TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Cryptographic library for ARM7TDMI processors

Jaroslav BÁN

MASTER'S THESIS

2007

TECHNICAL UNIVERSITY OF KOŠICE
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
Department of Electronics and Multimedia Communications

Cryptographic library for ARM7TDMI processors

Revision 2007-5-15

MASTER'S THESIS

Jaroslav BÁN

Supervisor: Assoc. Prof. Miloš Drutarovský, Ph.D.

Consultant:

Košice 2007

Metadata Sheet

Author:	Jaroslav BÁN
Thesis title:	Cryptographic library for ARM7TDMI processors
Subtitle:	
Language:	English
Type of Thesis:	Master’s Thesis
Number of Pages:	238
Degree:	Inžinier
University:	Technical University of Košice
Faculty:	Faculty of Electrical Engineering and Informatics (FEI)
Department:	Department of Electronics and Multimedia Communications (DEMC)
Study Specialization:	Electronics and Telecommunication Engineering
Study Programme:	
Town:	Košice
Supervisor:	Assoc. Prof. Miloš Drutarovský, Ph.D.
Consultant:	
Date of Submission:	2. 5. 2007
Date of Defence:	31. 5. 2007
Keywords:	Applied cryptography, AES, SHA, ECC, RSA, ARM7TDMI, ARM assembler, C library
Category Conspectus:	Computer science; Programming. Software.
Thesis Citation:	BÁN, Jaroslav: Cryptographic library for ARM7TDMI processors. Master’s Thesis. Košice: Technical University of Košice, Faculty of Electrical Engineering and Informatics, 2007. 238 p.
Title SK:	Cryptographic library for ARM7TDMI processors
Subtitle SK:	
Keywords SK:	Aplikovaná kryptografia, AES, SHA, ECC, RSA, ARM7TDMI, ARM asembler, C knižnica

Abstract in English

Cryptography is increasingly finding applications in embedded devices. One of the processors of choice for embedded devices is the ARM7TDMI. This processor’s fast 32- by 32-bit multiplier, efficient barrel shifter and powerful addressing modes make it very suitable for cryptographic applications, whether they are symmetric block ciphers or arbitrary precision arithmetic. This paper presents a cryptographic library optimized for the ARM7TDMI processor. The library implements the Advanced Encryption Standard (AES), Secure Hash Algorithms (SHA), the RSA algorithm and the Elliptic Curve Digital Signature Algorithm (ECDSA) over all the curves recommended by the Standards for Efficient Cryptography Group (SECG). The library is both small (38 kB of code memory) and fast (low-level routines written in assembler, C-callable) and strikes a balance between generality, speed and memory requirements. The performance achieved is comparable to available commercial and open-source libraries. One of the results is a general routine for fast reduction that solves the problem of having to write a different routine for each elliptic curve. The library was developed using the GNU toolchain and the µVision3 IDE.

Abstract in Slovak

Kryptografia nachádza stále väčšie uplatnenie vo vložených systémoch. Jeden z rozšírených procesorov pre vložené systémy je ARM7TDMI. Tento procesor sa vďaka jeho 32 krát 32-bitovej násobičke, rýchlym bitovým operáciam a širokým možnostiam adresovania veľmi hodí na kryptografické aplikácie, či už symetrické blokové šifry alebo prácu s veľkými číslami. Táto práca opisuje kryptografickú knižnicu optimalizovanú pre procesor ARM7TDMI. Knižnica realizuje algoritmy Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA), RSA a Elliptic Curve Digital Signature Algorithm (ECDSA) na všetkých krivkách odporúčaných skupinou Standards for Efficient Cryptography Group (SECG). Knižnica je malá (38 kB programovej pamäti) a rýchla (nízko-úrovňové moduly sú písané v asembleri a dajú sa volať z jazyku C) a snaží sa nájsť rovnováhu medzi všeobecnosťou, rýchlosťou a pamäťovými nárokmi. Dosiahnuté výsledky sú porovnateľné s dostupnými komerčnými a open-source knižnicami. Jeden z výsledkov je všeobecná funkcia na rýchlu redukciu, ktorá znamená, že sa nemusí písať rýchla redukcia pre každú eliptickú krivku. Knižnica bola vyvinutá pomocou nástrojov GNU a vývojového prostredia µVision3.

TECHNICKÁ UNIVERZITA V KOŠICIACH

Fakulta elektrotechniky a informatiky

Katedra elektroniky a multimediálnych telekomunikácií
akademický rok: 2006/2007

ZADANIE
DIPLOMOVEJ PRÁCE

pre: **Jaroslav Bán**

Odbor: Elektronika a telekomunikačná technika

Študijný program:

Vzhľadom k tomu, že ste splnili požiadavky učebného plánu, zadáva Vám dekan fakulty na návrh vedúceho vedecko-pedagogického pracoviska v zmysle zákona o VŠ č.131/2002 a Študijného poriadku TU §15, ods. 3, túto tému záverečnej práce:

Kryptografická knižnica pre procesory s jadrom ARM7TDMI

POKYNY PRE VYPRACOVANIE

Osnova práce:

Navrhnite štruktúru kryptografickej knižnice pre procesory s jadrom ARM7TDMI, ktorá umožní využitie kryptografických algoritmov (ECC, RSA, AES, SHA, ...) v základných kryptografických módoch a protokoloch. Knižnica má podporovať minimálne parametre špecifikované v aktuálnych normách NIST pre jednotlivé kryptografické algoritmy. Navrhnutú knižnicu implementujte v GNU vývojových nástrojoch pre procesory s jadrom ARM. Kritické časti knižnice optimalizujte v asembleri procesora ARM s cieľom minimalizácie nárokov na pamäť RAM prípadne veľkosť programovej pamäte. Otestujte možnosť použitia vytvorenej knižnice s prekladačom Real View firmy Keil ARM pre vložené procesory. Funkčnosť knižnice overte pomocou dostupných technických prostriedkov na báze jadra ARM7TDMI a vytvorte príklady demonštrujúce interoperabilitu navrhnutej knižnice s typickými kryptografickými knižnicami na PC platforme (napr. OpenSSL, Miracle a pod.). Dosiahnuté výsledky (rýchlosť, veľkosť, funkčnosť) navrhnutej knižnice porovnajte so známymi implementáciami pre vložené procesory na báze jadra ARM7TDMI.

Rozsah laboratórnych a grafických prác : podľa potreby

Rozsah záverečnej práce: odporúčaný počet strán 40 a viac

Zoznam odporúčanej literatúry:

1. J.A. Menezes, P.C. Oorschot, S.A. Vanstone: Handbook of Applied Cryptography, CRC Press, New York, 1997.
2. D. Hankerson, A. Menezes, S. Vanstone: Guide to Elliptic Curve Cryptography, Springer-Verlag, New York, 2004.
3. www.nist.gov
4. www.keil.com
5. www.codesourcery.com
6. www.openssl.org

Vedúci záverečnej práce: doc.Ing. Miloš Drutarovský,CSc.

Konzultant:

Dátum zadania záverečnej práce: 2.4.2007

Dátum odovzdania záverečnej práce: 2.5.2007

V Košiciach, dňa 2.4.2007

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
-02-

v z. Kololor

prof.Ing. Liberios Vokorokos,PhD.

dekan fakulty

prof.Ing. Dušan Levický,CSc.

vedúci vedecko-pedagogického pracoviska

Thesis Assignment

Slovak

Navrhnite štruktúru kryptografickej knižnice pre procesory s jadrom ARM7TDMI, ktorá umožní využitie kryptografických algoritmov (ECC, RSA, AES, SHA, …) v základných kryptografických módoch a protokoloch. Knižnica má podporovať minimálne parametre špecifikované v aktuálnych normách NIST pre jednotlivé kryptografické algoritmy. Navrhnutú knižnicu implementujte v GNU vývojových nástrojoch pre procesory s jadrom ARM. Kritické časti knižnice optimalizujte v asembleri procesora ARM s cieľom minimalizácie nárokov na pamäť RAM prípadne veľkosť programovej pamäte. Otestujte možnosť použitia vytvorenej knižnice s prekladačom RealView firmy Keil ARM pre vložené procesory. Funkčnosť knižnice overte pomocou dostupných technických prostriedkov na báze jadra ARM7TDMI a vytvorte príklady demonštrujúce interoperabilitu navrhnutej knižnice s typickými kryptografickými knižnicami na PC platforme (napr. OpenSSL, MIRACL a pod.). Dosiahnuté výsledky (rýchlosť, veľkosť, funkčnosť) navrhnutej knižnice porovnajte so známymi implementáciami pre vložené procesory na báze jadra ARM7TDMI.

English

Design the structure of a cryptographic library for ARM7TDMI processors, which would allow the use of cryptographic algorithms (ECC, RSA, AES, SHA ...) in basic cryptographic modes and protocols. The library is to support at least the parameters specified in current NIST norms for the individual cryptographic algorithms. Implement the library in the GNU toolchain for ARM processors. Optimize critical parts of the library in ARM assembler with the goal of minimizing RAM requirements and/or code memory requirements. Test the possibility of using the library with Keil ARM's RealView compiler for embedded processors. Verify the functionality of the library on available ARM7TDMI-based systems and create programs demonstrating the interoperability of the library with typical PC-based cryptographic libraries (e.g. OpenSSL, MIRACL, etc.). Compare the results obtained with the library (speed, size, functionality) with known implementations for ARM7TDMI-based embedded processors.

Declaration

Vyhlasujem, že som celú diplomovú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice, 2. 5. 2007

....

signature

Acknowledgement

I would like to thank my family for their unconditional love and support and for patiently letting me use the computer for months on end. I would like to thank my supervisor, Prof. Drutarovský, for his valuable recommendations and careful review of this thesis and of the library. Any bugs, errors, omissions, or just general sloppiness that he missed must be the result of skillful obfuscation on my part.

Preface

Cryptography is a fascinating subject. It is a truly interdisciplinary field – mathematics, quantum mechanics, software, hardware, electronics, telecommunications, export controls, politics, and war. It defiles pure and abstract mathematics by finding real-world applications for its lofty theorems. The cryptographer has to leave the ivory tower and has to consider mundane things like the current consumption of a circuit and cache timing attacks. The advertised N-bit security – no, N is not enough – the advertised M-bit security is negated by a side-channel attack or a weak key or a short password or a padding scheme or an error message.

The topic of this thesis was chosen intentionally to serve multiple purposes. It was to be an exercise in ARM assembler programming, embedded device programming, C programming, and a deeper study of modern cryptographic primitives. It was also supposed to be a learning experience in LaTeX desktop publishing. Unfortunately, this didn’t work out for lack of time (the thesis is written in Microsoft Word).

Table of Contents

List of Illustrations **13**

List of Tables **14**

List of Symbols and Abbreviations **15**

Introduction **16**

1 The ARM7TDMI processor architecture **18**

2 Software development **20**

2.1 Choice of development environment 20

2.2 GNU Toolchain 20

2.3 Interfacing µVision3 with the GNU toolchain 20

2.4 Simulation conditions 21

3 Library structure **22**

3.1 Design philosophy 22

3.2 C vs. ARM assembler 22

3.3 Data structures 23

3.3.1 vect 23

3.3.2 Other data types 25

3.4 Temporary storage 25

3.5 Error detection and reporting 25

3.6 Application programming interface (API) 26

3.7 Directory structure 26

4 Library verification and testing **27**

4.1 Test vectors 27

4.2 Random inputs, algebraic theorems 27

4.3 Serial port interface 28

5 Library components **29**

5.1 Advanced Encryption Standard (AES) 29

5.1.1 Implementation details 30

5.1.2 Implementation results 30

5.2 Secure Hash Algorithm (SHA) 30

5.2.1 Implementation details 31

5.2.2 Implementation results 32

5.3 Common routines 32
5.3.1 Copy array 32
5.3.2 Bit manipulation 33
5.3.3 Comparison 33
5.3.4 Convert number to string 33
5.3.5 Convert string to number 33
5.3.6 Other routines 34
5.4 Prime field arithmetic 34
5.4.1 Data structures 35
5.4.2 Modular addition and doubling 35
5.4.3 Modular subtraction 35
5.4.4 Modular halving 35
5.4.5 Multiplication 35
5.4.6 Squaring 37
5.4.7 Montgomery reduction 38
5.4.8 Modular exponentiation 39
5.4.9 Greatest common divisor (GCD) 39
5.4.10 Partial Montgomery inversion 40
5.4.11 Modular inversion 41
5.4.12 Long division – remainder (modulo) 41
5.4.13 Long division – quotient 41
5.4.14 Modular square root 41
5.4.15 Fast reduction 42
5.5 Pseudo-random number generators (PRNGs) 42
5.5.1 Linear congruential 43
5.5.2 AES in counter mode 43
5.5.3 SHA of a counter 43
5.5.4 Blum-Blum-Shub 43
5.5.5 Timings 44
5.6 Finding large prime numbers 44
5.6.1 The Rabin-Miller primality test 45
5.6.2 Prime finding algorithm 45
5.6.3 Timing results 46
5.6.4 Implementation details 47

5.6.5 Asymptotic behavior ... 47
5.7 The RSA algorithm ... 47
5.7.1 Chinese remainder theorem speed-up ... 48
5.7.2 Implementation timings ... 50
5.8 Binary field arithmetic ... 50
5.8.1 Addition ... 51
5.8.2 Division by z ("halving") ... 51
5.8.3 Multiplication ... 51
5.8.4 Squaring ... 52
5.8.5 Partial Montgomery inversion ... 53
5.8.6 Modular inversion ... 53
5.8.7 Fast reduction ... 54
5.9 Elliptic curve cryptography (ECC) ... 54
5.9.1 Group formulation ... 54
5.9.2 Point addition ... 55
5.9.3 Recommended curves ... 57
5.9.4 Timings of prime field operations ... 58
5.9.5 Timings of binary field operations ... 59
5.9.6 Projective coordinates ... 60
5.9.7 EC point arithmetic over GF(p) ... 61
5.9.8 EC point arithmetic over GF(2^m) ... 63
5.9.9 EC point multiplication ... 64
5.9.10 EC point compression ... 65
5.10 Elliptic Curve Digital Signature Algorithm (ECDSA) ... 65
6 Code size, comparisons ... 67
7 Conclusion ... 69
Bibliography ... 70
Appendices ... 74

List of Illustrations

Figure 1. Structure of the ARM cryptographic library with dependencies 22

List of Tables

Table 1 AES implementation timings 30
Table 2 SHA function characteristics (all sizes in bits) 31
Table 3 SHA implementation timings 32
Table 4 Operand scanning form of multiplication 36
Table 5 Product scanning form of multiplication 36
Table 6 Product scanning form of squaring 38
Table 7 Performance of Pseudo-Random Number Generators 44
Table 8 Average time to generate an m-bit prime with k = 8 (Rabin-Miller test iterations) 46
Table 9 RSA operation timings [ms] 50
Table 10 Prime field operation timings (in cycles) 58
Table 11 Binary field operation timings (in cycles) 60
Table 12 Timings of ECDSA signature generation (G) and verification (V) 66
Table 13 Codesize of the library modules in bytes 67
Table 14 Comparison with the Wakan Crypto Toolkit on the ARM 68
Table 15 Comparison with published ECDSA generation timings on the ARM 68

List of Symbols and Abbreviations

ACL	ARM Cryptographic Library
AES	Advanced Encryption Standard
ARM	Advanced Risc Machine
BBS	Blum-Blum-Shub (pseudo-random number generator)
CBC	Cipher-block chaining (block cipher mode)
CRT	Chinese Remainder Theorem
CTR	Counter (block cipher mode)
ECB	Electronic Codebook (block cipher mode)
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
FIPS	Federal Information Processing Standard
GCD	Greatest Common Divisor
$GF(2^m)$	Galois Field (binary)
$GF(p)$	Galois Field (prime)
GNU	GNU's Not Unix
IDE	Integrated Development Environment
LSB	Least significant bit (or byte)
MSB	Most significant bit (or byte)
NIST	National Institute of Standards and Technology
PRNG	Pseudo-Random Number Generator
RISC	Reduced Instruction Set Computer
RSA	Rivest-Shamir-Adleman (algorithm)
SECG	Standards for Efficient Cryptography Group
SHA	Secure Hash Algorithm

Introduction

Cryptography is the study of message secrecy in the presence of an adversary. An excellent historical overview of cryptography can be found in David Kahn's fascinating "The Codebreakers" [3].

Modern cryptography relies in large part on time-proven cryptographic primitives (hash function, symmetric cipher, public key cipher ...), which can be used as components in cryptographic protocols (key exchange, digital signature ...). An overview of some common cryptographic primitives can be found in [4].

Embedded systems by their nature have limited system resources, but still need information security in some applications. This creates a niche market for efficient cryptography implementations that can run on limited resources. Examples of such applications would be smart cards [5] and wireless sensor networks [6].

The ARM architecture [9] has seen widespread use in embedded systems, especially in its most popular embodiment, the ARM7TDMI processor core [12].

One of the goals of this thesis was to gain a better understanding of the implemented cryptographic primitives. The following questions were to be answered:

- What routines does this particular cryptographic module depend on? Examine the inter-dependencies.
- Where are the performance bottlenecks? Which routine dominates execution time?
- What are the strengths and weaknesses of the ARM architecture for cryptographic applications?
- What are the tradeoffs involved (memory/speed)?

The best way to answer these questions is to write and actually implement all these cryptographic primitives from the ground up.

But if the lessons learned were to have any kind of general application, the library had to be as general as possible. It is one thing to optimize a routine for one specific size / curve / bit length. It is another matter to optimize it for a wide range of sizes / curves / bit lengths.

A general approach forces us to use methods that will be efficient over a wide range of parameters.

This work does not exist in a vacuum. It is not world-changing. There are many different (and much more elegant and general) libraries out there. Still, we did not choose to optimize an existing library. We also didn’t try to draw inspiration from existing libraries because we wanted a library specifically suited to the ARM architecture.

The methods we used are standard for any optimization: a high quality debugger and performance analyzer.

As to the literature, we consistently found one resource to be more useful than the others – the “Guide to Elliptic Curve Cryptography” [1]. Kudos to the authors.

Chapter 1 deals with the ARM7TDMI. Chapter 2 looks at the software development process and tools. Chapter 3 looks at the structure of the library. Finally, the bulk of the thesis is chapter 5 – description and analysis of the library components.

1 The ARM7TDMI processor architecture

The ARM architecture was created in the 1980's at Acorn Computers Ltd. It is a von Neumann-type 32-bit RISC architecture. Since its first version, the ARM architecture has gone through a number of versions or families. The ARM7TDMI actually uses ARM architecture version 4 (ARMv4). This can lead to confusion, as there is also an architecture version ARMv7. The TDMI stands for:

T – Thumb (16-bit) instruction set

D – on-chip debugging support

M – high performance multiplier yielding a full 64-bit result

I – EmbeddedICE hardware

The ARM7TDMI has seen widespread use in embedded devices because of its high performance and low power consumption. It has:

- a large register set (16 x 32-bit registers): r0 – r15 where r13 = stack pointer, r14 = link register, r15 = program counter
- an orthogonal instruction set
- fixed instruction width (32 bits) – this leads to low code density; thumb was introduced to address this
- conditional execution of (almost) every instruction – fewer branches necessary
- barrel shifter – allows for fast shifts and rotates by any number of bits
- 3-stage pipeline
- wide selection of addressing modes

An interesting feature is that there is no dedicated stack pointer. Any register can be used as a stack pointer and the stack can be ascending or descending, full or empty. Because of this, when a subroutine is called, the return address doesn't get pushed onto the stack, but instead gets stored in the link register. It is then up to the user to push it onto the stack if necessary. Another feature is that the program counter can be used as a base pointer in memory addressing, allowing PC-relative addressing.

LDM and STM (load and store multiple) are two very powerful ARM instructions – they allow any combination of registers to be loaded from/stored to memory.

The NXP ARM7TDMI processors [25] that we simulated on are little-endian, so the library was written assuming a little-endian processor.

For a much better introduction to ARM7TDMI processors see “The Insider's Guide to the Philips ARM7-Based Microcontrollers” [13].

There are a few documents that can be considered indispensable for anyone developing low-level software for the ARM.

The ARM architecture reference manual (the ARM ARM) includes detailed descriptions of each instruction [9]. There is also a quick reference card [11]. For optimization, the instruction timings are critical [12]. Also, there is an overview of ARM programming techniques [14].

2 Software development

This chapter describes some of the choices involved in getting from the assignment to the final product. As this library was written by one person, some issues that arise in bigger projects were non-existent. For a much better treatment of the subject, see Fred Brooks‘ seminal book “The Mythical Man-Month“ [8].

2.1 Choice of development environment

The obvious choice for an integrated development environment was Keil’s µVision3 IDE (version 3.50), because of its excellent debugger and the availability of a free evaluation version [15]. Another option was the open-source Eclipse IDE [16], [17].

Keil (acquired by ARM in October 2005) also provides a real-time kernel – the RTX kernel [18]. Another option for a real-time kernel that we had experience with was FreeRTOS [19]. In the end we decided against an OS and decided to write the library for “bare metal”.

2.2 GNU Toolchain

The ARM assembler / C compiler we chose was CodeSourcery’s GNU toolchain for ARM processors (version 2006q3-27) [21]. This was necessary because Keil’s evaluation version of the RealView linker will not link object files over a total of 16 kB. Here another possibility was Martin Thomas’ WinARM GNU-toolchain [20].

2.3 Interfacing µVision3 with the GNU toolchain

The µVision IDE allows external tools to be used, but the problem is that the GNU toolchain expects UNIX-like paths (/usr/bin/etc...), while the µVision IDE calls external compilers with command-line parameters that use Windows paths (\My Documents\...). The slashes are different and Windows has spaces in its paths, which causes problems when passed as a command line parameter.

To integrate these two, we had to create a layer of “glue“ programs to interface between the two environments. This glue layer basically converts the slashes, removes the spaces and does any other command-line option replacement that is necessary. These programs can be found in the “Tools” directory (see section 3.7).

Also, RealView uses DWARF3 debug data and the GNU toolchain uses DWARF2 debug data [22]. So we found that it is only possible to link a GNU-compiled library under RealView if all the debug information is removed on the GNU side, and the RealView side is compiled with the –dwarf2 option.

2.4 Simulation conditions

The simulation is set up with the Keil MCB2130 demonstration board [23] in mind. This board is built around a NXP (formerly Philips) LPC2138 [25] with a 12MHz crystal. Using the on-chip PLL we multiply this by 5 to get a processor clock frequency of 60 MHz. Also, we made use of the LPC2138’s memory accelerator module (MAM), which speeds up access to the on-chip flash memory. So all the timings in this thesis essentially assume a 60MHz ARM7TDMI.

The timings are the result of simulation in the μVision3 simulator/debugger. We have found the simulator to be very accurate also in our previous work on digital filters [24], so the use of simulation results instead of actual hardware timings is justified.

Where input-dependent timings were involved, we list the average of 4-8 measurements.

We also made use of μVision’s excellent Performance Analyzer [26], an indispensable tool for finding a bottleneck quickly. For more complicated analysis (prime generation - see section 5.6) we also used μVision’s execution profiling capability.

3 Library structure

Figure 1 shows the structure of the library. Note that the “Common” routines are used throughout the C part of the library. Also note that the ECDSA routines do not explicitly require the SHA hash algorithm, as they only accept the resulting value of any hash. Also note that the “Primes” module requires a pseudo-random number generator (PRNG), but also that the Blum-Blum-Shub PRNG requires the “Primes” module.

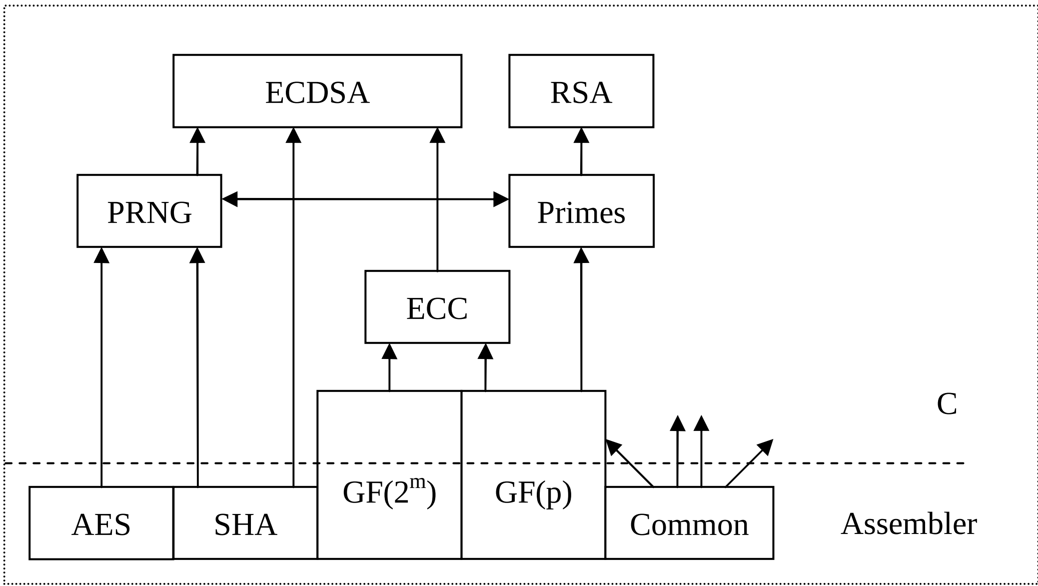

Figure 1. Structure of the ARM cryptographic library with dependencies

3.1 Design philosophy

We chose a bottom-up approach. We first looked at the big picture of what an algorithm needs (a good example of such a tree can be found in [1], p. 75, or). Then we started implementing the routines from the bottom up. This allowed us to test the code as it was being written. Also, it allowed the realistic capabilities of the routines to shape their API instead of having to adapt assembler routines to some prescribed interface.

3.2 C vs. ARM assembler

We tried to write as much as practically possible in assembler. On the other hand, the non-critical / rarely called routines were done in C.

Interfacing C and assembler is the role of the Procedure Call Standard for the ARM Architecture (AAPCS, [10]). This document specifies how a C compiler passes arguments to a function and how it recovers the (optional) return value.

For our purposes, the AAPCS stipulates that the first four parameters of the called function are passed in registers r0 – r3. There are rules for what the compiler should do if a parameter doesn’t fit into 32 bits, but we never had to pass anything bigger. Any subsequent arguments are stored on the stack.

The called routine can corrupt registers r0 - r3 and r12, but has to preserve registers r4 - r11, r13 and r14. This means that if a routine wants to use these registers, it will have to push them onto the stack before it modifies them and pop them after it’s done. Note that r15 is the program counter and r13 is the stack pointer. A routine could use r13, but this is rarely done. If there is a return value, it is returned in r0.

3.3 Data structures

The title of Nicklaus Wirth’s book states “Algorithms + Data Structures = Programs” [7]. The data structures are the overlooked but more important of the two. In “The Mythical Man Month”, Fred Brooks clarifies the relationship between the two: “Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowchart; it’ll be obvious” [8]. To avoid mystifying the reader, it behooves us to first mention our data structures.

3.3.1 vect

The large numbers (say 1024 bits or more) that arise in modern cryptography are usually represented as structs (called “bignum”s) which contain fields indicating their length, sign and a pointer to the actual array of digits (see [59], [60], [61], [62], [63], [64], [65]).

We considered this layout, but found it to be overkill in our case, as we were not building a generic number-theoretic library. Also, most of the low-level library routines would be written in assembler, and this would complicate matters. Since we would not be working with negative numbers, we considered the following layout:

L	A[0]	A[1]	...	A[L-1]

Where L is the number of 32-bit words that follow - A[0] is the least significant word and memory addressing increases to the right (we assume a little-endian layout of the words because most operations process them from least significant to most significant).

This way, instead of passing a pointer to a struct, we would be passing a pointer to an array of 32-bit integers and the first integer would contain the length (and optionally the sign).

While this is more elegant, it would lead to many complications – it would require very complicated routines to handle the different combinations of lengths, which we wouldn't really need as most of the time we would be adding / multiplying ... numbers of identical lengths. Also, it would make merging and splitting numbers difficult.

In the end we settled on the simplest option:

A[0]	A[1]	...	A[L-1]

Where L (the length of the array) is passed to the routine as a parameter.

In C code we refer to a pointer to this data structure (to an array of 32-bit words) as a "vect". We could also have called it a "bignum" or "longint", but vect was chosen because it has connotations of a pointer (in geometry) as well as because of the analogy of scalar–vector to integer–array of integers.

So a "vect" pointer always has a variable passed with it (usually called "len") that determines the length of the array. But immediately it became necessary to have arrays that are multiples of "len" (for example multiplying two n-bit numbers produces a 2n-bit number). To distinguish between the two without having to pass an extra "len2" variable, we created the type "vect2", which points to an array of size 2 times "len".

This led to a proliferation of "vectN" types. Currently there is even a vect11.

Lastly it became necessary to have pointers to fixed-length arrays, and for lack of a better idea, we called them "vectN", where N is the length of the array. They are the following: vect4 in AES and vect16, vect23, vect26, vect68 in SHA.

3.3.2 Other data types

The library uses different redundant (most are ints or pointers to ints) data types to provide information to the user. The different types are described below.

The “size_t” type is used to store the length of a vect array.

The “bool_t” type is a boolean variable (it can be TRUE = -1 or FALSE = 0).

The “uint” acts as a short for “unsigned int”.

The “prng” type is used to pass a pointer to a pseudo-random number generator.

3.4 Temporary storage

When a routine requires temporary storage, there are a few options of how to get it – the stack, dynamic allocation or having the caller provide it. We chose the last option throughout, since the ARM architecture is often used in embedded environments with constrained memory. We let the user choose how to allocate the memory and then simply have the user pass a pointer to it.

3.5 Error detection and reporting

The error reporting structure of a library is an important indicator of the quality of a library. This is especially critical in cryptography. Our library has no such structure. This is due to a few factors:

- The rule is “Don’t detect errors that you cannot fix.” – There is no point in detecting fatal errors, since there is nothing we can do about them.
- We do not do dynamic memory allocation inside the library.
- The idea was to design this library in such a way, that a developer could take the optimized assembler routines and incorporate them into a high-quality general cryptographic library with an error reporting system.

3.6 Application programming interface (API)

We chose an API interface that mimics both the ARM instruction set (most ARM instructions can specify a destination register and two operands) and the format of mathematical equations: result = function(operand1, operand2, …). Most of the functions in the library follow a predictable pattern:

function(result, operand1, operand2, ...);

This allows the user to intuitively rewrite equations into function calls.

Care was taken to make the routines work in-place (result and operand1 are the same memory location), and the user is warned where this is not the case. These warnings can be found both in the header file `acl.h` and in the code of the individual routines.

Some functions on the other hand (e.g. elliptic curve point operations) work only in-place.

3.7 Directory structure

The library source code is divided into directories according to its logical structure as described in Chapter 5. There are other directories in the distribution, which are not part of the library itself:

Tools – contains the "glue" code necessary to interface mVision3 and the GNU toolchain (see section 2.3).

Unused – contains some source code that wasn't necessary. Notably it includes alternative fast reduction routines that were written before the general one (see section 5.4.15).

Tests – contains projects that perform library testing and timing. The directory includes projects that use GNU tools, as well as corresponding RealView projects (these have an "rv" suffix).

The main directory contains the library header file (acl.h), the main library project (acl.uv2) and the library archive itself (acl.a).

4 Library verification and testing

Cryptographic libraries usually contain self-testing routines. In our case we did not make the tests part of the library, because they also serve the purpose of time measurement and they can also be considered examples of how to use this library.

Still, verification is a critical part of the design of any cryptographic library. There are two different approaches: test vectors (known answer tests) and random input tests (verifying a theorem).

Test vectors verify the accuracy of an implementation: does it provide the correct answer? But they do so only for a small number of inputs (small coverage). This coverage can be increased by having a large number of iterations before the answer is arrived at.

Random inputs tests test the consistency of an implementation: are encryption and decryption inverses? Does an implementation verify a signature that it has generated?

4.1 Test vectors

Most cryptographic primitives come with pre-computed known answers to known inputs. This approach was used when testing the AES, SHA and EC point compression. The AES was tested using test vectors that include 20 000 iterations of the cipher (although the tables go all the way to 400 000 iterations) [33]. One of the tests of the SHA involves hashing a 1 000 000 byte message.

We included the test vectors in the program as “vects” or strings and used string to “vect” conversion routines to read them (see section 5.3.5).

4.2 Random inputs, algebraic theorems

Some cryptographic primitives were tested using known mathematical equations / theorems and random data. This was the case in:

The prime finding algorithm – here we used Fermat’s little theorem $a^p = a \pmod p$.

RSA – we encoded and decoded a message and checked whether it was the same as the original.

ECC – again, Fermat's little theorem – a point multiplied by its order is the same as the original.

ECDSA – generate a signature and see if the library verifies it.

Notice that internal consistency of the results doesn't mean that the results are actually accurate. Because of this, a combination of known answer and random input tests is necessary (as we did for example with ECC).

4.3 Serial port interface

Part of our assignment was to test the library using the OpenSSL toolkit [61]. We have instead tested it using the official test vectors and random inputs, which is a more rigorous test, but we have also provided a serial port interface that allows it to be interfaced to an external client / server.

Also, we have provided string conversion routines that should make interfacing straightforward.

We also used the serial port to output the measured times. We set up the timers to increment every microsecond and printed out the resulting times after converting them to decimal numbers.

5 Library components

The library consists of several modules with varying degrees of dependence on the rest of the library. The modules are presented here in logical order from least dependent to most dependent.

5.1 Advanced Encryption Standard (AES)

The AES is a widely used symmetric-key block cipher [29]. It was standardized by the NIST for U.S. government use in a very open and transparent selection and adoption process. The winning proposal adopted as the AES was Rijndael [27], [28].

Like any other block cipher, the AES cipher can be operated in different modes of operation [31]. We implemented the following modes: ECB, CBC and CTR. Further modes can be easily added, since the cipher modes are implemented as wrappers of the core encryption/decryption routines.

The AES implementation was written entirely in assembler and it doesn't depend on any other part of the library. The only part of the library that depends on it is the AES-based pseudo-random number generator (see section 5.5.2).

The function interface was chosen to be as straightforward as possible. A bigger library with multiple symmetric ciphers and modes will usually provide a more elegant and unified application programming interface (API).

Here are the AES function prototypes:

```
void acl_aes_key_en(vect key_out, vect key_in, size_t key_size);
void acl_aes_key_de(vect key_out, vect key_in, size_t key_size);
void acl_aes_ecb_en(vect4 out, vect4 in, vect exp_key, size_t key_size);
void acl_aes_ecb_de(vect4 out, vect4 in, vect exp_key, size_t key_size);
void acl_aes_cbc_en(vect4 out, vect4 in, vect exp_key, \
                    size_t key_size, vect4 state);
void acl_aes_cbc_de(vect4 out, vect4 in, vect exp_key, \
                    size_t key_size, vect4 state);
void acl_aes_cntr(vect4 out, vect4 in, vect exp_key, \
                  size_t key_size, vect4 counter);
```

5.1.1 Implementation details

The authoritative resource on implementing the Advanced Encryption Standard is of course FIPS publication 197 [30]. As suggested in that document, we represented the columns of the state as 32-bit words ([30], section 3.5). Also, we used the equivalent inverse cipher ([30], section 5.3.5), which means that there is an additional step when expanding the key for decryption. This can be seen in Table 1, where the times to expand a key for decryption are at least twice as long as for encryption.

Using a 256 x 32-bit look-up table for both directions (forward and inverse) made it possible to integrate the SubBytes, ShiftRows and MixColumns steps. A larger table was not necessary as the ARM instruction set allows for efficient shifting and cyclic rotates. To generate the look-up tables, we used the MATLAB implementation of AES by Prof. J. Buchholz [32].

The implementation is one of the few parts of the library that is not endianness-neutral, as it assumes a little-endian memory layout.

5.1.2 Implementation results

For testing we used the Monte Carlo test vectors submitted with the original Rijndael AES proposal [33]. The running times are listed in Table 1. The code size of our AES implementation is ≈ 4200 bytes.

Table 1 AES implementation timings

Key bit length	128	192	256
Key expansion for encryption [cycles]	972	1062	1269
Key expansion for decryption [cycles]	2446	2856	3383
Encrypt/decrypt 128-bit block [cycles]	1012	1196	1379
Throughput @ 60 MHz [kB/s]	926	784	680

5.2 Secure Hash Algorithm (SHA)

The SHA hash functions [34] (SHA-1, SHA-224, SHA-256, SHA-384, SHA-512) are NSA-designed cryptographic hash functions published by NIST and widely used in different cryptographic protocols. Table 2 lists some properties of the algorithms. Note that SHA-224 is identical to SHA-256, except that the resulting digest is truncated from 256 to 224 bits. The same is true for SHA-384 in relation to SHA-512.

Table 2 SHA function characteristics (all sizes in bits)

Algorithm	Output size	Internal state size	Block size	Word size	Rounds
SHA-1	160	160	512	32	80
SHA-256/224	256/224	256	512	32	64
SHA-512/384	512/384	512	1024	64	80

5.2.1 Implementation details

The authoritative document on SHA implementation is of course FIPS Publication 180-2 [35]. In our implementations we chose to not expand the W's for each round, but to instead keep them in a circular buffer, to save memory.

We chose to implement only message lengths that are multiples of 8 bits. The SHA algorithms can handle arbitrary bit-lengths, but most applications do not make use of this feature.

We have found that the implementation of SHA-256 was actually less complicated than that of SHA-1. This is due to the irregular structure of SHA-1 (a different function every 20 rounds). On the other hand, SHA-512 was pretty complicated, since it involves rotates of 64-bit words, which something not native to a 32-bit architecture.

The function interface for each SHA algorithm includes prototypes for three operations: initialization, update, and finish-up.

The initialization function resets the state variable. The update function needs to be called for each byte of the message to be hashed. Finally, to obtain the final hash, the finish-up function must be called (to finish any unprocessed data in the buffer).

The function prototypes are listed below. The number after vect is the size of the state array in 32-bit words and therefore the memory requirement of the respective SHA algorithm.

```
void acl_sha1_init(vect23 state);
void acl_sha1(vect23 state, byte data);
void acl_sha1_done(vect23 state);
void acl_sha224_init(vect26 state);
```

```
void acl_sha256_init(vect26 state);
void acl_sha256(vect26 state, byte data);       // (== acl_sha224)
void acl_sha256_done(vect26 state);             // (== acl_sha224_done)
void acl_sha384_init(vect68 state);
void acl_sha512_init(vect68 state);
void acl_sha512(vect68 state, byte data);       // (== acl_sha384)
void acl_sha512_done(vect68 state);             // (== acl_sha384_done)
```

5.2.2 Implementation results

The running times of the various SHA algorithms are listed in Table 3. The timings were obtained by timing the hashing of a 1 000 000 byte message and then calculating the average time per byte.

Note that the speeds of SHA-1 and SHA-256 are comparable. This is partly due to the fact that SHA-256 has only 64 rounds, where SHA-1 has 80.

Table 3 SHA implementation timings

Type	Time [μs/byte]	Throughput [kb/s]	Memory [bytes]	Code size [bytes]
SHA-1	1.360	717.8	92	652
SHA-256	1.524	640.8	104	812
SHA-512	2.431	401.6	272	1608

5.3 Common routines

We have found that a number of auxiliary routines were necessary for recurring operations that were required in C routines. For example copying an array or setting a bit could be done in C, but would take up too much code memory. These routines were all written in assembler.

5.3.1 Copy array

```
void acl_mov(vect res, vect src, size_t len);
void acl_mov32(vect res, uint val, size_t len);
```

The first routine copies “len” ints from “src” to “res”. The second one initializes array “res” to a 32-bit value “val”. This means that the higher (more significant) words of “res” are cleared to zero. This can be used to clear the entire array if “val” = 0.

5.3.2 Bit manipulation

```
uint acl_bit(vect a, uint pos, size_t len);
void acl_bit_set(vect a, uint pos);
void acl_bit_clr(vect a, uint pos);
```

We needed three routines for bit manipulation: the first one reads the value of a bit, the second sets a bit and the third one clears a bit. Note the “len” in the first function – this ensures that if a bit is read from a position that is beyond “len” 32-bit words, the result is zero.

5.3.3 Comparison

```
int acl_cmp(vect a, vect b, size_t len);
bool_t acl_zero(vect a, size_t len);
```

The first routine compares two arrays and returns 1 if “a” is greater than “b”, -1 if “a” is less than “b” and 0 if “a” equals “b”. The second routine returns TRUE if the array is zero, FALSE otherwise.

5.3.4 Convert number to string

```
void acl_hex2str_dec(bytes res, size_t len_r, vect a, size_t len);
```

Converts integer in “a” to a decimal string in “res”. This routine was used in tests that print the elapsed time.

```
void acl_hex2str_le(bytes res, vect a, size_t len);
```

Converts little-endian integer in “a” to a hexadecimal string in “res”. This routine was used in elliptic curve point compression routines.

5.3.5 Convert string to number

```
void acl_str2hex_le(vect res, size_t len, bytes str, size_t len_s);
void acl_str2hex_be(vect res, bytes str, size_t len);
```

These routines convert a string of hexadecimal characters in “str” to either a little-endian or a big-endian number in “res”. These routines were used in AES testing and elliptic curve point decompression.

