我们的 RSA 密码系统中，我们将随机数生成函数、加密函数、解密函数使用汇编程序替代了 C 语言程序，**但是可惜的是，性能并没有提升，下面将详细说明。**

1）随机数函数

代码对比：

① C 语言：

```
void Self_Rand(int* a)
{
    *a = rand() % 90000 + 10000;
}
```

② 汇编：

```
void Self_Rand(int* a)
```

```
{
    int result;
    int divisor;
    divisor = 90000;
    result = rand();
    __asm
    {

        CMP result, divisor
        BCC loop4
        B loop1

        loop1:
        CMP result, divisor
        BCS loop2

        loop2:
        SUBS result, result, divisor
        CMP result, divisor
        BCC loop4
        B loop2

        loop4:
        ADDS result, result, #10000

    }
    *a = result;
}
```

性能对比：

为了显示修改前后的性能对比，我们使用 keil 软件，在调用该函数的前一条语句和后一条语句设置断点，通过运行到两个断点的时间差判断函数的运行速度。

① C 语言

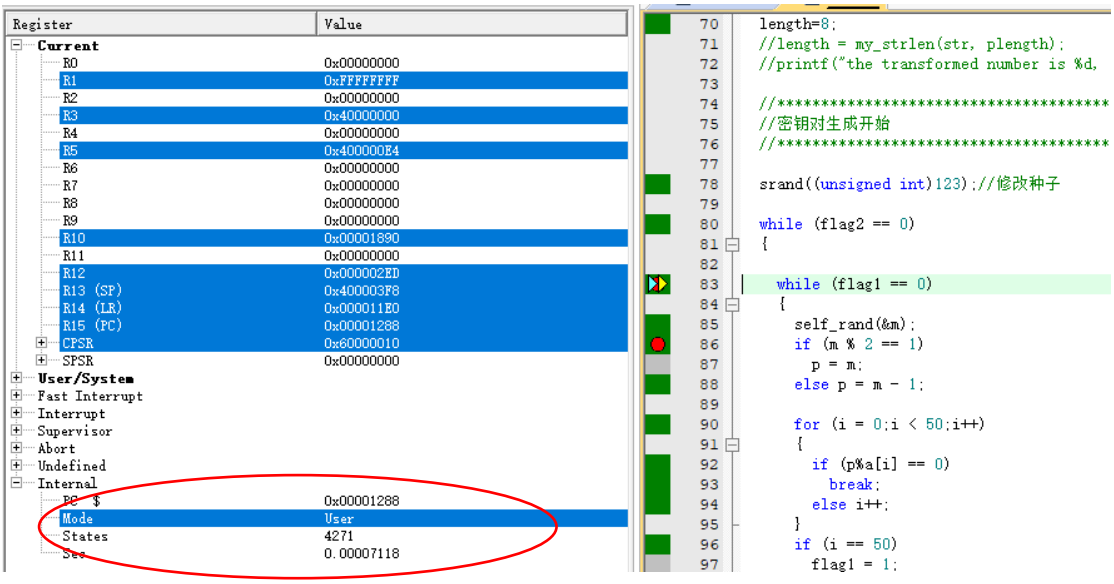

图 5.6 C 语言随机数函数第一个断点情况

运行到第一个断点的时间 0.00007118s，用了 4271 个状态；

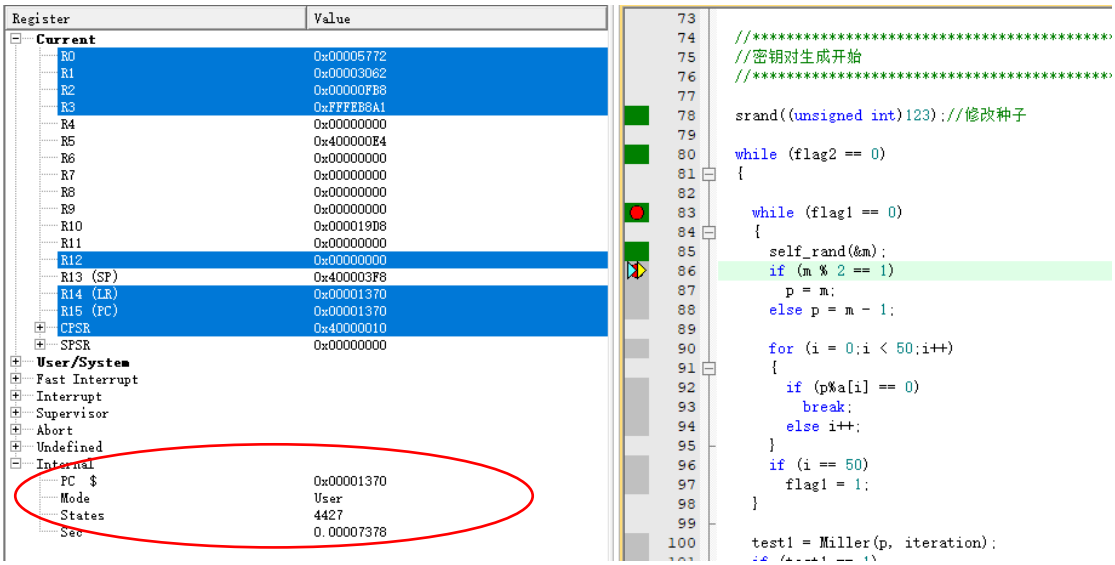

图 5.7 C 语言随机数函数第二个断点情况

运行到第一个断点的时间 0.00007378s，用了 4427 个状态；