5.3.6 Other routines

```
uint acl_ctz(vect a, size_t len);
```

Count trailing zeroes of “a”. Used for example in the Rabin-Miller test [43], or the modular square root [38]. In both cases we have to “let $p - 1 = 2^k m$, where m is odd”. This is essentially k = number of trailing zeroes of p – 1.

```
int acl_log2(vect a, size_t len);
```

Return position of highest non-zero bit. If the number is zero, return -1. Used during exponentiation and elliptic curve point multiplication to find the highest bit of exponent/multiplier.

```
void acl_rsh(vect a, uint k, size_t len);
```

Right shift “a” by “k” bits. Used for example in the modular square root.

```
uint acl_rev(uint a);
```

Return “a” with byte order reversed. This routine is currently unused, but is included here as it was necessary at one point to convert numbers from little-endian to big-endian.

5.4 Prime field arithmetic

Much of the library depends on GF(p) routines. Also, this is where the ARM7TDMI excels. So in a way, this part of the library is indispensable.

For us, the authoritative resource on efficient prime field arithmetic was the famous Chapter 14 of the Handbook of Applied Cryptography [2] and section 2.2 of the Guide to Elliptic Curve Cryptography [1]. In a way, this chapter closely mirrors these two quoted chapters and we will refer to them throughout.

5.4.1 Data structures

Multiple precision numbers are represented as arrays of 32-bit words (“vects”) of variable length. The layout is little endian both in terms of bytes as well as words. See [1], Fig. 2.1.

5.4.2 Modular addition and doubling

```
uint acl_p_mod_add(vect res, vect a, vect b, vect m, size_t len);
uint acl_p_mod_add32(vect res, vect a, uint b, vect m, size_t len);
```

The addition algorithm closely follows Algorithm 2.7 in [1]. As with other routines, there is a 32-bit version of the routine. Also, we have added a feature that turns the modular addition into a normal addition if m = 0.

```
uint acl_p_mod_dbl(vect a, uint k, vect m, size_t len);
```

Also, by optimizing the addition routine for “res” = “a” = “b”, we made a modular doubling routine. Note that the doubling routine does not have the m = 0 feature. Later we added a parameter, “k”, which determines how many times “a” should be doubled.

Finally, because of the needs of the long division routine (section 5.4.13) we made the addition and doubling routines return 1 if m was subtracted from the result and 0 otherwise.

5.4.3 Modular subtraction

```
void acl_p_mod_sub(vect res, vect a, vect b, vect m, size_t len);
void acl_p_mod_sub32(vect res, vect a, uint b, vect m, size_t len);
```

These routines are analogous to their addition counterparts. They follow Algorithm 2.8 in [1].

5.4.4 Modular halving

```
void acl_p_mod_hlv(vect a, uint k, vect m, size_t len);
```

This modular halving (see [1], eq. 2.2, p. 42) routine basically repeats “k” times:

If a mod 2 = 0 then a = a / 2 else a = (a + m) / 2.

5.4.5 Multiplication

```
void acl_p_mul(vect2 res, vect a, vect b, size_t len);
```

The multiplication of two numbers "a" and "b" of length "len" results in a number "res", twice the length.

We chose the product scanning form of multiplication ([1], Algorithm 2.10; Table 5), because it is superior to the operand scanning form ([1], Algorithm 2.9; Table 4) on the ARM.

Table 4 Operand scanning form of multiplication

(order in which elements A[i] B[j] are multiplied and added)

	A[0]	A[1]	A[2]	A[3]
B[0]	1	2	3	4
B[1]	5	6	7	8
B[2]	9	10	11	12
B[3]	13	14	15	16

Table 5 Product scanning form of multiplication

(order in which elements A[i] B[j] are multiplied and added)

	A[0]	A[1]	A[2]	A[3]
B[0]	1	3	6	10
B[1]	2	5	9	13
B[2]	4	8	12	15
B[3]	7	11	14	16

The reason is that both these forms have to perform two memory fetches, a multiplication and an addition in each iteration, but the operand scanning form has to additionally store the intermediate result, while the product scanning form doesn't have to do this. Since ARM is a RISC architecture, this extra step of storing the result makes a huge difference. If we were implementing multiplication on a processor that has read-modify-write type instructions, this might not be the case.

Next we had to decide between two possibilities of multiplying two 32-bit values to obtain a 64-bit value. One option is to use the umlal instruction. This instruction multiplies the values in two registers and adds the 64-bit result to a second set of two registers. The problem is that the carry from the 64-bit addition is not available, so a "zero" register has to be used to prevent an overflow. Note that we have to use a 96-bit sum (sum1-sum3), because we are adding along a "diagonal" (see Table 5) and the result quickly overflows 64 bits:

```
mov      zero, #0
umlal    sum1, zero, pro1, pro2
adds     sum2, zero
adc      sum3, #0
```

The other option was to use the umull instruction, which does not perform the 64-bit addition – it simply multiplies the values of two registers and stores the result in a second set of registers:

```
umull    tmp1, tmp2, pro1, pro2
adds     sum1, tmp1
adcs     sum2, tmp2
adc      sum3, #0
```

It turns out that the umlal block requires 10 cycles and 6 registers [12]. The umull block requires 9 cycles and 7 registers. Since we were able to get the extra register, we chose the umull instruction.

5.4.6 Squaring

```
void acl_p_sqr(vect2 res, vect a, size_t len);
```

Squaring is essentially a multiplication with "a" = "b", so the square (Table 6) is symmetric and we only need to calculate one half of the multiplications and then double the number that we obtain (see [1], Algorithm 2.13).

The problem is that this way, the products A[i]A[i] on the diagonal would be in the result twice. To solve this problem, we add all the products that are not on the diagonal; we add only one-half of each product on the diagonal; and we double the result at the end. The least significant bit of A[0]A[0] gets left out this way, so we add it at the very end.

Table 6 Product scanning form of squaring

(order in which elements A[i] A[j] are multiplied and added)

	A[0]	A[1]	A[2]	A[3]
A[0]	1			
A[1]	2	4		
A[2]	3	6	8	
A[3]	5	7	9	10

5.4.7 Montgomery reduction

```
void acl_p_mont_red(vect res, vect2 a, vect m, uint m_inv, size_t len);
```

To perform modular multiplication (squaring), we need to somehow reduce the resulting double length number to single length again. The easiest way to do this on the ARM is Montgomery reduction [36], [37]. We used a method that the authors of [37] would classify as “Separated Product Scanning form”. Separated, because we multiply first and then we pass the result to the reduction routine.

The library actually started out with an integrated Montgomery reduction. This had the advantage of requiring no intermediate storage – basically a prototype:

```
montgomery_multiplication(result, a, b, length);
```

The disadvantage was that there was no way to use a squaring when a = b. Also, the “fine integration” of multiplication and reduction, while reducing memory usage, was not conducive to further optimization, since the overhead required to perform two multiplications in a single loop was such that there were not enough registers to make use of LDM and STM instructions. Once this fundamental problem was recognized, we separated the two steps (multiplication and reduction) and things instantly improved.

The moral of the story is rather general:

Lesson Learned 1 On optimization on the ARM

Don’t “optimize” by merging things.

Merging two routines destroys the modularity of your library. Merging two loops isn’t worth it: you save one jump, but you lose precious registers that you need if you want to optimize using the LDM / STM (load / store multiple registers) instructions – which is really the best way to optimize on the ARM.

It should also be noted that Montgomery reduction is one of the major bottlenecks in this library. It dominates the execution time of the prime finding algorithm (section 5.6) and is a major component of any exponentiation (e.g. RSA decryption, section 5.7).

5.4.8 Modular exponentiation

```
void acl_p_mont_pre(vect r_mod_m, vect r2_mod_m, uint *m_inv, \
                    vect m, size_t len);
void acl_p_mont_exp(vect res, vect x, vect e, size_t len_e, vect m, \
                    vect3 tmp, uint m_inv, vect r2_mod_m, size_t len);
```

Once we have multiplication, squaring and Montgomery reduction, we can implement modular exponentiation ([2], Algorithm 14.94). The variables R, R^2 and $–m^{-1}$ are calculated by the first routine. The second one is the exponentiation itself.

5.4.9 Greatest common divisor (GCD)

```
bool_t acl_p_coprime(vect a, vect b, vect2 tmp, size_t len);
```

It was never actually necessary to return the *value* of the GCD of two numbers, only to check whether it is equal to one. So we integrated this test into a routine that only returns TRUE or FALSE. The test uses the extended binary Euclidean algorithm.

Algorithm 1 Coprimality test

Input: positive integers a, b.

Output: TRUE if gcd(a, b) = 1, FALSE otherwise

1. u = a, v = b
2. If (u mod 2 = 0) and (v mod 2 = 0) then return FALSE
3. If (u mod 2 = 1) and (v mod 2 = 1) then goto step 10
4. If (v mod 2 = 0) then swap pointers to u and v
5. Goto step 8
6. Swap pointers to u and v
7. u = u – v

8. t = number of trailing zeroes of u
9. u = u >> t (t-bit right shift)
10. If u > v then goto step 7
11. If u < v then goto step 6
12. If u = 1 return TRUE else return FALSE

5.4.10 Partial Montgomery inversion

```
int acl_p_mont_inv(vect res, vect a, vect m, vect3 tmp, size_t len);
```

The basic building block of our modular inversion routine is partial Montgomery inversion ([1], Algorithm 2.23). The actual assembler routine is slightly different than Algorithm 2, but this is its conceptual version.

Partial Montgomery inversion is basically the binary inversion algorithm ([1], Algorithm 2.22) without the halving. The advantage is that we don't have to keep halving two numbers x_1 and x_2, when at the end of the routine we are interested in only one of them. Instead of modular halving, we perform a much simpler left-shift operation.

We used multi-bit shifting, as this is one of ARM's strengths.

Algorithm 2 Partial Montgomery inversion over GF(p)

Input: a, m, where m is odd, 2 < m, 1 < a

Output: (x, k) where $x = a^{-1} 2^k \bmod m$

1. u = a, v = m, $x_1 = 1$, $x_2 = 0$, k = 0, swp = 0
2. if (u mod 2 = 0) then goto step 6, else goto step 10
3. Swap u and v, swap x_1 and x_2, let swp = 1 - swp
4. u = u – v
5. $x_1 = x_1 + x_2$
6. t = number of trailing zeroes of u
7. u = u >> t (t-bit right shift)
8. $x_2 = x_2 << t$ (t-bit left shift)
9. k = k + t

10. If u > v then goto step 4

11. If u < v then goto step 3

12. If u ≠ 1 then return NON-INVERTIBLE

13. If swp = 1 then return (x_1, k), else return (m - x_2, k)

5.4.11 Modular inversion

```
void acl_p_mod_inv(vect res, vect a, uint e, vect m, \
                   vect3 tmp, size_t len);
```

This routine first checks to see whether a = 0 (then it returns 0 – not invertible) or a = 1 (then it returns 2^e mod m). If not, it calls Algorithm 2 and then halves (or doubles) the result x = a^{-1} 2^k mod m until it is equal to res = a^{-1} 2^e mod m. So if this routine is called with e = 0, it returns the modular inversion. If it is called with e = 2 · 32 · word-length(m), it performs a Montgomery inversion.

5.4.12 Long division – remainder (modulo)

```
void acl_p_mod(vect res, vect a, size_t len_a, vect m, size_t len);
```

Used in various routines to adjust a number that is of the same order as the modulus, but it would be too dangerous to repeatedly subtract the modulus, since we do not know just how much bigger the number is – 10 subtractions would be ok, but a 1000? This routine is not used in speed-critical loops, only in places where it is run once.

5.4.13 Long division – quotient

```
void acl_p_div(vect a, size_t len_a, vect m, vect tmp, size_t len);
```

It is sad that this routine had to be included in this library. Everything was so elegant – no long division. But alas. This routine is used in only one place - the calculation of the decryption exponent “d” in the RSA algorithm (see section 5.7).

5.4.14 Modular square root

```
bool_t acl_p_sqrt(vect res, vect a, vect m, vect8 tmp, size_t len);
```

The modular square root is only needed when decompressing points on Elliptic curves. The algorithm used was copied from the Mersennewiki article [38].

5.4.15 Fast reduction

```
void acl_p_fr(vect res, vect2 a, list data, size_t len);
```

This routine is used in elliptic curve arithmetic over GF(p). It takes as an argument “data” which is a pointer to a list of exponents which define the fast reduction to use. For example, $m = (2^{256} - 2^{224} + 2^{192} + 2^{96} - 1)$ is encoded as the list (256, 224, ~192, ~96, 0).

Algorithm 3 Fast reduction over GF(p) (simplified)

Input: In[2n bits], list of exponents (e, ±exp1, ±exp2, … 0)

Output: Out[n bits] = In mod $(2^{e} \pm 2^{exp1} \pm 2^{exp2} \ldots \pm 1)$

1) Get first exponent – e
2) Out = In[top e bits]; In[top e bits] = 0
3) If Out = 0, then Out = In[bottom e bits]; return
4) For each next exponent ±exp: In = In ± (Out << exp)
5) Go to step 1

The list format is a little more complicated - it uses one’s complement instead of two’s complement and there is an additional input option. It is described in more detail in the source code of the routine. Suffice it to say that this routine can handle all the SECG-recommended GF(p) curves. For timings and discussion see Table 10.

5.5 Pseudo-random number generators (PRNGs)

A pseudo-random number generator is a deterministic algorithm that uses arithmetic methods to approximate a sequence of random bits. Cryptographically secure PRNGs are cryptographic primitives [4]. For a rigorous treatment see [31]. The PRNGs in this library have a unified interface – PRNG_function(vect res, size_t len). This allows PRNGs to be passed as parameters to routines that need them.

5.5.1 Linear congruential

```
void acl_prng_lc_init(uint seed);
void acl_prng_lc(vect res, size_t len);
```

This PRNG is included because of its small size and high speed. It is used for bootstrapping purposes (to initialize the other generators) in the absence of a better (e.g. hardware) random number generator. It is also used in probabilistic primality tests, as all that is required there is that the values generated not be the same. Otherwise it is not considered safe for cryptographic applications.

The linear congruence that was chosen (from [39]) is:

$$x_{n+1} = (279470273 \cdot x_n) \bmod (2^{32} - 5)$$

For each 32-bit word of array "res", a new 32-bit value x_n is calculated and stored in "res".

5.5.2 AES in counter mode

```
void acl_prng_aes_init(prng rnd);
void acl_prng_aes(vect res, size_t len);
```

This PRNG is an AES encryptor in counter mode initialized to a random key and a random counter value (for this a random number generator must be passed to the initialization routine).

For each 32-bit word of array "res", an AES encryption with inputs set to zero is performed and the top 32 bits of the result are stored in "res".

5.5.3 SHA of a counter

```
void acl_prng_sha_init(prng rnd);
void acl_prng_sha(vect res, size_t len);
```

This PRNG is the SHA-1 hash of a 160-bit counter initialized to a random value (for this a random number generator must be passed to the initialization routine).

For each 32-bit word of array "res", a SHA-1 hash of the counter is calculated and the top 32 bits of the result are stored in "res".

5.5.4 Blum-Blum-Shub

```
void acl_prng_bbs_init(prng rnd_fast, prng rnd_strong, vect7 tmp);
```

```
void acl_prng_bbs(vect res, size_t len);
```

The Blum-Blum-Shub PRNG [41] generates the following sequence of numbers:

$$x_{n+1} = x_n^2 \bmod (p \cdot q)$$

Where p and q are two primes both congruent to 3 modulo 4. To generate the primes and the initial value of x, the initialization routine needs random number generators – a strong one for the primes and a fast one for the primality tests.

For each bit of “res”, one iteration of the generator is run, and the LSB of the Montgomery representation of x determines the value stored in “res”.

5.5.5 Timings

The throughput performance of the individual PRNGs is listed in Table 7.

Table 7 Performance of Pseudo-Random Number Generators

PRNG	Throughput [kb/s]
LC	8050
AES	226
SHA-1	56.2
BBS	0.961

Note that the LC works on a single 32-bit value, AES is being run with a 128-bit key, SHA-1 hashes a 160-bit counter, and BBS runs on a 512-bit modulus. The BBS PRNG requires a squaring and a Montgomery reduction for each bit it generates. More than one bit could be read off of the state variable after each iteration, with a proportionate increase in throughput.

5.6 Finding large prime numbers

Various cryptographic primitives require the finding (generation) of random large primes. The most obvious example would be RSA, but for example the Blum-Blum-Shub PRNG also needs large primes to work. For related work see [42].

5.6.1 The Rabin-Miller primality test

```
bool_t acl_p_rm_test2(vect m, vect3 tmp, uint m_inv, \
                      vect r_mod_m, size_t len);
bool_t acl_p_rm_test(vect a, vect m, vect4 tmp, uint m_inv, \
                     vect r_mod_m, vect r2_mod_m, size_t len);
```

For more on the Rabin-Miller test see [43]. The reason why we first perform the Rabin-Miller test with a = 2 is that modular exponentiation (square-and-multiply method) is much faster if instead of a full modular multiplication, we multiply by 2, as this is equivalent to a modular addition.

5.6.2 Prime finding algorithm

```
void acl_p_rnd_prime(vect res, vect7 tmp, uint k, uint also_set, \
                     prng rnd_fast, prng rnd_strong, size_t len);
```

The algorithm that we implemented is essentially the same as the one in [4], section 4.14, p. 24.

The “also_set” parameter can be used to set an additional bit in the prime generated. It is used in the BBS PRNG to make sure that the prime generated is congruent to 3 modulo 4 and in RSA to make sure that the two most significant bits of the primes are set. This is to ensure that their product is full-length.

Algorithm 4 Finding a large probable prime

Input: random number generator, Rabin-Miller parameter k, number “also_set”

Output: probable prime N

1. Generate a random number N
2. N = N + 2
3. Set the LSB, MSB, and bit at position “also_set”
4. If N has a small divisor, go to step 2.

   (~80% of candidates fail here; ~5% of time spent here)
5. If N fails Rabin-Miller test with a = 2, go to step 2.

   (~20% of candidates fail here; ~70% of time spent here)
6. If N fails any one of k Rabin-Miller tests with random a, go to step 2.

(~0% of candidates fail here; ~25% of time spent here for k = 8 Rabin-Miller test iterations)

7. Return N

5.6.3 Timing results

For each bit length 10 - 30 primes were generated. Note that this is a small statistical sample, so the results are not too reliable. The average times are listed in Table 8. The standard deviation was about t / 2.

Table 8 Average time to generate an m-bit prime with k = 8 (Rabin-Miller test iterations)

m [bits]	192	256	384	512	768	1024
t [s]	0.310	0.388	1.14	4.01	12.5	30.1

Around 70% of the time is spent in step 5. The breakdown of the time spent there (as measured by the µVision3 Performance Analyzer) is:

⅔ of the time: Montgomery reduction ~ 1 modular multiplication

⅓ of the time: squaring ~ ½ of a modular multiplication

Each squaring is followed by a Montgomery reduction, but a squaring takes only about half the time a multiplication takes, so the reduction becomes the bottleneck.

Table 8 can be summarized in the formula $t = (670 + 30\ k) \cdot (m / 32)^3\ \mu s$, where k is the number of Rabin-Miller tests with random “a” to be performed.

In our tests, we chose k = 8. Note that this is overkill, since in our tests of ~14000 candidates, none we rejected by the Rabin-Miller test with “a” random (step 6 of Algorithm 4). Also note that the random Rabin-Miller tests add a significant amount of time to the test (25% for k = 8), while contributing little more than peace of mind. This means that they are more a verification of the primality rather than part of the finding of a prime.

Another possible improvement would be to implement more special Rabin-Miller tests with small primes (3, 5, 7, …) to speed up the verification phase.

5.6.4 Implementation details

To perform the small – divisor test of step 4, note that we can test multiple small prime divisors p_1, p_2, p_3, ... p_n by pre-computing their product “PoP” (product of the first n primes) and then calculating gcd(N, PoP), where N is the candidate being tested. If the resulting greatest common divisor is > 1, then at least one of the primes $p_1 \dots p_n$ is a divisor of N and thus N is composite.

Since our gcd routine only accepts inputs that are of the same length, we pre-calculated a table of products-of-primes for each length from 1 to 32.

To calculate the “PoP” table, we used CALC, a scripting language that supports modular arithmetic [64].

The side-effect to the fact that the PoP grows with growing bit-length was that the rejection ratio of the small divisor test grew from 79% at m = 256 to 83% at m = 1024.

5.6.5 Asymptotic behavior

Theoretically, we would expect a $t \sim m^4$ dependence. The probability of a number being prime is proportional to $1 / \log(N) = 1 / \log(2^m) \sim 1 / m$, so to find a prime we need to test ~ m candidates.

The dominant step is the Rabin-Miller test with a = 2, which requires m squarings. Each squaring in turn takes an amount of time proportional to m^2.

So we have t ~ (candidates to test) (squarings per candidate) (time per squaring) ~ $(m)\ (m)\ (m^2) \sim m^4$.

This discrepancy between our best fit (m^3) and theory (m^4) is probably due to the fact that our measurements were done for small m and on a relatively small statistical sample. We still expect the asymptotic behavior to be $t \sim m^4$.

5.7 The RSA algorithm

For an introduction to RSA see the famous RSA paper [44] or the Wikipedia article [45]. Encryptions and decryptions in RSA are essentially modular exponentiations (ct is the ciphertext and pt the plaintext):

$ct = pt^e \bmod n$ encryption using public key (e, n)

$pt = ct^d \bmod n$ decryption using private key (d, n)

N is the product of two large primes p and q. Notice that some plaintexts do not encode very well: pt = 0 or 1 produces ct = pt. This is why in real life an additional padding scheme is necessary. To speed up encryption, a small exponent is chosen.

5.7.1 Chinese remainder theorem speed-up

To speed up decryption, the Chinese remainder theorem (CRT, [2], Note 14.75) is used. The CRT allows the modular exponentiation modulo n (= p · q) to be calculated using two modular exponentiations modulo p and q. Since p and q are half as long as n, the CRT-based method is theoretically 4 times faster. Assuming that the time to perform an exponentiation is proportional to m^3, where m is the bit-length of p and q:

$$Speedup = \frac{Exp(2m)}{2Exp(m)} = \frac{k(2m)^3}{2km^3} = \frac{8km^3}{2km^3} = 4$$

The CRT speed-up requires some additional values to be calculated (see [46]) during RSA key generation:

Algorithm 5 RSA key generation

Output: Public key (e, n), Private key (d, n) or (p, q, dmp1, dmq1, iqmp)

1. Choose e
2. Find two large random primes p, q
3. n = p · q
4. $\phi = (p - 1)(q - 1) = n - p - q + 1$
5. If $\gcd(e, \phi) \neq 1$ then goto step 2
6. $t = \phi^{-1} \bmod e$
7. $t = t \cdot \phi$
8. t = t - 1
9. t = t / e
10. $d = -t \bmod \phi$
11. dmp1 = d mod (p – 1)
12. dmq1 = d mod (q – 1)
13. $iqmp = q^{-1} \bmod p$

Note that steps 6 through 10 in the preceding algorithm calculate d = e^{-1} mod ϕ in a very complicated fashion. This was necessary, as ϕ is an even number and our inversion routine requires modular halving, which is not possible with an even modulus. This is also the only place where long division (quotient calculation) was necessary. This is how it works:

$$e \cdot \left[e^{-1}(\bmod \phi)\right] + \phi \cdot \left[\phi^{-1}(\bmod e)\right] = 1(\bmod e)$$

Therefore,

$$e^{-1}(\bmod \phi) = \frac{1 - \phi \cdot \left[\phi^{-1}(\bmod e)\right]}{e}$$

In our library, the two primes have to be generated first. Then they are passed along with e to the following routine:

```
bool_t acl_rsa_pre(vect2 n, vect2 d, vect dmp1, vect dmq1, vect iqmp, \
                   vect2 e, vect p, vect q, vect6 tmp, size_t len);
```

This routine takes as an input p, q and e and calculates the remaining values necessary for RSA encryption, decryption and decryption using the Chinese remainder theorem (CRT):

Algorithm 6 RSA decryption using CRT

Input: ciphertext ct, RSA private key (p, q, dmp1, dmq1, iqmp)

Output: plaintext pt

1. s_p = ct mod p
2. $s_p = s_p^{dmp1}$ mod p
3. s_q = ct mod q
4. $s_q = s_q^{dmq1}$ mod q
5. t = $s_p - s_q$
6. t = (t · iqmp) mod p
7. t = t · q
8. pt = t + s_q

This is achieved by the following routine:

```
void acl_rsa_crt(vect2 pt, vect2 ct, \
                 vect p, vect r2_mod_p, uint p_inv, \
                 vect q, vect r2_mod_q, uint q_inv,
                 vect dmp1, vect dmq1, vect iqmp, vect4 tmp, size_t len);
```

5.7.2 Implementation timings

The timings for the various RSA operations are listed in Table 9.

Table 9 RSA operation timings [ms]

Bit length of n	512	768	1024	1536	2048
Encode (e = 65537)	3.61	7.50	12.8	27.7	48.1
Decode	122	387	892	2874	6668
Decode with CRT	41.4	119	261	804	1814

Encryption with e = 65537: $t \approx 12.5\ (m / 32)^2\ \mu s$

Decryption: $t \approx 26.3\ (m / 32)^3\ \mu s$

Decryption using CRT speed-up: $t \approx 7.5\ (m / 32)^3\ \mu s$

Note that the CRT method is ~ 3.5 times faster than normal decryption as opposed to the theoretical / asymptotic 4 times faster.

5.8 Binary field arithmetic

Elliptic curve cryptography over binary fields requires basic field operations over $GF(2^m)$. This section closely follows the treatment in [1], chapter 2.3. We chose a polynomial basis representation (as opposed to a normal basis representation). Polynomials $a_n z^n + \dots a_1 z + a_0$ are stored as "vects", each bit representing the coefficient that multiplies the corresponding power of z. Operations over $GF(2^m)$ also require a reduction polynomial, which plays the same role as the modulus in GF(p).

5.8.1 Addition

```
void acl_xor(vect res, vect a, vect b, size_t len);
void acl_xor32(vect res, vect a, uint b, size_t len);
```

Addition and subtraction over GF(2^m) are identical to the an exclusive or (xor) operation. The first routine xors “a” and “b” and puts the result in “res”. The 32-bit version of this routine xors only the least significant word of “a” and copies the result to “res”. We will denote this addition operation as ⊕.

5.8.2 Division by z (“halving”)

```
void acl_2_mod_hlv(vect a, uint k, vect poly, size_t len);
```

This routine basically repeats “k” times:

If a mod z = 0 then a = a / z else a = (a ⊕ poly) / z

Here “poly” is the reduction polynomial. Note that this is identical to the halving over GF(p) (5.4.4), but instead of an addition, we perform an exclusive or.

5.8.3 Multiplication

```
void acl_2_mul(vect2 res, vect a, vect b, size_t len);
```

This routine is identical to the multiplication over GF(p) (5.4.5), except that instead of multiplying A[i] and B[j] as integers, we have to multiply them as binary field elements (see [1], section 2.3.2). This in turn is the same as integer multiplication (shift-and-add) except that instead of adding, we have to exclusive or. The following code fragment shows how this is done. The 32-bit polynomials to be multiplied are in registers “pro1” and “pro2”. The intermediate result is in registers “res2” and “res1”.

```
adds    pro1, pro1
eorcs   res1, pro2
adds    res1, res1
adc     res2, res2
```

The first instruction copies the MSB of “pro1” into the carry and shifts “pro1” left by one bit. If the carry is set, the second instruction xors “pro2” into the intermediate result. The next two instructions shift the 64-bit intermediate result 1 bit to the left. This is repeated 32 times for each bit of “pro1”. This method multiplies (1 bit) · (32 bits) in 4 cycles = 8 bits^2 / cycle.

Another method would be to store the field elements in such a way that data bits would be separated by 3 zero bits:

8 bits in a 32-bit word: 000H000G000F000E000D000C000B000A (binary)

This would take up four times as much space, but would allow the following multiplication:

```
umlal   res1, res2, pro1, pro2
and     res1, mask
and     res2, mask
```

The first instruction multiplies 8 bits by 8 bits and adds them to the intermediate result. The next two instructions (mask = 00010001000100010001000100010001 binary) return the intermediate result back into the sparse format. This method multiplies (8 bits) · (8 bits) in 9 cycles = 7.1 bits^2 / cycle.

In our case, the first method wins. But on a different processor (even an ARM other than ARM7TDMI), the results may vary.

Still, this is a far cry from the (32 bits) · (32 bits) in 9 cycles = 113.8 bits^2 / cycle that the code fragment in integer multiplication achieves. As a result, we can expect the GF(2^m) multiplication to be ~ 10 times slower than GF(p) multiplication.

Binary field multiplication could benefit from the use of Karatsuba-Ofman multiplication [47]. It is much easier to add binary field elements (exclusive or) than prime field elements (carry propagation, reduction). Still, the minor speed-up did not justify the added memory requirements and complexity for us.

5.8.4 Squaring

```
void acl_2_sqr(vect2 res, vect a, size_t len);
```

Squaring in binary fields with a polynomial representation is essentially inserting zeroes between the bits of the polynomial. A byte (HGFEDCBA) squared would look like this: (0H0G0F0E0D0C0B0A) (binary).

This is usually done via a lookup table. We could have used a 256 entry table that converts a byte into the corresponding 16-bit value, but since squaring is not a bottleneck, we chose instead to have a 16 entry table that converts a 4-bit value into the corresponding byte.

5.8.5 Partial Montgomery inversion

```
int acl_2_mont_inv(vect res, vect a, vect poly, vect3 tmp, size_t len);
```

This routine is analogous to its GF(p) counterpart (section 5.4.10). The difference is that we do not have to worry about subtraction, since it equivalent to addition and therefore to the exclusive or operation

Algorithm 7 Partial Montgomery inversion over GF(2^m)

Input: a, poly, where poly mod z = 0, 1 < a

Output: (x, k) where $x = a^{-1} z^k$ mod poly

1. u = a, v = m, $x_1 = 1$, $x_2 = 0$, k = 0, swp = 0
2. if (u mod z = 0) then goto step 6, else goto step 1010
3. Swap u and v, swap x_1 and x_2, let swp = 1 - swp
4. $u = u \oplus v$
5. $x_1 = x_1 \oplus x_2$
6. t = number of trailing zeroes of u
7. u = u >> t (t-bit right shift)
8. $x_2 = x_2$ << t (t-bit left shift)
9. k = k + t
10. If u = 1 then if swp = 1 then return (x_1, k), else return (x_2, k)
11. If u > v then goto step 4
12. If u < v then goto step 3
13. Return NON-INVERTIBLE

5.8.6 Modular inversion

```
void acl_2_mod_inv(vect res, vect a, vect poly, vect3 tmp, size_t len);
```

This routine first checks to see whether a = 0 (then it returns 0 – not invertible) or a = 1 (then it returns 1). If not, it calls Algorithm 7 and then divides the result $x = a^{-1} z^k$ by z k times to obtain the modular inversion.

5.8.7 Fast reduction

```
void acl_2_fr(vect res, vect2 a, list data, size_t len);
```

This routine is used in elliptic curve arithmetic over $GF(2^m)$. It takes as an argument “data” which is a pointer to a list of exponents which define the fast reduction to use. For example, the reduction polynomial $(z^{571} + z^{10} + z^5 + z^2 + 1)$ is encoded as the list (571, 10, 5, 2, 0).

Algorithm 8 Fast reduction over $GF(2^m)$

Input: In[2n bits], list of exponents (e, exp1, exp2, … 0)

Output: Out[n bits] = In mod $(z^e + z^{exp1} + z^{exp2} \ldots + 1)$

1) Get first exponent – e
2) Out = In[top e bits]; In[top e bits] = 0
3) If Out = 0, then Out = In[bottom e bits]; return
4) For each next exponent exp: In = In ⊕ (Out << exp)
5) Go to step 1

For timings and discussion see Table 11.

5.9 Elliptic curve cryptography (ECC)

For an introduction to ECC, see the Wikipedia article [48], the Certicom online tutorial [49], and [51]. The authoritative resource for software implementations of ECC is [1].

5.9.1 Group formulation

For our purposes, an elliptic curve is a set of points (x, y) that fulfill the following equations:

$y^2 = x^3 + ax + b \pmod m$ for prime fields

$y^2 \oplus xy = x^3 \oplus ax^2 \oplus b$ (modulo reduction polynomial p) for binary fields

The number of points on an elliptic curve #E varies between these bounds:

$$q + 1 - 2\sqrt{q} \leq (\#E) \leq q + 1 + 2\sqrt{q} \qquad \textbf{Equation 1}$$

where q = p or 2^m (this is known as Hasse's theorem). This is important when trying to store this number, since it can take up an additional 32-bits of storage if $q \approx 2^{32k}$.

We can turn the points on an elliptic curve into elements of an additive group, if we can find: an identity element, an inverse element for each element (a negative) and an addition operation.

The identity element is called the point at infinity, is denoted "O", and is often represented as (0, 0). Note that (0, 0) will not be confused with an actual point on the curve as long as b is non-zero.

The negative of a point (x, y) is (x, m – y) for prime fields and $(x, x \oplus y)$ for binary fields. Group theory requires that the negative also be a group element. This means that the negative point must also lie on the curve. That this is true can be easily verified.

Every point on the curve has its "order" – if we multiply a point by its order, we get the original point. The order of a point has to divide the number of points of the curve.

5.9.2 Point addition

The addition operation required to create a group is defined in this way:

Algorithm 9 Addition of two points $P \neq Q$ on an elliptic curve (prime, binary field)

Input: points $P(x_P, y_P)$ and $Q(x_Q, y_Q)$

Output: point $R(x_R, y_R) = P + Q$

1. $\lambda = (y_Q - y_P) / (x_Q - x_P)$
2. $x_R = \lambda^2 - x_P - x_Q$
3. $y_R = \lambda(x_P - x_R) - y_P$

or:

1. $\lambda = (y_Q \oplus y_P) / (x_Q \oplus x_P)$
2. $x_R = \lambda^2 \oplus \lambda \oplus x_P \oplus x_Q \oplus a$
3. $y_R = \lambda(x_P \oplus x_R) \oplus x_R \oplus y_P$

Notice that the division in step 1 of Algorithm 9 is not possible if $x_Q = x_P$. This case needs to be handled separately.

Let us now assume that we are trying to add two points with the same x-coordinate. Since a point's negative also has the same x-coordinate, there are two possibilities:

- The two points have different y-coordinates – we are adding a point and its negative and so the result is O, the point at infinity.
- The two points have the same y-coordinate – we are doubling a point. We need a special algorithm for this case.

Algorithm 10 Doubling a point P on an elliptic curve (prime, binary field)

Input: point $P(x_P, y_P)$

Output: point $R(x_R, y_R) = P + P$

1. $\lambda = (3x_P^2 + a) / (2y_P)$
2. $x_R = \lambda^2 - 2x_P$
3. $y_R = \lambda(x_P - x_R) - y_P$

or:

1. $\lambda = x_P \oplus (y_P / x_P)$
2. $x_R = \lambda^2 \oplus \lambda \oplus a$
3. $y_R = x_P^2 \oplus (\lambda \oplus 1)\, x_R$

Using these two operations (doubling and addition) we can multiply a point P by any positive integer k to obtain the point kP (see [1], section 3.3).

The problem of reversing this operation – to determine k given kP and P – is known as the elliptic curve discrete logarithm problem (ECDLP). It is the difficulty of calculating the discrete logarithm that allows elliptic curves to be used in public-key cryptographic protocols.

5.9.3 Recommended curves

The problem of finding an elliptic curve suitable for cryptography is beyond the scope of this paper. Suffice it to say that there are curves that are recommended by various groups / agencies. For example:

- NIST (U.S. Department of Commerce, National Institute of Standards and Technology) recommended curves [50].
- SECG (Standards for Efficient Cryptography Group) recommended curves [52].

The NIST curves are actually a subset of the SECG curves. It was our assignment to implement all the SECG-recommended curves. To store the information about individual curves, we created the following C struct:

```
typedef struct {
    const char *s;      // name of curve
    uint t;             // type of curve (see below)
    size_t l;           // length of m, g (2x), a, b in 32-bit words
    vect m;             // pointer to modulus or reduction polynomial
    list fr;            // pointer to fast reduction data
    vect2 g;            // pointer to base point (x, y, affine)
    vect a;             // pointer to a
    vect b;             // pointer to b
    vect n;             // order of base point
    size_t ln;          // length of order in 32-bit words
    uint h;             // cofactor - currently has no effect
    void *f;            // pointers to field specific ecc functions
} ecc_t;
```

This struct essentially stores pointers to all the relevant information about a curve. ECC routines are simply passed a pointer to this struct for a given curve.

Each curve recommendation also contains a “base point” which is chosen randomly and is used in cryptographic protocols to establish a common starting point.

In our library, the directory “Curves” stores the definitions of all the SECG curves.

Some of the SECG-recommended curves are members of a special class of curves, called “Koblitz curves” after Neal Koblitz ([1], section 3.4). These curves allow very efficient point doubling. We did not make use of this special feature and treated these curves as ordinary elliptic curves.

5.9.4 Timings of prime field operations

We have already described all the field operations that are required to implement a point multiplication on an elliptic curve. The timings of prime field operations for SECG-recommended curves are listed in Table 10.

Table 10 Prime field operation timings (in cycles)

M = multiplication, FR = fast reduction, I = inversion

Curve	M	FR	I	FR/M	I/M
secp112r1	369	427	19170	1.16	52
secp112r2	369	427	19660	1.16	53
secp128r1	369	937	19415	2.54	53
secp128r2	369	926	19664	2.51	53
secp160k1	540	552	26434	1.02	49
secp160r1	540	550	26936	1.02	50
secp160r2	540	552	27576	1.02	51
secp192k1	743	591	36670	0.80	49
secp192r1	743	566	35942	0.76	48
secp224k1	975	658	46775	0.67	48
secp224r1	975	665	47958	0.68	49
secp256k1	1237	696	58285	0.56	47
secp256r1	1237	3066	59432	2.48	48
secp384r1	2583	1314	116999	0.51	45
secp521r1	4945	1061	212098	0.21	43

The shaded fields in Table 10 show the cases where fast reduction is slower than multiplication. This means that in these cases our general “fast” reduction routine has failed in speeding up reduction and is in fact slower than the general Montgomery reduction (which would take about the same time as a multiplication).

The reasons for this lackluster performance are:

- Overhead. The routine is being run with small bit-lengths and the overhead is still a factor (secp112, secp160).
- Unfortunate choice of exponents. These reduction polynomials have a large second exponent which means that the fast reduction requires many iterations (secp128, secp256r1).

Possible solutions were:

- Allow user to select choice between Montgomery and fast reduction. The problem is that this would require major structural changes to the library (coordinates would have to be converted to / from the Montgomery domain).
- Allow the user to supply a dedicated fast reduction routine. This too would require structural changes to the library and the user always has the option of optimizing the general fast reduction routine for a single curve.
- Do nothing. The slowdown is not that big and a common fast reduction routine saves code memory and prevents a “balkanization” of fast reduction routines.

Needless to say, we chose the last option. Being able to use a curve (albeit slowly and inefficiently) at no additional cost is a Good Thing sometimes.

5.9.5 Timings of binary field operations

The timings of binary field operations for SECG-recommended curves are listed in Table 11.

Notice that multiplication over binary fields is about 10 times slower than over prime fields. Also, modular inversion in binary fields is not as fast as it could be if it were optimized for a specific curve (see [1], p. 60). This is due to a general, but very slow modular “halving” (division by z) routine (see 5.8.2).

Table 11 Binary field operation timings (in cycles)

M = multiplication, FR = fast reduction, I = inversion

Curve	M	FR	I	FR/M	I/M
sect113r1	2551	505	29064	0.20	11
sect113r2	2551	505	29203	0.20	11
sect131r1	3939	880	40740	0.22	10
sect131r2	3939	881	40522	0.22	10
sect163k1	5631	903	57879	0.16	10
sect163r1	5631	903	56885	0.16	10
sect163r2	5631	904	56313	0.16	10
sect193r1	7627	726	75509	0.10	10
sect193r2	7627	726	76443	0.10	10
sect233k1	9927	806	100844	0.08	10
sect233r1	9927	806	100892	0.08	10
sect239k1	9927	1188	101726	0.12	10
sect283k1	12531	1205	130938	0.10	10
sect283r1	12531	1205	131458	0.10	10
sect409k1	25987	1136	262051	0.04	10
sect409r1	25987	1136	261278	0.07	10
sect571k1	49647	1896	490443	0.04	10
sect571r1	49647	1896	490017	0.00	10

5.9.6 Projective coordinates

Notice that both the point addition and point doubling algorithms require a modular inversion. But the cost of a modular inversion is prohibitingly large. This is expressed in the ratio I/M (inversion to multiplication) – in our case, for prime fields, this ratio is about 50 (Table 10, last column) and for binary fields, it is about 10 (Table 11, last column).

To avoid having to calculate an inversion during every point operation, projective coordinates were introduced ([1], section 3.2.1).

A point in projective coordinates is represented by three numbers (x, y, z). It corresponds to the affine (2-coordinate) point (x / z^c, y / z^d). With projective coordinates, instead of dividing x and y by a value, we multiply z by a (different) value in such a way that the ratios stay the same.

This allows us to trade an inversion for a few multiplications. If the ratio I/M were smaller than the number of multiplications that projective coordinates require, projective coordinates would not be necessary.

The idea is to take a point in affine coordinates and convert it into projective coordinates. This is done by setting z = 1: (x, y) → (x, y, 1). Then operations (additions, doublings) are done in projective coordinates, which is faster. At the end we convert the point back to affine coordinates by doing a single inversion of z and multiplying x and y by z^{-1} the appropriate number of times: (x, y, z) → ($x \cdot z^{-c}$, $y \cdot z^{-d}$). These conversions are done by the following routines:

```
void acl_ecc_pro(vect3 a, vect2 b, size_t len);
void acl_p_ecc_aff(vect3 a, vect4 tmp, ecc_t *c);
void acl_2_ecc_aff(vect3 a, vect5 tmp, ecc_t *c);
```

Notice that points in affine coordinates are stored as two consecutive “vects” - x, y. Points in projective coordinates are stored as three consecutive “vects” – x, y, z.

To represent the point at infinity (O) in projective coordinates, we chose to use the point (x, y, 0), so any point with z = 0 represents the point at infinity.

Much work (not ours) has gone into finding the best choice of c and d (the exponents of z) in projective coordinates.

5.9.7 EC point arithmetic over GF(p)

For curves over prime fields, based on [1], Table 3.3, we chose Jacobian projective coordinates, where (x, y, z) corresponds to (x / z^2, y / z^3). This choice leads to the following point addition and doubling algorithms. They are taken from [1], Algorithms 3.21 and 3.22, p. 91, with minor modifications.

The C prototype for point doubling is:

```
void acl_p_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c);
```

Algorithm 11 Point doubling in Jacobian coordinates over GF(p)

Input: P(x, y, z), parameters a, b of curve $y^2 = x^3 + ax + b$

Output: (x, y, z) = 2P

1. If z = 0 then return(x, y, z) (since 2 O = O)
2. If a = 0 then
   - 2.1. $t_1 = x^2$
   - 2.2. $t_2 = 2\ t_1$
3. Else if a = -3 then
   - 3.1. $t_1 = z^2$
   - 3.2. $t_2 = x – t_1$
   - 3.3. $t_1 = t_1 + x$
   - 3.4. $t_1 = t_1 \cdot t_2$
   - 3.5. $t_2 = 2\ t_1$
4. Else
   - 4.1. $t_2 = z^2$
   - 4.2. $t_2 = t_2^2$
   - 4.3. $t_2 = t_2 \cdot a$
   - 4.4. $t_1 = x^2$
   - 4.5. $t_2 = t_2 + t_1$
   - 4.6. $t_1 = 2\ t_1$
5. $t_2 = t_2 + t_1$
6. y = 2 y
7. $z = z \cdot y$
8. $y = y^2$
9. $t_1 = y \cdot x$
10. $y = y^2$
11. y = y / 2
12. $x = t_2^2$
13. $x = x – t_1$
14. $x = x – t_1$
15. $t_1 = t_1 – x$
16. $t_1 = t_1 \cdot t_2$
17. $y = t_1 – y$
18. Return(x, y, z)

Notice that some values of “a” lead to a speed-up. For this reason, many curves choose a = -3 (this is in reality a = m – 3, since this is modular arithmetic).

Point addition is complicated if we try to add two points in projective coordinates. It is faster and simpler to only add points in projective and affine coordinates. The following algorithm adds “Jacobian + Affine = Jacobian”. The C prototype is:

```
void acl_p_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c);
```

Algorithm 12 Point addition (Jacobian = Jacobian + Affine) over GF(p)

Input: P(x, y, z), Q(X, Y), parameters a, b of curve $y^2 = x^3 + ax + b$

Output: (x, y, z) = P + Q

1. If X = 0 and Y = 0 then return (x, y, z) (since P + O = P)
2. If z = 0 then return (X, Y, 1) (since O + Q = Q)
3. $t_1 = z^2$
4. $t_2 = t_1 \cdot z$
5. $t_1 = t_1 \cdot X$
6. $t_2 = t_2 \cdot Y$
7. $t_1 = t_1 – x$
8. $t_2 = t_2 – y$
9. If $t_1 = 0$ (if P and Q have the same x-coordinate) then

9.1. If $t_2 = 0$ (if P and Q have the same y-coordinate) then return the result of Algorithm 11 with (x, y, z) or (X, Y, 1) as input (since P = Q, it follows that P + Q = 2P = 2Q)

9.2. Return (x, y, 0) (since the two points have the same x-coordinate, their sum is O)

10. $z = z \cdot t_1$
11. $t_3 = t_1^2$
12. $t_1 = t_1 \cdot t_3$
13. $t_3 = t_3 \cdot x$
14. $x = t_2^2$
15. $x = x - t_3$
16. $x = x - t_3$
17. $x = x - t_1$
18. $t_3 = t_3 - x$
19. $t_3 = t_3 \cdot t_2$
20. $t_1 = t_1 \cdot y$
21. $y = t_3 - t_1$
22. Return (x, y, z)

5.9.8 EC point arithmetic over GF(2^m)

For curves over binary fields, based on [1], Table 3.4, we chose López-Dahab projective coordinates, where (x, y, z) corresponds to (x / z, y / z^2). This choice leads to the following point addition and doubling algorithms. They are taken from [1], Algorithms 3.24 and 3.25, pp. 94-95, with minor modifications.

The C prototype for point doubling is:

```
void acl_2_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c);
```

Algorithm 13 Point doubling in López-Dahab coordinates over GF(2^m)

Input: P(x, y, z), parameters a, b of curve $y^2 + xy = x^3 + ax^2 + b$

Output: (x, y, z) = 2P

1. If z = 0 then return(x, y, 0) (since 2 O = O)
2. $t_1 = z^2$
3. $t_2 = x^2$
4. $z = t_1 \cdot t_2$
5. $x = t_2^2$
6. $t_1 = t_1^2$
7. $t_2 = b \cdot t_1$
8. $x = x + t_2$
9. $y = y^2$
10. $t_1 = a \cdot z$
11. $y = y + t_1$
12. $y = y + t_2$
13. $y = y \cdot x$
14. $t_1 = z \cdot t_2$
15. $y = y + t_1$
16. Return(x, y, z)

Again, just like with Jacobian coordinates, point addition is complicated if we try to add two points in projective coordinates. It is faster and simpler to only add “López-Dahab + Affine = López-Dahab”. The C prototype is:

```
void acl_2_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c);
```

Algorithm 14 Point addition in López-Dahab coordinates over GF(2^m)

Input: P(x, y, z), Q(X, Y), parameters a, b of curve $y^2 + xy = x^3 + ax^2 + b$

Output: (x, y, z) = P + Q

1. If X = 0 and Y = 0 then return (x, y, z) (since P + O = P)
2. If z = 0 then return (X, Y, 1) (since O + Q = Q)
3. $t_1 = z \cdot X$
4. $t_2 = z^2$
5. $x = x + t_1$
6. $t_1 = z \cdot x$
7. $t_3 = t_2 \cdot Y$
8. $y = y + t_3$
9. If x = 0 (if P and Q have the same x-coordinate) then
   - 9.3. If y = 0 (if P and Q have the same y-coordinate) then return the result of Algorithm 13 with (X, Y, 1) as input (since P = Q, it follows that P + Q = 2P = 2Q; but we have already modified x and y so we cannot pass P)
   - 9.4. Return (x, y, 0) (since the two points have the same x-coordinate, their sum is O)
10. $z = t_1^2$
11. $t_3 = t_1 \cdot y$
12. $t_2 = t_2 \cdot a$
13. $t_1 = t_1 + t_2$
14. $t_2 = x^2$
15. $x = t_1 \cdot t_2$
16. $t_2 = y^2$
17. $x = x + t_2$
18. $x = x + t_3$
19. $t_2 = X \cdot z$
20. $t_2 = t_2 + x$
21. $t_1 = z^2$
22. $t_3 = t_3 + z$
23. $y = t_2 \cdot t_3$
24. $t_2 = X + Y$
25. $t_3 = t_1 \cdot t_2$
26. $y = y + t_3$
27. Return (x, y, z)

5.9.9 EC point multiplication

```
void acl_ecc_pre(vectN pre, vect2 p, uint w, uint s, vect8 tmp, \
                 ecc_t *c);
void acl_ecc_mul(vect3 res, vect p, vect q, uint w, uint s, \
                 vect k, vect l, size_t len_kl, vect5 tmp, ecc_t *c);
```

ECDSA (section 5.10) requires the computation of a linear combination of two points kP + lQ. EC point multiplication is analogous to exponentiation in RSA. But since an elliptic curve forms an additive group, instead of a square-and-multiply algorithm, we have a double-and-add algorithm.

The method we used to speed up this multiplication is a comb with interleaving (see [1], section 3.3). The first routine does the pre-computation required for a comb of width “w” and spacing “s”. This has to be done for both points if two points are used. The number of bytes required by such a table is (2 · length) · (2^w – 1), where length is the number of bytes required to store one coordinate.

The multiplication routine takes two points and two multipliers and calculates the linear combination. The routine can also be used with only one point (if “q” = 0) or without pre-computation (if “w” = 1).

To test the EC arithmetic, we used the MAGMA online calculator [65].

5.9.10 EC point compression

```
void acl_p_ecc_p2str(bytes str, vect2 a, bool_t comp, vect tmp, \
                     ecc_t *c);
void acl_2_ecc_p2str(bytes str, vect2 a, bool_t comp, vect5 tmp, \
                     ecc_t *c);
bool_t acl_p_ecc_str2p(vect2 a, bytes str, vect9 tmp, ecc_t *c);
bool_t acl_2_ecc_str2p(vect2 a, bytes str, vect6 tmp, ecc_t *c);
```

We implemented routines that allow the conversion of EC points from/to a string. These routines use point compression as described in [51]. To test this routine, we used the string representations of base points in [52].

5.10 Elliptic Curve Digital Signature Algorithm (ECDSA)

```
void acl_ecdsa_gen(vect r, vect s, vect e, size_t len_e, vect dA, \
                   vectN base, uint wi, uint sp, \
                   prng rnd_strong, vect9 tmp, ecc_t *c);
bool_t acl_ecdsa_ver(vect r, vect s, vect e, size_t len_e, vectN qA, \
                     vectN base, uint wi, uint sp, vect10 tmp, ecc_t *c);
```

ECDSA is the equivalent of the Digital Signature Algorithm over elliptic curves. The authoritative document is FIPS publication 186-2, [53]. For a more accessible summary see [1], section 4.4.1, or [54]. This is the only ECC-based protocol that we implemented in this library, but the framework is in place and other protocols can be easily added.

The timings of our implementation are listed in Table 12.

Table 12 Timings of ECDSA signature generation (G) and verification (V) with or without pre-computation (P) (comb with w=4); all times in milliseconds

Curve	G	G+P	V	V+P
secp112r1	24	10	32	14
secp112r2	26	10	32	14
secp128r1	39	16	58	24
secp128r2	45	18	62	26
secp160k1	41	17	59	26
secp160r1	44	18	61	27
secp160r2	45	18	59	27
secp192k1	58	21	82	32
secp192r1	62	22	85	33
secp224k1	83	34	113	51
secp224r1	89	31	126	47
secp256k1	108	39	150	60
secp256r1	248	87	354	135
secp384r1	336	117	467	179
secp521r1	654	240	938	368

Curve	G	G+P	V	V+P
sect113r1	64	25	94	40
sect113r2	65	26	92	40
sect131r1	114	49	164	77
sect131r2	117	48	171	77
sect163k1	154	63	233	104
sect163r1	193	77	274	124
sect163r2	163	68	246	110
sect193r1	286	111	386	180
sect193r2	276	111	407	180
sect233k1	328	132	500	222
sect233r1	364	140	546	234
sect239k1	351	140	550	230
sect283k1	501	194	798	326
sect283r1	589	208	855	340
sect409k1	1363	528	2109	888
sect409r1	1534	587	2360	944
sect571k1	3603	1334	5623	2277
sect571r1	4162	1477	6262	2394

The comb with w = 4 was on average 2.5 times faster than normal multiplication. Further improvement is possible with larger combs at the cost of exponentially larger memory requirements.

These timings were obtained by simulation on an ARM7TDMI processor at 60MHz as an average of four measurements.

6 Code size, comparisons

The codesizes (ARM and Thumb mode) of the individual modules of our library are listed in Table 13.

Table 13 Codesize of the library modules in bytes

Module	ARM	Thumb
AES	4128	4128
SHA	3072	3072
Common	1196	1196
GF(p)	6836	5504
Primes	3372	2904
PRNG	1008	684
RSA	876	520
GF(2^m)	1612	1528
Curves	6844	6844
ECC	8988	5456
Total	37932	31836

To put the timings and codesize of our library into perspective, here is some data that we were able to find in the literature.

Our main source for comparison is the Wakan Crypto Toolkit [62]. This is a multi-platform cryptographic library written in C. We hope that quoting their publicly available information falls under "fair use".

The Wakan Crypto Toolkit, as described in [62], was tested on a 20 MHz ARM processor. We simulated our library at 60 MHz. The following normalizations were necessary to adjust the Wakan data to our conditions:

AES: (20 kb/s at 20 MHz, 256 bits data) · (60 MHz / 20 MHz) · (256 bits data / 128 bits data) = 120 kb/s.

SHA-1: (88 kb/s at 20 MHz) · (60 MHz / 20 MHz) = 264 kb/s.

RSA: (13853 ms at 20 MHz) · (20 MHz / 60 MHz) = 4618 ms.

RSA key generation: (50.3 s at 20 MHz) · (20 MHz / 60 MHz) = 16.8 ms.

Table 14 Comparison with the Wakan Crypto Toolkit on the ARM

	Speed/time		Codesize (bytes)	
	ACL	Wakan	ACL	Wakan
AES-256	680 kB/s	120 kB/s	4128	2952
SHA-1	717.8 kB/s	264 kB/s	652	1204
RSA decrypt w/ CRT (2048 bits)	1814 ms	4618 ms	~9900	~13000
1024-bit RSA key generation	~8 s	~17 s		

Another comparison that we can make is with two master's theses:

E. Turan [57] reports an ECDSA signature generation on the B-233 (sect233r1) curve in 76.6 ms using Karatsuba multiplication and no pre-computation.

Normalizing: (76.6 ms reported at 80 MHz) · (80 MHz / 60 MHz) = 102.1 ms.

H.K. Tanik [58] reports an ECDSA signature generation on the P-224 (secp224r1) curve in 106.96 ms using Montgomery reduction and no pre-computation.

Normalizing: (106.96 ms reported at 80 MHz) · (80 MHz / 60 MHz) = 142.6 ms.

Table 15 Comparison with published ECDSA generation timings on the ARM

Curve	ACL [ms]	In [57] [ms]	In [58] [ms]
B-233	364	102.1	-
P-224	89	-	142.6

The B-233 implementation is faster than ours because it uses Karatsuba multiplication and is optimized for a single curve.

The P-224 implementation is slower than ours because it uses Montgomery reduction where fast reduction would be a faster option.

We can conclude that the performance of our library is comparable to the published results for cryptography on the ARM7TDMI. Perhaps we are approaching the state of the art.

7 Conclusion

We have implemented a small, general and fast cryptographic library for the ARM7TDMI architecture. The low-level routines are written in assembler and callable from C. It supports the following cryptographic primitives:

- AES (128, 192, 256)
- SHA (1, 224, 256, 384, 512)
- Pseudo-random number generation (LC, AES, SHA, BBS)
- Prime number generation
- RSA (with CRT)
- ECC (all the SECG-recommended curves, with point compression)
- ECDSA

All this in 38 kilobytes of code (32 kB in Thumb mode).

The library is written to run on “bare metal”. It is optimized for both speed and code size. It uses freely available development tools – the GNU toolchain and the μVision3 IDE (evaluation version).

The main lesson from ARM assembler optimization is to avoid merging things (routines, loops).

A lot of time and effort was saved by writing a single fast reduction routine for all elliptic curves (actually two – one for prime and one for binary fields).

Further optimization is possible in ECC if a single curve is selected – the general fast reduction routine can be replaced by a dedicated one (at the cost of losing generality).

Further work could be done to speed up Koblitz curves, since these are treated as ordinary curves.

Binary field multiplication could be implemented using the Karatsuba-Ofman algorithm (at the cost of more memory).

Inversion over binary fields can be made faster. A major part of it is modular halving, but this can be made faster only for “suitable” curves. If the ratio I/M (which is now at 10) could be made smaller, the use of projective coordinates could be reconsidered.

Bibliography

[1] D. HANKERSON, A. MENEZES, AND S.A. VANSTONE, Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004.

[2] A. MENEZES, P. VAN OORSCHOT AND S. VANSTONE, Handbook of Applied Cryptography, CRC Press, 1997. Chapters available online at www.cacr.math.uwaterloo.ca/hac

[3] D. KAHN, The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet. Charles Scribner's Sons, New York. 1996.

[4] P. GARRETT, Cryptographic Primitives, in *Public-key cryptography : American Mathematical Society short course*, January 13-14, 2003, Baltimore, Maryland. Available online at: www.math.umn.edu/~garrett/crypto/overview.pdf

[5] A. D. WOODBURY, D. V. BAILEY, AND C. PAAR. Elliptic Curve Cryptography on Smart Cards without Coprocessors. *Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, September 2000. Bristol, UK.

[6] G. GAUBATZ, J.-P. KAPS, E. ÖZTÜRK AND B. SUNAR, State of the Art in Ultra-Low Power Public Key Cryptography for Wireless Sensor Networks. *2nd IEEE International Workshop on Pervasive Computing and Communication Security (PerSec 2005)*, Kauai Island, Hawaii, March 2005.

[7] NICKLAUS WIRTH, Algorithms + Data Structures = Programs, Prentice-Hall, 1975.

[8] F. P. BROOKS, The Mythical Man Month, Addison Wesley, 1995.

[9] ARM, ARM Architecture Reference Manual, ARM DDI0100, available online at www.altera.com/literature/third-party/ddi0100e_arm_arm.pdf

[10] ARM, Procedure Call Standard for the ARM Architecture, ARM AAPCS, available online at www.arm.com/miscPDFs/8031.pdf

[11] ARM, ARM Instruction Set Quick Reference Card, ARM QRC0001I, available online at www.arm.com/documentation/Instruction_Set/index.html

[12] ARM, ARM7TDMI Product Overview, ARM DVI0027B, available online at www.arm.com/documentation/ARMProcessor_Cores/index.html

[13] T. MARTIN, The Insider's Guide to the NXP ARM7 Based Microcontrollers, Hitex Ltd., available online at www.hitex.co.uk/arm/lpc2000book

[14] ARM, ARM Programming Techniques, ARM DUI0021A, available online at www.ee.ic.ac.uk/pcheung/teaching/ee2_computing/arm/Progtech.pdf

[15] KEIL, ARM Development Tools, available online at www.keil.com/arm

[16] ECLIPSE IDE, available online at www.eclipse.org

[17] J.P. LYNCH, ARM Cross Development with Eclipse, available online.

[18] KEIL, RTX Real-Time Kernel, description at www.keil.com/rl-arm/kernel.asp

[19] FreeRTOS, Free Real-Time Kernel, available online at www.freertos.org

[20] M. Thomas, WinARM, a GNU toolchain for Windows, available online at www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects

[21] CodeSourcery, GNU Toolchain for ARM Processors, available online at www.codesourcery.com/gnu_toolchains/arm

[22] ARM, Application Note 150, available online at www.arm.com/pdfs/AN150B_Building_Linux_Applications_with_RVCT.zip

[23] Keil, MCB2130 Evaluation Board, description at www.keil.com/mcb2130

[24] J. Bán, M. Varchola, Digital Filtering on the ARM7TDMI Architecture, Semestral project, KEMT, Technical University of Kosice, May 2006.

[25] NXP, LPC21xx Product Description, available online at www.nxp.com/pip/LPC2132FBD64.html

[26] Keil Software, µVision3 Quick Start, available online at www.keil.com/product/brochures/uv3.pdf

[27] J. Daemen, V. Rijmen, The Design of Rijndael, Springer-Verlag, 2001.

[28] J. Daemen, V. Rijmen, AES Proposal: Rijndael, available online at csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf

[29] Wikipedia, Advanced Encryption Standard, available online at en.wikipedia.org/wiki/Advanced_Encryption_Standard

[30] NIST, FIPS Publication 197, Advanced Encryption Standard, available online at csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[31] NIST, Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation, available online at csrc.nist.gov/CryptoToolkit/modes/800-38_Series_Publications/SP800-38A.pdf

[32] J. J. Buchholz, MATLAB implementation of the Advanced Encryption Standard, available online at buchholz.hs-bremen.de/aes/aes.htm

[33] J. Daemen, V. Rijmen, Rijndael Test Vectors, available online at csrc.nist.gov/CryptoToolkit/aes/rijndael/rijndael-vals.zip

[34] Wikipedia, SHA hash functions, available online at en.wikipedia.org/wiki/SHA_hash_functions

[35] NIST, FIPS Publication 180-2, Secure Hashing Standard, available online at csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[36] P. Montgomery, Modular Multiplication without Trial Division, Mathematics of Computation, vol. 44, no. 170, Apr. 1985, pp. 519-521.

[37] Ç.K. Koç, T. Acar, and B.S. Kaliski Jr., Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro, 16(3):26–33, June 1996.

[38] MersenneWiki, Modular square root, available online at mersennewiki.org/index.php/Modular_Square_Root

[39] P. L'ECUYER, Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure, *Mathematics of Computation*, Vol. 68, No. 225 (Jan., 1999), pp. 249-260

[40] NIST, Recommendation for Random Number Generation Using Deterministic Random Bit Generators, available online at csrc.nist.gov/publications/nistpubs/ 800-90/SP800-90revised_March2007.pdf

[41] L. BLUM, M. BLUM, M. SHUB, A simple unpredictable random number generator, SIAM Journal on Computing 15 (1986), 364–383.

[42] R.L. RIVEST, Finding Four Million Random Primes. Advances in Cryptology, Crypto '90, Springer-Verlag, 1991. 625-626.

[43] D. MARKER, Rabin-Miller Primality Test, available online at www.math.uic.edu/ ~marker/math435/rm.pdf

[44] R.L. RIVEST, A. SHAMIR, AND L. ADLEMAN, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, Vol.21, Nr.2, 1978, S.120-126, available online at theory.lcs.mit.edu/~rivest/rsapaper.pdf

[45] WIKIPEDIA, RSA Algorithm, available online at en.wikipedia.org/wiki/RSA

[46] S.M. YEN, D. KIM, Cryptanalysis of Two Protocols for RSA with CRT Based on Fault Infection, Proc. Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC '04), L. Breveglieri and I. Koren, eds., pp. 381-385, 2004.

[47] A. KARATSUBA AND YU OFMAN, Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk SSSR Vol. 145 (1962), pp. 293–294. Translation in Physics-Doklady 7 (1963), pp. 595–596.

[48] CERTICOM, Online Elliptic Curve Cryptography Tutorial, available online at www.certicom.com

[49] WIKIPEDIA, Elliptic curve cryptography, available online at en.wikipedia.org/wiki/ Elliptic_curve_cryptography

[50] NIST, Recommended Elliptic Curves for Government Use, July 1999. Available online at csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.pdf

[51] SECG, Standards for Efficient Cryptography 1, Elliptic Curve Cryptography, available online at www.secg.org/download/aid-385/sec1_final.pdf

[52] SECG, Standards for Efficient Cryptography 2, Recommended Elliptic Curve Domain Parameters, available online at www.secg.org/download/aid-386/ sec2_final.pdf

[53] NIST, FIPS Publication 186-2, Digital Signature Standard, available online at csrc.nist.gov/publications/ fips/fips186-2/fips186-2-change1.pdf

[54] WIKIPEDIA, Elliptic Curve DSA, available online at en.wikipedia.org/wiki/ Elliptic_Curve_DSA

[55] M.BROWN, D.HANKERSON, J.LÓPEZ, A.MENEZES. Software implementation of the NIST elliptic curves over prime fields. In: Progress in Cryptology CT-RSA 2001, D.Naccache, editor, vol 2020, LNCS, pp. 250-265, 2001.

[56] T. HASEGAWA, J. NAKAJIMA, AND M. MATSUI. A practical implementation of elliptic curve cryptosystems over GF (p) on a 16-bit microcomputer. In Public Key Cryptography PKC '98, volume 1431 of Lecture Notes in Computer Science, pages 182-194. Springer-Verlag, 1998.

[57] E. TURAN, ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(2^k). M.S. Thesis, Department of Electrical & Computer Engineering, Oregon State University, June 15, 2001.

[58] H. K. TANIK. ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(p). M.S. Thesis, Department of Computer Science, Oregon State University, June 19, 2001.

[59] SHAMUS SOFTWARE, MIRACL (Multiprecision Integer and Rational Arithmetic C/C++ Library) available online at www.shamus.ie

[60] TOM ST. DENIS, LibTomCrypt, available online at libtom.org

[61] OpenSSL, available online at www.openssl.org

[62] ASCOM, Wakan Crypto Software Toolkit, www.ascom.com/secsol/art/wakan_art

[63] The GNU Crypto Project, available online at www.gnu.org/software/gnu-crypto

[64] CALC, C-style arbitrary precision calculator, available online at sourceforge.net/ projects/calc/

[65] MAGMA, High performance software for Algebra, Number Theory, and Geometry, available online at magma.maths.usyd.edu.au/magma

Appendices

Appendix A: CD containing the electronic version of this document and the library

Appendix B: ARM Cryptographic Library Source Code

Appendix B: ARM Cryptographic Library Source Code

Source file 1 acl.h

```
/*
                   ARM Cryptographic Library
    -------------------------------------------------------
    Author: Jaroslav Ban      Supervisor: Milos Drutarovsky

    Department of Electronics and Multimedia Communications
           Technical University of Kosice (Slovakia)

    Version: 1.00                Last revision: 2007-04-24
*/

#ifndef ACL_H
#define ACL_H

/*
 Notes:
 - certain parts of this library may be covered by certain patents
   in certain coutries; if you want to use said parts in said coutries,
   make sure to first obtain a license from the respective patent holder(s)
 - most routines will work if you use them in-place (result == input)
 - some routines will not work with "len" small (assuming len ~ 4 and more)
 - AES routines are byte-order dependent (little-endian)
 - most routines assume that m (the modulus) is odd,
   and won't work properly with m even
*/

#define TRUE -1
#define FALSE 0

/* The vect type is an array of 32-bit words with the LSW first (offset +0)
   and the MSW last (offset +4*(length-1)), its size indicated by "len".
   The following types "vectN" imply an N-times bigger array than vect. */
typedef unsigned int * vect;
typedef unsigned int * vect2;
typedef unsigned int * vect3;
typedef unsigned int * vect4;
typedef unsigned int * vect5;
typedef unsigned int * vect6;
typedef unsigned int * vect7;
typedef unsigned int * vect8;
typedef unsigned int * vect9;
typedef unsigned int * vect10;
typedef unsigned int * vect11;
typedef unsigned int * vectN;

/* The following vect types indicate the size of the array in ints. */
typedef unsigned int * vect16;
typedef unsigned int * vect23;
typedef unsigned int * vect26;
typedef unsigned int * vect68;

typedef char byte;
```

```
typedef char * bytes;
typedef unsigned int uint;      // 32-bit integers
typedef int bool_t;             // booleans
typedef unsigned int size_t;    // lengths (in multiples of 32-bits)
typedef unsigned int * list;    // various lists

/* Pseudo-random number generator function type */
typedef void (* prng)(vect, size_t);

/* Lengths of "vects" (bits -> 32-bit words) */
#define ACL_64 2
#define ACL_96 3
#define ACL_128 4
#define ACL_192 6
#define ACL_224 7
#define ACL_256 8
#define ACL_384 12
#define ACL_512 16
#define ACL_768 24
#define ACL_1024 32
#define ACL_1536 48
#define ACL_2048 64
#define ACL_4096 128

/* AES */
void acl_aes_key_en(vect key_out, vect key_in, size_t key_size);
                                                // expand key for encryption
void acl_aes_key_de(vect key_out, vect key_in, size_t key_size);
                                                // expand key for decryption
void acl_aes_ecb_en(vect4 out, vect4 in, vect exp_key, size_t key_size);
                                                // encrypt in ecb mode
void acl_aes_ecb_de(vect4 out, vect4 in, vect exp_key, size_t key_size);
                                                // decrypt in ecb mode
void acl_aes_cbc_en(vect4 out, vect4 in, vect exp_key, \
                    size_t key_size, vect4 state);  // encrypt in cbc mode
void acl_aes_cbc_de(vect4 out, vect4 in, vect exp_key, \
                    size_t key_size, vect4 state);  // decrypt in cbc mode
void acl_aes_cntr(vect4 out, vect4 in, vect exp_key, \
                  size_t key_size, vect4 counter); // encrypt in counter mode

/* SHA */
void acl_sha1_init(vect23 state);
void acl_sha1(vect23 state, byte data);
void acl_sha1_done(vect23 state);
void acl_sha224_init(vect26 state);
void acl_sha256_init(vect26 state);
void acl_sha256(vect26 state, byte data);       // (== acl_sha224)
void acl_sha256_done(vect26 state);             // (== acl_sha224_done)
void acl_sha384_init(vect68 state);
void acl_sha512_init(vect68 state);
void acl_sha512(vect68 state, byte data);       // (== acl_sha384)
void acl_sha512_done(vect68 state);             // (== acl_sha384_done)

// Functions with a '32' suffix are versions of the original functions where
// one of the operands is a 32-bit value (the more significant bits being zero).

/* The Commons */
void acl_mov(vect res, vect src, size_t len);         // res = src
```

```
void acl_mov32(vect res, uint val, size_t len);       // res = val[32-bit]
void acl_xor(vect res, vect a, vect b, size_t len);   // res = a xor b
void acl_xor32(vect res, vect a, uint b, size_t len); // res = a xor b[32-bit]
uint acl_ctz(vect a, size_t len);           // count trailing zeroes
int acl_log2(vect a, size_t len);           // position of highest non-zero bit
uint acl_bit(vect a, uint pos, size_t len); // return value of bit
void acl_bit_set(vect a, uint pos);         // set bit at given position
void acl_bit_clr(vect a, uint pos);         // clear bit at given position
int acl_cmp(vect a, vect b, size_t len);    // compare two arrays
bool_t acl_zero(vect a, size_t len);        // returns true if the array is zero
void acl_rsh(vect a, uint k, size_t len);   // a = a >> k
uint acl_rev(uint a);                   // return value with byte order reversed
void acl_hex2str_dec(bytes res, size_t len_r, vect a, size_t len);
                            // convert number to string(decimal)
void acl_hex2str_le(bytes res, vect a, size_t len);
                            // convert number (little endian) to string(hex)
void acl_str2hex_le(vect res, size_t len, bytes str, size_t len_s);
                            // convert string(hex) to number (little endian)
void acl_str2bytes(vect res, bytes str, size_t len);
                            // convert string(hex) to array of bytes
void acl_str2hex_be(vect res, bytes str, size_t len);
                            // convert string(hex) to number (big endian)

/* GF(p) */
uint acl_p_mod_add(vect res, vect a, vect b, vect m, size_t len);
                                                // res = (a + b) mod m
uint acl_p_mod_add32(vect res, vect a, uint b, vect m, size_t len);
                                                // res = (a + b[32-bit]) mod m
void acl_p_mod_sub(vect res, vect a, vect b, vect m, size_t len);
                                                // res = (a - b) mod m
void acl_p_mod_sub32(vect res, vect a, uint b, vect m, size_t len);
                                                // res = (a - b[32-bit]) mod m
uint acl_p_mod_dbl(vect a, uint k, vect m, size_t len); // a = a*(2^k) mod m
void acl_p_mod_hlv(vect a, uint k, vect m, size_t len); // a = a/(2^k) mod m
void acl_p_mul(vect2 res, vect a, vect b, size_t len);
                                            // res = a * b, res != a, res != b
void acl_p_sqr(vect2 res, vect a, size_t len);  // res = a * a, res != a
void acl_p_mod(vect res, vect a, size_t len_a, vect m, size_t len);
                                                // res = a mod m, res != a !!!
void acl_p_div(vect a, size_t len_a, vect m, vect tmp, size_t len);
                                                // a = a div m
bool_t acl_p_sqrt(vect res, vect a, vect m, prng rnd, vect8 tmp, size_t len);
                                                // res^2 = a mod m, res != a
void acl_p_fr(vect res, vect2 a, list data, size_t len);
                                        // res = fast reduction(a), res != a
bool_t acl_p_coprime(vect a, vect b, vect2 tmp, size_t len);
                                                // true if gcd(a,b) == 1
int acl_p_mont_inv(vect res, vect a, vect m, vect3 tmp, size_t len);
                // res = a^(-1)*(+-2^k) mod m, m odd, a!=0, a!=1, m mod 2 == 1
void acl_p_mod_inv(vect res, vect a, uint e, vect m, vect3 tmp, size_t len);
                            // res = a^(-1)*(2^e) mod m, m mod 2 == 1, res != a

/* Montgomery */
uint acl_p_mont_m_inv(vect m);              // returns -m^(-1) mod 2^32
void acl_p_mont_pre(vect r_mod_m, vect r2_mod_m, uint *m_inv, \
                    vect m, size_t len);    // precomputation for montgomery
void acl_p_mont_red(vect res, vect2 a, vect m, uint m_inv, size_t len);
                                            // res = a*r^(-1) mod m, res!=a !!!
```

```
void acl_p_mont_exp(vect res, vect x, vect e, size_t len_e, vect m, vect3 tmp, \
                    uint m_inv, vect r2_mod_m, size_t len);   // res = x^e mod m

/* Pseudorandom number generators */
void acl_prng_lc_init(uint seed);        // linear congruential prng
void acl_prng_lc(vect res, size_t len);  // for bootstrapping purposes
void acl_prng_aes_init(prng rnd);
void acl_prng_aes(vect res, size_t len);            // aes
void acl_prng_sha_init(prng rnd);
void acl_prng_sha(vect res, size_t len);            // sha-1
void acl_prng_bbs_init(prng rnd_fast, prng rnd, vect7 tmp);
                       // note that vect7 here means 7*ACL_PRNG_BBS_SIZE
void acl_prng_bbs(vect res, size_t len);            // blum-blum-shub

/* Primes */
extern uint *acl_pop_table;             // product-of-small-primes table
bool_t acl_p_rm_test(vect a, vect m, vect4 tmp, uint m_inv, \
                     vect r_mod_m, vect r2_mod_m, size_t len);
                                        // rabin-miller test
bool_t acl_p_rm_test2(vect m, vect3 tmp, uint m_inv, \
                      vect r_mod_m, size_t len);
                                        // rabin-miller test with a == 2
void acl_p_rnd_prime(vect res, vect7 tmp, uint k, uint also_set, \
                     prng rnd_fast, prng rnd_strong, size_t len);
                                        // generate random prime

/* RSA */
bool_t acl_rsa_pre(vect2 n, vect2 d, vect dmp1, vect dmq1, vect iqmp, \
                   vect2 e, vect p, vect q, vect6 tmp, size_t len);
                                        // n, d, dmp1, dmq1, iqmp = f(e, p, q)
void acl_rsa_crt(vect2 pt, vect2 ct, \
                 vect p, vect r2_mod_p, uint p_inv, \
                 vect q, vect r2_mod_q, uint q_inv,
                 vect dmp1, vect dmq1, vect iqmp, vect4 tmp, size_t len);
                                        // pt = rsa_inv(ct) (using crt)

/* GF(2^m) */
void acl_2_mul(vect2 res, vect a, vect b, size_t len);
                                        // res = a * b, res != a, res != b
void acl_2_sqr(vect2 res, vect a, size_t len);  // res = a * a, res != a
void acl_2_fr(vect res, vect2 a, list data, size_t len);
                                        // res = fast reduction(a), res != a
int acl_2_mont_inv(vect res, vect a, vect poly, vect3 tmp, size_t len);
                    // res = a^(-1)*z^k mod poly, a!=0, a!=1, poly mod z == 1
void acl_2_mod_hlv(vect a, uint k, vect poly, size_t len);
                                    // a = a/(z^k) mod poly, poly mod z == 1
void acl_2_mod_inv(vect res, vect a, vect poly, vect3 tmp, size_t len);
                        // res = a^(-1) mod poly, poly mod z == 1, res != a

/* ECC curve struct */
typedef struct {
    const char *s;      // name of curve
    uint t;             // type of curve (see below)
    size_t l;           // length of m, g (2x), a, b in 32-bit words
    vect m;             // pointer to modulus or reduction polynomial
    list fr;            // pointer to fast reduction data
    vect2 g;            // pointer to base point (x, y, affine)
    vect a;             // pointer to a or: 0, 1, -3
```

```
    vect b;             // pointer to b or: 0 .. ACL_MAX_B (see acl_config.h)
    vect n;             // order of base point
    size_t ln;          // length of order in 32-bit words
    uint h;             // cofactor - currently has no effect
    void *f;            // pointers to field specific ecc functions
} ecc_t;

/* ECC function struct */
typedef struct {
    bool_t (*chk)(vect, vect, ecc_t *);   // add point compression/decompression
    void (*dbl)(vect, vect, ecc_t *);
    void (*add)(vect, vect, vect, ecc_t *);
    void (*aff)(vect, vect, ecc_t *);
    void (*p2str)(bytes, vect, bool_t, vect, ecc_t *);
    bool_t (*str2p)(vect, bytes, vect, ecc_t *);
} ecc_func_t;

/* ECC curve type flags */
#define ECC_P 0     // curve is over GF(p)
#define ECC_2 1     // curve is over GF(2)
#define ECC_K 2     // koblitz - currently has no effect
#define ECC_A 4     // almost prime - used for acl_secp112r1 and acl_secp112r2
                    // the modulus c->m     = the almost prime,
                    //             c->m+len = the real prime.
                    // the reduction polynomial is for the almost prime.

#define ECC_F_MASK 1    // field mask
#define ECC_K_MASK 2    // koblitz mask
#define ECC_A_MASK 4    // almost prime mask

/* SECG-recommended curves */
extern const ecc_t                acl_secp112r1, acl_secp112r2;
extern const ecc_t                acl_secp128r1, acl_secp128r2;
extern const ecc_t acl_secp160k1, acl_secp160r1, acl_secp160r2;
extern const ecc_t acl_secp192k1, acl_secp192r1;
extern const ecc_t acl_secp224k1, acl_secp224r1;
extern const ecc_t acl_secp256k1, acl_secp256r1;
extern const ecc_t                acl_secp384r1;
extern const ecc_t                acl_secp521r1;

extern const ecc_t                acl_sect113r1, acl_sect113r2;
extern const ecc_t                acl_sect131r1, acl_sect131r2;
extern const ecc_t acl_sect163k1, acl_sect163r1, acl_sect163r2;
extern const ecc_t                acl_sect193r1, acl_sect193r2;
extern const ecc_t acl_sect233k1, acl_sect233r1;
extern const ecc_t acl_sect239k1;
extern const ecc_t acl_sect283k1, acl_sect283r1;
extern const ecc_t acl_sect409k1, acl_sect409r1;
extern const ecc_t acl_sect571k1, acl_sect571r1;

/* NIST-recommended curves */
#define P_192 acl_secp192r1
#define P_224 acl_secp224r1
#define P_256 acl_secp256r1
#define P_384 acl_secp384r1
#define P_521 acl_secp521r1

#define K_163 acl_sect163k1
```

```
#define K_233 acl_sect233k1
#define K_283 acl_sect283k1
#define K_409 acl_sect409k1
#define K_571 acl_sect571k1

#define B_163 acl_sect163r2
#define B_233 acl_sect233r1
#define B_283 acl_sect283r1
#define B_409 acl_sect409r1
#define B_571 acl_sect571r1

/* ECC function sets */
extern const ecc_func_t acl_p_ecc_func;
extern const ecc_func_t acl_2_ecc_func;

/* ECC points are represented thus:
   Jacobian/Lopez-Dahab coordinates: vect3 (x, y, z)
   Affine coordinates:               vect2 (x, y)      */

/* ECC arithmetic - GF(p) */
bool_t acl_p_ecc_chk(vect2 a, vect4 tmp, ecc_t *c);   // is point a on curve c?
void acl_p_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c); // a = a + b
void acl_p_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c);       // a = a + a
void acl_p_ecc_aff(vect3 a, vect4 tmp, ecc_t *c);       // (x,y,z) -> (x',y')

/* ECC arithmetic - GF(2^m) */
bool_t acl_2_ecc_chk(vect2 a, vect4 tmp, ecc_t *c);   // is point a on curve c?
void acl_2_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c); // a = a + b
void acl_2_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c);       // a = a + a
void acl_2_ecc_aff(vect3 a, vect5 tmp, ecc_t *c);       // a(x,y,z) -> a(x',y')

/* ECC arithmetic - generic */
void acl_ecc_pro(vect3 a, vect2 b, size_t len);         // b(x,y) -> a(x,y,1)
void acl_ecc_neg(vect3 a, ecc_t *c);                    // a = -a
void acl_ecc_pre(vectN pre, vect2 p, uint w, uint s, vect8 tmp, ecc_t *c);
                                                    // precomputation (comb)
void acl_ecc_mul(vect3 res, vect p, vect q, uint w, uint s, vect k, vect l, \
                 size_t len_kl, vect5 tmp, ecc_t *c);
                                            // linear combination of two points

/* ECC point <-> string conversion */
void acl_p_ecc_p2str(bytes str, vect2 a, bool_t comp, vect tmp, ecc_t *c);
                                // point a(x,y) -> string(comp/no comp)
void acl_2_ecc_p2str(bytes str, vect2 a, bool_t comp, vect5 tmp, ecc_t *c);
                                // point a(x,y) -> string(comp/no comp)
bool_t acl_p_ecc_str2p(vect2 a, bytes str, vect9 tmp, ecc_t *c);
                                // string(comp/no comp) -> point a(x,y)
bool_t acl_2_ecc_str2p(vect2 a, bytes str, vect6 tmp, ecc_t *c);
                                // string(comp/no comp) -> point a(x,y)

/* ECC protocols */
void acl_ecdsa_gen(vect r, vect s, vect e, size_t len_e, vect dA, \
                   vectN base, uint wi, uint sp, \
                   prng rnd_strong, vect9 tmp, ecc_t *c);
                                                    // generate ecdsa signature
bool_t acl_ecdsa_ver(vect r, vect s, vect e, size_t len_e, vectN qA, \
                     vectN base, uint wi, uint sp, vect10 tmp, ecc_t *c);
                                                    // verify ecdsa signature
```

```
/* Macros to make the ECC arithmetic more readable */
#define acl_ecc_chk(a, b, c) ((ecc_func_t *) c->f)->chk(a, b, c)
#define acl_ecc_dbl(a, b, c) ((ecc_func_t *) c->f)->dbl(a, b, c)
#define acl_ecc_add(a, b, c, d) ((ecc_func_t *) d->f)->add(a, b, c, d)
#define acl_ecc_aff(a, b, c) ((ecc_func_t *) c->f)->aff(a, b, c)
#define acl_ecc_p2str(a, b, c, d, e) ((ecc_func_t *) e->f)->p2str(a, b, c, d, e)
#define acl_ecc_str2p(a, b, c, d) ((ecc_func_t *) d->f)->str2p(a, b, c, d)

#endif
```