由此可得，使用 C 语言时此函数运行时间为 0.0000026s，使用状态数为 156。

② 汇编

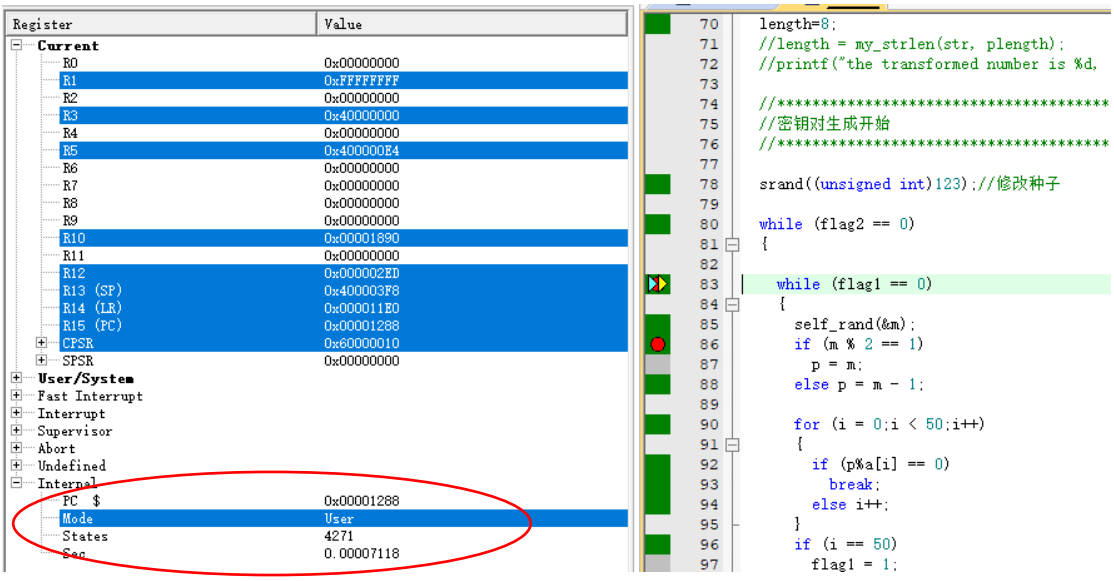

图 5.8 汇编随机数函数第一个断点情况

运行到第一个断点的时间 0.00007118s，用了 4271 个状态；

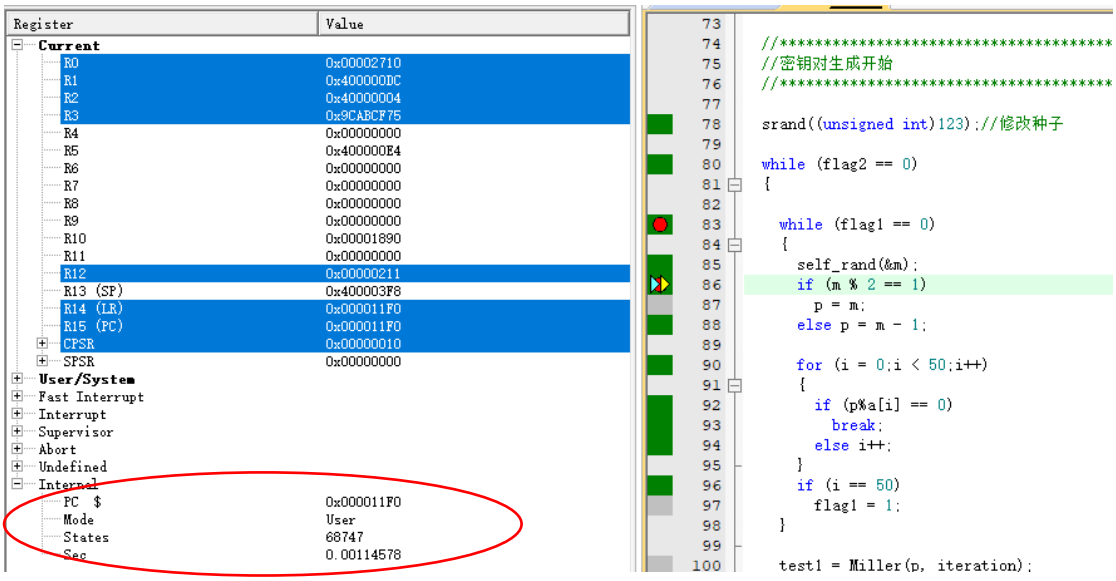

图 5.9 汇编随机数函数第二个断点情况

运行到第二个断点的时间 0.00114578s，用了 68747 个状态；

由此可得，使用汇编语言时此函数运行时间为 0.00107462s，使用状态数为 64476。

结果及原因分析：

由上可见，在使用汇编后性能并没有提升，反而下降。

原因是所写的汇编程序并不优秀，在限制生成随机数的范围时需要进行取余操作，取余又是通过除法来实现的，在使用汇编实现取余时，我们运用了大量的减法操作，极大的提升了算法复杂度，并且我们没有使用移位相减的方法，导致

我们的取余算法非常慢；而经过编译器优化过的 C 语言，虽然将 C 语言处理成汇编语言的过程会花费时间，但是编译器生成的汇编语言是非常优化的，其一共花费的时间是小于汇编函数的。

当然，这种差距在处理随机数生成函数时并不会让人察觉，因为生成的随机数并不大，处理起来都很快，但是在进行加密解密操作的时候，这种差距就很明显了，这点会在下文详细说明。

2）解密函数

代码对比：

① C 语言：

```
long long Encryption(long long Plaintext,  long long e,  long long n) //
加密函数
{

    long long text = 1;
    Plaintext = Plaintext % n;
    while (e > 0) {
        if (e % 2 == 1) text = (text*Plaintext) % n;
        Plaintext = (Plaintext*Plaintext) % n;
        e = e / 2;
    }
    return text;

}
```