Source file 2 acl_config.h

```
#ifndef ACL_CONFIG_H
#define ACL_CONFIG_H

// Some magic numbers

// in routine acl_p_ecc_chk
#define ACL_MAX_B 9     // = maximum "small" value of b in ECC over GF(p)
                        // any larger value is considered a pointer
                        // the largest "small" value used in SECG curves is 7
                        // (secp160k1, secp256k1)

// in routines acl_p_ecc_chk and acl_2_ecc_chk
#define ACL_CHK_INF_ON_CURVE 0
    // determines how routines acl_p_ecc_chk and acl_2_ecc_chk treat
    // the point at infinity (x == 0, y == 0)
    //  0 -> acl_x_ecc_chk(point-at-infinity) returns FALSE
    //  1 -> acl_x_ecc_chk(point-at-infinity) returns TRUE
    // depending on where the acl_x_ecc_chk routines are used,
    // this can be used to make the point at infinity valid/invalid
    // or to force the user to check for the point at infinity separately

    // for example, in our library testing programs, the acl_x_ecc_chk routines
    // are only used to check for a valid base point. the point at infinity
    // is not a valid base point, so we set ACL_CHK_INF_ON_CURVE to 0.
    // this way we don't have to test for the point at infinity separately.

// There is an magic number in GF_p/acl_p_fr - called BORDER

// PRNG configuration
#define ACL_PRNG_AES_SIZE 4     // 4, 6, 8 = 128, 192, 256
#define ACL_PRNG_BBS_MONT 1     // get bits from montgomery representation
                                //   of x^(2i) (see acl_prng_bbs.c)
#define ACL_PRNG_BBS_SIZE 8     // length of p, q
#define ACL_PRNG_BBS_K 8        // rabin-miller parameter for BBS primes

// number of entries in acl_pop_table
// to change this, you must also generate a new table
#define ACL_POP_SIZE 32

#endif
```

Source file 3 acl_int.h

```
#ifndef ACL_INT_H
#define ACL_INT_H

/* Macros to make the Montgomery arithmetic more readable */
#define acl_p_mul_mont(out, in1, in2) acl_p_mul(tmp, in1, in2, len); \
                                      acl_p_mont_red(out, tmp, m, m_inv, len)
#define acl_p_sqr_mont(out, in)       acl_p_sqr(tmp, in, len);       \
                                      acl_p_mont_red(out, tmp, m, m_inv, len)

/* Macros to make the field arithmetic more readable */
#define acl_p_mul_sr(out, in1, in2) acl_p_mul(tmp, in1, in2, len); \
                                    acl_p_mod(out, tmp, 2*len, m, len)
#define acl_p_mul_fr(out, in1, in2) acl_p_mul(tmp, in1, in2, len); \
                                    acl_p_fr(out, tmp, fr, len)
#define acl_p_sqr_fr(out, in)       acl_p_sqr(tmp, in, len);       \
                                    acl_p_fr(out, tmp, fr, len)
#define acl_2_mul_fr(out, in1, in2) acl_2_mul(tmp, in1, in2, len); \
                                    acl_2_fr(out, tmp, fr, len)
#define acl_2_sqr_fr(out, in)       acl_2_sqr(tmp, in, len);       \
                                    acl_2_fr(out, tmp, fr, len)
#define xx a
#define xx1 a
#define xx2 b

#endif
```

Source file 4 acl_gen_tabs.m

```
function gen_tabs()
% script generating AES tables (forward, inverse)
% this code is taken from the following matlab implementation of aes:
% http://buchholz.hs-bremen.de/aes/aes.htm

[s_box, i_box] = s_box_gen;
mod_pol = bin2dec ('100011011');

% rcon
fid = fopen('acl_aes_rcon.txt', 'wt');
fprintf(fid, '@ aes rcon\n');
h = 1;
for i = 1:14
    fprintf(fid, '.byte 0x%02x\n', h);
    h = poly_mult (h, 2, mod_pol);
end
status = fclose(fid);

% forward sbox
fid = fopen('acl_aes_fwd_sbox.txt', 'wt');
fprintf(fid, '@ aes forward sbox\n');
for i = 1:64
    fprintf(fid, '.byte 0x%02x, 0x%02x, 0x%02x, 0x%02x\n', s_box(4*i-3), s_box(4*i-2), 
s_box(4*i-1), s_box(4*i));
end
status = fclose(fid);

% inverse sbox
fid = fopen('acl_aes_inv_sbox.txt', 'wt');
```

```
fprintf(fid, '@ aes inverse sbox\n');
for i = 1:64
    fprintf(fid, '.byte 0x%02x, 0x%02x, 0x%02x, 0x%02x\n', i_box(4*i-3), i_box(4*i-2),
i_box(4*i-1), i_box(4*i));
end
status = fclose(fid);

% forward table
fid = fopen('acl_aes_fwd_table.txt', 'wt');
fprintf(fid, '@ aes forward table (sub, mix)\n');
for i = 1:256
    h = s_box(i);
    r2 = poly_mult(h, 2, mod_pol);
    r3 = poly_mult(h, 3, mod_pol);
    fprintf(fid, '.int 0x%02x%02x%02x%02x\n', r3, h, h, r2);
end
status = fclose(fid);

% inverse table
fid = fopen('acl_aes_inv_table.txt', 'wt');
fprintf(fid, '@ aes inverse table (sub, mix)\n');
for i = 1:256
    h = i_box(i);
    rb = poly_mult(h, 11, mod_pol);
    rd = poly_mult(h, 13, mod_pol);
    r9 = poly_mult(h, 9, mod_pol);
    re = poly_mult(h, 14, mod_pol);
    fprintf(fid, '.int 0x%02x%02x%02x%02x\n', rb, rd, r9, re);
end
status = fclose(fid);
```

Source file 5 acl_aes_tables.s

```
@ tables used by the aes routines

                .global acl_aes_rcon
                .global acl_aes_fwd_sbox
                .global acl_aes_inv_sbox
                .global acl_aes_fwd_table
                .global acl_aes_inv_table
                .text
acl_aes_rcon:           .include "./aes/acl_aes_rcon.txt"       @ 14
acl_aes_fwd_sbox:       .include "./aes/acl_aes_fwd_sbox.txt"   @ 256
acl_aes_inv_sbox:       .include "./aes/acl_aes_inv_sbox.txt"   @ 256
                .align 2
acl_aes_fwd_table:      .include "./aes/acl_aes_fwd_table.txt"  @ 1k
acl_aes_inv_table:      .include "./aes/acl_aes_inv_table.txt"  @ 1k
                .end
```

Source file 6 acl_aes_key_en.s

```
@ void acl_aes_key_en(vect key_out, vect key_in, size_t key_size);
@   expands aes encryption key key_in to key_out
@   assuming a little endian processor
@ on entry:
@   r0 = pointer to expanded key (output)
@   r1 = pointer to key (input)
```

```
@   r2 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)

                .global acl_aes_key_en
                .text
                .arm

out             .req    r0      @ outputs
in              .req    r1      @ inputs
nk              .req    r2      @ nk
ff              .req    r3      @ mask value
tmp             .req    r4      @ temp
cnt             .req    r5      @ loop counter
sbox            .req    r6      @ pointer to sbox
rcon            .req    r7      @ pointer to rcon
rnd             .req    r8      @ round counter
acc             .req    r9      @ accumulator
st              .req    r12     @ substitution

acl_aes_key_en:
                push    {r4-r9}
                ldr     sbox, =acl_aes_fwd_sbox
                ldr     rcon, =acl_aes_rcon
                mov     ff, #0xff

                @ copy key from in to out
                mov     cnt, nk
aake_lp:        ldr     acc, [in], #4
                str     acc, [out], #4
                subs    cnt, #1
                bne     aake_lp

                @ number of rounds = 3*nk + 28 (cnt==0)
                add     rnd, nk, nk, lsl #1
                add     rnd, #28

                @ rnd mod nk == 0
aake_zero:      ldr     st, [out, #-4]
                ror     st, #8
                and     tmp, ff, st
                ldrb    acc, [sbox, tmp]
                and     tmp, ff, st, lsr #8
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #8
                and     tmp, ff, st, lsr #16
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #16
                mov     tmp, st, lsr #24
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #24
                ldrb    tmp, [rcon], #1
                eor     acc, tmp
                b       aake_drain

                @ nk > 6 ?
aake_cmp_6:     cmp     nk, #6
                bls     aake_drain
                @ (rnd mod nk == 4) and (nk > 6)
                ldr     st, [out, #-4]
```

```
                and     tmp, ff, st
                ldrb    acc, [sbox, tmp]
                and     tmp, ff, st, lsr #8
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #8
                and     tmp, ff, st, lsr #16
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #16
                mov     tmp, st, lsr #24
                ldrb    tmp, [sbox, tmp]
                orr     acc, tmp, lsl #24
                b       aake_drain

                @ rnd mod nk == 4 ?
aake_try_4:     cmp     cnt, #4
                beq     aake_cmp_6
                @ rnd mod nk != 4
aake_drain:     ldr     tmp, [out, -nk, lsl #2]
                eor     acc, tmp
                str     acc, [out], #4
                subs    rnd, #1
                beq     aake_done
                add     cnt, #1
                cmp     cnt, nk
                bne     aake_try_4
                mov     cnt, #0
                b       aake_zero

aake_done:      pop     {r4-r9}
                bx      lr
                .end
```

Source file 7 acl_aes_key_de.s

```
@ void acl_aes_key_de(vect key_out, vect key_in, size_t key_size);
@   expands aes decryption key key_in to key_out
@   assuming a little endian processor
@ on entry:
@   r0 = pointer to expanded key (output)
@   r1 = pointer to key (input)
@   r2 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)

                .global acl_aes_key_de
                .text
                .arm

ptr             .req    r0      @ expanded key
acc             .req    r1      @ accumulator
nk              .req    r2      @ nk
ff              .req    r3      @ mask value
tmp             .req    r4      @ temp
st0             .req    r5      @ temp
st1             .req    r6      @ temp
sbox            .req    r7      @ pointer to sbox
inv_table       .req    r12     @ pointer to inverse table

acl_aes_key_de:
```

```
                push    {r0, r2, r14}
                bl      acl_aes_key_en
                pop     {r0, r2, r14}
                push    {r4-r7}
                ldr     sbox, =acl_aes_fwd_sbox
                ldr     inv_table, =acl_aes_inv_table
                mov     ff, #0xff
                lsl     nk, #2
                add     nk, #20
                add     ptr, #16
aakd_lp:        ldr     tmp, [ptr]
                and     st0, ff, tmp
                and     st1, ff, tmp, lsr #8
                ldrb    st0, [sbox, st0]
                ldrb    st1, [sbox, st1]
                ldr     acc, [inv_table, st0, lsl #2]
                ldr     st1, [inv_table, st1, lsl #2]
                eor     acc, st1, ror #24
                and     st0, ff, tmp, lsr #16
                mov     st1, tmp, lsr #24
                ldrb    st0, [sbox, st0]
                ldrb    st1, [sbox, st1]
                ldr     st0, [inv_table, st0, lsl #2]
                ldr     st1, [inv_table, st1, lsl #2]
                eor     acc, st0, ror #16
                eor     acc, st1, ror #8
                str     acc, [ptr], #4
                subs    nk, #1
                bne     aakd_lp
                pop     {r4-r7}
                bx      lr

                .end
```

Source file 8 acl_aes_en.s

```
@ no c prototype exists for this function, as it is only called from assembler
@   core encryption routine for aes
@   based on Federal Information Processing Standards Publication 197
@ on entry:
@   r2 = key = pointer to already expanded key
@   r3 = rnd = key size - 4: 128, 6: 192, 8: 256
@   r4-r7 = st0-st3 = 16 input data bytes
@ returns:
@   r4-r7 = st0-st3 = 16 output data bytes
@ corrupts:
@   r0-r12

                .global acl_aes_en
                .text
                .arm

ff              .req    r0      @ holds mask value
lut             .req    r1      @ aes look up table
key             .req    r2      @ expanded key
rnd             .req    r3      @ round counter
st0             .req    r4      @ aes state word 0
```

```
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3
tmp             .req    r8      @ temp
key0            .req    r8      @ key tmp 0
acc             .req    r9      @ xor accumulator
key1            .req    r9      @ key tmp 1
nst0            .req    r10     @ next state word 0
nst1            .req    r11     @ next state word 1
nst2            .req    r12     @ next state word 2

                @ rnd = number of rounds - 1
acl_aes_en:     add     rnd, #5
                mov     ff, #0xff
                ldr     lut, =acl_aes_fwd_table

                @ add round key
                ldm     key!, {key0, key1}
                eor     st0, key0
                eor     st1, key1
                ldm     key!, {key0, key1}
                eor     st2, key0
                eor     st3, key1

                @ 1. column
aae_lp:         and     tmp, ff, st0                    @ st 0 0
                ldr     acc, [lut, tmp, lsl #2]
                and     tmp, ff, st1, lsr #8            @ st 1 1
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #24
                and     tmp, ff, st2, lsr #16           @ st 2 2
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #16
                mov     tmp, st3, lsr #24               @ st 3 3
                ldr     tmp, [lut, tmp, lsl #2]
                eor     nst0, acc, tmp, ror #8          @ store new st 0

                @ 2. column
                and     tmp, ff, st1                    @ st 1 0
                ldr     acc, [lut, tmp, lsl #2]
                and     tmp, ff, st2, lsr #8            @ st 2 1
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #24
                and     tmp, ff, st3, lsr #16           @ st 3 2
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #16
                mov     tmp, st0, lsr #24               @ st 0 3
                ldr     tmp, [lut, tmp, lsl #2]
                eor     nst1, acc, tmp, ror #8          @ store new st 1

                @ 3. column
                and     tmp, ff, st2                    @ st 2 0
                ldr     acc, [lut, tmp, lsl #2]
                and     tmp, ff, st3, lsr #8            @ st 3 1
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #24
                and     tmp, ff, st0, lsr #16           @ st 0 2
                ldr     tmp, [lut, tmp, lsl #2]
```

```
eor     acc, tmp, ror #16
mov     tmp, st1, lsr #24                @ st 1 3
ldr     tmp, [lut, tmp, lsl #2]
eor     nst2, acc, tmp, ror #8           @ store new st 2

@ 4. column
and     tmp, ff, st3                     @ st 3 0
ldr     acc, [lut, tmp, lsl #2]
and     tmp, ff, st0, lsr #8             @ st 0 1
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #24
and     tmp, ff, st1, lsr #16            @ st 1 2
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #16
mov     tmp, st2, lsr #24                @ st 2 3
ldr     tmp, [lut, tmp, lsl #2]
eor     st3, acc, tmp, ror #8            @ store new st 3

@ add round key
ldm     key!, {key0, key1}
eor     st0, nst0, key0
eor     st1, nst1, key1
ldm     key!, {key0, key1}
eor     st2, nst2, key0
eor     st3, key1

@ decrement counter
subs    rnd, #1                          @ do all the rounds,
bne     aae_lp                           @ except the last one

@ last round
ldr     lut, =acl_aes_fwd_sbox

@ 1. column
and     tmp, ff, st0                     @ st 0 0
ldrb    acc, [lut, tmp]
and     tmp, ff, st1, lsr #8             @ st 1 1
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #8
and     tmp, ff, st2, lsr #16            @ st 2 2
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #16
mov     tmp, st3, lsr #24                @ st 3 3
ldrb    tmp, [lut, tmp]
orr     nst0, acc, tmp, lsl #24          @ store new st 0

@ 2. column
and     tmp, ff, st1                     @ st 1 0
ldrb    acc, [lut, tmp]
and     tmp, ff, st2, lsr #8             @ st 2 1
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #8
and     tmp, ff, st3, lsr #16            @ st 3 2
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #16
mov     tmp, st0, lsr #24                @ st 0 3
ldrb    tmp, [lut, tmp]
orr     nst1, acc, tmp, lsl #24          @ store new st 1
```

```
                @ 3. column
                and     tmp, ff, st2                      @ st 2 0
                ldrb    acc, [lut, tmp]
                and     tmp, ff, st3, lsr #8              @ st 3 1
                ldrb    tmp, [lut, tmp]
                orr     acc, tmp, lsl #8
                and     tmp, ff, st0, lsr #16             @ st 0 2
                ldrb    tmp, [lut, tmp]
                orr     acc, tmp, lsl #16
                mov     tmp, st1, lsr #24                 @ st 1 3
                ldrb    tmp, [lut, tmp]
                orr     nst2, acc, tmp, lsl #24           @ store new st 2

                @ 4. column
                and     tmp, ff, st3                      @ st 3 0
                ldrb    acc, [lut, tmp]
                and     tmp, ff, st0, lsr #8              @ st 0 1
                ldrb    tmp, [lut, tmp]
                orr     acc, tmp, lsl #8
                and     tmp, ff, st1, lsr #16             @ st 1 2
                ldrb    tmp, [lut, tmp]
                orr     acc, tmp, lsl #16
                mov     tmp, st2, lsr #24                 @ st 2 3
                ldrb    tmp, [lut, tmp]
                orr     st3, acc, tmp, lsl #24            @ store new st 3

                @ add round key
                ldm     key!, {key0, key1}
                eor     st0, nst0, key0
                eor     st1, nst1, key1
                ldm     key!, {key0, key1}
                eor     st2, nst2, key0
                eor     st3, st3, key1
                bx      lr

                .end
```

Source file 9 acl_aes_de.s

```
@ no c prototype exists for this function, as it is only called from assembler
@   core decryption routine for aes
@   based on Federal Information Processing Standards Publication 197
@ on entry:
@   r2 = key = pointer to already expanded key
@   r3 = rnd = key size - 4: 128, 6: 192, 8: 256
@   r4-r7 = st0-st3 = 16 input data bytes
@ returns:
@   r4-r7 = st0-st3 = 16 output data bytes
@ corrupts:
@   r0-r12

                .global acl_aes_de
                .text
                .arm

ff              .req    r0      @ holds mask value
```

```
lut             .req    r1      @ aes look up table
key             .req    r2      @ expanded key
rnd             .req    r3      @ round counter
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3
tmp             .req    r8      @ temp
key0            .req    r8      @ key tmp 0
acc             .req    r9      @ xor accumulator
key1            .req    r9      @ key tmp 1
nst0            .req    r10     @ next state word 0
nst1            .req    r11     @ next state word 1
nst2            .req    r12     @ next state word 2

                @ rnd = number of rounds - 1
acl_aes_de:     add     rnd, #5
                mov     ff, #0xff
                ldr     lut, =acl_aes_inv_table
                add     acc, rnd, #2
                add     key, acc, lsl #4

                @ add round key
                ldmdb   key!, { key0, key1 }
                eor     st2, key0
                eor     st3, key1
                ldmdb   key!, { key0, key1 }
                eor     st0, key0
                eor     st1, key1

                @ 1. column
aad_lp:         and     tmp, ff, st0                    @ st 0 0
                ldr     acc, [lut, tmp, lsl #2]
                and     tmp, ff, st3, lsr #8            @ st 3 1
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #24
                and     tmp, ff, st2, lsr #16           @ st 2 2
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #16
                mov     tmp, st1, lsr #24               @ st 1 3
                ldr     tmp, [lut, tmp, lsl #2]
                eor     nst0, acc, tmp, ror #8          @ store new st 0

                @ 2. column
                and     tmp, ff, st1                    @ st 1 0
                ldr     acc, [lut, tmp, lsl #2]
                and     tmp, ff, st0, lsr #8            @ st 0 1
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #24
                and     tmp, ff, st3, lsr #16           @ st 3 2
                ldr     tmp, [lut, tmp, lsl #2]
                eor     acc, tmp, ror #16
                mov     tmp, st2, lsr #24               @ st 2 3
                ldr     tmp, [lut, tmp, lsl #2]
                eor     nst1, acc, tmp, ror #8          @ store new st 1

                @ 3. column
                and     tmp, ff, st2                    @ st 2 0
```

```
ldr     acc, [lut, tmp, lsl #2]
and     tmp, ff, st1, lsr #8            @ st 1 1
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #24
and     tmp, ff, st0, lsr #16           @ st 0 2
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #16
mov     tmp, st3, lsr #24               @ st 3 3
ldr     tmp, [lut, tmp, lsl #2]
eor     nst2, acc, tmp, ror #8          @ store new st 2

@ 4. column
and     tmp, ff, st3                    @ st 3 0
ldr     acc, [lut, tmp, lsl #2]
and     tmp, ff, st2, lsr #8            @ st 2 1
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #24
and     tmp, ff, st1, lsr #16           @ st 1 2
ldr     tmp, [lut, tmp, lsl #2]
eor     acc, tmp, ror #16
mov     tmp, st0, lsr #24               @ st 0 3
ldr     tmp, [lut, tmp, lsl #2]
eor     st3, acc, tmp, ror #8           @ store new st 3

@ add round key
ldmdb   key!, { key0, key1 }
eor     st2, nst2, key0
eor     st3, key1
ldmdb   key!, { key0, key1 }
eor     st0, nst0, key0
eor     st1, nst1, key1

@ decrement counter
subs    rnd, #1                         @ do all the rounds,
bne     aad_lp                          @ except the last one

@ last round
ldr     lut, =acl_aes_inv_sbox

@ 1. column
and     tmp, ff, st0                    @ st 0 0
ldrb    acc, [lut, tmp]
and     tmp, ff, st3, lsr #8            @ st 3 1
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #8
and     tmp, ff, st2, lsr #16           @ st 2 2
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #16
mov     tmp, st1, lsr #24               @ st 1 3
ldrb    tmp, [lut, tmp]
orr     nst0, acc, tmp, lsl #24         @ store new st 0

@ 2. column
and     tmp, ff, st1                    @ st 1 0
ldrb    acc, [lut, tmp]
and     tmp, ff, st0, lsr #8            @ st 0 1
ldrb    tmp, [lut, tmp]
orr     acc, tmp, lsl #8
```

```
        and     tmp, ff, st3, lsr #16           @ st 3 2
        ldrb    tmp, [lut, tmp]
        orr     acc, tmp, lsl #16
        mov     tmp, st2, lsr #24               @ st 2 3
        ldrb    tmp, [lut, tmp]
        orr     nst1, acc, tmp, lsl #24         @ store new st 1

        @ 3. column
        and     tmp, ff, st2                    @ st 2 0
        ldrb    acc, [lut, tmp]
        and     tmp, ff, st1, lsr #8            @ st 1 1
        ldrb    tmp, [lut, tmp]
        orr     acc, tmp, lsl #8
        and     tmp, ff, st0, lsr #16           @ st 0 2
        ldrb    tmp, [lut, tmp]
        orr     acc, tmp, lsl #16
        mov     tmp, st3, lsr #24               @ st 3 3
        ldrb    tmp, [lut, tmp]
        orr     nst2, acc, tmp, lsl #24         @ store new st 2

        @ 4. column
        and     tmp, ff, st3                    @ st 3 0
        ldrb    acc, [lut, tmp]
        and     tmp, ff, st2, lsr #8            @ st 2 1
        ldrb    tmp, [lut, tmp]
        orr     acc, tmp, lsl #8
        and     tmp, ff, st1, lsr #16           @ st 1 2
        ldrb    tmp, [lut, tmp]
        orr     acc, tmp, lsl #16
        mov     tmp, st0, lsr #24               @ st 0 3
        ldrb    tmp, [lut, tmp]
        orr     st3, acc, tmp, lsl #24          @ store new st 3

        @ add round key
        ldmdb   key!, { key0, key1 }
        eor     st2, nst2, key0
        eor     st3, key1
        ldmdb   key!, { key0, key1 }
        eor     st0, nst0, key0
        eor     st1, nst1, key1
        bx      lr

        .end
```

Source file 10 acl_aes_ecb_en.s

```
@ void acl_aes_ecb_en(vect4 out, vect4 in, vect exp_key, size_t key_size);
@   encrypt 16 bytes in ecb mode (little endian)
@ on entry:
@   r0 = pointer to 16 output data bytes
@   r1 = pointer to 16 input data bytes
@   r2 = pointer to already expanded key
@   r3 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)

        .global acl_aes_ecb_en
        .text
        .arm
```

```
out             .req    r0      @ outputs
in              .req    r1      @ inputs
key             .req    r2      @ expanded key
nk              .req    r3      @ key size
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3

acl_aes_ecb_en: push    {r4-r11, r14}
                push    {out}
                ldm     in, {st0, st1, st2, st3}
                bl      acl_aes_en
                pop     {out}
                stm     out, {st0, st1, st2, st3}
                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 11 acl_aes_ecb_de.s

```
@ void acl_aes_ecb_de(vect4 out, vect4 in, vect exp_key, size_t key_size);
@   decrypt 16 bytes in ecb mode (little endian)
@ on entry:
@   r0 = pointer to 16 output data bytes
@   r1 = pointer to 16 input data bytes
@   r2 = pointer to already expanded key
@   r3 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)

                .global acl_aes_ecb_de
                .text
                .arm

out             .req    r0      @ outputs
in              .req    r1      @ inputs
key             .req    r2      @ expanded key
nk              .req    r3      @ key size
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3

acl_aes_ecb_de: push    {r4-r11, r14}
                push    {out}
                ldm     in, {st0, st1, st2, st3}
                bl      acl_aes_de
                pop     {out}
                stm     out, {st0, st1, st2, st3}
                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 12 acl_aes_cbc_en.s

```
@ void acl_aes_cbc_en(vect4 out, vect4 in, vect exp_key, \
@                     size_t key_size, vect4 state);
@   encrypt 16 bytes in cbc mode (little endian)
@ on entry:
@   r0 = pointer to 16 output data bytes
@   r1 = pointer to 16 input data bytes
@   r2 = pointer to already expanded key
@   r3 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)
@   [sp] = pointer to 16 state bytes

                .global acl_aes_cbc_en
                .text
                .arm

out             .req    r0      @ outputs
in              .req    r1      @ inputs
key             .req    r2      @ expanded key
nk              .req    r3      @ key size
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3
hlp0            .req    r8      @
hlp1            .req    r9      @
hlp2            .req    r10     @
hlp3            .req    r11     @
state           .req    r12     @ pointer to state

acl_aes_cbc_en:
                ldr     state, [sp]
                push    {r4-r11, r14}
                push    {out, state}
                ldm     state, {hlp0, hlp1, hlp2, hlp3}
                ldm     in, {st0, st1, st2, st3}
                eor     st0, hlp0
                eor     st1, hlp1
                eor     st2, hlp2
                eor     st3, hlp3
                bl      acl_aes_en
                pop     {out, state}
                stm     out, {st0, st1, st2, st3}
                stm     state, {st0, st1, st2, st3}
                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 13 acl_aes_cbc_de.s

```
@ void acl_aes_cbc_de(vect4 out, vect4 in, vect exp_key, \
@                     size_t key_size, vect4 state);
@   decrypt 16 bytes in cbc mode (little endian)
@ on entry:
@   r0 = pointer to 16 output data bytes
@   r1 = pointer to 16 input data bytes
@   r2 = pointer to already expanded key
@   r3 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)
```

```
@   [sp] = pointer to 16 state bytes

                .global acl_aes_cbc_de
                .text
                .arm

out             .req    r0      @ outputs
in              .req    r1      @ inputs
key             .req    r2      @ expanded key
nk              .req    r3      @ key size
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3
hlp0            .req    r8      @
hlp1            .req    r9      @
hlp2            .req    r10     @
hlp3            .req    r11     @
state           .req    r12     @ pointer to state

acl_aes_cbc_de: ldr     state, [sp]
                push    {r4-r11, r14}
                ldm     state, {hlp0, hlp1, hlp2, hlp3}
                push    {out, hlp0, hlp1, hlp2, hlp3}
                ldm     in, {st0, st1, st2, st3}
                stm     state, {st0, st1, st2, st3}
                bl      acl_aes_de
                pop     {out, hlp0, hlp1, hlp2, hlp3}
                eor     st0, hlp0
                eor     st1, hlp1
                eor     st2, hlp2
                eor     st3, hlp3
                stm     out, {st0, st1, st2, st3}
                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 14 acl_aes_cntr.s

```
@ void acl_aes_cntr(vect4 out, vect4 in, vect exp_key, \
@                   size_t key_size, vect4 counter);
@   encrypt/decrypt 16 bytes in counter mode (little endian)
@ on entry:
@   r0 = pointer to 16 output data bytes
@   r1 = pointer to 16 input data bytes
@   r2 = pointer to already expanded key
@   r3 = key size - 4: 128, 6: 192, 8: 256 (use constants ACL_xxx)
@   [sp] = pointer to 16 counter bytes

                .global acl_aes_cntr
                .text
                .arm

out             .req    r0      @ outputs
in              .req    r1      @ inputs
key             .req    r2      @ expanded key
```

```
nk              .req    r3      @ key size
st0             .req    r4      @ aes state word 0
st1             .req    r5      @ aes state word 1
st2             .req    r6      @ aes state word 2
st3             .req    r7      @ aes state word 3
hlp0            .req    r8      @
hlp1            .req    r9      @
hlp2            .req    r10     @
hlp3            .req    r11     @
cntr            .req    r12     @ pointer to counter

acl_aes_cntr:   ldr     cntr, [sp]
                push    {r4-r11, r14}
                push    {out, in}
                ldm     cntr, {st0, st1, st2, st3}
                adds    hlp0, st0, #1
                adcs    hlp1, st1, #0
                adcs    hlp2, st2, #0
                adcs    hlp3, st3, #0
                stm     cntr, {hlp0, hlp1, hlp2, hlp3}
                bl      acl_aes_en
                pop     {out, in}
                ldm     in, {hlp0, hlp1, hlp2, hlp3}
                eor     hlp0, st0
                eor     hlp1, st1
                eor     hlp2, st2
                eor     hlp3, st3
                stm     out, {hlp0, hlp1, hlp2, hlp3}
                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 15 acl_sha1.s

```
                .global acl_sha1_init
                .global acl_sha1
                .global acl_sha1_done
                .text
                .arm

@ based on Federal Information Processing Standards Publication 180-2
@       (+ Change Notice to include SHA-224)
@ and on http://en.wikipedia.org/wiki/SHA_hash_functions
@ which translates very well into ARM code

@ throughout this file:
@   state[0..4] == hash state
@   state[5..20] == tmp storage
@   state[21..22] == length

@ void acl_sha1_init(vect23 state);
@   initialize sha-1 hash
@ on entry:
@   r0 = pointer to state

st              .req    r0      @
```

```
cnt             .req    r1      @
tmp             .req    r2      @
tab             .req    r3      @

acl_sha1_init_table:
                .int    0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476
                .int    0xc3d2e1f0

acl_sha1_init:  adr     tab, acl_sha1_init_table
                mov     cnt, #5
as1i_lp:        ldr     tmp, [tab], #4
                str     tmp, [st], #4
                subs    cnt, #1
                bne     as1i_lp
                add     st, #4*16
                mov     tmp, #0
                str     tmp, [st], #4
                str     tmp, [st]
                bx      lr

                .unreq  st
                .unreq  cnt
                .unreq  tmp
                .unreq  tab

@ void acl_sha1(vect23 state, byte data);
@   update sha-1 hash with byte "data"
@ on entry:
@   r0 = pointer to state
@   r1 = next byte to update the hash

st              .req    r0      @
chr             .req    r1      @
buf             .req    r2      @

len             .req    r3      @
tmp0            .req    r12     @

aa              .req    r1      @
bb              .req    r2      @
cc              .req    r3      @
dd              .req    r4      @
ee              .req    r5      @
rnd             .req    r6      @
acc             .req    r7      @
kay             .req    r8      @
tmp1            .req    r9      @
tmp2            .req    r10     @
tmp3            .req    r12     @


as1c_big:       ldr     len, [st, #4*21]
                add     len, #1
                str     len, [st, #4*21]
                b       as1_core

acl_sha1:       add     buf, st, #4*5
                ldr     len, [st, #4*22]
```

```
                eor     tmp0, len, #3 << 3
                lsl     tmp0, #23
                strb    chr, [buf, tmp0, lsr #26]
                adds    len, #8
                str     len, [st, #4*22]
                bcs     as1c_big
                tst     len, #63 << 3
                bxne    lr

as1_core:       push    {r4-r10}
                ldmia   st!, {aa, bb, cc, dd, ee}
                mov     rnd, #0

                @ rnd = 0-15
                ldr     kay, =0x5a827999
as1c_lp1:       ldr     acc, [st, rnd, lsl #2]
                and     tmp2, bb, cc
                bic     tmp1, dd, bb
                orr     tmp1, tmp2
                add     acc, tmp1
                add     acc, kay
                add     acc, ee
                add     acc, aa, ror #27
                mov     ee, dd
                mov     dd, cc
                mov     cc, bb, ror #2
                mov     bb, aa
                mov     aa, acc
                add     rnd, #1
                cmp     rnd, #16
                bne     as1c_lp1
                b       as1c_entry

                @ rnd = 16-79
as1c_main_lp:   add     acc, ee
                add     acc, aa, ror #27
                mov     ee, dd
                mov     dd, cc
                mov     cc, bb, ror #2
                mov     bb, aa
                mov     aa, acc

                @ get w(rnd)
as1c_entry:     mov     tmp3, rnd, lsl #28
                ldr     acc, [st, tmp3, lsr #26]
                add     tmp1, tmp3, #2 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                eor     acc, tmp2
                add     tmp1, #6 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                eor     acc, tmp2
                add     tmp1, #5 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                eor     acc, tmp2
                ror     acc, #31
                str     acc, [st, tmp3, lsr #26]

                @ get f(rnd)
```

```
                cmp     rnd, #60
                bhs     as1c_60
                cmp     rnd, #40
                bhs     as1c_40
                cmp     rnd, #20
                bhs     as1c_20

                @ rnd = 16-19
                and     tmp2, bb, cc
                bic     tmp1, dd, bb
                orr     tmp1, tmp2
                add     acc, tmp1
                add     acc, kay

                add     rnd, #1
                cmp     rnd, #20
                bne     as1c_main_lp
                ldr     kay, =0x6ed9eba1
                b       as1c_main_lp

                @ rnd = 20-39
as1c_20:        eor     tmp1, cc, dd
                eor     tmp1, bb
                add     acc, tmp1
                add     acc, kay

                add     rnd, #1
                cmp     rnd, #40
                bne     as1c_main_lp
                ldr     kay, =0x8f1bbcdc
                b       as1c_main_lp

                @ rnd = 40-59
as1c_40:        and     tmp2, bb, cc
                orr     tmp1, bb, cc
                and     tmp1, dd
                orr     tmp1, tmp2
                add     acc, tmp1
                add     acc, kay

                add     rnd, #1
                cmp     rnd, #60
                bne     as1c_main_lp
                ldr     kay, =0xca62c1d6
                b       as1c_main_lp

                @ rnd = 60-79
as1c_60:        eor     tmp1, cc, dd
                eor     tmp1, bb
                add     acc, tmp1
                add     acc, kay

                add     rnd, #1
                cmp     rnd, #80
                bne     as1c_main_lp

                add     acc, ee
                add     acc, aa, ror #27
```

```
                mov     ee, dd
                mov     dd, cc
                mov     cc, bb, ror #2
                mov     bb, aa
                mov     aa, acc

                ldmdb   st!, {rnd, acc, tmp1, tmp2}
                add     bb, rnd
                add     cc, acc
                add     dd, tmp1
                add     ee, tmp2
                ldr     tmp1, [st, #-4]!
                add     aa, tmp1
                stmia   st, {aa, bb, cc, dd, ee}

                pop     {r4-r10}
                bx      lr

@ void acl_sha1_done(vect23 state);
@   finish sha-1 hash - the result is in state[0..4]
@ on entry:
@   r0 = pointer to state

len1            .req    r1      @
len2            .req    r3      @

acl_sha1_done:  push    {lr}
                ldr     len1, [st, #4*22]
                ldr     len2, [st, #4*21]
                push    {len1, len2}
                mov     chr, #0x80
                b       as1d_entry

as1d_lp1:       mov     chr, #0
as1d_entry:     bl      acl_sha1
                ldr     len1, [st, #4*22]
                lsr     len1, #3
                tst     len1, #3
                bne     as1d_lp1
                and     len1, #63
                cmp     len1, #56
                bhi     as1d_lp1

                add     buf, st, #4*5
                beq     as1d_done
                mov     len2, #0
as1d_lp2:       str     len2, [buf, len1]
                add     len1, #4
                cmp     len1, #56
                bne     as1d_lp2

as1d_done:      pop     {len1, len2}
                str     len1, [buf, #4*15]
                str     len2, [buf, #4*14]
                bl      as1_core
                pop     {lr}
                bx      lr
```

```
                .end
```

Source file 16 acl_sha256.s

```
                .global acl_sha224_init
                .global acl_sha256_init
                .global acl_sha256
                .global acl_sha256_done
                .text
                .arm

@ based on Federal Information Processing Standards Publication 180-2
@       (+ Change Notice to include SHA-224)
@ and on http://en.wikipedia.org/wiki/SHA_hash_functions
@ which translates very well into ARM code

@ throughout this file:
@   state[0..7] == hash state
@   state[8..23] == tmp storage
@   state[24..25] == length

acl_sha256_table:
                .int    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5
                .int    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
                .int    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3
                .int    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
                .int    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc
                .int    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da
                .int    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7
                .int    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967
                .int    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13
                .int    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85
                .int    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3
                .int    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070
                .int    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5
                .int    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
                .int    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208
                .int    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

@ void acl_sha224_init(vect26 state);
@   initialize sha-224 hash
@ on entry:
@   r0 = pointer to state

st              .req    r0      @
cnt             .req    r1      @
tmp             .req    r2      @
tab             .req    r3      @

acl_sha224_init_table:
                .int    0xc1059ed8, 0x367cd507, 0x3070dd17, 0xf70e5939
                .int    0xffc00b31, 0x68581511, 0x64f98fa7, 0xbefa4fa4

acl_sha224_init:
                adr     tab, acl_sha224_init_table
                b       as256i_entry
```

```
@ void acl_sha256_init(vect26 state);
@   initialize sha-256 hash
@ on entry:
@   r0 = pointer to state

acl_sha256_init_table:
                .int    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a
                .int    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19

acl_sha256_init:
                adr     tab, acl_sha256_init_table
as256i_entry:   mov     cnt, #8
as256i_lp:      ldr     tmp, [tab], #4
                str     tmp, [st], #4
                subs    cnt, #1
                bne     as256i_lp
                add     st, #4*16
                mov     tmp, #0
                str     tmp, [st], #4
                str     tmp, [st]
                bx      lr

                .unreq  st
                .unreq  cnt
                .unreq  tmp
                .unreq  tab

@ void acl_sha256(vect26 state, byte data);
@   update sha-256 hash with byte "data"
@ on entry:
@   r0 = pointer to state
@   r1 = next byte to update the hash

st              .req    r0      @
chr             .req    r1      @
buf             .req    r2      @

len             .req    r3      @
tmp0            .req    r12     @

aa              .req    r1      @
bb              .req    r2      @
cc              .req    r3      @
dd              .req    r4      @
ee              .req    r5      @
ff              .req    r6      @
gg              .req    r7      @
hh              .req    r8      @
rnd             .req    r9      @
acc             .req    r10     @
tmp1            .req    r11     @
tmp2            .req    r12     @
tmp3            .req    r14     @

as256c_big:     ldr     len, [st, #4*24]
                add     len, #1
                str     len, [st, #4*24]
                b       as256_core
```

```
acl_sha256:     add     buf, st, #4*8
                ldr     len, [st, #4*25]
                eor     tmp0, len, #3 << 3
                lsl     tmp0, #23
                strb    chr, [buf, tmp0, lsr #26]
                adds    len, #8
                str     len, [st, #4*25]
                bcs     as256c_big
                tst     len, #63 << 3
                bxne    lr

as256_core:     push    {r4-r11, r14}
                ldmia   st!, {aa, bb, cc, dd, ee, ff, gg, hh}
                mov     rnd, #0