② 汇编：

```
AREA Encryption, CODE, READONLY
    EXPORT Encryption
    ENTRY
ruzhan
    PUSH     {r4-r12, lr}
    MOV      r4, r0       ;明文存在 5 和 4
    MOV      r5, r1
    MOV      r10, r2      ;e 存在 10
    MOV      r11, r3      ;n 存在 11
    MOV      r6, #0x01
    MOV      r7, #0x00
    B        judge        ;跳转

judge
```

```
    CMP       r10, #0x00       ;判断 e 和 0
    BGT       jishu            ;e>0 往下进行
    B         jieshu

jishu
    MOV       r0, r10                  ;e 存到 r0
    ADD       r1, r10, r0, LSR #31
    ASRS      r1, r1, #1
    SUB       r1, r10, r1, LSL #1
    CMP       r1, #0x01
    BNE       chengfa
    MOV       r0, r6           ;text=text*plaintext
    UMULL     r8, r3, r0, r4
    MLA       r1, r7, r4, r3
    MLA       r1, r6, r5, r1
    MOV       r2, r11          ;n 变成 64 位存在 r3 r2 中
    ASR       r3, r11, #31
    MOV       r0, r8           ;积存在 r1 r0
    MOV       r6, r0
    MOV       r7, r1
    B         quyu1


chengfa                        ;Plaintext=Plaintext*plaintext
    MOV       r0, r4
    UMULL     r8, r3, r0, r0
    MLA       r3, r5, r0, r3
    MLA       r1, r4, r5, r3
    MOV       r2, r11
    ASR       r3, r11, #31      ;n 变成 64 位存在 r3 r2 中
    MOV       r0, r8           ;积存在 r1 r0
    B         quyu3

quyu1
    CMP       r1, #0x00
    BGT       quyu2
    CMP       r0, r2
    BGT       quyu2
    B         chengfa

quyu2
    SUBS      r0, r0, r2
    SBC       r1, r1, r3
    CMP       r1, #0x00
    BGT       quyu2
```

```
    CMP       r0, r2
    BGT       quyu2
    MOV       r7, r1
    MOV       r6, r0          ;text 存在 r7, r6
    B         chengfa

quyu3
    CMP       r1, #0x00
    BGT       quyu4
    CMP       r0, r2
    BGT       quyu4
    B         chufa

quyu4
    SUBS      r0, r0, r2
    SBC       r1, r1, r3
    CMP       r1, #0x00
    BGT       quyu4
    CMP       r0, r2
    BGT       quyu4
    MOV       r5, r1
    MOV       r4, r0          ;text 存在 r7, r6
    B         chufa

chufa
    ASR       r10, r10, #1     ;e=e/2
    CMP       r10, #2
    B         judge

jieshu
    POP       {r4-r12, pc}
    SWI       #0x12


    END
```

性能对比：

此项目中的汇编版本的加密函数是写在.s 文件，通过 main.c 文件调用实现的，因此在测试性能时，我们创建了一个测试项目，通过观察函数运行的时间来判断性能。而由于加密函数的特殊性，明文和公钥的复杂度(大小)都会影响函数运行的时间，因此我们采用了[明文=312 E=18229 N=18511（数值较小）]、[明

文=20201225 E=837211133 N=837269857（数值较大）]两组数据来比较性能

Ⅰ 输入明文=312 E=18229 N=18511（数值较小）

① C 语言

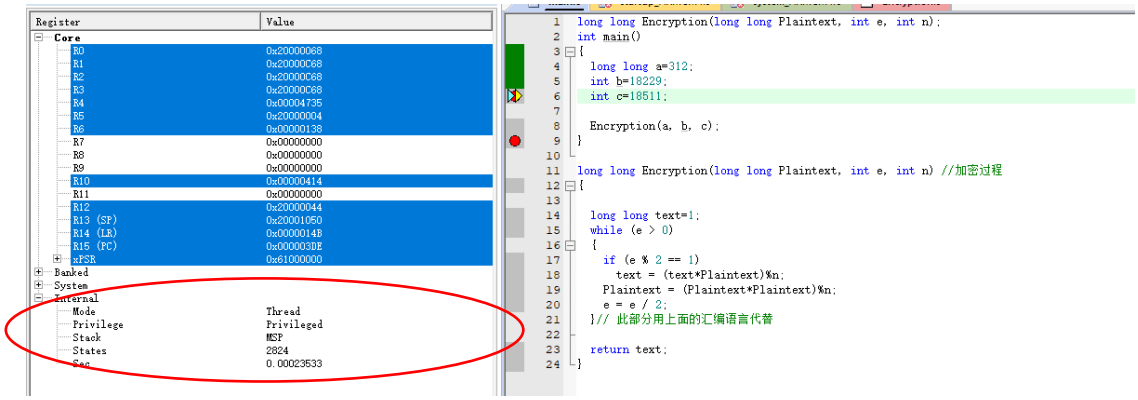

图 5.10 输入较小数 C 语言加密函数第一个断点情况

运行到第一个断点的时间 0.00023533s，用了 2824 个状态；

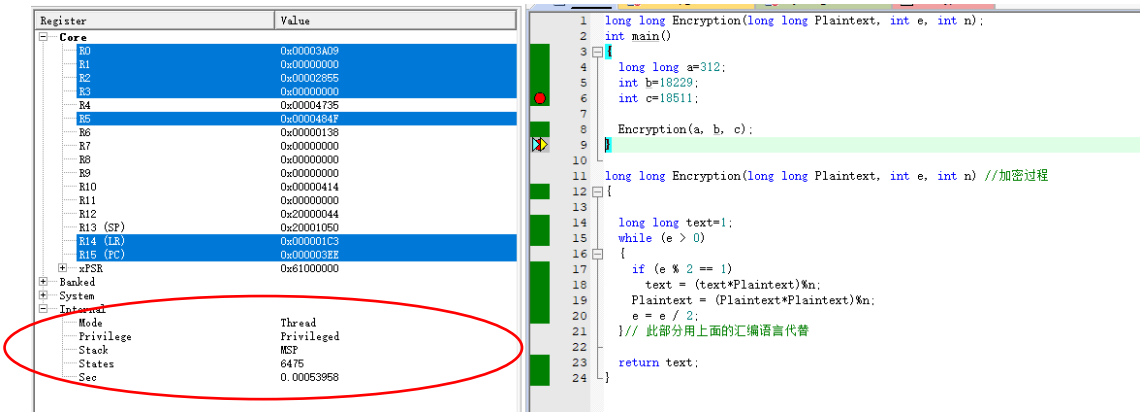

图 5.11 输入较小数 C 语言加密函数第二个断点情况

运行到第二个断点的时间 0.00053958s，用了 6475 个状态；

由此可得，输入较小数时，使用 C 语言时此函数运行时间为 0.00030425s，使用状态数为 3651。

② 汇编

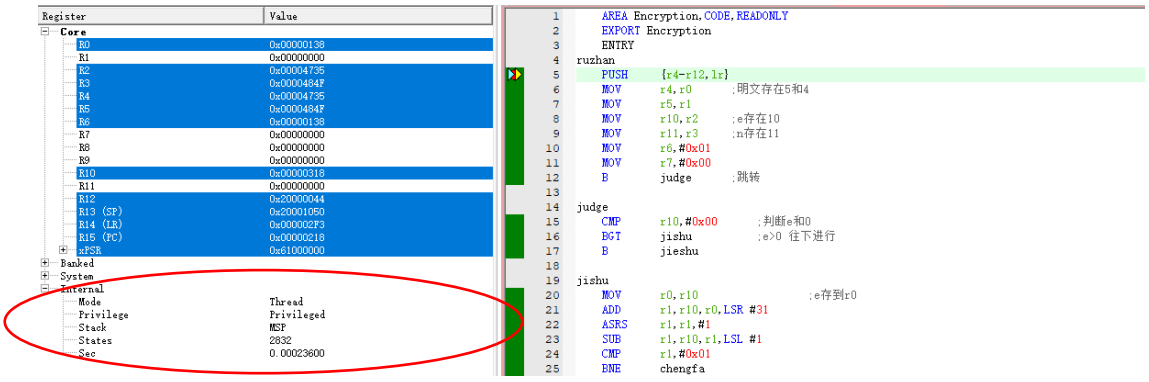

图 5.12 输入较小数汇编加密函数第一个断点情况

运行到第一个断点的时间 0.000236s，用了 2832 个状态；

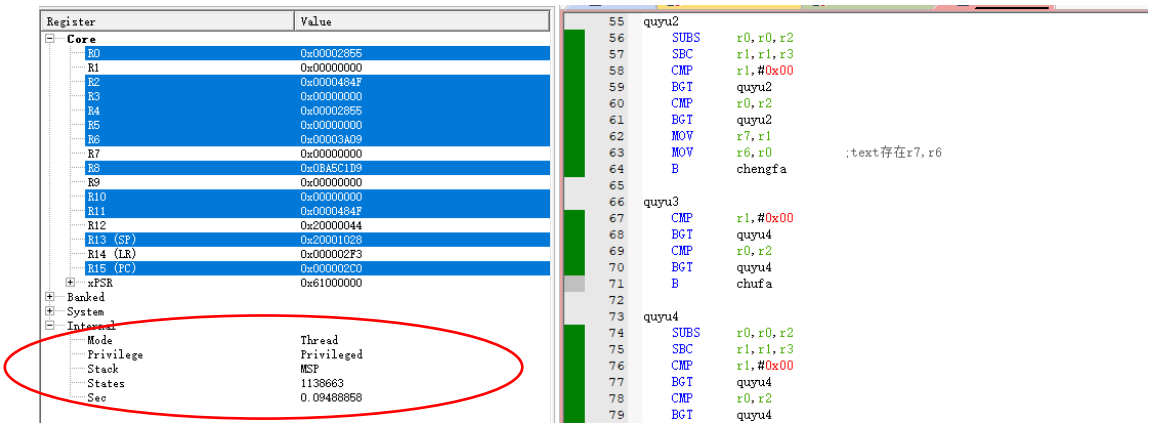

图 5.13 输入较小数汇编加密函数第二个断点情况

运行到第二个断点的时间 0.09488858s，用了 1138663 个状态；

由此可得，使用汇编语言输入较小数时此函数运行时间为 0.094349s，使用状态数为 1135831。

Ⅱ 明文=20201225 E=837211133 N=837269857（数值较大）

① C 语言

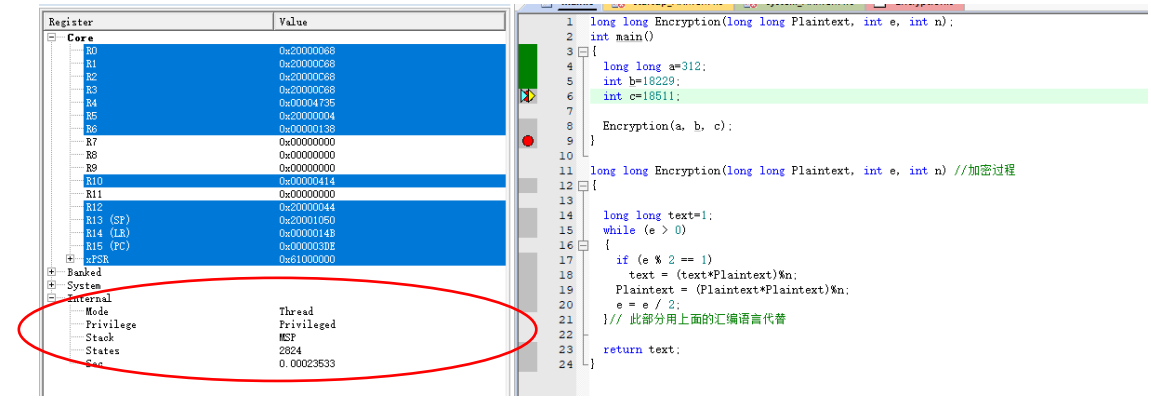

图 5.14 输入较大数 C 语言加密函数第一个断点情况

运行到第一个断点的时间 0.00023533s，用了 2824 个状态；

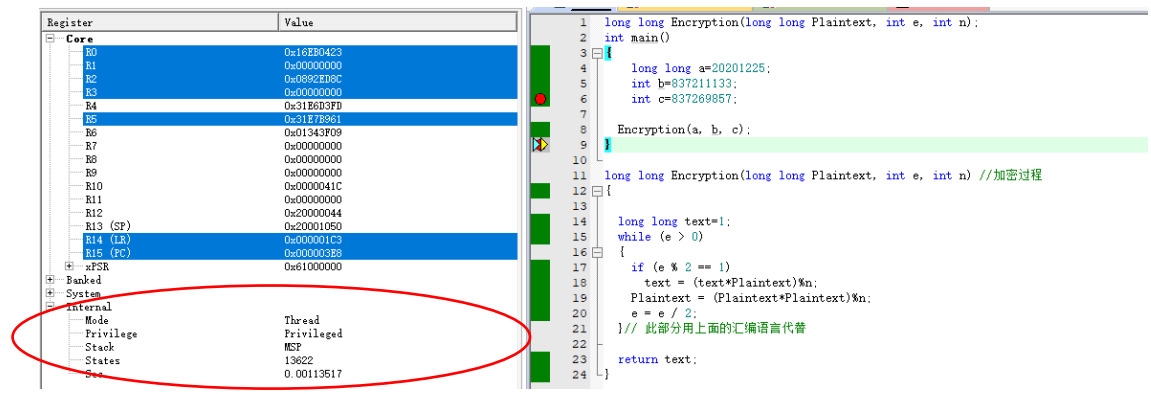

图 5.15 输入较大数 C 语言加密函数第二个断点情况