                @ get w(rnd)
as256c_main_lp: mov     tmp1, rnd, lsl #28
                ldr     acc, [st, tmp1, lsr #26]
                cmp     rnd, #16
                blo     as256c_skip
                add     tmp1, #1 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                mov     tmp3, tmp2, ror #7
                eor     tmp3, tmp2, ror #18
                eor     tmp3, tmp2, lsr #3
                add     acc, tmp3
                add     tmp1, #8 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                add     acc, tmp2
                add     tmp1, #5 << 28
                ldr     tmp2, [st, tmp1, lsr #26]
                mov     tmp3, tmp2, ror #17
                eor     tmp3, tmp2, ror #19
                eor     tmp3, tmp2, lsr #10
                add     acc, tmp3
                mov     tmp1, rnd, lsl #28
                str     acc, [st, tmp1, lsr #26]

                @ get k(rnd), ch, s1, hh -> tmp1
as256c_skip:    adr     tmp1, acl_sha256_table
                ldr     tmp1, [tmp1, rnd, lsl #2]
                add     acc, tmp1
                and     tmp1, ee, ff
                bic     tmp2, gg, ee
                eor     tmp1, tmp2
                add     acc, tmp1
                mov     tmp1, ee, ror #6
                eor     tmp1, ee, ror #11
                eor     tmp1, ee, ror #25
                add     acc, tmp1
                add     tmp1, acc, hh

                @ get s0, maj -> tmp2
                mov     acc, aa, ror #2
                eor     acc, aa, ror #13
                eor     acc, aa, ror #22
                and     tmp2, aa, bb
```

```
                eor     tmp3, aa, bb
                and     tmp3, cc
                eor     tmp2, tmp3
                add     tmp2, acc

                mov     hh, gg
                mov     gg, ff
                mov     ff, ee
                add     ee, dd, tmp1
                mov     dd, cc
                mov     cc, bb
                mov     bb, aa
                add     aa, tmp1, tmp2

                add     rnd, #1
                cmp     rnd, #64
                bne     as256c_main_lp

                ldmdb   st!, {rnd, acc, tmp1, tmp2}
                add     ee, rnd
                add     ff, acc
                add     gg, tmp1
                add     hh, tmp2
                ldmdb   st!, {rnd, acc, tmp1, tmp2}
                add     aa, rnd
                add     bb, acc
                add     cc, tmp1
                add     dd, tmp2
                stmia   st, {aa, bb, cc, dd, ee, ff, gg, hh}

                pop     {r4-r11, r14}
                bx      lr

@ void acl_sha256_done(vect26 state);
@   finish sha-256 hash - the result is in state[0..4]
@ on entry:
@   r0 = pointer to state

len1            .req    r1      @
len2            .req    r3      @

acl_sha256_done:
                push    {lr}
                ldr     len1, [st, #4*25]
                ldr     len2, [st, #4*24]
                push    {len1, len2}
                mov     chr, #0x80
                b       as256d_entry

as256d_lp1:     mov     chr, #0
as256d_entry:   bl      acl_sha256
                ldr     len1, [st, #4*25]
                lsr     len1, #3
                tst     len1, #3
                bne     as256d_lp1
                and     len1, #63
                cmp     len1, #56
                bhi     as256d_lp1
```

```
                add     buf, st, #4*8
                beq     as256d_done
                mov     len2, #0
as256d_lp2:     str     len2, [buf, len1]
                add     len1, #4
                cmp     len1, #56
                bne     as256d_lp2

as256d_done:    pop     {len1, len2}
                str     len1, [buf, #4*15]
                str     len2, [buf, #4*14]
                bl      as256_core
                pop     {lr}
                bx      lr

                .end
```

Source file 17 acl_sha512.s

```
                .global acl_sha384_init
                .global acl_sha512_init
                .global acl_sha512
                .global acl_sha512_done
                .text
                .arm

@ based on Federal Information Processing Standards Publication 180-2
@       (+ Change Notice to include SHA-224)
@ and on http://en.wikipedia.org/wiki/SHA_hash_functions
@ which translates very well into ARM code

@ throughout this file:
@   state[0..15] == hash state
@   state[16..31] == a-h
@   state[32..63] == tmp storage
@   state[64..67] == length

acl_sha512_table:
                .int    0x428a2f98, 0xd728ae22, 0x71374491, 0x23ef65cd
                .int    0xb5c0fbcf, 0xec4d3b2f, 0xe9b5dba5, 0x8189dbbc
                .int    0x3956c25b, 0xf348b538, 0x59f111f1, 0xb605d019
                .int    0x923f82a4, 0xaf194f9b, 0xab1c5ed5, 0xda6d8118
                .int    0xd807aa98, 0xa3030242, 0x12835b01, 0x45706fbe
                .int    0x243185be, 0x4ee4b28c, 0x550c7dc3, 0xd5ffb4e2
                .int    0x72be5d74, 0xf27b896f, 0x80deb1fe, 0x3b1696b1
                .int    0x9bdc06a7, 0x25c71235, 0xc19bf174, 0xcf692694
                .int    0xe49b69c1, 0x9ef14ad2, 0xefbe4786, 0x384f25e3
                .int    0x0fc19dc6, 0x8b8cd5b5, 0x240ca1cc, 0x77ac9c65
                .int    0x2de92c6f, 0x592b0275, 0x4a7484aa, 0x6ea6e483
                .int    0x5cb0a9dc, 0xbd41fbd4, 0x76f988da, 0x831153b5
                .int    0x983e5152, 0xee66dfab, 0xa831c66d, 0x2db43210
                .int    0xb00327c8, 0x98fb213f, 0xbf597fc7, 0xbeef0ee4
                .int    0xc6e00bf3, 0x3da88fc2, 0xd5a79147, 0x930aa725
                .int    0x06ca6351, 0xe003826f, 0x14292967, 0x0a0e6e70
                .int    0x27b70a85, 0x46d22ffc, 0x2e1b2138, 0x5c26c926
                .int    0x4d2c6dfc, 0x5ac42aed, 0x53380d13, 0x9d95b3df
```

```
                .int    0x650a7354, 0x8baf63de, 0x766a0abb, 0x3c77b2a8
                .int    0x81c2c92e, 0x47edaee6, 0x92722c85, 0x1482353b
                .int    0xa2bfe8a1, 0x4cf10364, 0xa81a664b, 0xbc423001
                .int    0xc24b8b70, 0xd0f89791, 0xc76c51a3, 0x0654be30
                .int    0xd192e819, 0xd6ef5218, 0xd6990624, 0x5565a910
                .int    0xf40e3585, 0x5771202a, 0x106aa070, 0x32bbd1b8
                .int    0x19a4c116, 0xb8d2d0c8, 0x1e376c08, 0x5141ab53
                .int    0x2748774c, 0xdf8eeb99, 0x34b0bcb5, 0xe19b48a8
                .int    0x391c0cb3, 0xc5c95a63, 0x4ed8aa4a, 0xe3418acb
                .int    0x5b9cca4f, 0x7763e373, 0x682e6ff3, 0xd6b2b8a3
                .int    0x748f82ee, 0x5defb2fc, 0x78a5636f, 0x43172f60
                .int    0x84c87814, 0xa1f0ab72, 0x8cc70208, 0x1a6439ec
                .int    0x90befffa, 0x23631e28, 0xa4506ceb, 0xde82bde9
                .int    0xbef9a3f7, 0xb2c67915, 0xc67178f2, 0xe372532b
                .int    0xca273ece, 0xea26619c, 0xd186b8c7, 0x21c0c207
                .int    0xeada7dd6, 0xcde0eb1e, 0xf57d4f7f, 0xee6ed178
                .int    0x06f067aa, 0x72176fba, 0x0a637dc5, 0xa2c898a6
                .int    0x113f9804, 0xbef90dae, 0x1b710b35, 0x131c471b
                .int    0x28db77f5, 0x23047d84, 0x32caab7b, 0x40c72493
                .int    0x3c9ebe0a, 0x15c9bebc, 0x431d67c4, 0x9c100d4c
                .int    0x4cc5d4be, 0xcb3e42b6, 0x597f299c, 0xfc657e2a
                .int    0x5fcb6fab, 0x3ad6faec, 0x6c44198c, 0x4a475817

@ void acl_sha384_init(vect68 state);
@   initialize sha-384 hash
@ on entry:
@   r0 = pointer to state

st              .req    r0      @
cnt             .req    r1      @
tmp             .req    r2      @
tab             .req    r3      @

acl_sha384_init_table:
                .int    0xcbbb9d5d, 0xc1059ed8
                .int    0x629a292a, 0x367cd507
                .int    0x9159015a, 0x3070dd17
                .int    0x152fecd8, 0xf70e5939
                .int    0x67332667, 0xffc00b31
                .int    0x8eb44a87, 0x68581511
                .int    0xdb0c2e0d, 0x64f98fa7
                .int    0x47b5481d, 0xbefa4fa4

acl_sha384_init:
                adr     tab, acl_sha384_init_table
                b       as512i_entry

@ void acl_sha512_init(vect68 state);
@   initialize sha-512 hash
@ on entry:
@   r0 = pointer to state

acl_sha512_init_table:
                .int    0x6a09e667, 0xf3bcc908
                .int    0xbb67ae85, 0x84caa73b
                .int    0x3c6ef372, 0xfe94f82b
                .int    0xa54ff53a, 0x5f1d36f1
                .int    0x510e527f, 0xade682d1
```

```
                .int    0x9b05688c, 0x2b3e6c1f
                .int    0x1f83d9ab, 0xfb41bd6b
                .int    0x5be0cd19, 0x137e2179

acl_sha512_init:
                adr     tab, acl_sha512_init_table
as512i_entry:   mov     cnt, #16
as512i_lp1:     ldr     tmp, [tab], #4
                str     tmp, [st], #4
                subs    cnt, #1
                bne     as512i_lp1
                add     st, #4*48
                mov     cnt, #4
                mov     tmp, #0
as512i_lp2:     str     tmp, [st], #4
                subs    cnt, #1
                bne     as512i_lp2
                bx      lr

                .unreq  st
                .unreq  cnt
                .unreq  tmp
                .unreq  tab

@ void acl_sha512(vect68 state, byte data);
@   update sha-512 hash with byte "data"
@ on entry:
@   r0 = pointer to state
@   r1 = next byte to update the hash

st              .req    r0      @
chr             .req    r1      @
buf             .req    r2      @

len             .req    r3      @
tmp0            .req    r12     @

kay             .req    r1      @
rnd             .req    r3      @
acch            .req    r4      @
accl            .req    r5      @
wh              .req    r6      @
wl              .req    r7      @
tmp1            .req    r8      @
tmp2            .req    r9      @
tmp3            .req    r10     @
tmp4            .req    r11     @
tmp5            .req    r12     @
tmp6            .req    r14     @

as512c_big:     ldr     len, [st, #4*66]
                adds    len, #1
                str     len, [st, #4*66]
                ldr     len, [st, #4*65]
                adcs    len, #0
                str     len, [st, #4*65]
                ldr     len, [st, #4*64]
                adc     len, #0
```

```
                str     len, [st, #4*64]
                b       as512_core

acl_sha512:     add     buf, st, #4*32
                ldr     len, [st, #4*67]
                eor     tmp0, len, #3 << 3
                lsl     tmp0, #22
                strb    chr, [buf, tmp0, lsr #25]
                adds    len, #8
                str     len, [st, #4*67]
                bcs     as512c_big
                tst     len, #127 << 3
                bxne    lr

as512_core:     push    {r4-r11, r14}
                add     buf, st, #4*16
                ldmia   st!, {r4-r11}
                stmia   buf!, {r4-r11}
                ldmia   st!, {r4-r11}
                stmia   buf!, {r4-r11}
                adr     kay, acl_sha512_table
                mov     rnd, #0

                @ get w[rnd]
as512c_main_lp: mov     tmp1, rnd, lsl #28
                ldr     acch, [buf, tmp1, lsr #25]
                add     tmp1, #1 << 27
                ldr     accl, [buf, tmp1, lsr #25]
                cmp     rnd, #16
                blo     as512c_skip

                @ add sigma0(w[rnd+1])
                add     tmp1, #1 << 27
                ldr     wh, [buf, tmp1, lsr #25]
                add     tmp1, #1 << 27
                ldr     wl, [buf, tmp1, lsr #25]
                mov     tmp2, wl, lsr #1
                eor     tmp2, wh, lsl #31
                eor     tmp2, wl, lsr #8
                eor     tmp2, wh, lsl #24
                eor     tmp2, wl, lsr #7
                eor     tmp2, wh, lsl #25
                adds    accl, tmp2
                mov     tmp2, wh, lsr #1
                eor     tmp2, wl, lsl #31
                eor     tmp2, wh, lsr #8
                eor     tmp2, wl, lsl #24
                eor     tmp2, wh, lsr #7
                adc     acch, tmp2

                @ add w[rnd+9]
                add     tmp1, #(8 << 28) - (1 << 27)
                ldr     wh, [buf, tmp1, lsr #25]
                add     tmp1, #1 << 27
                ldr     wl, [buf, tmp1, lsr #25]
                adds    accl, wl
                adc     acch, wh
```

```
                @ add sigma1(w[rnd+14])
                add     tmp1, #(5 << 28) - (1 << 27)
                ldr     wh, [buf, tmp1, lsr #25]
                add     tmp1, #1 << 27
                ldr     wl, [buf, tmp1, lsr #25]
                mov     tmp2, wl, lsr #19
                eor     tmp2, wh, lsl #13
                eor     tmp2, wl, lsl #3
                eor     tmp2, wh, lsr #29
                eor     tmp2, wl, lsr #6
                eor     tmp2, wh, lsl #26
                adds    accl, tmp2
                mov     tmp2, wh, lsr #19
                eor     tmp2, wl, lsl #13
                eor     tmp2, wh, lsl #3
                eor     tmp2, wl, lsr #29
                eor     tmp2, wh, lsr #6
                adc     acch, tmp2

                @ store new w[rnd]
                mov     tmp1, rnd, lsl #28
                str     acch, [buf, tmp1, lsr #25]
                add     tmp1, #1 << 27
                str     accl, [buf, tmp1, lsr #25]

                @ add k[rnd]
as512c_skip:    ldmia   kay!, {wh, wl}
                adds    accl, wl
                adc     acch, wh

                @ add sigma1(e)
                add     st, #4*8
                ldmia   st!, {wh, wl}
                mov     tmp2, wl, lsr #14
                eor     tmp2, wh, lsl #18
                eor     tmp2, wl, lsr #18
                eor     tmp2, wh, lsl #14
                eor     tmp2, wh, lsr #9
                eor     tmp2, wl, lsl #23
                adds    accl, tmp2
                mov     tmp2, wh, lsr #14
                eor     tmp2, wl, lsl #18
                eor     tmp2, wh, lsr #18
                eor     tmp2, wl, lsl #14
                eor     tmp2, wl, lsr #9
                eor     tmp2, wh, lsl #23
                adc     acch, tmp2

                @ add ch(e,f,g)
                ldmia   st!, {tmp1, tmp2}
                and     tmp1, wh
                and     tmp2, wl
                ldmia   st!, {tmp3, tmp4}
                bic     tmp3, wh
                bic     tmp4, wl
                eor     tmp1, tmp3
                eor     tmp2, tmp4
                adds    accl, tmp2
```

```
                adc     acch, tmp1

                @ add h
                ldmia   st!, {tmp1, tmp2}
                adds    tmp2, accl
                adc     tmp1, acch

                @ get maj(a,b,c)
                sub     st, #4*16
                ldmia   st, {wh, wl, tmp3, tmp4, tmp5, tmp6}
                eor     acch, wh, tmp3
                eor     accl, wl, tmp4
                and     acch, tmp5
                and     accl, tmp6
                and     tmp3, wh
                and     tmp4, wl
                eor     acch, tmp3
                eor     accl, tmp4

                @ add sigma0(a)
                mov     tmp4, wl, lsr #28
                eor     tmp4, wh, lsl #4
                eor     tmp4, wh, lsr #2
                eor     tmp4, wl, lsl #30
                eor     tmp4, wh, lsr #7
                eor     tmp4, wl, lsl #25
                adds    tmp4, accl
                mov     tmp3, wh, lsr #28
                eor     tmp3, wl, lsl #4
                eor     tmp3, wl, lsr #2
                eor     tmp3, wh, lsl #30
                eor     tmp3, wl, lsr #7
                eor     tmp3, wh, lsl #25
                adc     tmp3, acch

                @ shift
                adds    tmp4, tmp2
                adc     tmp3, tmp1
                stmia   st!, {tmp3, tmp4}  @ a = t1 + t2
                ldmia   st, {acch, accl, tmp3, tmp4, tmp5, tmp6}  @ b, c, d
                stmia   st!, {wh, wl}                             @ a
                stmia   st!, {acch, accl, tmp3, tmp4}             @    b, c
                adds    tmp2, tmp6
                adc     tmp1, tmp5
                ldmia   st, {acch, accl, tmp3, tmp4, tmp5, tmp6}  @ e, f, g
                stmia   st!, {tmp1, tmp2}  @ e = d + t1
                stmia   st!, {acch, accl, tmp3, tmp4, tmp5, tmp6} @ f, g, h
                sub     st, #4*16

                add     rnd, #1
                cmp     rnd, #80
                bne     as512c_main_lp

                mov     rnd, #4
as512c_lp2:     ldmdb   st, {acch, accl, wh, wl}
                ldmdb   buf!, {tmp1, tmp2, tmp3, tmp4}
                adds    tmp4, wl
                adc     tmp3, wh
```

```
                adds    tmp2, accl
                adc     tmp1, acch
                stmdb   st!, {tmp1, tmp2, tmp3, tmp4}
                subs    rnd, #1
                bne     as512c_lp2

                pop     {r4-r11, r14}
                bx      lr

@ void acl_sha512_done(vect68 state);
@   finish sha-512 hash - the result is in state[0..15]
@ on entry:
@   r0 = pointer to state

len1            .req    r1      @
len2            .req    r3      @

acl_sha512_done:
                push    {lr}
                ldr     len1, [st, #4*67]
                ldr     len2, [st, #4*66]
                push    {len1, len2}
                ldr     len1, [st, #4*65]
                ldr     len2, [st, #4*64]
                push    {len1, len2}
                mov     chr, #0x80
                b       as512d_entry

as512d_lp1:     mov     chr, #0
as512d_entry:   bl      acl_sha512
                ldr     len1, [st, #4*67]
                lsr     len1, #3
                tst     len1, #3
                bne     as512d_lp1
                and     len1, #127
                cmp     len1, #112
                bhi     as512d_lp1

                add     buf, st, #4*32
                beq     as512d_done
                mov     len2, #0
as512d_lp2:     str     len2, [buf, len1]
                add     len1, #4
                cmp     len1, #112
                bne     as512d_lp2

as512d_done:    pop     {len1, len2}
                str     len1, [buf, #4*29]
                str     len2, [buf, #4*28]
                pop     {len1, len2}
                str     len1, [buf, #4*31]
                str     len2, [buf, #4*30]
                bl      as512_core
                pop     {lr}
                bx      lr

                .end
```

Source file 18 acl_mov.s

```
@ void acl_mov(vect res, vect src, size_t len);
@   copies the array "src" to the array "res"
@ on entry:
@   r0 = pointer to destination
@   r1 = pointer to source
@   r2 = length of input/output arrays in 32-bit words

                .global acl_mov
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
len             .req    r2      @
tmp1            .req    r3      @
tmp2            .req    r12     @

acl_mov:        ldmia   src!, {tmp1, tmp2}
                subs    len, #2
                stmhsia dest!, {tmp1, tmp2}
                bhi     acl_mov
                strlo   tmp1, [dest]
                bx      lr

                .end
```

Source file 19 acl_mov32.s

```
@ void acl_mov32(vect res, uint val, size_t len);
@   initializes the array "res" to zero, except the lowest 32-bit word,
@   which is set to the 32-bit constant "val"
@ on entry:
@   r0 = pointer to result array
@   r1 = value
@   r2 = length of result array in 32-bit words

                .global acl_mov32
                .text
                .arm

dest            .req    r0      @
val             .req    r1      @
len             .req    r2      @
tmp1            .req    r3      @
tmp2            .req    r12     @

acl_mov32:      str     val, [dest], #4
                sub     len, #1
                mov     tmp1, #0
                mov     tmp2, #0
am32_lp1:       subs    len, #2
                stmhsia dest!, {tmp1, tmp2}
                bhi     am32_lp1
                strlo   tmp1, [dest]
```

```
                bx      lr

                .end
```

Source file 20 acl_bit.s

```
@ uint acl_bit(vect a, uint pos, size_t len);
@   return value of bit at position "pos" in array "a"
@   returns 0 if pos >= 32*len
@ on entry:
@   r0 = pointer to input array
@   r1 = position of bit to be read (0 -> lsb)
@   r2 = length of input array in 32-bit words
@ returns:
@   r0 = value of bit

                .global acl_bit
                .text
                .arm

src             .req    r0      @
pos             .req    r1      @
len             .req    r2      @
shift           .req    r3      @
tmp             .req    r12     @

acl_bit:        and     shift, pos, #31
                mov     pos, pos, lsr #5
                cmp     pos, len
                bhs     ab_zero
                ldr     tmp, [src, pos, lsl #2]
                mov     tmp, tmp, lsr shift
                and     r0, tmp, #1
                bx      lr

ab_zero:        mov     r0, #0
                bx      lr

                .end
```

Source file 21 acl_bit_clr.s

```
@ void acl_bit_clr(vect a, uint pos);
@   clear bit at position "pos" in array "a"
@ on entry:
@   r0 = pointer to input/output array
@   r1 = position of bit to be cleared (0 -> lsb)

                .global acl_bit_clr
                .text
                .arm

src             .req    r0      @
pos             .req    r1      @
shift           .req    r2      @
tmp             .req    r3      @
```

```
mask            .req    r12     @

acl_bit_clr:    and     shift, pos, #31
                mov     mask, #1
                mov     mask, mask, lsl shift
                mov     pos, pos, lsr #5
                ldr     tmp, [src, pos, lsl #2]
                bic     tmp, mask
                str     tmp, [src, pos, lsl #2]
                bx      lr

                .end
```

Source file 22 acl_bit_set.s

```
@ void acl_bit_set(vect a, uint pos);
@   set bit at position "pos" in array "a"
@ on entry:
@   r0 = pointer to input/output array
@   r1 = position of bit to be set (0 -> lsb)

                .global acl_bit_set
                .text
                .arm

src             .req    r0      @
pos             .req    r1      @
shift           .req    r2      @
tmp             .req    r3      @
mask            .req    r12     @

acl_bit_set:    and     shift, pos, #31
                mov     mask, #1
                mov     mask, mask, lsl shift
                mov     pos, pos, lsr #5
                ldr     tmp, [src, pos, lsl #2]
                orr     tmp, mask
                str     tmp, [src, pos, lsl #2]
                bx      lr

                .end
```

Source file 23 acl_cmp.s

```
@ int acl_cmp(vect a, vect b, size_t len);
@   compares two arrays; returns -1 if a<b, 0 if a==b, 1 if a>b
@ on entry:
@   r0 = pointer to first array
@   r1 = pointer to second array
@   r2 = length of input arrays in 32-bit words
@ returns:
@   r0 = result

                .global acl_cmp
                .text
                .arm
```

```
src1            .req    r0      @
src2            .req    r1      @
len             .req    r2      @
tmp1            .req    r3      @
tmp2            .req    r12     @

acl_cmp:        sub     len, #1
acmp_lp:        ldr     tmp1, [src1, len, lsl #2]
                ldr     tmp2, [src2, len, lsl #2]
                cmp     tmp1, tmp2
                blo     acmp_less
                bhi     acmp_more
                subs    len, #1
                bhs     acmp_lp
                mov     r0, #0
                bx      lr
acmp_less:      mov     r0, #-1
                bx      lr
acmp_more:      mov     r0, #1
                bx      lr

                .end
```

Source file 24 acl_zero.s

```
@ bool_t acl_zero(vect a, size_t len);
@   returns true if array is zero, false otherwise
@ on entry:
@   r0 = pointer to input array
@   r1 = length of input array in 32-bit words
@ returns:
@   r0 = result

                .global acl_zero
                .text
                .arm

src             .req    r0      @
len             .req    r1      @
tmp             .req    r2      @

acl_zero:       ldr     tmp, [src], #4
                cmp     tmp, #0
                bne     azro_nope
                subs    len, #1
                bne     acl_zero
                mov     r0, #-1
                bx      lr
azro_nope:      mov     r0, #0
                bx      lr

                .end
```

Source file 25 acl_xor.s

```
@ void acl_xor(vect res, vect a, vect b, size_t len);
@   res = a xor b
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = pointer to second operand
@   r3 = length of input/output arrays in 32-bit words

                .global acl_xor
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
src2            .req    r2      @
len             .req    r3      @
tmp1            .req    r4      @
tmp2            .req    r5      @
tmp3            .req    r6      @
tmp4            .req    r12     @

acl_xor:        push    {r4-r6}
ax_lp1:         ldmia   src1!, {tmp1, tmp2}
                ldmia   src2!, {tmp3, tmp4}
                eor     tmp1, tmp3
                eor     tmp2, tmp4
                subs    len, #2
                stmhsia dest!, {tmp1, tmp2}
                bhi     ax_lp1
                strlo   tmp1, [dest]
                pop     {r4-r6}
                bx      lr

                .end
```

Source file 26 acl_xor32.s

```
@ void acl_xor32(vect res, vect a, uint b, size_t len);
@   res = a xor b [32-bit]
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = second operand
@   r3 = length of input/output arrays in 32-bit words

                .global acl_xor32
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
src2            .req    r2      @
len             .req    r3      @
tmp             .req    r12     @

acl_xor32:      ldr     tmp, [src1], #4
                eor     tmp, src2
```

```
                str     tmp, [dest], #4
                cmp     dest, src1
                bxeq    lr
                sub     src2, len, #1
                b       acl_mov

                .end
```

Source file 27 acl_log2.s

```
@ int acl_log2(vect a, size_t len);
@   returns the position of the most significant non-zero bit
@   -1 -> all bits are zero,  0 -> lsb,  ...
@ on entry:
@   r0 = pointer to input array
@   r1 = length of input array in 32-bit words
@ returns:
@   r0 = result

                .global acl_log2
                .text
                .arm

src             .req    r0      @
len             .req    r1      @
tmp             .req    r2      @

acl_log2:       lsl     len, #5
al2_lp1:        subs    len, #32
                blo     al2_zero
                ldr     tmp, [src, len, lsr #3]
                cmp     tmp, #0
                beq     al2_lp1
al2_lp2:        subpl   len, #1
                addpls  tmp, tmp
                bpl     al2_lp2
al2_zero:       add     r0, len, #31
                bx      lr

                .end
```

Source file 28 acl_ctz.s

```
@ uint acl_ctz(vect a, size_t len);
@   count trailing zeroes; if whole number is zero, returns 32*len
@ on entry:
@   r0 = pointer to input array
@   r1 = length of input array in 32-bit words
@ returns:
@   r0 = result

                .global acl_ctz
                .text
                .arm

src             .req    r0      @
```

```
len             .req    r1      @
tmp             .req    r2      @
res             .req    r3      @

acl_ctz:        mov     res, #0
actz_lp1:       ldr     tmp, [src], #4
                cmp     tmp, #0
                bne     actz_lp2
                add     res, #32
                subs    len, #1
                bne     actz_lp1
                bx      lr
actz_lp2:       movs    tmp, tmp, rrx
                addcc   res, #1
                bcc     actz_lp2
                mov     r0, res
                bx      lr

                .end
```

Source file 29 acl_rev.s

```
@ uint acl_rev(uint a);
@   return 32-bit int with byte order reversed
@   taken directly from "programming techniques", arm doc. dui 0021a
@ on entry:
@   r0 = input
@ returns:
@   r0 = result

                .global acl_rev
                .text
                .arm

val             .req    r0      @
tmp             .req    r1      @

acl_rev:        eor     tmp, val, val, ror #16
                bic     tmp, #0xff0000
                mov     val, val, ror #8
                eor     val, tmp, lsr #8
                bx      lr

                .end
```

Source file 30 acl_rsh.s

```
@ void acl_rsh(vect a, uint k, size_t len);
@   a = a >> k    (right shift by k bits)
@ on entry:
@   r0 = pointer to input/output
@   r1 = number of bits to shift by
@   r2 = length of input/output array in 32-bit words

                .global acl_rsh
                .text
```

```
                .arm

dest            .req    r0      @
kay             .req    r1      @
len             .req    r2      @
tmp1            .req    r3      @
tmp2            .req    r4      @
cnt             .req    r5      @
shift_r         .req    r6      @
shift_l         .req    r12     @

acl_rsh:        push    {r4-r6}

ar_lp1:         add     dest, len, lsl #2
                mov     cnt, len
                mov     tmp2, #0
                mov     shift_r, kay
                rsbs    shift_l, shift_r, #32
                movmi   shift_r, #32
                movmi   shift_l, #0

ar_lp2:         ldr     tmp1, [dest, #-4]
                orr     tmp2, tmp1, lsr shift_r
                str     tmp2, [dest, #-4]!
                mov     tmp2, tmp1, lsl shift_l
                subs    cnt, #1
                bne     ar_lp2
                subs    kay, shift_r
                bne     ar_lp1

                pop     {r4-r6}
                bx      lr

                .end
```

Source file 31 acl_hex2str_dec.s

```
@ void acl_hex2str_dec(bytes res, size_t len_r, vect a, size_t len);
@   converts a[len] to a decimal string in res[len_r]
@ on entry:
@   r0 = pointer to result
@   r1 = number of characters in res
@   r2 = pointer to input
@   r3 = length of input array in 32-bit words

                .global acl_hex2str_dec
                .text
                .arm

dest            .req    r0      @
len_d           .req    r1      @
src             .req    r2      @
len             .req    r3      @
cnt             .req    r4      @
tmp             .req    r5      @
ind_s           .req    r6      @
ind_d           .req    r7      @
```

```
carry           .req    r12     @

acl_hex2str_dec:
                push    {r4-r7}

                @ clear accumulator
                mov     cnt, len_d
                mov     tmp, #0
ah2sd_init_lp1: subs    cnt, #1
                strb    tmp, [dest, cnt]
                bne     ah2sd_init_lp1

                mov     ind_s, len, lsl #3
                sub     ind_s, #1
                b       ah2sd_entry

                @ mul by 16 and adjust
ah2sd_main_lp:  sub     ind_d, len_d, #1
                mov     carry, #0
ah2sd_adj_lp1:  ldrb    tmp, [dest, ind_d]
                add     tmp, carry, tmp, lsl #4
                mov     carry, #0
                mov     cnt, #16
ah2sd_adj_lp2:  cmp     tmp, #10*16
                subhs   tmp, #10*16
                addhs   carry, cnt
                lsrs    cnt, #1
                lslne   tmp, #1
                bne     ah2sd_adj_lp2
                lsr     tmp, #4
                strb    tmp, [dest, ind_d]
                subs    ind_d, #1
                bhs     ah2sd_adj_lp1

                @ fetch next nibble (msb first)
ah2sd_entry:    ldrb    carry, [src, ind_s, lsr #1]
                tst     ind_s, #1
                lsrne   carry, #4
                and     carry, #0x0f

                @ add it to the lsb
                sub     ind_d, len_d, #1
ah2sd_adj_lp3:  ldrb    tmp, [dest, ind_d]
                add     tmp, carry
                mov     carry, #0
ah2sd_adj_lp4:  cmp     tmp, #10
                subhs   tmp, #10
                addhs   carry, #1
                bhi     ah2sd_adj_lp4
                strb    tmp, [dest, ind_d]
                cmp     carry, #0
                beq     ah2sd_adj_done
                subs    ind_d, #1
                bhs     ah2sd_adj_lp3

ah2sd_adj_done: subs    ind_s, #1
                bhs     ah2sd_main_lp
```

```
                @ convert to characters
                mov     cnt, #0
                mov     carry, #' '
ah2sd_end_lp1:  ldrb    tmp, [dest, cnt]
                cmp     tmp, #0
                bne     ah2sd_end_ent
                strb    carry, [dest, cnt]
                add     cnt, #1
                cmp     cnt, len_d
                bne     ah2sd_end_lp1

                @ entire number is zero
                sub     cnt, #1
                mov     tmp, #'0'
                strb    tmp, [dest, cnt]
                b       ah2sd_done

ah2sd_end_lp2:  ldrb    tmp, [dest, cnt]
ah2sd_end_ent:  add     tmp, #'0'
                strb    tmp, [dest, cnt]
                add     cnt, #1
                cmp     cnt, len_d
                bne     ah2sd_end_lp2

ah2sd_done:     pop     {r4-r7}
                bx      lr

                .end
```

Source file 32 acl_hex2str_le.s

```
@ void acl_hex2str_le(bytes res, vect a, size_t len);
@   converts little-endian number in a to string(hex chars)[len]
@ on entry:
@   r0 = pointer to result string
@   r1 = pointer to input
@   r2 = length of result string in bytes

                .global acl_hex2str_le
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
len             .req    r2      @
tmp             .req    r3      @
acc             .req    r12     @

acl_hex2str_le: ldrb    acc, [src], #1
                subs    len, #1
                bxmi    lr
                and     tmp, acc, #0x0f
                cmp     tmp, #10
                addlo   tmp, #'0'
                addhs   tmp, #'A' - 10
                strb    tmp, [dest, len]
```

```
                subs    len, #1
                bxmi    lr
                mov     tmp, acc, lsr #4
                cmp     tmp, #10
                addlo   tmp, #'0'
                addhs   tmp, #'A' - 10
                strb    tmp, [dest, len]
                b       acl_hex2str_le

                .end
```

Source file 33 acl_str2bytes.s

```
@ void acl_str2bytes(vect res, bytes str, size_t len);
@   converts string(hex chars) to array of 4*len bytes
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input string
@   r2 = length of result array in 32-bit words

                .global acl_str2bytes
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
len             .req    r2      @
cnt             .req    r3      @
tmp             .req    r4      @
acc             .req    r5      @
ind             .req    r12     @

asc2hex:        subs    tmp, #'0'
                cmp     tmp, #10
                movlo   pc, lr
                subs    tmp, #'A' - '0'
                cmp     tmp, #6
                addlo   tmp, #10
                movlo   pc, lr
                sub     tmp, #'a' - 'A' - 10
                mov     pc, lr

acl_str2bytes: push    {r4-r5, r14}
                mov     cnt, #0
                mov     ind, #0
as2b_main_lp:   mov     acc, #0
                ldrb    tmp, [src, cnt]
                cmp     tmp, #0
                beq     as2b_str_end
                bl      asc2hex
                mov     acc, tmp, lsl #4
                add     cnt, #1

                ldrb    tmp, [src, cnt]
                cmp     tmp, #0
                beq     as2b_str_end
                bl      asc2hex
                orr     acc, tmp
```

```
                add     cnt, #1

as2b_str_end:   strb    acc, [dest, ind]
                add     ind, #1
                cmp     ind, len, lsl #2
                bne     as2b_main_lp

                pop     {r4-r5, r14}
                bx      lr

                .end
```

Source file 34 acl_str2hex_be.s

```
@ void acl_str2hex_be(vect res, bytes str, size_t len);
@   converts string(hex chars) to big-endian number in res[len]
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input string
@   r2 = length of result array in 32-bit words

                .global acl_str2hex_be
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
len             .req    r2      @
cnt             .req    r3      @
tmp             .req    r4      @
acc             .req    r5      @
ind             .req    r12     @

asc2hex:        subs    tmp, #'0'
                cmp     tmp, #10
                movlo   pc, lr
                subs    tmp, #'A' - '0'
                cmp     tmp, #6
                addlo   tmp, #10
                movlo   pc, lr
                sub     tmp, #'a' - 'A' - 10
                mov     pc, lr

acl_str2hex_be: push    {r4-r5, r14}
                mov     cnt, #0
                mov     ind, #0
as2hb_main_lp:  mov     acc, #0
                ldrb    tmp, [src, cnt]
                cmp     tmp, #0
                beq     as2hb_str_end
                bl      asc2hex
                mov     acc, tmp, lsl #4
                add     cnt, #1

                ldrb    tmp, [src, cnt]
                cmp     tmp, #0
                beq     as2hb_str_end
                bl      asc2hex
```

```
                orr     acc, tmp
                add     cnt, #1

as2hb_str_end:  strb    acc, [dest, ind]
                add     ind, #1
                cmp     ind, len, lsl #2
                bne     as2hb_main_lp

                pop     {r4-r5, r14}
                bx      lr

                .end
```

Source file 35 acl_str2hex_le.s

```
@ void acl_str2hex_le(vect res, size_t len, bytes str, size_t len_s);
@   converts string(hex chars) to little-endian number in res[len]
@ on entry:
@   r0 = pointer to result
@   r1 = length of result array in 32-bit words
@   r2 = pointer to input string
@   r3 = length of input string in bytes, if 0 -> null terminated

                .global acl_str2hex_le
                .text
                .arm

dest            .req    r0      @
len             .req    r1      @
src             .req    r2      @
cnt             .req    r3      @
tmp             .req    r4      @
acc             .req    r5      @
ind             .req    r12     @

asc2hex:        subs    tmp, #'0'
                cmp     tmp, #10
                movlo   pc, lr
                subs    tmp, #'A' - '0'
                cmp     tmp, #6
                addlo   tmp, #10
                movlo   pc, lr
                sub     tmp, #'a' - 'A' - 10
                mov     pc, lr

acl_str2hex_le: push    {r4-r5, r14}

                @ find end of string
                cmp     cnt, #0
                bne     as2hl_skip
as2hl_init_lp:  ldrb    tmp, [src, cnt]
                cmp     tmp, #0
                addne   cnt, #1
                bne     as2hl_init_lp

as2hl_skip:     mov     ind, #0
as2hl_main_lp:  mov     acc, #0
```

```
                subs    cnt, #1
                bmi     as2hl_str_end
                ldrb    tmp, [src, cnt]
                bl      asc2hex
                mov     acc, tmp

                subs    cnt, #1
                bmi     as2hl_str_end
                ldrb    tmp, [src, cnt]
                bl      asc2hex
                orr     acc, tmp, lsl #4

as2hl_str_end:  strb    acc, [dest, ind]
                add     ind, #1
                cmp     ind, len, lsl #2
                bne     as2hl_main_lp

                pop     {r4-r5, r14}
                bx      lr

                .end
```

Source file 36 acl_p_mod_add.s

```
@ uint acl_p_mod_add32(vect res, vect a, uint b, vect m, size_t len);
@   res = (a + b [32-bit] ) mod m,  returns number of subtractions of m

@ uint acl_p_mod_add(vect res, vect a, vect b, vect m, size_t len);
@   res = (a + b) mod m,  returns number of subtractions of m

@ if m == 0, these additions work just like ordinary additions
@ (they're not modular)

@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = (pointer to) second operand
@   r3 = pointer to m
@   [sp] = length of input/output arrays in 32-bit words

                .global acl_p_mod_add32
                .global acl_p_mod_add
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
carry           .req    r1      @
src2            .req    r2      @
emm             .req    r3      @
len             .req    r4      @
cnt             .req    r5      @
tmp1            .req    r6      @
tmp2            .req    r7      @
total           .req    r12     @

acl_p_mod_add32:
```

```
                push    {r4-r7}
                mov     total, #0
                ldr     len, [sp, #4*4]
                ldr     tmp1, [src1]
                adds    tmp1, src2
                str     tmp1, [dest]

                mov     cnt, #1
apma_lp0:       ldr     tmp1, [src1, cnt, lsl #2]
                adcs    tmp1, #0
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apma_lp0
                b       apma_entry

acl_p_mod_add:  push    {r4-r7}
                mov     total, #0
                ldr     len, [sp, #4*4]
                mov     cnt, #0
                msr     cpsr_f, #0
apma_lp1:       ldr     tmp1, [src1, cnt, lsl #2]
                ldr     tmp2, [src2, cnt, lsl #2]
                adcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apma_lp1
apma_entry:     mov     carry, #1
                bcs     apma_subtract
                mov     carry, #0

                @ a > m ?
apma_cmp:       cmp     emm, #0
                beq     apma_ret
                sub     cnt, len, #1
apma_lp2:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                cmp     tmp1, tmp2
                blo     apma_ret
                bhi     apma_subtract
                subs    cnt, #1
                bhs     apma_lp2

                @ a = a - m
apma_subtract:  add     total, #1
                mov     cnt, #0
                msr     cpsr_f, #(1<<29)
apma_lp4:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                sbcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apma_lp4
                sbcs    carry, #0
                bne     apma_subtract
```

```
apma_ret:       mov     r0, total
                pop     {r4-r7}
                bx      lr

                .end
```

Source file 37 acl_p_mod_dbl.s

```
@ uint acl_p_mod_dbl(vect a, uint k, vect m, size_t len);
@   a = (2^k)*a mod m
@   returns number of times it had to subtract m
@ on entry:
@   r0 = pointer to input/result
@   r1 = number of times to double
@   r2 = pointer to m
@   r3 = length of input/output arrays in 32-bit words

                .global acl_p_mod_dbl
                .text
                .arm

dest            .req    r0      @
kay             .req    r1      @
emm             .req    r2      @
len             .req    r3      @
carry           .req    r4      @
tmp1            .req    r5      @
tmp2            .req    r6      @
total           .req    r7      @
cnt             .req    r12     @

acl_p_mod_dbl:  push    {r4-r7}
                mov     total, #0

                @ a = 2 * a
apmd_again:     mov     cnt, #0
                msr     cpsr_f, #0
apmd_lp1:       ldr     tmp1, [dest, cnt, lsl #2]
                adcs    tmp1, tmp1
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apmd_lp1
                mov     carry, #1
                bcs     apmd_subtract
                mov     carry, #0