运行到第二个断点的时间 0.0013517s，用了 13622 个状态；

由此可得，输入较大数时，使用 C 语言时此函数运行时间为 0.00089984s，使用状态数为 10798。

② 汇编

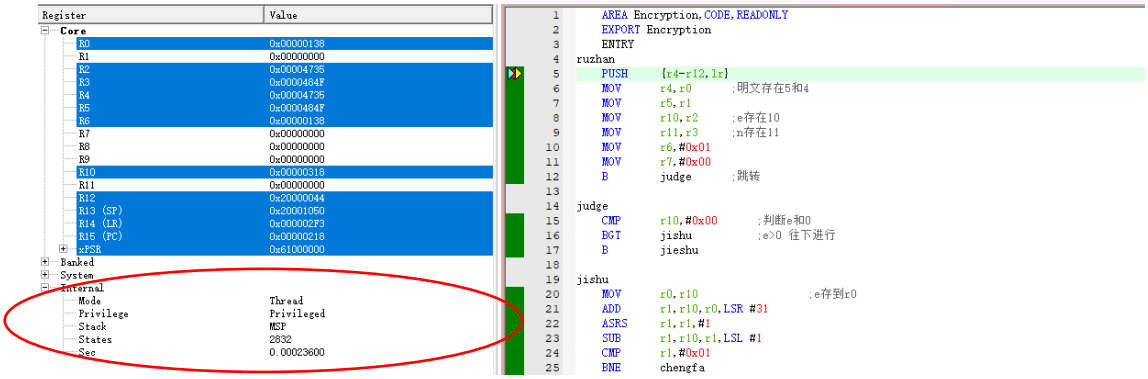

图 5.16 输入较大数汇编加密函数第一个断点情况

运行到第一个断点的时间 0.000236s，用了 2832 个状态；

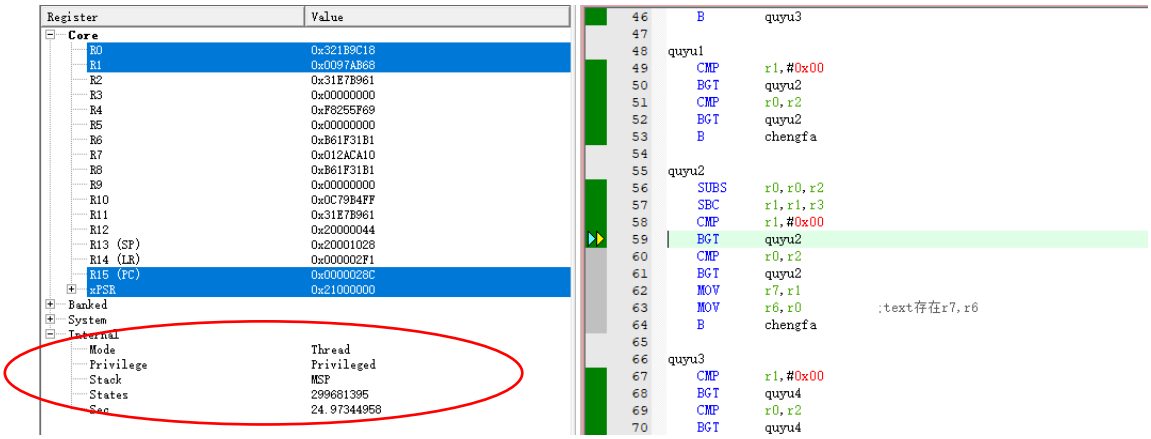

图 5.17 输入较大数汇编加密函数第二个断点情况

在运行到 25s 时程序还没有结束，程序实际运行时间要比 25s 还要长；

由此可得，使用汇编语言输入较大数时此函数运行时间要比 C 语言长的多。

结果及原因分析：

由上可见，在使用汇编后性能并没有提升，反而下降。

主要原因与其实与随机数函数的汇编版本性能下降原因相同，就是我们在汇编程序中不可避免的使用除法，然而我们在用汇编实现除法运算的算法非常复杂，算法复杂度较高。当输入的数较小时，算法复杂度的差距还并不明显，然而当输入的数据变大时，算法复杂度的差距也就显现了出来，查阅资料得知，keil 在实

现 C 语言的除法时是调用库函数进行的，其算法复杂度要比我们在汇编函数中使用的算法复杂度低，因此我们的汇编函数并没有使我们的程序性能提升，这也是一个遗憾。

由于使用汇编函数后性能并没有提升，因此加密函数我以.s 文件的形式放在源代码中，加密函数还是采用 C 函数实现。

而解密函数和加密函数的原理一模一样在这里不再赘述。

3. 与最先进程序对比

说明：由于能查到资料的限制，我们并不清楚与我们进行对比的是否为世界上最先进的 RSA 系统，并且大概率不是最先进的系统。

我们所实现的实际上是一个简单的 RSA 密码系统，与先进水平相比，其有以下不足：

1）公钥和密钥的长度

由于 C 语言变量最长为 64 位，我们所生成的公钥和密钥的长度不能太大，在我们的项目中，公钥和密钥的大小小于十进制的 6 位，换算成二进制也仅有 20 多位，而在世界上先进水平使用的密钥位 2048 位。

2）明文与密文的形式

在先进的 RSA 密码系统中，会先对输入的明文进行编码转换为数字，加密后的数字也会进行编码，而在我们的 RSA 密码系统中，输入和输出的均为数字，由于代码数量的限制，未对字符进行编码处理。

4. 代码行数

我们的代码行数是通过 Visual studio 2017 来计算的，如图 5.18 所示，在查找界面输入^b*[^:b#/]+.*$来查找正则表达式。

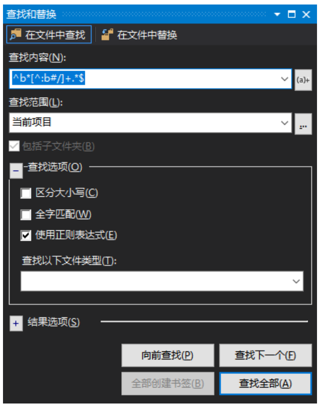

图 5.18 计算代码行数

如图 5.19 所示，包含注释行在内，我们的代码量为 1038 行，即使减去注释，代码也达到了 750 行。

六、外部器件拓展

为了更好地展示我们算法的运行成果，我们进行了外部器件的拓展，使用了 LCD1602 显示器来显示我们的运行结果。

1. LCD1602

LCD1602 液晶显示器（如图 6.1）是广泛使用的一种字符型液晶显示模块。它是由字符型液晶显示屏（LCD）、控制驱动主电路 HD44780 及其扩展驱动电路 HD44100，以及少量电阻、电容元件和结构件等装配在 PCB 板上而组成。

LCD1602 共有两行显示，每行可以显示 16 个字符，对于 RSA 密码系统的展示来说足够使用。

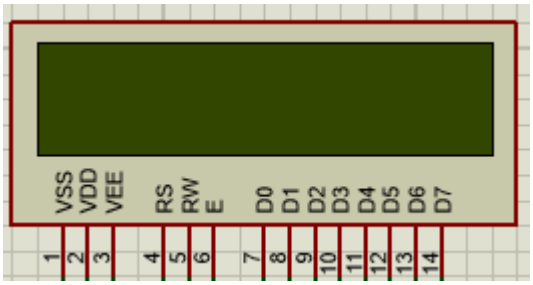

图 6.1 LCD1602 示意图

2. 实现方式

为了使用 LCD 来展示结果，我们在代码中加入了基于 LCD1602 的显示模块并设计了相应的函数。包括 LCD1602 的清零函数，忙碌判断逻辑，字符与字符串显示函数，命令输入函数与坐标控制等等（详见代码）。如此一来，在原先的 RSA 密码系统中引入显示模块，即可在 LCD1602 上展示我们的运行结果了。