                @ a > m ?
                sub     cnt, len, #1
apmd_lp2:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                cmp     tmp1, tmp2
                blo     apmd_next
                bhi     apmd_subtract
                subs    cnt, #1
                bhs     apmd_lp2
```

```
                @ a = a - m
apmd_subtract:  add     total, #1
                mov     cnt, #0
                msr     cpsr_f, #(1<<29)
apmd_lp3:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                sbcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apmd_lp3
                sbcs    carry, #0
                bne     apmd_subtract

apmd_next:      subs    kay, #1
                bne     apmd_again
                mov     r0, total
                pop     {r4-r7}
                bx      lr

                .end
```

Source file 38 acl_p_mod_sub.s

```
@ void acl_p_mod_sub32(vect res, vect a, uint b, vect m, size_t len);
@   res = (a - b [32-bit] ) mod m

@ void acl_p_mod_sub(vect res, vect a, vect b, vect m, size_t len);
@   res = (a - b) mod m

@ if m == 0, these subtractions work just like ordinary subtractions
@ (they're not modular)

@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = (pointer to) second operand
@   r3 = pointer to m
@   [sp] = length of input/output arrays in 32-bit words

                .global acl_p_mod_sub32
                .global acl_p_mod_sub
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
src2            .req    r2      @
emm             .req    r3      @
len             .req    r4      @
cnt             .req    r5      @
tmp1            .req    r6      @
tmp2            .req    r12     @

acl_p_mod_sub32:
                push    {r4-r6}
                ldr     len, [sp, #4*3]
```

```
                ldr     tmp1, [src1]
                subs    tmp1, src2
                str     tmp1, [dest]

                mov     cnt, #1
apms_lp0:       ldr     tmp1, [src1, cnt, lsl #2]
                sbcs    tmp1, #0
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apms_lp0
                bcs     apms_ret
                b       apms_add

acl_p_mod_sub:  push    {r4-r6}
                ldr     len, [sp, #4*3]
                mov     cnt, #0
                msr     cpsr_f, #(1<<29)
apms_lp1:       ldr     tmp1, [src1, cnt, lsl #2]
                ldr     tmp2, [src2, cnt, lsl #2]
                sbcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apms_lp1
                bcs     apms_ret

                @ add m to result (carry == 0)
apms_add:       teq     emm, #0
                beq     apms_ret
                mov     cnt, #0
apms_lp2:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                adcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apms_lp2
                bcc     apms_add

apms_ret:       pop     {r4-r6}
                bx      lr

                .end
```

Source file 39 acl_p_mod_hlv.s

```
@ void acl_p_mod_hlv(vect a, uint k, vect m, size_t len);
@   k times: a = (a + m)/2 mod m  or  res = (a/2) mod m
@ on entry:
@   r0 = pointer to input/result
@   r1 = k
@   r2 = pointer to m
@   r3 = length of input/output arrays in 32-bit words

                .global acl_p_mod_hlv
                .text
```

```
                .arm

dest            .req    r0      @
kay             .req    r1      @
emm             .req    r2      @
len             .req    r3      @
tmp1            .req    r4      @
tmp2            .req    r5      @
cnt             .req    r12     @

acl_p_mod_hlv:  push    {r4-r5}
                ldr     tmp1, [dest]

apmh_again:     mov     cnt, len
                msr     cpsr_f, #0
                tst     tmp1, #1
                beq     apmh_lp2

                @ a = a + m
                mov     cnt, #0
apmh_lp1:       ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [emm, cnt, lsl #2]
                adcs    tmp1, tmp2
                str     tmp1, [dest, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apmh_lp1

                @ a = a/2
apmh_lp2:       sub     cnt, #1
                ldr     tmp1, [dest, cnt, lsl #2]
                movs    tmp1, tmp1, rrx
                str     tmp1, [dest, cnt, lsl #2]
                teq     cnt, #0
                bne     apmh_lp2

                subs    kay, #1
                bne     apmh_again
                pop     {r4-r5}
                bx      lr

                .end
```

Source file 40 acl_p_mul.s

```
@ void acl_p_mul(vect2 res, vect a, vect b, size_t len);
@   res[2*len] = a[len] * b[len]
@   does not work in-place (res != a, res != b)
@   works only for len >= 3
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = pointer to second operand
@   r3 = length of input arrays in 32-bit words (output is twice as long)

                .global acl_p_mul
                .text
```

```
                .arm

dest            .req    r0      @
src1            .req    r1      @
src2            .req    r2      @
len             .req    r3      @
tmp2            .req    r3      @
ind             .req    r4      @
pro1            .req    r5      @
pro2            .req    r6      @
pro3            .req    r7      @
pro4            .req    r8      @
sum1            .req    r9      @
sum2            .req    r10     @
sum3            .req    r11     @
tmp1            .req    r12     @
cnt             .req    r14     @

acl_p_mul:      push    {r4-r11, r14}
                push    {len}
                add     src2, #4
                ldmia   src1!, {pro1, pro2}
                ldmda   src2!, {pro3, pro4}
                umull   tmp1, sum1, pro1, pro3
                str     tmp1, [dest], #4
                mov     sum2, #0
                mov     sum3, #0
                mov     ind, #2
                mov     cnt, #2
                b       apm_entry

                @ first half
apm_h1_lp1:     sub     src1, ind, lsl #2
                add     ind, #1
                add     src2, ind, lsl #2
                mov     cnt, ind
                tst     cnt, #1
                ldrne   pro2, [src1], #4
                ldrne   pro3, [src2], #-4
                bne     apm_h1_entry

apm_h1_lp2:     ldmia   src1!, {pro1, pro2}
                ldmda   src2!, {pro3, pro4}
apm_entry:      umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
apm_h1_entry:   umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
                bhi     apm_h1_lp2

                @ got a diagonal
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, sum3
```

```
                mov     sum3, #0

                ldr     len, [sp]
                cmp     ind, len
                bne     apm_h1_lp1

                @ second half
apm_h2_lp1:     add     src2, ind, lsl #2
                sub     ind, #1
                sub     src1, ind, lsl #2
                mov     cnt, ind
                tst     cnt, #1
                ldrne   pro2, [src1], #4
                ldrne   pro3, [src2], #-4
                bne     apm_h2_entry

apm_h2_lp2:     ldmia   src1!, {pro1, pro2}
                ldmda   src2!, {pro3, pro4}
                umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
apm_h2_entry:   umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
                bhi     apm_h2_lp2

                @ got a diagonal
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, sum3
                mov     sum3, #0

                cmp     ind, #2
                bne     apm_h2_lp1

                umull   tmp1, tmp2, pro2, pro4
                adds    sum1, tmp1
                adc     sum2, tmp2
                stmia   dest, {sum1, sum2}
                pop     {r0, r4-r11, r14}
                bx      lr

                .end
```

Source file 41 acl_p_sqr.s

```
@ void acl_p_sqr(vect2 res, vect a, size_t len);
@   res[2*len] = a[len] * a[len]
@   does not work in-place (res != a)
@   works only for len > 4
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input
@   r2 = length of input array in 32-bit words (output is twice as long)
```

```
                .global acl_p_sqr
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
len             .req    r2      @
tmp2            .req    r2      @
src2            .req    r3      @
ind             .req    r4      @
pro1            .req    r5      @
pro2            .req    r6      @
pro3            .req    r7      @
pro4            .req    r8      @
sum1            .req    r9      @
sum2            .req    r10     @
sum3            .req    r11     @
tmp1            .req    r12     @
cnt             .req    r14     @

acl_p_sqr:      push    {r4-r11, r14}
                push    {len}
                mov     src2, src1
                ldr     pro1, [src2], #4
                umull   tmp1, sum1, pro1, pro1
                movs    sum1, sum1, lsr #1
                mov     tmp1, tmp1, rrx
                str     tmp1, [dest], #4
                mov     sum2, #0
                mov     sum3, #0
                mov     ind, #2
                mov     cnt, #1
                b       aps_entry

                @ first quarter
aps_h1_lp1:     mov     tmp1, ind, lsr #1
                sub     src1, tmp1, lsl #2
                add     tmp1, #1
                add     src2, tmp1, lsl #2
                add     ind, #1
                mov     cnt, ind, lsr #1
aps_entry:      tst     cnt, #1
                ldrne   pro2, [src1], #4
                ldrne   pro3, [src2], #-4
                bne     aps_h1_entry

aps_h1_lp2:     ldmia   src1!, {pro1, pro2}
                ldmda   src2!, {pro3, pro4}
                umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
aps_h1_entry:   umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
```

```
                bhi     aps_h1_lp2

                @ got half a diagonal
                tst     ind, #1
                beq     aps_h1_skip
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, sum3
                mov     sum3, #0
                ldr     len, [sp]
                cmp     ind, len
                bne     aps_h1_lp1
                b       aps_h2_lp1

                @ add center/2
aps_h1_skip:    umull   tmp1, tmp2, pro3, pro3
                movs    tmp2, tmp2, lsr #1
                movs    tmp1, tmp1, rrx
                addcss  sum1, #0x80000000
                str     sum1, [dest], #4
                adcs    sum1, sum2, tmp1
                adc     sum2, sum3, tmp2
                mov     sum3, #0
                ldr     len, [sp]
                cmp     ind, len
                bne     aps_h1_lp1

                @ second quarter
aps_h2_lp1:     mov     tmp1, ind, lsr #1
                add     src2, tmp1, lsl #2
                sub     tmp1, #1
                sub     src1, tmp1, lsl #2
                sub     ind, #1
                mov     cnt, ind, lsr #1
                tst     cnt, #1
                ldrne   pro2, [src1], #4
                ldrne   pro3, [src2], #-4
                bne     aps_h2_entry

aps_h2_lp2:     ldmia   src1!, {pro1, pro2}
                ldmda   src2!, {pro3, pro4}
                umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
aps_h2_entry:   umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
                bhi     aps_h2_lp2

                @ got half a diagonal
                tst     ind, #1
                beq     aps_h2_skip
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, sum3
```

```
                mov     sum3, #0
                b       aps_h2_lp1

                @ add center/2
aps_h2_skip:    umull   tmp1, tmp2, pro3, pro3
                movs    tmp2, tmp2, lsr #1
                movs    tmp1, tmp1, rrx
                addcss  sum1, #0x80000000
                str     sum1, [dest], #4
                adcs    sum1, sum2, tmp1
                adc     sum2, sum3, tmp2
                mov     sum3, #0
                cmp     ind, #2
                bne     aps_h2_lp1

                stmia   dest!, {sum1, sum2}
                ldr     len, [sp]
                sub     dest, len, lsl #3
                add     src1, #4
                sub     src1, len, lsl #2
                mov     cnt, len
                ldr     pro1, [src1]
                movs    pro1, pro1, lsr #1

                @ multiply by 2
aps_sh_lp:      ldmia   dest, {sum1, sum2}
                adcs    sum1, sum1
                adcs    sum2, sum2
                stmia   dest!, {sum1, sum2}
                sub     cnt, #1
                teq     cnt, #0
                bne     aps_sh_lp

                pop     {r0, r4-r11, r14}
                bx      lr

                .end
```

Source file 42 acl_p_mont_red.s

```
@ void acl_p_mont_red(vect res, vect2 a, vect m, uint m_inv, size_t len);
@   res = a*r^(-1) mod m
@   does not work in-place (res != a)
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input
@   r2 = pointer to modulus
@   r3 = -m^-1 mod 2^32
@   [sp] = length of input/output arrays in 32-bit words

                .global acl_p_mont_red
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
emm             .req    r2      @
```

```
minv            .req    r3      @
tmp2            .req    r3      @
len             .req    r3      @
ind             .req    r4      @
pro1            .req    r5      @
pro2            .req    r6      @
pro3            .req    r7      @
pro4            .req    r8      @
sum1            .req    r9      @
sum2            .req    r10     @
sum3            .req    r11     @
tmp1            .req    r12     @
cnt             .req    r14     @

acl_p_mont_red: push    {r4-r11, r14}
                push    {minv}

                mov     sum1, #0
                mov     sum2, #0
                mov     sum3, #0
                mov     ind, #1
                b       apmr_entry

                @ first half
apmr_h1_lp1:    sub     dest, ind, lsl #2
                mov     cnt, ind
                add     ind, #1
                add     emm, ind, lsl #2
                tst     cnt, #1
                ldrne   pro2, [dest], #4
                ldrne   pro3, [emm], #-4
                bne     apmr_h1_entry

apmr_h1_lp2:    ldmia   dest!, {pro1, pro2}
                ldmda   emm!, {pro3, pro4}
                umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
apmr_h1_entry:  umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
                bhi     apmr_h1_lp2

                @ calculate next q
apmr_entry:     ldr     tmp1, [src], #4
                adds    sum1, tmp1
                adcs    sum2, #0
                adc     sum3, #0
                ldr     minv, [sp]
                mul     pro1, sum1, minv
                str     pro1, [dest], #4
                ldr     pro2, [emm], #-4
                umull   tmp1, tmp2, pro1, pro2
                adds    sum1, tmp1
                adcs    sum1, sum2, tmp2
```

```
                adc     sum2, sum3, #0
                mov     sum3, #0
                ldr     len, [sp, #4*10]
                cmp     ind, len
                bne     apmr_h1_lp1

                @ second half
                sub     dest, #4
                add     emm, #8
                sub     ind, #1
                mov     cnt, ind
                b       apmr_h2_entry1

apmr_h2_lp1:    ldr     tmp1, [src], #4
                adds    sum1, tmp1
                str     sum1, [dest]
                adcs    sum1, sum2, #0
                adc     sum2, sum3, #0
                mov     sum3, #0

                add     dest, ind, lsl #2
                sub     ind, #1
                mov     cnt, ind
                sub     emm, ind, lsl #2
apmr_h2_entry1: tst     cnt, #1
                ldrne   pro2, [dest], #-4
                ldrne   pro3, [emm], #4
                bne     apmr_h2_entry2

apmr_h2_lp2:    ldmda   dest!, {pro1, pro2}
                ldmia   emm!, {pro3, pro4}
                umull   tmp1, tmp2, pro1, pro4
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
apmr_h2_entry2: umull   tmp1, tmp2, pro2, pro3
                adds    sum1, tmp1
                adcs    sum2, tmp2
                adc     sum3, #0
                subs    cnt, #2
                bhi     apmr_h2_lp2

                @ finished yet?
                cmp     ind, #1
                bne     apmr_h2_lp1

                ldmia   src, {pro1, pro2}
                adds    sum1, pro1
                adcs    sum2, pro2
                adc     sum3, #0
                stmia   dest!, {sum1, sum2}
                ldr     len, [sp, #4*10]
                sub     dest, len, lsl #2
                sub     emm, len, lsl #2

                @ is there a carry?
                cmp     sum3, #0
                bne     apmr_subtract
```

```
                @ result > m ?
                sub     cnt, len, #1
apmr_cmp_lp:    ldr     pro1, [dest, cnt, lsl #2]
                ldr     pro2, [emm, cnt, lsl #2]
                cmp     pro1, pro2
                blo     apmr_ret
                bhi     apmr_subtract
                subs    cnt, #1
                bhs     apmr_cmp_lp

                @ result = result - m
apmr_subtract:  msr     cpsr_f, #(1<<29)
apmr_sub_lp:    ldr     pro1, [dest]
                ldr     pro2, [emm], #4
                sbcs    pro1, pro2
                str     pro1, [dest], #4
                sub     len, #1
                teq     len, #0
                bne     apmr_sub_lp

apmr_ret:       pop     {r0, r4-r11, r14}
                bx      lr

                .end
```

Source file 43 acl_p_mont_m_inv.s

```
@ uint acl_p_mont_m_inv(vect m);
@   precomputation for mongomery multiplication
@   returns -m^-1 mod 2^32  (m must be odd)
@ on entry:
@   r0 = pointer to m
@ returns:
@   r0 = result

                .global acl_p_mont_m_inv
                .text
                .arm

src             .req    r0      @
q               .req    r0      @
m               .req    r1      @
acc             .req    r2      @
mask            .req    r3      @

acl_p_mont_m_inv:
                ldr     m, [src]
                mov     q, #0
                mov     acc, #0
                mov     mask, #1
apmmi_lp:       tst     acc, #1
                addeq   acc, m
                orreq   q, mask
                mov     acc, acc, lsr #1
                adds    mask, mask
                bcc     apmmi_lp
```

```
        bx      lr

        .end
```

Source file 44 acl_p_mont_pre.c

```
// precomputation for montgomery arithmetic

// r_mod_m = 2^(32*len) mod m
// r2_mod_m = 2^(64*len) mod m
// m_inv = -m^(-1) mod 2^32
// len is the length of r_mod_m, r2_mod_m and m in 32-bit words

#include "..\acl.h"
void acl_p_mont_pre(vect r_mod_m, vect r2_mod_m, uint *m_inv, \
                    vect m, size_t len)
{
    int i;

    if(m_inv) *m_inv = acl_p_mont_m_inv(m);
    if(r_mod_m) {
        i = acl_log2(m, len);
        acl_mov32(r_mod_m, 0, len);
        acl_bit_set(r_mod_m, i);
        i = 32 * len - i;
        acl_p_mod_dbl(r_mod_m, i, m, len);
    }
    if(r2_mod_m) {
        if(r_mod_m) {
            acl_mov(r2_mod_m, r_mod_m, len);
            i = 32 * len;
        } else {
            i = acl_log2(m, len);
            acl_mov32(r2_mod_m, 0, len);
            acl_bit_set(r2_mod_m, i);
            i = 64 * len - i;
        }
        acl_p_mod_dbl(r2_mod_m, i, m, len);
    }
}
```

Source file 45 acl_p_mont_exp.c

```
// res = x^e mod m  (using montgomery exponentiation)
// len is the length of res, x, m, r2_mod_m in 32-bit words
// len_e is the length of e in 32-bit words
// tmp is (3 x len) big; tmp and x_r are used for temporary storage

#include "..\acl.h"
#include "..\acl_int.h"

void acl_p_mont_exp(vect res, vect x, vect e, size_t len_e, vect m, vect3 tmp, \
                    uint m_inv, vect r2_mod_m, size_t len)
{
    int i; vect x_r;   // tmp tmp x_r
```

```
    x_r = tmp + 2*len;
    i = acl_log2(e, len_e);
    if(i != -1) {
        acl_p_mul_mont(x_r, x, r2_mod_m);
        acl_mov(res, x_r, len);
        while(i--) {
            acl_p_sqr_mont(res, res);
            if(acl_bit(e, i, len_e)) acl_p_mul_mont(res, res, x_r);
        }
        acl_mov(tmp, res, len);
        acl_mov32(tmp + len, 0, len);
        acl_p_mont_red(res, tmp, m, m_inv, len);
    } else acl_mov32(res, 1, len);
}
```

Source file 46 acl_p_coprime.s

```
@ bool_t acl_p_coprime(vect a, vect b, vect2 tmp, size_t len);
@   returns true if gcd(a,b) == 1
@ on entry:
@   r0 = pointer to a
@   r1 = pointer to b
@   r2 = pointer to temporary array (2 x len 32-bit words)
@   r3 = length of input/output arrays in 32-bit words

                .global acl_p_coprime
                .text
                .arm

aa              .req    r0      @
shift_r         .req    r0      @
bb              .req    r1      @
shift_l         .req    r1      @
uu              .req    r2      @
len             .req    r3      @
vv              .req    r4      @
tmp1            .req    r5      @
tmp2            .req    r6      @
cnt             .req    r12     @

acl_p_coprime:  push    {r4-r6}
                add     vv, uu, len, lsl #2

                @ initialization
                sub     cnt, len, #1
apco_init_lp:   ldr     tmp1, [aa, cnt, lsl #2]
                str     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [bb, cnt, lsl #2]
                str     tmp2, [vv, cnt, lsl #2]
                subs    cnt, #1
                bhs     apco_init_lp

                @ if both u and v are even
                orr     cnt, tmp1, tmp2
                tst     cnt, #1
                beq     apco_no
```

```
                @ if both u and v are odd
                and     cnt, tmp1, tmp2
                tst     cnt, #1
                bne     apco_compare

                @ if u is even and v is odd
                tst     tmp1, #1
                beq     apco_u_again

                @ if u is odd and v is even
                mov     tmp1, uu
                mov     uu, vv
                mov     vv, tmp1
                b       apco_u_again

                @ swap u and v
apco_v_bigger:  mov     tmp1, uu
                mov     uu, vv
                mov     vv, tmp1

                @ u = u - v
apco_u_bigger:  mov     cnt, #0
                msr     cpsr_f, #(1<<29)
apco_u_lp1:     ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                sbcs    tmp1, tmp2
                str     tmp1, [uu, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len
                bne     apco_u_lp1

                @ count trailing zeroes of u
apco_u_again:   ldr     tmp1, [uu]
                mov     shift_r, #0
                msr     cpsr_f, #(1<<29)
apco_u_lp2:     movs    tmp1, tmp1, rrx
                addcc   shift_r, #1
                bcc     apco_u_lp2
                rsb     shift_l, shift_r, #32

                @ right shift u
                add     uu, len, lsl #2
                mov     cnt, len
                mov     tmp2, #0
apco_u_lp3:     ldr     tmp1, [uu, #-4]
                orr     tmp2, tmp1, lsr shift_r
                str     tmp2, [uu, #-4]!
                mov     tmp2, tmp1, lsl shift_l
                subs    cnt, #1
                bne     apco_u_lp3

                @ shifted by 32 bits?
                cmp     shift_r, #32
                beq     apco_u_again

                @ compare u and v
apco_compare:   mov     shift_r, #0
                subs    cnt, len, #1
```

```
apco_cmp_lp1:   ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                cmp     tmp1, tmp2
                bhi     apco_u_bigger
                blo     apco_v_bigger
                orrs    shift_r, tmp1
                subeq   len, #1
                subs    cnt, #1
                bhs     apco_cmp_lp1

                @ u (== v) == 1 ?
                cmp     len, #1
                bne     apco_no
                cmp     tmp1, #1
                bne     apco_no
                mov     r0, #-1
                pop     {r4-r6}
                bx      lr

apco_no:        mov     r0, #0
                pop     {r4-r6}
                bx      lr

                .end
```

Source file 47 acl_p_mont_inv.s

```
@ int acl_p_mont_inv(vect res, vect a, vect m, vect3 tmp, size_t len);
@   res = +- (a^-1) * (2^k) mod m     (m must be odd)
@   returns 0 if a is non-invertible, +- k otherwise
@   a != 0  and  a != 1
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to a
@   r2 = pointer to m
@   r3 = pointer to temporary array (size: 3*len ints)
@   [sp] = length of input/output arrays in 32-bit words

                .global acl_p_mont_inv
                .text
                .arm

x1              .req    r0      @
aa              .req    r1      @
shift_r         .req    r1      @
mm              .req    r2      @
shift_l         .req    r2      @
uu              .req    r3      @
vv              .req    r4      @
x2              .req    r5      @
cnt             .req    r6      @
kay             .req    r7      @
len_x           .req    r8      @
len_u           .req    r9      @
tmp1            .req    r10     @
tmp2            .req    r11     @
swap            .req    r12     @
```

```
acl_p_mont_inv: push    {r4-r11}
                mov     swap, #0
                ldr     len_u, [sp, #4*8]
                add     vv, uu, len_u, lsl #2
                add     x2, vv, len_u, lsl #2

                @ initialization
apmi_init:      mov     cnt, len_u
                mov     kay, #0
apmi_init_lp1:  subs    cnt, #1
                ldr     tmp1, [aa, cnt, lsl #2]
                str     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [mm, cnt, lsl #2]
                str     tmp2, [vv, cnt, lsl #2]
                str     kay, [x1, cnt, lsl #2]
                str     kay, [x2, cnt, lsl #2]
                bne     apmi_init_lp1

                mov     cnt, #1
                str     cnt, [x1]
                mov     len_x, #1

                tst     tmp1, #1
                bne     apmi_compare
                b       apmi_u_again

apmi_v_bigger:  mov     tmp1, uu
                mov     uu, vv
                mov     vv, tmp1
                mov     tmp1, x1
                mov     x1, x2
                mov     x2, tmp1
                eor     swap, #1

                @ u = u - v
apmi_u_bigger:  mov     cnt, #0
                msr     cpsr_f, #(1<<29)
apmi_u_lp1:     ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                sbcs    tmp1, tmp2
                str     tmp1, [uu, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len_u
                bne     apmi_u_lp1

                @ x1 = x1 + x2
                mov     cnt, #0
                msr     cpsr_f, #0
apmi_u_lp2:     ldr     tmp1, [x1, cnt, lsl #2]
                ldr     tmp2, [x2, cnt, lsl #2]
                adcs    tmp1, tmp2
                str     tmp1, [x1, cnt, lsl #2]
                add     cnt, #1
                teq     cnt, len_x
                bne     apmi_u_lp2

                @ make x1 and x2 longer?
```

```
                mov     tmp1, #1
                strcs   tmp1, [x1, len_x, lsl #2]
                addcs   len_x, #1

                @ count trailing zeroes of u
apmi_u_again:   ldr     tmp1, [uu]
                mov     shift_r, #0
                msr     cpsr_f, #(1<<29)
apmi_u_lp3:     movs    tmp1, tmp1, rrx
                addcc   shift_r, #1
                bcc     apmi_u_lp3
                rsb     shift_l, shift_r, #32
                add     kay, shift_r

                @ right shift u
                add     uu, len_u, lsl #2
                mov     cnt, len_u
                mov     tmp2, #0
apmi_u_lp4:     ldr     tmp1, [uu, #-4]
                orr     tmp2, tmp1, lsr shift_r
                str     tmp2, [uu, #-4]!
                mov     tmp2, tmp1, lsl shift_l
                subs    cnt, #1
                bne     apmi_u_lp4

                @ left shift x2
                mov     cnt, len_x
                mov     tmp2, #0
apmi_u_lp5:     ldr     tmp1, [x2]
                orr     tmp2, tmp1, lsl shift_r
                str     tmp2, [x2], #4
                mov     tmp2, tmp1, lsr shift_l
                subs    cnt, #1
                bne     apmi_u_lp5
                cmp     tmp2, #0
                addne   len_x, #1
                strne   tmp2, [x2], #4
                sub     x2, len_x, lsl #2

                @ shifted by 32 bits?
                cmp     shift_r, #32
                beq     apmi_u_again

                @ compare u and v
apmi_compare:   mov     shift_r, #0
                subs    cnt, len_u, #1
apmi_cmp_lp1:   ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                cmp     tmp1, tmp2
                bhi     apmi_u_bigger
                blo     apmi_v_bigger
                orrs    shift_r, tmp1
                subeq   len_u, #1
                subs    cnt, #1
                bhs     apmi_cmp_lp1

                @ u (== v) == 1 ?
                cmp     len_u, #1
```

```
                bne     apmi_not_inv
                cmp     tmp1, #1
                bne     apmi_not_inv
                cmp     swap, #0
                beq     apmi_done

                ldr     cnt, [sp, #4*8]
apmi_mov_lp1:   subs    cnt, #1
                ldr     tmp1, [x1, cnt, lsl #2]
                str     tmp1, [x2, cnt, lsl #2]
                bne     apmi_mov_lp1
                rsb     kay, kay, #0

apmi_done:      mov     r0, kay
                pop     {r4-r11}
                bx      lr

apmi_not_inv:   mov     r0, #0
                pop     {r4-r11}
                bx      lr

                .end
```

Source file 48 acl_p_mod_inv.c

```
// res = a^(-1)*(2^e) mod m, m mod 2 == 1, res != a

#include "..\acl.h"

void acl_p_mod_inv(vect res, vect a, uint e, vect m, vect3 tmp, size_t len)
{
    int k; uint m_inv;

    k = e;
    acl_mov32(res, 0, len);
    if(!acl_zero(a, len)) {
        res[0] = 1;
        if(acl_cmp(res, a, len)) {
            k = acl_p_mont_inv(res, a, m, tmp, len);
            if(k == 0)
                acl_mov32(res, 0, len);
            else if(k < 0) {
                k = -k;
                acl_p_mod_sub(res, m, res, m, len);
            }
        }
        if(k < e)
            acl_p_mod_dbl(res, e - k, m, len);
        else {
            k = k - e;
            m_inv = acl_p_mont_m_inv(m);
            while(k >= 32*len) {
                k -= 32*len;
                acl_mov32(tmp + len, 0, len);
                acl_mov(tmp, res, len);
                acl_p_mont_red(res, tmp, m, m_inv, len);
            }
```

```
            if((k > len) && (k >= 32)) {
                acl_mov32(tmp, 0, 2*len);
                acl_mov(tmp + len - (k >> 5), res, len);
                acl_p_mont_red(res, tmp, m, m_inv, len);
                k -= 32*(k >> 5);
            }
            if(k) acl_p_mod_hlv(res, k, m, len);
        }
    }
}
```

Source file 49 acl_p_mod.c

```
// res[len] = a[len_a] mod m[len]
// does not work in-place (res != a) !!!

#include "..\acl.h"

void acl_p_mod(vect res, vect a, size_t len_a, vect m, size_t len)
{
    int k;

    if(len_a < len) {
        acl_mov(res, a, len_a);
        acl_mov32(res + len_a, 0, len - len_a); k = 0;
    } else {
        acl_mov(res, a + (len_a - len), len); k = 32 * (len_a - len);
    }
    while(acl_cmp(res, m, len) >= 0) { acl_rsh(res, 1, len); k++; }
    while(k--) {
        acl_p_mod_dbl(res, 1, m, len);
        if(acl_bit(a, k, len_a)) acl_p_mod_add32(res, res, 1, m, len);
    }
}
```

Source file 50 acl_p_div.c

```
// a[len_a] = a[len_a] div m[len]  (tmp[len])

#include "..\acl.h"

void acl_p_div(vect a, size_t len_a, vect m, vect tmp, size_t len)
{
    int k; int h;

    if(len_a < len) {
        acl_mov(tmp, a, len_a);
        acl_mov32(tmp + len_a, 0, len - len_a); k = 0;
    } else {
        acl_mov(tmp, a + (len_a - len), len); k = 32 * (len_a - len);
    }
    while(acl_cmp(tmp, m, len) >= 0) { acl_rsh(tmp, 1, len); k++; }
    for(h = 32*len_a; h > k; h--) acl_bit_clr(a, h - 1);
    while(k--) {
        h = acl_p_mod_dbl(tmp, 1, m, len);
        if(acl_bit(a, k, len_a)) h += acl_p_mod_add32(tmp, tmp, 1, m, len);
```

```
        if(h) acl_bit_set(a, k);
        else acl_bit_clr(a, k);
    }
}
```

Source file 51 acl_p_sqrt.c

```
// res[len]^2 = a[len_a] mod m[len]       res != a
// assuming m is an odd prime
// returns TRUE if square root exists, FALSE otherwise
// taken from http://mersennewiki.org/index.php/Modular_Square_Root

#include "..\acl.h"
#include "..\acl_int.h"

bool_t acl_p_sqrt(vect res, vect a, vect m, prng rnd, vect8 tmp, size_t len)
{
    uint m_inv, e, k, i; vect r_mod_m, r2_mod_m, t1, t2, t3;
    // tmp = tmp tmp tmp r_mod_m r2_mod_m t1 t2 t3

    r_mod_m = tmp + 3*len; r2_mod_m = r_mod_m + len;
    t1 = r2_mod_m + len; t2 = t1 + len; t3 = t2 + len;

    if(acl_zero(a, len)) {                      // if a == 0 -> res = 0;
        acl_mov(res, a, len); return TRUE;
    }
    acl_p_mod_sub32(tmp, a, 1, m, len);
    if(acl_zero(tmp, len)) {                    // if a == 1 -> res = 1;
        acl_mov(res, a, len); return TRUE;
    }
    acl_p_mont_pre(r_mod_m, r2_mod_m, &m_inv, m, len);
    acl_mov(t1, m, len);
    acl_rsh(t1, 1, len);
    acl_p_mont_exp(res, a, t1, len, m, tmp, m_inv, r2_mod_m, len);
    acl_p_mod_sub32(tmp, res, 1, m, len);
    if(!acl_zero(tmp, len)) return FALSE;       // if a^((m-1)/2) != 1, no sqrt
    switch(m[0] & 7) {
        case 3:
        case 7:                                 // if m mod 4 == 3
            acl_mov(t1, m, len);
            acl_rsh(t1, 2, len);
            acl_p_mod_add32(t1, t1, 1, 0, len);
            acl_p_mont_exp(res, a, t1, len, m, tmp, m_inv, r2_mod_m, len);
            return TRUE;
        case 5:                                 // if m mod 8 == 5
            acl_mov(t1, a, len);
            acl_p_mod_dbl(t1, 1, m, len);       // t1 = 2a
            acl_mov(t3, m, len);
            acl_rsh(t3, 3, len);                // t2 = (2a)^(m >> 3) == v
            acl_p_mont_exp(t2, t1, t3, len, m, tmp, m_inv, r2_mod_m, len);
            acl_p_mul_mont(t1, t1, r2_mod_m);   // into montgomery domain
            acl_p_mul_mont(t2, t2, r2_mod_m);
            acl_p_sqr_mont(res, t2);            // res = t2^2 == v^2
            acl_p_mul_mont(res, res, t1);       // res = 2a * v^2 == i
            acl_p_mod_sub(res, res, r_mod_m, m, len); // res = i - 1
            acl_p_mul_mont(res, res, t2);       // res = v * (i - 1)
            acl_p_mod_hlv(t1, 1, m, len);       // t1 = t1 / 2 == a
```

```
            acl_p_mul_mont(res, res, t1);       // res = a * v * (i - 1)
            acl_mov(tmp, res, len);             // out of montgomery domain
            acl_mov32(tmp + len, 0, len);
            acl_p_mont_red(res, tmp, m, m_inv, len);
            return TRUE;
        case 1:                                 // if m mod 8 == 1
            // res  t1  t2  t3
            // y    q   x   a
            //      v   d   w
            acl_p_mod_sub32(t1, m, 1, 0, len);
            e = acl_ctz(t1, len);
            acl_rsh(t1, e, len);                // t1 = q
            do {
                rnd(t2, len);               // t2 = x        res = x^q == z
                acl_p_mont_exp(res, t2, t1, len, m, tmp, m_inv, r2_mod_m, len);
                acl_p_mul_mont(res, res, r2_mod_m); // into montgomery domain
                acl_mov(t3, res, len);
                for(i=0; i<e-1; i++) { acl_p_sqr_mont(t3, t3); }
                acl_p_mod_add(t3, t3, r_mod_m, m, len);
            } while(!acl_zero(t3, len));        // repeat until x^m == -1
            acl_rsh(t1, 1, len);                // t1 = (q - 1)/2
            acl_p_mont_exp(t2, a, t1, len, m, tmp, m_inv, r2_mod_m, len);
            acl_p_mul_mont(t2, t2, r2_mod_m);   // x = a^((q - 1)/2)
            acl_p_mul_mont(t3, a, r2_mod_m);
            acl_p_mul_mont(t1, t2, t3);         // v = x * a
            acl_p_mul_mont(t3, t1, t2);         // w = v * x
            while(acl_cmp(t3, r_mod_m, len)) {  // while w != 1
                k = 0;
                acl_mov(t2, t3, len);           // t2 = w
                do {
                    k++;
                    acl_p_sqr_mont(t2, t2);
                } while(acl_cmp(t2, r_mod_m, len));
                acl_mov(t2, res, len);          // t2 = y
                for(i=0; i<e-k-1; i++) { acl_p_sqr_mont(t2, t2); }  // t2 = d
                acl_p_sqr_mont(res, t2);        // y = d^2
                e = k;
                acl_p_mul_mont(t1, t1, t2);     // v = d * v
                acl_p_mul_mont(t3, t3, res);    // w = w * y
            }                                   // w == 1, return v
            acl_mov(tmp, t1, len);              // out of montgomery domain
            acl_mov32(tmp + len, 0, len);
            acl_p_mont_red(res, tmp, m, m_inv, len);
            return TRUE;
        default:
            return FALSE;                   // m is composite - beyond our scope
    }
}
```

Source file 52 acl_p_fr.s

```
BORDER  =   512

@ void acl_p_fr(vect res, vect2 a, list data, size_t len);
@   res = a mod (2^exp1 +- 2^exp2 +- ... +- 2^0)
@         does not work in-place (res != a)
@
```

```
@   the terms are listed in a table pointed to by data:
@   - the first entry must be the highest exponent
@   - for a negative term (-2^exp), list the exponent (must be <= BORDER)
@   - for a positive term (+2^exp), list the ones' complement of the exponent
@
@   the table is terminated:
@   - by a zero, this is also considered a term (-2^0)
@   - by a value BORDER < val < 0x80000000, each of the 32 bits that is '1'
@     encodes a term in the form -2^(bit position)
@
@   examples of lists:
@       128, 1, 0
@       128, 97, 0
@       160, 32, 0x538d             @ 14, 12, 9, 8, 7, 3, 2, 0
@       160, 31, 0
@       192, 32, 0x11c9             @ 12, 8, 7, 6, 3, 0
@       192, 64, 0
@       224, 32, 0x1a93             @ 12, 11, 9, 7, 4, 1, 0
@       224, 96, ~0
@       256, 32, 0x03d1             @ 9, 8, 7, 6, 4, 0
@       256, 224, ~192, ~96, 0
@       384, 128, 96, ~32, 0
@       521, 0
@
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input
@   r2 = pointer to exponent table
@   r3 = length of input/output arrays in 32-bit words

                .global acl_p_fr
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
tab             .req    r2      @
len             .req    r3      @
carry           .req    r4      @
tmp1            .req    r5      @
tmp2            .req    r6      @
ind1            .req    r7      @
ind2            .req    r8      @
shift_r         .req    r9      @
shift_l         .req    r10     @
tmp3            .req    r11     @
tmp4            .req    r12     @
cnt             .req    r14     @

acl_p_fr:       push    {r4-r11, r14}
                push    {tab, len}
                b       apfr_again

                @ get exponent
apfr_main_lp:   mov     ind2, dest
                mov     cnt, len
                ldr     tmp1, [tab, #4]!
                cmp     tmp1, #0
```

```
                bmi     apfr_subtract
                cmp     tmp1, #BORDER
                bhi     apfr_mul

                @ src += dest << exp
                mov     ind1, tmp1, lsr #5
                add     ind1, src, ind1, lsl #2
                ands    shift_l, tmp1, #31
                beq     apfr_a_skip1
                rsb     shift_r, shift_l, #32

                @ add with shift
                msr     cpsr_f, #0
                mov     carry, #0
                tst     cnt, #1
                beq     apfr_a_lp1
                ldr     tmp1, [ind1]
                ldr     tmp3, [ind2], #4
                adcs    tmp1, tmp3, lsl shift_l
                mov     carry, tmp3, lsr shift_r
                str     tmp1, [ind1], #4
                sub     cnt, #1
                teq     cnt, #0
                beq     apfr_a_lp2
apfr_a_lp1:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
                orr     carry, tmp3, lsl shift_l
                adcs    tmp1, carry
                mov     carry, tmp3, lsr shift_r
                orr     carry, tmp4, lsl shift_l
                adcs    tmp2, carry
                mov     carry, tmp4, lsr shift_r
                stmia   ind1!, {tmp1, tmp2}
                sub     cnt, #2
                teq     cnt, #0
                bne     apfr_a_lp1

                @ propagate carry
apfr_a_lp2:     ldr     tmp1, [ind1]
                adcs    tmp1, carry
                str     tmp1, [ind1], #4
                mov     carry, #0
                bcs     apfr_a_lp2
                b       apfr_main_lp

                @ add
apfr_a_skip1:   msr     cpsr_f, #0
                tst     cnt, #1
                beq     apfr_a_lp3
                ldr     tmp1, [ind1]
                ldr     tmp3, [ind2], #4
                adcs    tmp1, tmp3
                str     tmp1, [ind1], #4
                sub     cnt, #1
                teq     cnt, #0
                beq     apfr_a_lp3_d
apfr_a_lp3:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
```

```
                adcs    tmp1, tmp3
                adcs    tmp2, tmp4
                stmia   ind1!, {tmp1, tmp2}
                sub     cnt, #2
                teq     cnt, #0
                bne     apfr_a_lp3
apfr_a_lp3_d:   bcc     apfr_a_skip2

                @ propagate carry
apfr_a_lp4:     ldr     tmp1, [ind1]
                adcs    tmp1, #0
                str     tmp1, [ind1], #4
                bcs     apfr_a_lp4

apfr_a_skip2:   ldr     tmp1, [tab]
                cmp     tmp1, #0
                bne     apfr_main_lp
                b       apfr_again

                @ src -= dest << exp
apfr_subtract:  mvn     tmp2, tmp1
                cmp     tmp2, #BORDER
                bhi     apfr_mul
                mov     ind1, tmp2, lsr #5
                add     ind1, src, ind1, lsl #2
                ands    shift_l, tmp2, #31
                beq     apfr_s_skip1
                rsb     shift_r, shift_l, #32

                @ subtract with shift
                msr     cpsr_f, #(1<<29)
                mov     carry, #0
                tst     cnt, #1
                beq     apfr_s_lp1
                ldr     tmp1, [ind1]
                ldr     tmp3, [ind2], #4
                sbcs    tmp1, tmp3, lsl shift_l
                mov     carry, tmp3, lsr shift_r
                str     tmp1, [ind1], #4
                sub     cnt, #1
                teq     cnt, #0
                beq     apfr_s_lp2
apfr_s_lp1:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
                orr     carry, tmp3, lsl shift_l
                sbcs    tmp1, carry
                mov     carry, tmp3, lsr shift_r
                orr     carry, tmp4, lsl shift_l
                sbcs    tmp2, carry
                mov     carry, tmp4, lsr shift_r
                stmia   ind1!, {tmp1, tmp2}
                sub     cnt, #2
                teq     cnt, #0
                bne     apfr_s_lp1