3. 连接方式

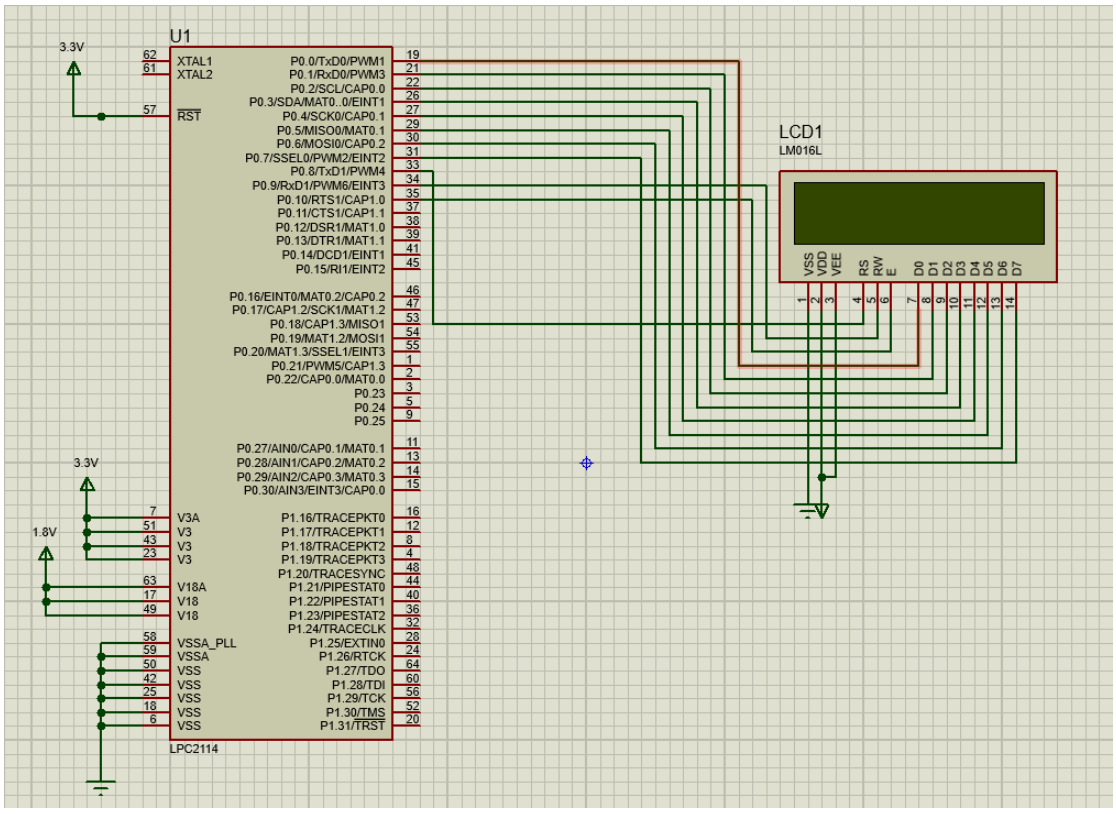

图 6.2 Proteus 中的硬件连接图

4. 部分结果展示

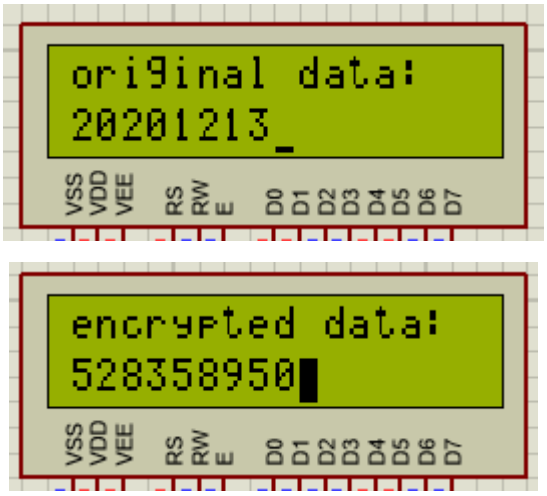

图 6.3 加密源码和加密终码展示

七、个人工作介绍

1. 节点时间表展示

工作内容	计划完成时间	实际完成时间
平台选择	10.14	10.15
查找优异算法	10.19	10.19
Keil5 测试	10.18	10.25
密钥对生成	11.13	11.14
显示模块编写	12.14	12.13
显示模块测试	12.15	12.15
总体代码的 Keil5 调试	12.16	12.16
Keil5 与 Proteus 联调	12.16	12.17

表 7.1 节点时间表

2. 本人分工

*在进行平台选择与测试时，我主要负责 Keil5 的安装与测试工作。

*算法查找方面，我主要负责质数判断算法的确定，最终找到了米勒拉宾算法这个适用于大质数判断的算法

*代码编写方面，我主要负责公钥对于密钥对的生成算法以及相应的功能函数设

计。并且完成了整个显示模块的编写与测试，并将其与主程序相结合。

*调试过程中我主要负责前期 Keil5 与 Proteus 的联合调试，测试整个 RSA 密码系统在 ARM 上运行的可行性以及外部器件是否正常工作。

3. 各环节技巧

1）开题阶段

在开题阶段我进行的最主要工作是 Keil5 平台的安装与测试。Keil5 的安装教程可以在网络上找到大量的资源。并且 Keil5 在完成安装后还要安装一些硬件包，以使其支持我们所选择的微控制器。

在对 Keil5 进行测试时，我首先从网络上找来了一些样例代码在 Keil5 上运行，以此来确定 Keil5 的基本功能是否满足我们的需求。在确定了 Keil5 作为开发平台后，我又自己写了一些简单的汇编程序在 Keil5 上运行并测试，以此来熟悉 Keil5 的使用操作。

2）代码编写阶段

在对代码进行最初的编写时，由于我们都是以 C 语言代码为基础的，因此我们首先在 Visual Studio 平台上进行开发与测试，这样的好处是我们可以通过输入输出直接判断代码功能的正确与否，修改起来非常方便，这样的开发方式我们之后正式在 Keil5 平台上进行开发扫清了许多障碍。

在编写秘钥对生成模块是，我使用的主要技巧是编写与验算相结合，已检测生成的公钥和秘钥对是否符合要求。

在编写显示模块时，我对显示模块的功能进行了拆分，并设置了相应的函数，包括清空，指令输入，显示输入等。这样当在主程序中调用显示模块显示指定字符串或者对 LCD 清屏时就非常简便。