                @ propagate borrow
apfr_s_lp2:     ldr     tmp1, [ind1]
                sbcs    tmp1, carry
```

```
                str     tmp1, [ind1], #4
                mov     carry, #0
                bcc     apfr_s_lp2
                b       apfr_main_lp

                @ subtract
apfr_s_skip1:   msr     cpsr_f, #(1<<29)
                tst     cnt, #1
                beq     apfr_s_lp3
                ldr     tmp1, [ind1]
                ldr     tmp3, [ind2], #4
                sbcs    tmp1, tmp3
                str     tmp1, [ind1], #4
                sub     cnt, #1
                teq     cnt, #0
                beq     apfr_s_lp3_d
apfr_s_lp3:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
                sbcs    tmp1, tmp3
                sbcs    tmp2, tmp4
                stmia   ind1!, {tmp1, tmp2}
                sub     cnt, #2
                teq     cnt, #0
                bne     apfr_s_lp3
apfr_s_lp3_d:   bcs     apfr_s_skip2

                @ propagate borrow
apfr_s_lp4:     ldr     tmp1, [ind1]
                sbcs    tmp1, #0
                str     tmp1, [ind1], #4
                bcc     apfr_s_lp4

apfr_s_skip2:   ldr     tmp1, [tab]
                cmp     tmp1, #~0
                bne     apfr_main_lp
                b       apfr_again

                @ src += dest * exp
apfr_mul:       mov     ind1, src
                mov     tab, tmp1
                mov     carry, #0
                tst     cnt, #1
                beq     apfr_u_lp1
                ldr     tmp1, [ind1]
                ldr     tmp3, [ind2], #4

                umull   shift_l, shift_r, tmp3, tab
                adds    tmp1, shift_l
                adc     carry, shift_r, #0

                str     tmp1, [ind1], #4
                sub     cnt, #1
                teq     cnt, #0
                beq     apfr_u_lp1_d

apfr_u_lp1:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
```

```
                umull   shift_l, shift_r, tmp3, tab
                adds    shift_l, carry
                adc     shift_r, #0
                adds    tmp1, shift_l
                adc     carry, shift_r, #0

                umull   shift_l, shift_r, tmp4, tab
                adds    shift_l, carry
                adc     shift_r, #0
                adds    tmp2, shift_l
                adc     carry, shift_r, #0

                stmia   ind1!, {tmp1, tmp2}
                subs    cnt, #2
                bne     apfr_u_lp1

apfr_u_lp1_d:   ldr     tmp1, [ind1]
                adds    tmp1, carry
                str     tmp1, [ind1], #4
                bcc     apfr_again

                @ propagate carry
apfr_u_lp2:     ldr     tmp1, [ind1]
                adcs    tmp1, #0
                str     tmp1, [ind1], #4
                bcs     apfr_u_lp2

                @ dest = src(hi), src(hi) = 0
apfr_again:     mov     tmp3, #0
                mov     tmp4, #0
                mov     cnt, len
                mov     ind2, dest
                ldr     tab, [sp]
                ldr     tmp1, [tab]
                mov     ind1, tmp1, lsr #5
                add     ind1, src, ind1, lsl #2
                ands    shift_r, tmp1, #31
                beq     apfr_m_lp2
                rsb     shift_l, shift_r, #32
                ldr     tmp1, [ind1]
                mov     tmp2, tmp1, lsl shift_l
                lsr     tmp2, shift_l
                str     tmp2, [ind1], #4
                mov     carry, tmp1, lsr shift_r

                @ move with shift
apfr_m_lp1:     ldmia   ind1, {tmp1, tmp2}
                orr     carry, tmp1, lsl shift_l
                str     carry, [ind2], #4
                mov     carry, tmp1, lsr shift_r
                orr     carry, tmp2, lsl shift_l
                subs    cnt, #2
                strhs   carry, [ind2], #4
                movhs   carry, tmp2, lsr shift_r
                stmhsia ind1!, {tmp3, tmp4}
                bhi     apfr_m_lp1
                strlo   tmp3, [ind1]
                b       apfr_c_lp1
```

```
                @ move
apfr_m_lp2:     ldmia   ind1, {tmp1, tmp2}
                subs    cnt, #2
                stmhsia ind2!, {tmp1, tmp2}
                stmhsia ind1!, {tmp3, tmp4}
                bhi     apfr_m_lp2
                strlo   tmp1, [ind2], #4
                strlo   tmp3, [ind1]

apfr_c_lp1:     ldr     tmp1, [ind2, #-4]!
                cmp     tmp1, #0
                bne     apfr_main_lp
                subs    len, #1
                bne     apfr_c_lp1

                @ dest = src(lo)
                pop     {tab, len}
apfr_d_lp1:     ldmia   src!, {tmp1, tmp2}
                subs    len, #2
                stmhsia dest!, {tmp1, tmp2}
                bhi     apfr_d_lp1
                strlo   tmp1, [dest]

                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 53 acl_prng_lc.s

```
@ void acl_prng_lc_init(uint seed);
@   initialize linear congruential prng
@ on entry:
@   r0 = seed

@ void acl_prng_lc(vect res, size_t len);
@   write output array with pseudorandom numbers from linear congruential prng
@   prng: x = (279470273 * x) mod (2^32-5)
@   parameters taken from:
@       Tables of Linear Congruential Generators of Different Sizes
@       and Good Lattice Structure
@       Pierre L'Ecuyer
@       Mathematics of Computation, Vol. 68, No. 225 (Jan., 1999), pp. 249-260
@
@ on entry:
@   r0 = pointer to result
@   r1 = length of input/output arrays in 32-bit words

                .global acl_prng_lc_init
                .global acl_prng_lc

                .data
acl_prng_lc_val:        .int    1

                .text
                .arm
```

```
seed            .req    r0      @
ptr             .req    r1      @

out             .req    r0      @
len             .req    r1      @
tmp1            .req    r2      @
tmp2            .req    r3      @
sum1            .req    r4      @
sum2            .req    r5      @
top             .req    r12     @

acl_prng_lc_init:
                ldr     ptr, =acl_prng_lc_val
                str     seed, [ptr]
                bx      lr

acl_prng_lc:    push    {r4-r5}
                ldr     tmp1, =acl_prng_lc_val
                ldr     sum1, [tmp1]
aplc_lp1:       ldr     tmp1, =279470273
                mov     tmp2, sum1
                umull   sum1, top, tmp1, tmp2
aplc_lp2:       mov     sum2, top, lsr #30
                adds    sum1, top
                adc     sum2, #0
                adds    sum1, top, lsl #2
                adc     sum2, #0
                movs    top, sum2
                bne     aplc_lp2
                cmp     sum1, #0
                moveq   sum1, #1
                str     sum1, [out], #4
                subs    len, #1
                bne     aplc_lp1
                ldr     tmp1, =acl_prng_lc_val
                str     sum1, [tmp1]
                pop     {r4-r5}
                bx      lr

                .end
```

Source file 54 acl_prng_aes.c

```
#include "..\acl.h"
#include "..\acl_config.h"

static uint apa_key[ACL_PRNG_AES_SIZE];
static uint apa_key_exp[(ACL_PRNG_AES_SIZE + 7)*4];
static uint apa_cntr[4];
static uint apa_tmp[4];
static const uint apa_in[4] = {0, 0, 0, 0};

void acl_prng_aes_init(prng rnd)
{
    rnd(apa_key, ACL_PRNG_AES_SIZE);
    acl_aes_key_en(apa_key_exp, apa_key, ACL_PRNG_AES_SIZE);
```

```
    rnd(apa_cntr, 4);
}

void acl_prng_aes(vect res, size_t len)
{
    int i;

    for(i = 0; i < len; i++) {
        acl_aes_cntr(apa_tmp, (vect) apa_in, apa_key_exp, \
                     ACL_PRNG_AES_SIZE, apa_cntr);
        res[i] = apa_tmp[0];
    }
}
```

Source file 55 acl_prng_sha.c

```
#include "..\acl.h"

static uint aps_state[23];
static uint aps_cntr[5];

void acl_prng_sha_init(prng rnd)
{
    rnd(aps_cntr, 5);
}

void acl_prng_sha(vect res, size_t len)
{
    int i, j;

    for(i = 0; i < len; i++) {
        acl_sha1_init(aps_state);
        for(j = 0; j < 20; j++) {
            acl_sha1(aps_state, ((byte *) aps_cntr)[j]);
        }
        acl_sha1_done(aps_state);
        res[i] = aps_state[0];
        acl_p_mod_add32(aps_cntr, aps_cntr, 1, 0, 5);
    }
}
```

Source file 56 acl_prng_bbs.c

```
#include "..\acl.h"
#include "..\acl_int.h"
#include "..\acl_config.h"

static uint apb_m[2*ACL_PRNG_BBS_SIZE];
static uint apb_tmp[4*ACL_PRNG_BBS_SIZE];
static uint apb_x[2*ACL_PRNG_BBS_SIZE];
#if ACL_PRNG_BBS_MONT == 0
static uint apb_y[2*ACL_PRNG_BBS_SIZE];
#endif
static uint apb_m_inv;

// m = product of two primes (each == 3 mod 4)
```

```
// x = random number coprime with m
// x = x * R (go into montgomery domain)
// note that vect7 here means 7*ACL_PRNG_BBS_SIZE

void acl_prng_bbs_init(prng rnd_fast, prng rnd, vect7 tmp)
{
    do {
        acl_p_rnd_prime(apb_x, tmp, ACL_PRNG_BBS_K, 1, \
                        rnd_fast, rnd, ACL_PRNG_BBS_SIZE);
        acl_p_rnd_prime(apb_x + ACL_PRNG_BBS_SIZE, tmp, ACL_PRNG_BBS_K, 1, \
                        rnd_fast, rnd, ACL_PRNG_BBS_SIZE);
    } while(!acl_cmp(apb_x, apb_x + ACL_PRNG_BBS_SIZE, ACL_PRNG_BBS_SIZE));
    acl_p_mul(apb_m, apb_x, apb_x + ACL_PRNG_BBS_SIZE, ACL_PRNG_BBS_SIZE);
    do {
        rnd(apb_x, 2*ACL_PRNG_BBS_SIZE);
    } while(!acl_p_coprime(apb_x, apb_m, tmp, 2*ACL_PRNG_BBS_SIZE));
    acl_p_mont_pre(0, apb_tmp, &apb_m_inv, apb_m, 2*ACL_PRNG_BBS_SIZE);
    acl_p_mul(tmp, apb_x, apb_tmp, 2*ACL_PRNG_BBS_SIZE);
    acl_p_mont_red(apb_x, tmp, apb_m, apb_m_inv, 2*ACL_PRNG_BBS_SIZE);
}

// for each bit: x = (x^2)/R
// note that the least significant bit can be taken from x^(2i) mod m (slower)
//   or from its montgomery representation x^(2i)*R mod m (faster)
// if multiplication by a non-zero number R modulo m is a one-to-one mapping
//   (ask a mathematician...) then this *should* be equivalent

void acl_prng_bbs(vect res, size_t len)
{
    int i;

    acl_mov32(res, 0, len);
    for(i = 0; i < 32*len; i++) {
        acl_p_sqr(apb_tmp, apb_x, 2*ACL_PRNG_BBS_SIZE);
        acl_p_mont_red(apb_x, apb_tmp, apb_m, apb_m_inv, 2*ACL_PRNG_BBS_SIZE);
#if ACL_PRNG_BBS_MONT == 0
        acl_mov(apb_tmp, apb_x, 2*ACL_PRNG_BBS_SIZE);
        acl_mov32(apb_tmp + 2*ACL_PRNG_BBS_SIZE, 0, 2*ACL_PRNG_BBS_SIZE);
        acl_p_mont_red(apb_y, apb_tmp, apb_m, apb_m_inv, 2*ACL_PRNG_BBS_SIZE);
        if(apb_y[0] & 1) acl_bit_set(res, i);
#else
        if(apb_x[0] & 1) acl_bit_set(res, i);
#endif
    }
}
```

Source file 57 gen_primes.txt

```
# this is a calc script. calc can be found here:
# http://isthe.com/chongo/tech/comp/calc/

# generate products of first couple of primes
# that will fit into n 32-bit words

max_n = 32;
base(16);
fp = fopen("acl_pop_table.txt", "w");
```

```
fprintf(fp, "@ products-of-primes tables");
p = 3;
prod = 1;
oprod = 1;
limit = 2^32;

for(i=0; i<max_n; i++) {
        while(prod < limit) {
                if(isprime(p)) {
                        oprod = prod;
                        prod *= p;
                }
                p++;
        }

        fprintf(fp, "\n.int ");
        out = oprod;
        for(k=0; k<i; k++) {
                fprintf(fp, "%x, ", out & 0xFFFFFFFF);
                out = out >> 32;
        }
        fprintf(fp, "%x", out & 0xFFFFFFFF);

        limit *= 2^32;
}
fprintf(fp, "\n");
fclose(fp);
```

Source file 58 acl_p_tables.s

```
@ tables used by the GF(p) routines

                .global acl_pop_table
                .text
                .align 2
acl_pop_table:  .include "./primes/acl_pop_table.txt"
                .end
```

Source file 59 acl_p_rm_test2.c

```
// rabin-miller test with a == 2
// returns false if 2 proves the compositeness of m, true otherwise
// len is the length of m, r_mod_m, res in 32-bit words
// tmp is used for temporary storage;
//   its size should be at least (3 x len) 32-bit words

#include "..\acl.h"
#include "..\acl_int.h"

bool_t acl_p_rm_test2(vect m, vect3 tmp, uint m_inv, vect r_mod_m, size_t len)
{
    int i, k; vect res;    // tmp tmp res

    res = tmp + 2*len;

    i = acl_log2(m, len);
```

```
    if(i < 2) return FALSE;
    k = 1;
    while(!acl_bit(m, k, len)) k++;
    acl_mov(res, r_mod_m, len);
    acl_p_mod_dbl(res, 1, m, len);
    while(i > k) {
        i--;
        acl_p_sqr_mont(res, res);
        if(acl_bit(m, i, len)) acl_p_mod_dbl(res, 1, m, len);
    }
    if(acl_cmp(res, r_mod_m, len) == 0) return TRUE;
    acl_p_mod_add(tmp, res, r_mod_m, m, len);
    if(acl_zero(tmp, len)) return TRUE;
    k--;
    while(k--) {
        acl_p_sqr_mont(res, res);
        acl_p_mod_add(tmp, res, r_mod_m, m, len);
        if(acl_zero(tmp, len)) return TRUE;
    }
    return FALSE;
}
```

Source file 60 acl_p_rm_test.c

```
// rabin-miller test with generic a
// returns false if a proves the compositeness of m, true otherwise
// len is the length of a, m, r_mod_m, r2_mod_m, res, a_r in 32-bit words
// tmp is used for temporary storage;
//   its size should be at least (4 x len) 32-bit words

#include "..\acl.h"
#include "..\acl_int.h"

bool_t acl_p_rm_test(vect a, vect m, vect4 tmp, uint m_inv, \
                     vect r_mod_m, vect r2_mod_m, size_t len)
{
    uint i, k; vect res, a_r;   // tmp tmp res a_r

    res = tmp + 2*len; a_r = res + len;
    i = acl_log2(m, len);
    if(i < 2) return FALSE;
    k = 1;
    while(!acl_bit(m, k, len)) k++;
    acl_p_mul_mont(a_r, a, r2_mod_m);
    acl_mov(res, a_r, len);
    while(i > k) {
        i--;
        acl_p_sqr_mont(res, res);
        if(acl_bit(m, i, len)) acl_p_mul_mont(res, res, a_r);
    }
    if(acl_cmp(res, r_mod_m, len) == 0) return TRUE;
    acl_p_mod_add(tmp, res, r_mod_m, m, len);
    if(acl_zero(tmp, len)) return TRUE;
    k--;
    while(k--) {
        acl_p_sqr_mont(res, res);
        acl_p_mod_add(tmp, res, r_mod_m, m, len);
```

```
        if(acl_zero(tmp, len)) return TRUE;
    }
    return FALSE;
}
```

Source file 61 acl_p_rnd_prime.c

```
// returns in res a random probable prime of length len
// runs the rabin-miller test k-times
// sets the msb and lsb; also sets bit "also_set";
// if you don't want to set any bit other than msb and lsb, set "also_set"
//   to zero - sets the lsb again
// len is the length of res in 32-bit words; tmp is (7 x len) ints big,
//   used for temporary storage

#include "..\acl.h"
#include "..\acl_config.h"

void acl_p_rnd_prime(vect res, vect7 tmp, uint k, uint also_set, \
                     prng rnd_fast, prng rnd_strong, size_t len)
{
    uint m_inv, cnt; vect ptr, tmp1, tmp2, r_mod_m, r2_mod_m, aa;

    // tmp tmp tmp1 tmp2 r_mod_m r2_mod_m aa
    tmp1 = tmp + 2*len; tmp2 = tmp1 + len; r_mod_m = tmp2 + len;
    r2_mod_m = r_mod_m + len; aa = r2_mod_m + len;

    rnd_strong(res, len);           // generate random number
    acl_bit_set(res, 0);            // make sure number is odd
    while(1) {
        acl_p_mod_add32(res, res, 2, 0, len); // increment candidate number
        acl_bit_set(res, 32*len-1); // make sure number is full-length
        acl_bit_set(res, also_set); // allow user to set arbitrary bit
        if(len <= ACL_POP_SIZE) {
            ptr = (vect) ((uint) &acl_pop_table + 2*len*(len-1));
        } else {
            ptr = (vect) ((uint) &acl_pop_table \
                          + 2*ACL_POP_SIZE*(ACL_POP_SIZE-1));
            acl_mov32(tmp, 0, len);
            acl_mov(tmp, ptr, ACL_POP_SIZE);
            ptr = tmp;
        }
        if(acl_p_coprime(res, ptr, tmp1, len)) {
            acl_p_mont_pre(r_mod_m, r2_mod_m, &m_inv, res, len);
            if(acl_p_rm_test2(res, tmp, m_inv, r_mod_m, len)) {
                cnt = k;
                do {
                    if(cnt-- == 0) return;
                    rnd_fast(tmp, len);
                    acl_p_mod(aa, tmp, len, res, len);
                } while(acl_p_rm_test(aa, res, tmp, m_inv, \
                                   r_mod_m, r2_mod_m, len));
            }
        }
    }
}
```

Source file 62 acl_rsa_pre.c

```
// calculate values necessary for RSA
// input: e, p, q (p and q have to be stored in ram - they get dec'd and inc'd)
// output: n, d
// output: dmp1, dmq1, iqmp (if 0, will not be generated)
// returns false if gcd(phi, e) != 1
// len is the length of p, q in 32-bit words
// tmp is used for temporary storage (6 x len) 32-bit words

#include "..\acl.h"

#define phi n

bool_t acl_rsa_pre(vect2 n, vect2 d, vect dmp1, vect dmq1, vect iqmp, \
                   vect2 e, vect p, vect q, vect6 tmp, size_t len)
{
    p[0]--; q[0]--;                                // p = p - 1,  q = q - 1
    acl_p_mul(phi, p, q, len);                     // phi = (p - 1) * (q - 1)
    if(!acl_p_coprime(e, phi, tmp, 2*len)) return FALSE;
    acl_p_mod_inv(d, phi, 0, e, tmp, 2*len);       // d = phi^(-1) mod e
    acl_p_mul(tmp, d, phi, 2*len);                 // tmp = d * phi
    acl_p_mod_sub32(tmp, tmp, 1, 0, 4*len);        // tmp = d * phi - 1
    acl_p_div(tmp, 4*len, e, tmp + 4*len, 2*len);  // tmp = tmp / e
    acl_p_mod(d, tmp, 4*len, phi, 2*len);          // d = tmp mod phi
    acl_p_mod_sub(d, phi, d, phi, 2*len);          // d = -d
    if(dmp1) acl_p_mod(dmp1, d, 2*len, p, len);    // dmp1 = d mod (p - 1)
    if(dmq1) acl_p_mod(dmq1, d, 2*len, q, len);    // dmq1 = d mod (q - 1)
    p[0]++; q[0]++;                                // p = p + 1,  q = q + 1
    if(iqmp) acl_p_mod_inv(iqmp, q, 0, p, tmp, len); // iqmp = q^(-1) mod p
    acl_mov32(tmp + len, 0, len);
    acl_mov(tmp, p, len);
    acl_p_mod_add(phi, phi, tmp, 0, 2*len);        // phi = phi + p
    acl_mov(tmp, q, len);
    acl_p_mod_add(phi, phi, tmp, 0, 2*len);        // phi = phi + q
    acl_p_mod_sub32(n, phi, 1, 0, 2*len);          // n = phi - 1
    return TRUE;
}
```

Source file 63 acl_rsa_crt.c

```
// RSA decryption using CRT
// pt = decrypt(ct)

#include "..\acl.h"

#define sq pt

void acl_rsa_crt(vect2 pt, vect2 ct, \
                 vect p, vect r2_mod_p, uint p_inv, \
                 vect q, vect r2_mod_q, uint q_inv,
                 vect dmp1, vect dmq1, vect iqmp, vect4 tmp, size_t len)
{
    vect sp;

    sp = tmp + 3*len;
```

```
    acl_p_mod(sp, ct, 2*len, p, len);       // sp = ct mod p
    acl_p_mont_exp(sp, sp, dmp1, len, p, tmp, p_inv, r2_mod_p, len);
                                            // sp = sp^dmp1 mod p
    acl_p_mod(sq, ct, 2*len, q, len);       // sq = ct mod q
    acl_p_mont_exp(sq, sq, dmq1, len, q, tmp, q_inv, r2_mod_q, len);
                                            // sq = sq^dmq1 mod q
    acl_p_mod_sub(sp, sp, sq, p, len);      // sp = (sp - sq) mod p
    acl_p_mul(tmp, sp, iqmp, len);          // tmp = sp * iqmp
    acl_p_mod(sp, tmp, 2*len, p, len);      // sp = tmp mod p
    acl_p_mul(tmp, sp, q, len);             // tmp = sp * q
    acl_mov32(pt + len, 0, len);
    acl_p_mod_add(pt, sq, tmp, 0, 2*len);   // pt = sq + tmp
}
```

Source file 64 acl_2_mod_hlv.s

```
@ void acl_2_mod_hlv(vect a, uint k, vect poly, size_t len);
@   k times: if a mod z == 1 then a = (a + poly)/z mod poly
@                            else a = (a/z) mod poly
@   poly mod z must be equal to 1
@ on entry:
@   r0 = pointer to input/result
@   r1 = number of times to halve
@   r2 = pointer to reduction polynomial
@   r3 = length of input/output arrays in 32-bit words

                .global acl_2_mod_hlv
                .text
                .arm

dest            .req    r0      @
kay             .req    r1      @
poly            .req    r2      @
len             .req    r3      @
tmp1            .req    r4      @
tmp2            .req    r5      @
cnt             .req    r12     @

acl_2_mod_hlv:  push    {r4-r5}

a2mh_again:     ldr     tmp1, [dest]
                mov     cnt, len
                msr     cpsr_f, #0
                tst     tmp1, #1
                beq     a2mh_lp2

                @ a = (a + poly)/z
a2mh_lp1:       sub     cnt, #1
                ldr     tmp1, [dest, cnt, lsl #2]
                ldr     tmp2, [poly, cnt, lsl #2]
                eor     tmp1, tmp2
                movs    tmp1, tmp1, rrx
                str     tmp1, [dest, cnt, lsl #2]
                teq     cnt, #0
                bne     a2mh_lp1
                subs    kay, #1
```

```
                bne     a2mh_again
                b       a2mh_ret

                @ a = a/z
a2mh_lp2:       sub     cnt, #1
                ldr     tmp1, [dest, cnt, lsl #2]
                movs    tmp1, tmp1, rrx
                str     tmp1, [dest, cnt, lsl #2]
                teq     cnt, #0
                bne     a2mh_lp2
                subs    kay, #1
                bne     a2mh_again
a2mh_ret:       pop     {r4-r5}
                bx      lr

                .end
```

Source file 65 acl_2_mul.s

```
@ void acl_2_mul(vect2 res, vect a, vect b, size_t len);
@   res[2*len] = a[len] * b[len]   over gf(2^m)
@   does not work in-place (res != a, res != b)
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to first operand
@   r2 = pointer to second operand
@   r3 = length of input arrays in 32-bit words (output is twice as long)

                .global acl_2_mul
                .text
                .arm

dest            .req    r0      @
src1            .req    r1      @
src2            .req    r2      @
len             .req    r3      @
ind             .req    r4      @
pro1            .req    r5      @
pro2            .req    r6      @
res1            .req    r7      @
res2            .req    r8      @
sum1            .req    r9      @
sum2            .req    r10     @
cnt             .req    r12     @

acl_2_mul:      push    {r4-r10}
                mov     sum1, #0
                mov     sum2, #0
                mov     ind, #1
                mov     cnt, #1
                b       a2m_h1_lp2

                @ first half
a2m_h1_lp1:     sub     src1, ind, lsl #2
                add     ind, #1
                add     src2, ind, lsl #2
                mov     cnt, ind
```

```
a2m_h1_lp2:     ldr     pro1, [src1], #4
                ldr     pro2, [src2], #-4
                mov     res1, #0
                mvn     res2, #1

a2m_h1_lp3:     adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adcs    res2, res2
                bcs     a2m_h1_lp3

                eors    sum2, res2, rrx
                eor     sum1, res1, rrx
                subs    cnt, #1
                bne     a2m_h1_lp2

                @ got a diagonal
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, #0
                cmp     ind, len
                bne     a2m_h1_lp1
```

```
                @ second half
a2m_h2_lp1:     add     src2, ind, lsl #2
                sub     ind, #1
                sub     src1, ind, lsl #2
                mov     cnt, ind

a2m_h2_lp2:     ldr     pro1, [src1], #4
                ldr     pro2, [src2], #-4
                mov     res1, #0
                mvn     res2, #1

a2m_h2_lp3:     adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adc     res2, res2

                adds    pro1, pro1
                eorcs   res1, pro2
                adds    res1, res1
                adcs    res2, res2
                bcs     a2m_h2_lp3

                eors    sum2, res2, rrx
                eor     sum1, res1, rrx
                subs    cnt, #1
                bne     a2m_h2_lp2
```

```
                @ got a diagonal
                str     sum1, [dest], #4
                mov     sum1, sum2
                mov     sum2, #0
                cmp     ind, #1
                bne     a2m_h2_lp1

                str     sum1, [dest]
                pop     {r4-r10}
                bx      lr

                .end
```

Source file 66 acl_2_sqr.s

```
@ void acl_2_sqr(vect2 res, vect a, size_t len);
@   res[2*len] = a[len] * a[len]   over gf(2^m)
@   does not work in-place (res != a)
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input
@   r2 = length of input array in 32-bit words (output is twice as long)

                .global acl_2_sqr
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
len             .req    r2      @
tab             .req    r3      @
mask            .req    r4      @
pro             .req    r5      @
tmp             .req    r6      @
res             .req    r12     @

a2s_table:      .byte   0x00, 0x01, 0x04, 0x05
                .byte   0x10, 0x11, 0x14, 0x15
                .byte   0x40, 0x41, 0x44, 0x45
                .byte   0x50, 0x51, 0x54, 0x55

acl_2_sqr:      push    {r4-r6}
                adr     tab, a2s_table
                mov     mask, #0xf

a2s_lp:         ldr     pro, [src], #4
                and     tmp, mask, pro
                ldrb    res, [tab, tmp]
                and     tmp, mask, pro, lsr #4
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #8
                and     tmp, mask, pro, lsr #8
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #16
                and     tmp, mask, pro, lsr #12
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #24
```

```
                str     res, [dest], #4

                and     tmp, mask, pro, lsr #16
                ldrb    res, [tab, tmp]
                and     tmp, mask, pro, lsr #20
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #8
                and     tmp, mask, pro, lsr #24
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #16
                and     tmp, mask, pro, lsr #28
                ldrb    tmp, [tab, tmp]
                eor     res, tmp, lsl #24
                str     res, [dest], #4
                subs    len, #1
                bne     a2s_lp

                pop     {r4-r6}
                bx      lr

                .end
```

Source file 67 acl_2_mont_inv.s

```
@ int acl_2_mont_inv(vect res, vect a, vect poly, vect3 tmp, size_t len);
@   res = (a^-1) * (z^k) mod poly       (poly mod z == 1)
@   returns 0 if a is non-invertible, k otherwise
@   a != 0  and  a != 1
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to a
@   r2 = pointer to reduction polynomial
@   r3 = pointer to temporary array (size: 3*len ints)
@   [sp] = length of input/output arrays in 32-bit words

                .global acl_2_mont_inv
                .text
                .arm

x1              .req    r0      @
aa              .req    r1      @
shift_r         .req    r1      @
mm              .req    r2      @
shift_l         .req    r2      @
uu              .req    r3      @
vv              .req    r4      @
x2              .req    r5      @
cnt             .req    r6      @
kay             .req    r7      @
len_x           .req    r8      @
len_u           .req    r9      @
tmp1            .req    r10     @
tmp2            .req    r11     @
swap            .req    r12     @

acl_2_mont_inv: push    {r4-r11}
                mov     swap, #0
```

```
                ldr     len_u, [sp, #4*8]
                add     vv, uu, len_u, lsl #2
                add     x2, vv, len_u, lsl #2

                @ initialization
                mov     cnt, len_u
                mov     kay, #0
a2mi_init_lp1:  subs    cnt, #1
                ldr     tmp1, [aa, cnt, lsl #2]
                str     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [mm, cnt, lsl #2]
                str     tmp2, [vv, cnt, lsl #2]
                str     kay, [x1, cnt, lsl #2]
                str     kay, [x2, cnt, lsl #2]
                bne     a2mi_init_lp1

                mov     cnt, #1
                str     cnt, [x1]
                mov     len_x, #1

                tst     tmp1, #1
                bne     a2mi_compare
                b       a2mi_u_again

a2mi_v_bigger:  mov     tmp1, uu
                mov     uu, vv
                mov     vv, tmp1
                mov     tmp1, x1
                mov     x1, x2
                mov     x2, tmp1
                eor     swap, #1

                @ u = u + v
a2mi_u_bigger:  mov     cnt, len_u
a2mi_u_lp1:     subs    cnt, #1
                ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                eor     tmp1, tmp2
                str     tmp1, [uu, cnt, lsl #2]
                bne     a2mi_u_lp1

                @ x1 = x1 + x2
                mov     cnt, len_x
a2mi_u_lp2:     subs    cnt, #1
                ldr     tmp1, [x1, cnt, lsl #2]
                ldr     tmp2, [x2, cnt, lsl #2]
                eor     tmp1, tmp2
                str     tmp1, [x1, cnt, lsl #2]
                bne     a2mi_u_lp2

                @ count trailing zeroes of u
a2mi_u_again:   ldr     tmp1, [uu]
                mov     shift_r, #0
                msr     cpsr_f, #(1<<29)
a2mi_u_lp3:     movs    tmp1, tmp1, rrx
                addcc   shift_r, #1
                bcc     a2mi_u_lp3
                rsb     shift_l, shift_r, #32
```

```
                add     kay, shift_r

                @ right shift u
                add     uu, len_u, lsl #2
                mov     cnt, len_u
                mov     tmp2, #0
a2mi_u_lp4:     ldr     tmp1, [uu, #-4]
                orr     tmp2, tmp1, lsr shift_r
                str     tmp2, [uu, #-4]!
                mov     tmp2, tmp1, lsl shift_l
                subs    cnt, #1
                bne     a2mi_u_lp4

                @ left shift x2
                mov     cnt, len_x
                mov     tmp2, #0
a2mi_u_lp5:     ldr     tmp1, [x2]
                orr     tmp2, tmp1, lsl shift_r
                str     tmp2, [x2], #4
                mov     tmp2, tmp1, lsr shift_l
                subs    cnt, #1
                bne     a2mi_u_lp5
                cmp     tmp2, #0
                addne   len_x, #1
                strne   tmp2, [x2], #4
                sub     x2, len_x, lsl #2

                @ shifted by 32 bits?
                cmp     shift_l, #0
                beq     a2mi_u_again

                @ u == 1 ?
a2mi_compare:   ldr     tmp1, [uu]
                cmp     tmp1, #1
                beq     a2mi_cmp_one

                @ compare u and v
a2mi_not_one:   mov     shift_r, #0
                subs    cnt, len_u, #1
a2mi_cmp_lp1:   ldr     tmp1, [uu, cnt, lsl #2]
                ldr     tmp2, [vv, cnt, lsl #2]
                cmp     tmp1, tmp2
                bhi     a2mi_u_bigger
                blo     a2mi_v_bigger
                orrs    shift_r, tmp1
                subeq   len_u, #1
                subs    cnt, #1
                bhs     a2mi_cmp_lp1

a2mi_not_inv:   mov     r0, #-1
                pop     {r4-r11}
                bx      lr

                @ u == 1 ?
a2mi_cmp_one:   mov     cnt, len_u
a2mi_cmp_lp2:   subs    cnt, #1
                beq     a2mi_done1
                ldr     tmp1, [uu, cnt, lsl #2]
```

```
                cmp     tmp1, #0
                bne     a2mi_not_one
                b       a2mi_cmp_lp2

a2mi_done1:     teq     swap, #0
                beq     a2mi_done2

                @ x2 = x1
                ldr     cnt, [sp, #4*8]
a2mi_mov_lp1:   subs    cnt, #1
                ldr     tmp1, [x1, cnt, lsl #2]
                str     tmp1, [x2, cnt, lsl #2]
                bne     a2mi_mov_lp1

a2mi_done2:     mov     r0, kay
                pop     {r4-r11}
                bx      lr

                .end
```

Source file 68 acl_2_mod_inv.c

```
// res = a^(-1) mod poly, poly mod z == 1, res != a

#include "..\acl.h"

void acl_2_mod_inv(vect res, vect a, vect poly, vect3 tmp, size_t len)
{
    uint k;

    acl_mov32(res, 0, len);
    if(!acl_zero(a, len)) {
        res[0] = 1;
        if(acl_cmp(res, a, len)) {
            k = acl_2_mont_inv(res, a, poly, tmp, len);
            if(k == 0) acl_mov32(res, 0, len);
            else acl_2_mod_hlv(res, k, poly, len);
        }
    }
}
```

Source file 69 acl_2_fr.s

```
@ void acl_2_fr(vect res, vect2 a, list data, size_t len);
@   res = a mod (z^exp1 + z^exp2 + ... + z^0)
@         does not work in-place (res != a)
@
@   the exponents are listed (highest exponent must be first)
@   in a null-terminated table pointed to by data;
@   the final zero is also considered an exponent
@   - thus all polynomials must end with z^0
@
@   examples of lists:
@       113, 9, 0
@       131, 8, 3, 2, 0
@       163, 7, 6, 3, 0
```

```
@       193, 15, 0
@       233, 74, 0
@       239, 158, 0
@       283, 12, 7, 5, 0
@       409, 87, 0
@       571, 10, 5, 2, 0
@
@ on entry:
@   r0 = pointer to result
@   r1 = pointer to input
@   r2 = pointer to exponent table
@   r3 = length of input/output arrays in 32-bit words

                .global acl_2_fr
                .text
                .arm

dest            .req    r0      @
src             .req    r1      @
tab             .req    r2      @
len             .req    r3      @
carry           .req    r4      @
tmp1            .req    r5      @
tmp2            .req    r6      @
ind1            .req    r7      @
ind2            .req    r8      @
shift_r         .req    r9      @
shift_l         .req    r10     @
tmp3            .req    r11     @
tmp4            .req    r12     @
cnt             .req    r14     @

acl_2_fr:       push    {r4-r11, r14}
                push    {tab, len}
                b       a2fr_entry

                @ src ^= dest << exp
a2fr_main_lp:   mov     ind2, dest
                mov     cnt, len
                ldr     tmp1, [tab, #4]!
                mov     ind1, tmp1, lsr #5
                add     ind1, src, ind1, lsl #2
                ands    shift_l, tmp1, #31
                beq     a2fr_x_lp2
                rsb     shift_r, shift_l, #32

                @ xor with shift
                mov     carry, #0
a2fr_x_lp1:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
                eor     tmp1, carry
                eor     tmp1, tmp3, lsl shift_l
                mov     carry, tmp3, lsr shift_r
                eor     tmp2, carry
                eor     tmp2, tmp4, lsl shift_l
                subs    cnt, #2
                movhs   carry, tmp4, lsr shift_r
                stmhsia ind1!, {tmp1, tmp2}
```

```
                bhi     a2fr_x_lp1
                strlo   tmp1, [ind1], #4

                ldr     tmp1, [ind1]
                eor     tmp1, carry
                str     tmp1, [ind1]
                b       a2fr_main_lp

                @ xor
a2fr_x_lp2:     ldmia   ind1, {tmp1, tmp2}
                ldmia   ind2!, {tmp3, tmp4}
                eor     tmp1, tmp3
                eor     tmp2, tmp4
                subs    cnt, #2
                stmhsia ind1!, {tmp1, tmp2}
                bhi     a2fr_x_lp2
                strlo   tmp1, [ind1]

                ldr     tmp1, [tab]
                cmp     tmp1, #0
                bne     a2fr_main_lp

                @ dest = src(hi), src(hi) = 0
a2fr_entry:     mov     ind2, dest
                ldr     tab, [sp]
                ldr     tmp1, [tab]
                mov     ind1, tmp1, lsr #5
                add     ind1, src, ind1, lsl #2
                and     shift_r, tmp1, #31
                rsb     shift_l, shift_r, #32
                ldr     tmp1, [ind1]
                mov     tmp2, tmp1, lsl shift_l
                lsr     tmp2, shift_l
                str     tmp2, [ind1], #4
                mov     carry, tmp1, lsr shift_r

                @ move with shift
                mov     tmp3, #0
                mov     tmp4, #0
                mov     cnt, len
a2fr_m_lp1:     ldmia   ind1, {tmp1, tmp2}
                orr     carry, tmp1, lsl shift_l
                str     carry, [ind2], #4
                mov     carry, tmp1, lsr shift_r
                orr     carry, tmp2, lsl shift_l
                subs    cnt, #2
                strhs   carry, [ind2], #4
                movhs   carry, tmp2, lsr shift_r
                stmhsia ind1!, {tmp3, tmp4}
                bhi     a2fr_m_lp1
                strlo   tmp3, [ind1], #4

a2fr_c_lp1:     ldr     tmp1, [ind2, #-4]!
                cmp     tmp1, #0
                bne     a2fr_main_lp
                subs    len, #1
                bne     a2fr_c_lp1
```

```
                @ dest = src(lo)
                pop     {tab, len}
a2fr_d_lp1:     ldmia   src!, {tmp1, tmp2}
                subs    len, #2
                stmhsia dest!, {tmp1, tmp2}
                bhi     a2fr_d_lp1
                strlo   tmp1, [dest]

                pop     {r4-r11, r14}
                bx      lr

                .end
```

Source file 70 acl_secp112r1.c

```
#include "..\acl.h"

const uint acl_secp112r1_m[] = {
    0xfffffffd, 0xffffffff, 0xffffffff, 0xffffffff,
    0xbead208b, 0x5e668076, 0x2abf62e3, 0x0000db7c
};

const uint acl_secp112r1_fr[] = { 128, 1, 0 };

const uint acl_secp112r1_g[] = {
    0xf9c2f098, 0x5ee76b55, 0x7239995a, 0x00000948,
    0x0ff77500, 0xc0a23e0e, 0xe5af8724, 0x0000a89c
};

const uint acl_secp112r1_b[] = {
    0x11702b22, 0x16eede89, 0xf8ba0439, 0x0000659e
};

const uint acl_secp112r1_o[] = {
    0xac6561c5, 0x5e7628df, 0x2abf62e3, 0x0000db7c
};

const ecc_t acl_secp112r1 = {
    "secp112r1",
    ECC_P + ECC_A,
    4,
    (vect) acl_secp112r1_m,
    (list) acl_secp112r1_fr,
    (vect2) acl_secp112r1_g,
    (vect) -3,
    (vect) acl_secp112r1_b,
    (vect) acl_secp112r1_o,
    4,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 71 acl_secp112r2.c

```
#include "..\acl.h"
```

```
const uint acl_secp112r2_m[] = {
    0xfffffffd, 0xffffffff, 0xffffffff, 0xffffffff,
    0xbead208b, 0x5e668076, 0x2abf62e3, 0x0000db7c
};

const uint acl_secp112r2_fr[] = { 128, 1, 0 };

const uint acl_secp112r2_g[] = {
    0xd0928643, 0xb4e1649d, 0x0ab5e892, 0x00004ba3,
    0x6e956e97, 0x3747def3, 0x46f5882e, 0x0000adcd
};

const uint acl_secp112r2_a[] = {
    0x5c0ef02c, 0x8a0aaaf6, 0xc24c05f3, 0x00006127
};

const uint acl_secp112r2_b[] = {
    0x4c85d709, 0xed74fcc3, 0xf1815db5, 0x000051de
};

const uint acl_secp112r2_o[] = {
    0x0520d04b, 0xd7597ca1, 0x0aafd8b8, 0x000036df
};

const ecc_t acl_secp112r2 = {
    "secp112r2",
    ECC_P + ECC_A,
    4,
    (vect) acl_secp112r2_m,
    (list) acl_secp112r2_fr,
    (vect2) acl_secp112r2_g,
    (vect) acl_secp112r2_a,
    (vect) acl_secp112r2_b,
    (vect) acl_secp112r2_o,
    4,
    4,
    (void *) &acl_p_ecc_func
};
```

Source file 72 acl_secp128r1.c