测试显示模块时，我先自己写了一个简单的主函数来显示特定字符串，尽早发现问题。这样在结合 RSA 主程序时就简化了测试过程，避免因主程序过于庞大而难以发现问题所在。

3）调试阶段

在 Proteus 中进行测试时，由于只能看到 LCD 显示屏的显示结果，当结果异常时很难查出哪里出了问题。所以我在测试时，在主函数中加入了一些特殊的字符串显示，这样当程序出现异常时，我就可以根据上次显示的字符串快速定位问题所在，提高了调试效率。

此外，在 Keil5 中通过逐句执行的方式查找问题也是常用的技巧之一。

4. 工作亮点

1）拓展欧几里得算法

在密钥对生成部分，解密钥即私钥中 D 的求解较为复杂。于是我采用拓展欧几里得算法来求解。

已知两个整数 a，b，拓展欧几里得算法可以在求得 a 和 b 的最大公约数的同时，找到整数 x 和 y 使它们满足等式：

$$ax + by = \gcd(a,b)$$

如果对两个整数 a，b 使用辗转相除法（欧几里得算法），则可以得到它们的最大公约数。在拓展欧几里得算法中，我们收集辗转相除中产生的式子，逆向计算即可得到$ax + by = \gcd(a,b)$的整数解，因此拓展欧几里得算法可以用来求模逆元，对 RSA 算法的实现至关重要。

2）显示模块

我在本项目中完成了整个显示模块的编写、测试与调用，并且对显示模块需要调用的微控制器输入输出端口进行了定义，并且完成了 Proteus 中电路图的连接。

我所编写的显示模块基于 LCD1602 显示器。在该显示模块中，我实现了在 LCD1602 的任意位置显示任意标准字符或者字符串。例如我可以指定在第一行的第一个位置处开始，显示一个字符串“hello world”，也可以指定在第二行的第五个位置处显示一个字符“H”。并且随时可以在主程序中对 LCD1602 进行清屏。同时设置了合适的延时函数，以达到更好的显示效果。

3）Keil5 与 Proteus 的共同调试

本项目中 Keil5 与 Proteus 的共同调试工作主要由我完成，我在调试中使用了上述的多种调试技巧，并且修改了一些不被支持的 C 语言函数，例如 itoa 等等。最终使得 Keil5 中生成的 16 进制文件可以烧写在 Proteus 中的微控制器上并且很好地执行我们的预期功能。

5. 遇到的问题与解决办法

1）初期安装测试 Keil5 时，遇到不明原因报错（Error:L6630E）

在 options 的 Linker 选项卡中取消选择 Use Memory Layout from Target Dialog，并清空下方文本框，即可解决。

2）在计算秘钥求最大公约数时效率很低

最开始用的是更相减损术求解最大公约数，但是这种方法在 RSA 密码系统中面对较大的数据时，速度很慢。于是改成了辗转相除法，速度大幅提高。

3）对于如何将 LCD1602 与 LPC2114 连接并不是特别清楚，因此在开始调试时不能正常点亮 LCD1602 并显示

我查阅了一些资料，并参考了网络上的一些资源，最终根据我们自己定义的端口进行了连接，并且对 LCD1602 和 LPC2114 进行了正确的电压设置。

4)LCD 不能正常显示相应的字符串

经过检查和测试，我们发现不能正确显示的原因在于没有对 LCD 显示模块中的一些函数设置正确的延时，导致无法对 LCD1602 进行有效地写入操作，因此 LCD 无显示。当延时设置正确后，即可正常显示需要显示的字符串。

5）在 Keil5 与 Proteus 共同调试时，有部分 C 语言库中的函数不被支持

我们自己重写了这些函数，并将其中的一些采用汇编语言实现，例如 itoa 函数。

6. 体会与建议

1）项目体会

通过本项目，我最深刻的体会是有序的项目管理对于项目开发非常重要。我们小组的三名成员通过合理的分工与时间规划，按时且高效的完成了项目，并且取得了不错的结果。虽然在过程中有些时候会无法完全按计划时间完成，但是总体上项目管理是非常有序且高效的，这对于我们整个项目非常重要。

此外，通过本次项目我还体会到前期准备工作的重要性。我们在前期经过慎重比较确定了我们的核心内容与算法，为整个项目奠定了坚实的基础。

还有就是在项目开发的过程中要学会灵活应变，及时调整。在发现 C 语言库中的部分函数时不能正常使用时，我们就准备通过自己编写的方式来代替原本的库，同样实现了功能。

2）课程建议

在课程中我们学习了 ARM 开发的一些基本知识，学习了汇编语言，但是课堂上的讲解过于理论化，常常在上完课之后不能很好地掌握相应的知识。因此我建议接下来在课堂上能插入一些实例教学环节，让同学们更好地掌握微控制器的原理。

参考文献：

[1] 孙伟. 公钥 RSA 加密算法的改进与实现[D].安徽大学,2014.

[2] 陈侨川. 一种基于 DES 和 RSA 算法的混合加密算法[D].云南大学,2015.

[3] Huawei C&C++ Secure Coding Standard V3.1

[4]元泽怀,李丽芳.单片机工程项目 C 语言编程规范实践教学研究[J].肇庆学院学报,2020,41(02):32-36.

[5] R. S. Dhakar, A. K. Gupta and P. Sharma, "Modified RSA Encryption Algorithm (MREA)," 2012 Second International Conference on Advanced Computing & Communication Technologies, Rohtak, Haryana, 2012, pp. 426-429, doi: 10.1109/ACCT.2012.74.

[6] Rahim R, Winata H, Zulkarnain I, et al. Prime number: an experiment Rabin-Miller and fast exponentiation[C]//J. Phys. Conf. Ser. 2017, 930(1): 012032.