```
#include "..\acl.h"

const uint acl_secp128r1_m[] = {
    0xffffffff, 0xffffffff, 0xffffffff, 0xfffffffd
};

const uint acl_secp128r1_fr[] = { 128, 97, 0 };

const uint acl_secp128r1_g[] = {
    0xa52c5b86, 0x0c28607c, 0x8b899b2d, 0x161ff752,
    0xdded7a83, 0xc02da292, 0x5bafeb13, 0xcf5ac839
};

const uint acl_secp128r1_b[] = {
    0x2cee5ed3, 0xd824993c, 0x1079f43d, 0xe87579c1
};
```

```
const uint acl_secp128r1_o[] = {
    0x9038a115, 0x75a30d1b, 0x00000000, 0xfffffffe
};

const ecc_t acl_secp128r1 = {
    "secp128r1",
    ECC_P,
    4,
    (vect) acl_secp128r1_m,
    (list) acl_secp128r1_fr,
    (vect2) acl_secp128r1_g,
    (vect) -3,
    (vect) acl_secp128r1_b,
    (vect) acl_secp128r1_o,
    4,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 73 acl_secp128r2.c

```
#include "..\acl.h"

const uint acl_secp128r2_m[] = {
    0xffffffff, 0xffffffff, 0xffffffff, 0xfffffffd
};

const uint acl_secp128r2_fr[] = { 128, 97, 0 };

const uint acl_secp128r2_g[] = {
    0xcdebc140, 0xe6fb32a7, 0x5e572983, 0x7b6aa5d8,
    0x5fc34b44, 0x7106fe80, 0x894d3aee, 0x27b6916a
};

const uint acl_secp128r2_a[] = {
    0xbff9aee1, 0xbf59cc9b, 0xd1b3bbfe, 0xd6031998
};

const uint acl_secp128r2_b[] = {
    0xbb6d8a5d, 0xdc2c6558, 0x80d02919, 0x5eeefca3
};

const uint acl_secp128r2_o[] = {
    0x0613b5a3, 0xbe002472, 0x7fffffff, 0x3fffffff
};

const ecc_t acl_secp128r2 = {
    "secp128r2",
    ECC_P,
    4,
    (vect) acl_secp128r2_m,
    (list) acl_secp128r2_fr,
    (vect2) acl_secp128r2_g,
    (vect) acl_secp128r2_a,
    (vect) acl_secp128r2_b,
    (vect) acl_secp128r2_o,
```

```
    4,
    4,
    (void *) &acl_p_ecc_func
};
```

Source file 74 acl_secp160k1.c

```
#include "..\acl.h"

const uint acl_secp160k1_m[] = {
    0xffffac73, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp160k1_fr[] = { 160, 32, 0x538d };

const uint acl_secp160k1_g[] = {
    0xdd4d7ebb, 0x3036f4f5, 0xa4019e76, 0xe37aa192, 0x3b4c382c,
    0xf03c4fee, 0x531733c3, 0x6bc28286, 0x318fdced, 0x938cf935
};

const uint acl_secp160k1_o[] = {
    0xca16b6b3, 0x16dfab9a, 0x0001b8fa, 0x00000000, 0x00000000, 0x00000001
};

const ecc_t acl_secp160k1 = {
    "secp160k1",
    ECC_P + ECC_K,
    5,
    (vect) acl_secp160k1_m,
    (list) acl_secp160k1_fr,
    (vect2) acl_secp160k1_g,
    (vect) 0,
    (vect) 7,
    (vect) acl_secp160k1_o,
    6,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 75 acl_secp160r1.c

```
#include "..\acl.h"

const uint acl_secp160r1_m[] = {
    0x7fffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp160r1_fr[] = { 160, 31, 0 };

const uint acl_secp160r1_g[] = {
    0x13cbfc82, 0x68c38bb9, 0x46646989, 0x8ef57328, 0x4a96b568,
    0x7ac5fb32, 0x04235137, 0x59dcc912, 0x3168947d, 0x23a62855
};

const uint acl_secp160r1_b[] = {
    0xc565fa45, 0x81d4d4ad, 0x65acf89f, 0x54bd7a8b, 0x1c97befc
```

```
};

const uint acl_secp160r1_o[] = {
    0xca752257, 0xf927aed3, 0x0001f4c8, 0x00000000, 0x00000000, 0x00000001
};

const ecc_t acl_secp160r1 = {
    "secp160r1",
    ECC_P,
    5,
    (vect) acl_secp160r1_m,
    (list) acl_secp160r1_fr,
    (vect2) acl_secp160r1_g,
    (vect) -3,
    (vect) acl_secp160r1_b,
    (vect) acl_secp160r1_o,
    6,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 76 acl_secp160r2.c

```
#include "..\acl.h"

const uint acl_secp160r2_m[] = {
    0xffffac73, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp160r2_fr[] = { 160, 32, 0x538d };

const uint acl_secp160r2_g[] = {
    0x3144ce6d, 0x30f7199d, 0x1f4ff11b, 0x293a117e, 0x52dcb034,
    0xa7d43f2e, 0xf9982cfe, 0xe071fa0d, 0xe331f296, 0xfeaffef2
};

const uint acl_secp160r2_b[] = {
    0xf50388ba, 0x04664d5a, 0xab572749, 0xfb59eb8b, 0xb4e134d3
};

const uint acl_secp160r2_o[] = {
    0xf3a1a16b, 0xe786a818, 0x0000351e, 0x00000000, 0x00000000, 0x00000001
};

const ecc_t acl_secp160r2 = {
    "secp160r2",
    ECC_P,
    5,
    (vect) acl_secp160r2_m,
    (list) acl_secp160r2_fr,
    (vect2) acl_secp160r2_g,
    (vect) -3,
    (vect) acl_secp160r2_b,
    (vect) acl_secp160r2_o,
    6,
    1,
    (void *) &acl_p_ecc_func
```

```
};
```

Source file 77 acl_secp192k1.c

```
#include "..\acl.h"

const uint acl_secp192k1_m[] = {
    0xffffee37, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp192k1_fr[] = { 192, 32, 0x11c9 };

const uint acl_secp192k1_g[] = {
    0xeae06c7d, 0x1da5d1b1, 0x80b7f434, 0x26b07d02, 0xc057e9ae, 0xdb4ff10e,
    0xd95e2f9d, 0x4082aa88, 0x15be8634, 0x844163d0, 0x9c5628a7, 0x9b2f2f6d
};

const uint acl_secp192k1_o[] = {
    0x74defd8d, 0x0f69466a, 0x26f2fc17, 0xfffffffe, 0xffffffff, 0xffffffff
};

const ecc_t acl_secp192k1 = {
    "secp192k1",
    ECC_P + ECC_K,
    6,
    (vect) acl_secp192k1_m,
    (list) acl_secp192k1_fr,
    (vect2) acl_secp192k1_g,
    (vect) 0,
    (vect) 3,
    (vect) acl_secp192k1_o,
    6,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 78 acl_secp192r1.c

```
#include "..\acl.h"

const uint acl_secp192r1_m[] = {
    0xffffffff, 0xffffffff, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp192r1_fr[] = { 192, 64, 0 };

const uint acl_secp192r1_g[] = {
    0x82ff1012, 0xf4ff0afd, 0x43a18800, 0x7cbf20eb, 0xb03090f6, 0x188da80e,
    0x1e794811, 0x73f977a1, 0x6b24cdd5, 0x631011ed, 0xffc8da78, 0x07192b95
};

const uint acl_secp192r1_b[] = {
    0xc146b9b1, 0xfeb8deec, 0x72243049, 0x0fa7e9ab, 0xe59c80e7, 0x64210519
};

const uint acl_secp192r1_o[] = {
```

```
    0xb4d22831, 0x146bc9b1, 0x99def836, 0xffffffff, 0xffffffff, 0xffffffff
};

const ecc_t acl_secp192r1 = {
    "secp192r1",
    ECC_P,
    6,
    (vect) acl_secp192r1_m,
    (list) acl_secp192r1_fr,
    (vect2) acl_secp192r1_g,
    (vect) -3,
    (vect) acl_secp192r1_b,
    (vect) acl_secp192r1_o,
    6,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 79 acl_secp224k1.c

```
#include "..\acl.h"

const uint acl_secp224k1_m[] = {
    0xffffe56d, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff
};

const uint acl_secp224k1_fr[] = { 224, 32, 0x1a93};

const uint acl_secp224k1_g[] = {
    0xb6b7a45c, 0x0f7e650e, 0xe47075a9, 0x69a467e9, 0x30fc28a1, 0x4df099df,
    0xa1455b33,
    0x556d61a5, 0xe2ca4bdb, 0xc0b0bd59, 0xf7e319f7, 0x82cafbd6, 0x7fba3442,
    0x7e089fed
};

const uint acl_secp224k1_o[] = {
    0x769fb1f7, 0xcaf0a971, 0xd2ec6184, 0x0001dce8, 0x00000000, 0x00000000,
    0x00000000, 0x00000001
};

const ecc_t acl_secp224k1 = {
    "secp224k1",
    ECC_P + ECC_K,
    7,
    (vect) acl_secp224k1_m,
    (list) acl_secp224k1_fr,
    (vect2) acl_secp224k1_g,
    (vect) 0,
    (vect) 5,
    (vect) acl_secp224k1_o,
    8,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 80 acl_secp224r1.c

```
#include "..\acl.h"

const uint acl_secp224r1_m[] = {
    0x00000001, 0x00000000, 0x00000000, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff
};

const uint acl_secp224r1_fr[] = { 224, 96, ~0 };

const uint acl_secp224r1_g[] = {
    0x115c1d21, 0x343280d6, 0x56c21122, 0x4a03c1d3, 0x321390b9, 0x6bb4bf7f,
    0xb70e0cbd,
    0x85007e34, 0x44d58199, 0x5a074764, 0xcd4375a0, 0x4c22dfe6, 0xb5f723fb,
    0xbd376388
};

const uint acl_secp224r1_b[] = {
    0x2355ffb4, 0x270b3943, 0xd7bfd8ba, 0x5044b0b7, 0xf5413256, 0x0c04b3ab,
    0xb4050a85
};

const uint acl_secp224r1_o[] = {
    0x5c5c2a3d, 0x13dd2945, 0xe0b8f03e, 0xffff16a2, 0xffffffff, 0xffffffff,
    0xffffffff
};

const ecc_t acl_secp224r1 = {
    "secp224r1",
    ECC_P,
    7,
    (vect) acl_secp224r1_m,
    (list) acl_secp224r1_fr,
    (vect2) acl_secp224r1_g,
    (vect) -3,
    (vect) acl_secp224r1_b,
    (vect) acl_secp224r1_o,
    7,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 81 acl_secp256k1.c

```
#include "..\acl.h"

const uint acl_secp256k1_m[] = {
    0xfffffc2f, 0xfffffffe, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff, 0xffffffff
};

const uint acl_secp256k1_fr[] = { 256, 32, 0x03d1 };

const uint acl_secp256k1_g[] = {
    0x16f81798, 0x59f2815b, 0x2dce28d9, 0x029bfcdb, 0xce870b07, 0x55a06295,
    0xf9dcbbac, 0x79be667e,
```

```
    0xfb10d4b8, 0x9c47d08f, 0xa6855419, 0xfd17b448, 0x0e1108a8, 0x5da4fbfc,
    0x26a3c465, 0x483ada77
};

const uint acl_secp256k1_o[] = {
    0xd0364141, 0xbfd25e8c, 0xaf48a03b, 0xbaaedce6, 0xfffffffe, 0xffffffff,
    0xffffffff, 0xffffffff
};

const ecc_t acl_secp256k1 = {
    "secp256k1",
    ECC_P + ECC_K,
    8,
    (vect) acl_secp256k1_m,
    (list) acl_secp256k1_fr,
    (vect2) acl_secp256k1_g,
    (vect) 0,
    (vect) 7,
    (vect) acl_secp256k1_o,
    8,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 82 acl_secp256r1.c

```
#include "..\acl.h"

const uint acl_secp256r1_m[] = {
    0xffffffff, 0xffffffff, 0xffffffff, 0x00000000, 0x00000000, 0x00000000,
    0x00000001, 0xffffffff
};

const uint acl_secp256r1_fr[] = { 256, 224, ~192, ~96, 0 };

const uint acl_secp256r1_g[] = {
    0xd898c296, 0xf4a13945, 0x2deb33a0, 0x77037d81, 0x63a440f2, 0xf8bce6e5,
    0xe12c4247, 0x6b17d1f2,
    0x37bf51f5, 0xcbb64068, 0x6b315ece, 0x2bce3357, 0x7c0f9e16, 0x8ee7eb4a,
    0xfe1a7f9b, 0x4fe342e2
};

const uint acl_secp256r1_b[] = {
    0x27d2604b, 0x3bce3c3e, 0xcc53b0f6, 0x651d06b0, 0x769886bc, 0xb3ebbd55,
    0xaa3a93e7, 0x5ac635d8
};

const uint acl_secp256r1_o[] = {
    0xfc632551, 0xf3b9cac2, 0xa7179e84, 0xbce6faad, 0xffffffff, 0xffffffff,
    0x00000000, 0xffffffff
};

const ecc_t acl_secp256r1 = {
    "secp256r1",
    ECC_P,
    8,
    (vect) acl_secp256r1_m,
```

```
    (list) acl_secp256r1_fr,
    (vect2) acl_secp256r1_g,
    (vect) -3,
    (vect) acl_secp256r1_b,
    (vect) acl_secp256r1_o,
    8,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 83 acl_secp384r1.c

```
#include "..\acl.h"

const uint acl_secp384r1_m[] = {
    0xffffffff, 0x00000000, 0x00000000, 0xffffffff, 0xfffffffe, 0xffffffff,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
};

const uint acl_secp384r1_fr[] = { 384, 128, 96, ~32, 0 };

const uint acl_secp384r1_g[] = {
    0x72760ab7, 0x3a545e38, 0xbf55296c, 0x5502f25d, 0x82542a38, 0x59f741e0,
    0x8ba79b98, 0x6e1d3b62, 0xf320ad74, 0x8eb1c71e, 0xbe8b0537, 0xaa87ca22,
    0x90ea0e5f, 0x7a431d7c, 0x1d7e819d, 0x0a60b1ce, 0xb5f0b8c0, 0xe9da3113,
    0x289a147c, 0xf8f41dbd, 0x9292dc29, 0x5d9e98bf, 0x96262c6f, 0x3617de4a
};

const uint acl_secp384r1_b[] = {
    0xd3ec2aef, 0x2a85c8ed, 0x8a2ed19d, 0xc656398d, 0x5013875a, 0x0314088f,
    0xfe814112, 0x181d9c6e, 0xe3f82d19, 0x988e056b, 0xe23ee7e4, 0xb3312fa7
};

const uint acl_secp384r1_o[] = {
    0xccc52973, 0xecec196a, 0x48b0a77a, 0x581a0db2, 0xf4372ddf, 0xc7634d81,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
};

const ecc_t acl_secp384r1 = {
    "secp384r1",
    ECC_P,
    12,
    (vect) acl_secp384r1_m,
    (list) acl_secp384r1_fr,
    (vect2) acl_secp384r1_g,
    (vect) -3,
    (vect) acl_secp384r1_b,
    (vect) acl_secp384r1_o,
    12,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 84 acl_secp521r1.c

```
#include "..\acl.h"
```

```
const uint acl_secp521r1_m[] = {
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0x000001ff
};

const uint acl_secp521r1_fr[] = { 521, 0 };

const uint acl_secp521r1_g[] = {
    0xc2e5bd66, 0xf97e7e31, 0x856a429b, 0x3348b3c1, 0xa2ffa8de, 0xfe1dc127,
    0xefe75928, 0xa14b5e77, 0x6b4d3dba, 0xf828af60, 0x053fb521, 0x9c648139,
    0x2395b442, 0x9e3ecb66, 0x0404e9cd, 0x858e06b7, 0x000000c6,
    0x9fd16650, 0x88be9476, 0xa272c240, 0x353c7086, 0x3fad0761, 0xc550b901,
    0x5ef42640, 0x97ee7299, 0x273e662c, 0x17afbd17, 0x579b4468, 0x98f54449,
    0x2c7d1bd9, 0x5c8a5fb4, 0x9a3bc004, 0x39296a78, 0x00000118
};

const uint acl_secp521r1_b[] = {
    0x6b503f00, 0xef451fd4, 0x3d2c34f1, 0x3573df88, 0x3bb1bf07, 0x1652c0bd,
    0xec7e937b, 0x56193951, 0x8ef109e1, 0xb8b48991, 0x99b315f3, 0xa2da725b,
    0xb68540ee, 0x929a21a0, 0x8e1c9a1f, 0x953eb961, 0x00000051
};

const uint acl_secp521r1_o[] = {
    0x91386409, 0xbb6fb71e, 0x899c47ae, 0x3bb5c9b8, 0xf709a5d0, 0x7fcc0148,
    0xbf2f966b, 0x51868783, 0xfffffffa, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0x000001ff
};

const ecc_t acl_secp521r1 = {
    "secp521r1",
    ECC_P,
    17,
    (vect) acl_secp521r1_m,
    (list) acl_secp521r1_fr,
    (vect2) acl_secp521r1_g,
    (vect) -3,
    (vect) acl_secp521r1_b,
    (vect) acl_secp521r1_o,
    17,
    1,
    (void *) &acl_p_ecc_func
};
```

Source file 85 acl_sect113r1.c

```
#include "..\acl.h"

const uint acl_sect113r1_fr[] = { 113, 9, 0 };

const uint acl_sect113r1_g[] = {
    0x3562c10f, 0xab1407d7, 0x616f35f4, 0x00009d73,
    0x5ed31886, 0xee84d131, 0x30277958, 0x0000a528
};

const uint acl_sect113r1_a[] = {
```

```
    0xe85820f7, 0xc7fe649c, 0x250ca6e7, 0x00003088
};

const uint acl_sect113r1_b[] = {
    0xe0e9c723, 0x0744188b, 0xe4d3e226, 0x0000e8be
};

const uint acl_sect113r1_o[] = {
    0x8a39e56f, 0x00d9ccec, 0x00000000, 0x00010000
};

const ecc_t acl_sect113r1 = {
    "sect113r1",
    ECC_2,
    4,
    (vect) 0,
    (list) acl_sect113r1_fr,
    (vect2) acl_sect113r1_g,
    (vect) acl_sect113r1_a,
    (vect) acl_sect113r1_b,
    (vect) acl_sect113r1_o,
    4,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 86 acl_sect113r2.c

```
#include "..\acl.h"

const uint acl_sect113r2_fr[] = { 113, 9, 0 };

const uint acl_sect113r2_g[] = {
    0xb8164797, 0x5ef52fcd, 0x6a7b26ca, 0x0001a57a,
    0x95baba1d, 0x674c06e6, 0xc94ed1fe, 0x0000b3ad
};

const uint acl_sect113r2_a[] = {
    0xc0aa55c7, 0x5a0dd6df, 0x18dbec7e, 0x00006899
};

const uint acl_sect113r2_b[] = {
    0xe059184f, 0x7bd4bf36, 0xa9ec9b29, 0x000095e9
};

const uint acl_sect113r2_o[] = {
    0x2496af93, 0x0108789b, 0x00000000, 0x00010000
};

const ecc_t acl_sect113r2 = {
    "sect113r2",
    ECC_2,
    4,
    (vect) 0,
    (list) acl_sect113r2_fr,
    (vect2) acl_sect113r2_g,
    (vect) acl_sect113r2_a,
```

```
    (vect) acl_sect113r2_b,
    (vect) acl_sect113r2_o,
    4,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 87 acl_sect131r1.c

```
#include "..\acl.h"

const uint acl_sect131r1_fr[] = { 131, 8, 3, 2, 0 };

const uint acl_sect131r1_g[] = {
    0x43638399, 0x0f9c1813, 0xdf9833c4, 0x81baf91f, 0x00000000,
    0x4ef9e150, 0xc8134b1b, 0x8c001f73, 0x8c6e7ea3, 0x00000007
};

const uint acl_sect131r1_a[] = {
    0x8c2570b8, 0x418ff3ff, 0x6b562144, 0xa11b09a7, 0x00000007
};

const uint acl_sect131r1_b[] = {
    0x78f9d341, 0xc6c72916, 0x884b63b9, 0x17c05610, 0x00000002
};

const uint acl_sect131r1_o[] = {
    0x9464b54d, 0x3123953a, 0x00000002, 0x00000000, 0x00000004
};

const ecc_t acl_sect131r1 = {
    "sect131r1",
    ECC_2,
    5,
    (vect) 0,
    (list) acl_sect131r1_fr,
    (vect2) acl_sect131r1_g,
    (vect) acl_sect131r1_a,
    (vect) acl_sect131r1_b,
    (vect) acl_sect131r1_o,
    5,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 88 acl_sect131r2.c

```
#include "..\acl.h"

const uint acl_sect131r2_fr[] = { 131, 8, 3, 2, 0 };

const uint acl_sect131r2_g[] = {
    0x1bb366a8, 0x652d2395, 0xf95031ad, 0x56dcd8f2, 0x00000003,
    0xe9eb240f, 0x6d9e265d, 0x7940a536, 0x48f06d86, 0x00000006
};
```

```
const uint acl_sect131r2_a[] = {
    0x176573b2, 0x415f07c2, 0xd7cafcbf, 0xe5a88919, 0x00000003
};

const uint acl_sect131r2_b[] = {
    0x018f2192, 0x734ce38f, 0xc55657ac, 0xb8266a46, 0x00000004
};

const uint acl_sect131r2_o[] = {
    0x049ba98f, 0x6954a233, 0x00000001, 0x00000000, 0x00000004
};

const ecc_t acl_sect131r2 = {
    "sect131r2",
    ECC_2,
    5,
    (vect) 0,
    (list) acl_sect131r2_fr,
    (vect2) acl_sect131r2_g,
    (vect) acl_sect131r2_a,
    (vect) acl_sect131r2_b,
    (vect) acl_sect131r2_o,
    5,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 89 acl_sect163k1.c

```
#include "..\acl.h"

const uint acl_sect163k1_fr[] = { 163, 7, 6, 3, 0 };

const uint acl_sect163k1_g[] = {
    0x5c94eee8, 0xde4e6d5e, 0xaa07d793, 0x7bbc11ac, 0xfe13c053, 0x00000002,
    0xccdaa3d9, 0x0536d538, 0x321f2e80, 0x5d38ff58, 0x89070fb0, 0x00000002
};

const uint acl_sect163k1_o[] = {
    0x99f8a5ef, 0xa2e0cc0d, 0x00020108, 0x00000000, 0x00000000, 0x00000004
};

const ecc_t acl_sect163k1 = {
    "sect163k1",
    ECC_2 + ECC_K,
    6,
    (vect) 0,
    (list) acl_sect163k1_fr,
    (vect2) acl_sect163k1_g,
    (vect) 1,
    (vect) 1,
    (vect) acl_sect163k1_o,
    6,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 90 acl_sect163r1.c

```
#include "..\acl.h"

const uint acl_sect163r1_fr[] = { 163, 7, 6, 3, 0 };

const uint acl_sect163r1_g[] = {
    0x7876a654, 0x567f787a, 0x89566789, 0xab438977, 0x69979697, 0x00000003,
    0xf41ff883, 0xe3c80988, 0x9d51fefc, 0xefafb298, 0x435edb42, 0x00000000
};

const uint acl_sect163r1_a[] = {
    0xd2782ae2, 0xbd88e246, 0x54ff8428, 0xefa84f95, 0xb6882caa, 0x00000007
};

const uint acl_sect163r1_b[] = {
    0xf958afd9, 0xca91f73a, 0x946bda29, 0xdcb40aab, 0x13612dcd, 0x00000007
};

const uint acl_sect163r1_o[] = {
    0xa710279b, 0xb689c29c, 0xffff48aa, 0xffffffff, 0xffffffff, 0x00000003
};

const ecc_t acl_sect163r1 = {
    "sect163r1",
    ECC_2,
    6,
    (vect) 0,
    (list) acl_sect163r1_fr,
    (vect2) acl_sect163r1_g,
    (vect) acl_sect163r1_a,
    (vect) acl_sect163r1_b,
    (vect) acl_sect163r1_o,
    6,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 91 acl_sect163r2.c

```
#include "..\acl.h"

const uint acl_sect163r2_fr[] = { 163, 7, 6, 3, 0 };

const uint acl_sect163r2_g[] = {
    0xe8343e36, 0xd4994637, 0xa0991168, 0x86a2d57e, 0xf0eba162, 0x00000003,
    0x797324f1, 0xb11c5c0c, 0xa2cdd545, 0x71a0094f, 0xd51fbc6c, 0x00000000
};

const uint acl_sect163r2_b[] = {
    0x4a3205fd, 0x512f7874, 0x1481eb10, 0xb8c953ca, 0x0a601907, 0x00000002,
};

const uint acl_sect163r2_o[] = {
    0xa4234c33, 0x77e70c12, 0x000292fe, 0x00000000, 0x00000000, 0x00000004
};
```

```
const ecc_t acl_sect163r2 = {
    "sect163r2",
    ECC_2,
    6,
    (vect) 0,
    (list) acl_sect163r2_fr,
    (vect2) acl_sect163r2_g,
    (vect) 1,
    (vect) acl_sect163r2_b,
    (vect) acl_sect163r2_o,
    6,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 92 acl_sect193r1.c

```
#include "..\acl.h"

const uint acl_sect193r1_fr[] = { 193, 15, 0 };

const uint acl_sect193r1_g[] = {
    0xd8c0c5e1, 0x79625372, 0xdef4bf61, 0xad6cdf6f, 0x0ff84a74, 0xf481bc5f,
    0x00000001,
    0xf7ce1b05, 0xb3201b6a, 0x1ad17fb0, 0xf3ea9e3a, 0x903712cc, 0x25e399f2,
    0x00000000
};

const uint acl_sect193r1_a[] = {
    0x11df7b01, 0x098ac8a9, 0x7b4087de, 0x69e171f7, 0x7a989751, 0x17858feb,
    0x00000000
};

const uint acl_sect193r1_b[] = {
    0x31478814, 0xc1c2e5d8, 0x1e5bbc7c, 0xacadaa7a, 0xe6c3a89f, 0xfdfb49bf,
    0x00000000
};

const uint acl_sect193r1_o[] = {
    0x920eba49, 0x8f443acc, 0xc7f34a77, 0x00000000, 0x00000000, 0x00000000,
    0x00000001
};

const ecc_t acl_sect193r1 = {
    "sect193r1",
    ECC_2,
    7,
    (vect) 0,
    (list) acl_sect193r1_fr,
    (vect2) acl_sect193r1_g,
    (vect) acl_sect193r1_a,
    (vect) acl_sect193r1_b,
    (vect) acl_sect193r1_o,
    7,
    2,
    (void *) &acl_2_ecc_func
```

```
};
```

Source file 93 acl_sect193r2.c

```
#include "..\acl.h"

const uint acl_sect193r2_fr[] = { 193, 15, 0 };

const uint acl_sect193r2_g[] = {
    0xae617e8f, 0xa651350a, 0x7e82ca14, 0x03f39e1a, 0x2e0367c8, 0xd9b67d19,
    0x00000000,
    0x4cdecf6c, 0x96f92722, 0xd9ca01f5, 0x29e7defb, 0x07c304ac, 0xce943356,
    0x00000001
};

const uint acl_sect193r2_a[] = {
    0x7702709b, 0x3ecd6997, 0x190b0bc4, 0xa6ed8667, 0x37c2ce3e, 0x63f35a51,
    0x00000001
};

const uint acl_sect193r2_b[] = {
    0x1d4316ae, 0xe3efb7f6, 0x856a5b16, 0x377e2ab2, 0x27d4d64c, 0xc9bb9e89,
    0x00000000
};

const uint acl_sect193r2_o[] = {
    0xd4ee99d5, 0x005413cc, 0x5aab561b, 0x00000001, 0x00000000, 0x00000000,
    0x00000001
};

const ecc_t acl_sect193r2 = {
    "sect193r2",
    ECC_2,
    7,
    (vect) 0,
    (list) acl_sect193r2_fr,
    (vect2) acl_sect193r2_g,
    (vect) acl_sect193r2_a,
    (vect) acl_sect193r2_b,
    (vect) acl_sect193r2_o,
    7,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 94 acl_sect233k1.c

```
#include "..\acl.h"

const uint acl_sect233k1_fr[] = { 233, 74, 0 };

const uint acl_sect233k1_g[] = {
    0xefad6126, 0x0a4c9d6e, 0x19c26bf5, 0x149563a4, 0x29f22ff4, 0x7e731af1,
    0x32ba853a, 0x00000172,
    0x56fae6a3, 0x56e0c110, 0xf18aeb9b, 0x27a8cd9b, 0x555a67c4, 0x19b7f70f,
    0x537dece8, 0x000001db
```

```
};

const uint acl_sect233k1_o[] = {
    0xf173abdf, 0x6efb1ad5, 0xb915bcd4, 0x00069d5b, 0x00000000, 0x00000000,
    0x00000000, 0x00000080
};

const ecc_t acl_sect233k1 = {
    "sect233k1",
    ECC_2 + ECC_K,
    8,
    (vect) 0,
    (list) acl_sect233k1_fr,
    (vect2) acl_sect233k1_g,
    (vect) 0,
    (vect) 1,
    (vect) acl_sect233k1_o,
    8,
    4,
    (void *) &acl_2_ecc_func
};
```

Source file 95 acl_sect233r1.c

```
#include "..\acl.h"

const uint acl_sect233r1_fr[] = { 233, 74, 0 };

const uint acl_sect233r1_g[] = {
    0x71fd558b, 0xf8f8eb73, 0x391f8b36, 0x5fef65bc, 0x39f1bb75, 0x8313bb21,
    0xc9dfcbac, 0x000000fa,
    0x01f81052, 0x36716f7e, 0xf867a7ca, 0xbf8a0bef, 0xe58528be, 0x03350678,
    0x6a08a419, 0x00000100
};

const uint acl_sect233r1_b[] = {
    0x7d8f90ad, 0x81fe115f, 0x20e9ce42, 0x213b333b, 0x0923bb58, 0x332c7f8c,
    0x647ede6c, 0x00000066
};

const uint acl_sect233r1_o[] = {
    0x03cfe0d7, 0x22031d26, 0xe72f8a69, 0x0013e974, 0x00000000, 0x00000000,
    0x00000000, 0x00000100
};

const ecc_t acl_sect233r1 = {
    "sect233r1",
    ECC_2,
    8,
    (vect) 0,
    (list) acl_sect233r1_fr,
    (vect2) acl_sect233r1_g,
    (vect) 1,
    (vect) acl_sect233r1_b,
    (vect) acl_sect233r1_o,
    8,
    2,
```

```
    (void *) &acl_2_ecc_func
};
```

Source file 96 acl_sect239k1.c

```
#include "..\acl.h"

const uint acl_sect239k1_fr[] = { 239, 158, 0 };

const uint acl_sect239k1_g[] = {
    0x193035dc, 0x7b2a6555, 0xc44cc2cc, 0xa8b2d126, 0x88a68727, 0x83e97309,
    0xb6a887a9, 0x000029a0,
    0x6553f0ca, 0x2a5dc6b7, 0xb275fc31, 0xe73510ac, 0x1c103089, 0x549bdb01,
    0x0804f12e, 0x00007631
};

const uint acl_sect239k1_o[] = {
    0x00e478a5, 0x1f1c1da8, 0xc67cb6e9, 0x005a79fe, 0x00000000, 0x00000000,
    0x00000000, 0x00002000
};

const ecc_t acl_sect239k1 = {
    "sect239k1",
    ECC_2 + ECC_K,
    8,
    (vect) 0,
    (list) acl_sect239k1_fr,
    (vect2) acl_sect239k1_g,
    (vect) 0,
    (vect) 1,
    (vect) acl_sect239k1_o,
    8,
    4,
    (void *) &acl_2_ecc_func
};
```

Source file 97 acl_sect283k1.c

```
#include "..\acl.h"

const uint acl_sect283k1_fr[] = { 283, 12, 7, 5, 0 };

const uint acl_sect283k1_g[] = {
    0x58492836, 0xb0c2ac24, 0x16876913, 0x23c1567a, 0x53cd265f, 0x62f188e5,
    0x3f1a3b81, 0x78ca4488, 0x0503213f,
    0x77dd2259, 0x4e341161, 0xe4596236, 0xe8184698, 0xe87e45c0, 0x07e5426f,
    0x8d90f95d, 0x0f1c9e31, 0x01ccda38
};

const uint acl_sect283k1_o[] = {
    0x1e163c61, 0x94451e06, 0x265dff7f, 0x2ed07577, 0xffffe9ae, 0xffffffff,
    0xffffffff, 0xffffffff, 0x01ffffff
};

const ecc_t acl_sect283k1 = {
    "sect283k1",
```

```
    ECC_2 + ECC_K,
    9,
    (vect) 0,
    (list) acl_sect283k1_fr,
    (vect2) acl_sect283k1_g,
    (vect) 0,
    (vect) 1,
    (vect) acl_sect283k1_o,
    9,
    4,
    (void *) &acl_2_ecc_func
};
```

Source file 98 acl_sect283r1.c

```
#include "..\acl.h"

const uint acl_sect283r1_fr[] = { 283, 12, 7, 5, 0 };

const uint acl_sect283r1_g[] = {
    0x86b12053, 0xf8cdbecd, 0x80e2e198, 0x557eac9c, 0x2eed25b8, 0x70b0dfec,
    0xe1934f8c, 0x8db7dd90, 0x05f93925,
    0xbe8112f4, 0x13f0df45, 0x826779c8, 0x350eddb0, 0x516ff702, 0xb20d02b4,
    0xb98fe6d4, 0xfe24141c, 0x03676854
};

const uint acl_sect283r1_b[] = {
    0x3b79a2f5, 0xf6263e31, 0xa581485a, 0x45309fa2, 0xca97fd76, 0x19a0303f,
    0xa5a4af8a, 0xc8b8596d, 0x027b680a
};

const uint acl_sect283r1_o[] = {
    0xefadb307, 0x5b042a7c, 0x938a9016, 0x399660fc, 0xffffef90, 0xffffffff,
    0xffffffff, 0xffffffff, 0x03ffffff
};

const ecc_t acl_sect283r1 = {
    "sect283r1",
    ECC_2,
    9,
    (vect) 0,
    (list) acl_sect283r1_fr,
    (vect2) acl_sect283r1_g,
    (vect) 1,
    (vect) acl_sect283r1_b,
    (vect) acl_sect283r1_o,
    9,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 99 acl_sect409k1.c

```
#include "..\acl.h"

const uint acl_sect409k1_fr[] = { 409, 87, 0 };
```

```
const uint acl_sect409k1_g[] = {
    0xe9023746, 0xb35540cf, 0xee222eb1, 0xb5aaaa62, 0xc460189e, 0xf9f67cc2,
    0x27accfb8, 0xe307c84c, 0x0efd0987, 0x0f718421, 0xad3ab189, 0x658f49c1,
    0x0060f05f,
    0xd8e0286b, 0x5863ec48, 0xaa9ca27a, 0xe9c55215, 0xda5f6c42, 0xe9ea10e3,
    0xe6325165, 0x918ea427, 0x3460782f, 0xbf04299c, 0xacba1dac, 0x0b7c4e42,
    0x01e36905
};

const uint acl_sect409k1_o[] = {
    0xe01e5fcf, 0x4b5c83b8, 0xe3e7ca5b, 0x557d5ed3, 0x20400ec4, 0x83b2d4ea,
    0xfffffe5f, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
    0x007fffff
};

const ecc_t acl_sect409k1 = {
    "sect409k1",
    ECC_2 + ECC_K,
    13,
    (vect) 0,
    (list) acl_sect409k1_fr,
    (vect2) acl_sect409k1_g,
    (vect) 0,
    (vect) 1,
    (vect) acl_sect409k1_o,
    13,
    4,
    (void *) &acl_2_ecc_func
};
```

Source file 100 acl_sect409r1.c

```
#include "..\acl.h"

const uint acl_sect409r1_fr[] = { 409, 87, 0 };

const uint acl_sect409r1_g[] = {
    0xbb7996a7, 0x60794e54, 0x5603aeab, 0x8a118051, 0xdc255a86, 0x34e59703,
    0xb01ffe5b, 0xf1771d4d, 0x441cde4a, 0x64756260, 0x496b0c60, 0xd088ddb3,
    0x015d4860,
    0x0273c706, 0x81c364ba, 0xd2181b36, 0xdf4b4f40, 0x38514f1f, 0x5488d08f,
    0x0158aa4f, 0xa7bd198d, 0x7636b9c5, 0x24ed106a, 0x2bbfa783, 0xab6be5f3,
    0x0061b1cf
};

const uint acl_sect409r1_b[] = {
    0x7b13545f, 0x4f50ae31, 0xd57a55aa, 0x72822f6c, 0xa9a197b2, 0xd6ac27c8,
    0x4761fa99, 0xf1f3dd67, 0x7fd6422e, 0x3b7b476b, 0x5c4b9a75, 0xc8ee9feb,
    0x0021a5c2
};

const uint acl_sect409r1_o[] = {
    0xd9a21173, 0x8164cd37, 0x9e052f83, 0x5fa47c3c, 0xf33307be, 0xaad6a612,
    0x000001e2, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x01000000
};
```

```
const ecc_t acl_sect409r1 = {
    "sect409r1",
    ECC_2,
    13,
    (vect) 0,
    (list) acl_sect409r1_fr,
    (vect2) acl_sect409r1_g,
    (vect) 1,
    (vect) acl_sect409r1_b,
    (vect) acl_sect409r1_o,
    13,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 101 acl_sect571k1.c

```
#include "..\acl.h"

const uint acl_sect571k1_fr[] = { 571, 10, 5, 2, 0 };

const uint acl_sect571k1_g[] = {
    0xa01c8972, 0xe2945283, 0x4dca88c7, 0x988b4717, 0x494776fb, 0xbbd1ba39,
    0xb4ceb08c, 0x47da304d, 0x93b205e6, 0x43709584, 0x01841ca4, 0x60248048,
    0x0012d5d4, 0xac9ca297, 0xf8103fe4, 0x82189631, 0x59923fbc, 0x026eb7a8,
    0x3ef1c7a3, 0x01cd4c14, 0x591984f6, 0x320430c8, 0x7ba7af1b, 0xb620b01a,
    0xf772aedc, 0x4fbebbb9, 0xac44aea7, 0x9d4979c0, 0x006d8a2c, 0xffc61efc,
    0x9f307a54, 0x4dd58cec, 0x3bca9531, 0x4f4aeade, 0x7f4fbf37, 0x0349dc80
};

const uint acl_sect571k1_o[] = {
    0x637c1001, 0x5cfe778f, 0x1e91deb4, 0xe5d63938, 0xb630d84b, 0x917f4138,
    0xb391a8db, 0xf19a63e4, 0x131850e1, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x02000000
};

const ecc_t acl_sect571k1 = {
    "sect571k1",
    ECC_2 + ECC_K,
    18,
    (vect) 0,
    (list) acl_sect571k1_fr,
    (vect2) acl_sect571k1_g,
    (vect) 0,
    (vect) 1,
    (vect) acl_sect571k1_o,
    18,
    4,
    (void *) &acl_2_ecc_func
};
```

Source file 102 acl_sect571r1.c

```
#include "..\acl.h"
```

```
const uint acl_sect571r1_fr[] = { 571, 10, 5, 2, 0 };

const uint acl_sect571r1_g[] = {
    0x8eec2d19, 0xe1e7769c, 0xc850d927, 0x4abfa3b4, 0x8614f139, 0x99ae6003,
    0x5b67fb14, 0xcdd711a3, 0xf4c0d293, 0xbde53950, 0xdb7b2abd, 0xa5f40fc8,
    0x955fa80a, 0x0a93d1d2, 0x0d3cd775, 0x6c16c0d4, 0x34b85629, 0x0303001d,
    0x1b8ac15b, 0x1a4827af, 0x6e23dd3c, 0x16e2f151, 0x0485c19b, 0xb3531d2f,
    0x461bb2a8, 0x6291af8f, 0xbab08a57, 0x84423e43, 0x3921e8a6, 0x1980f853,
    0x009cbbca, 0x8c6c27a6, 0xb73d69d7, 0x6dccfffe, 0x42da639b, 0x037bf273
};

const uint acl_sect571r1_b[] = {
    0x2955727a, 0x7ffeff7f, 0x39baca0c, 0x520e4de7, 0x78ff12aa, 0x4afd185a,
    0x56a66e29, 0x2be7ad67, 0x8efa5933, 0x84ffabbd, 0x4a9a18ad, 0xcd6ba8ce,
    0xcb8ceff1, 0x5c6a97ff, 0xb7f3d62f, 0xde297117, 0x2221f295, 0x02f40e7e
};

const uint acl_sect571r1_o[] = {
    0x2fe84e47, 0x8382e9bb, 0x5174d66e, 0x161de93d, 0xc7dd9ca1, 0x6823851e,
    0x08059b18, 0xff559873, 0xe661ce18, 0xffffffff, 0xffffffff, 0xffffffff,
    0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0x03ffffff
};

const ecc_t acl_sect571r1 = {
    "sect571r1",
    ECC_2,
    18,
    (vect) 0,
    (list) acl_sect571r1_fr,
    (vect2) acl_sect571r1_g,
    (vect) 1,
    (vect) acl_sect571r1_b,
    (vect) acl_sect571r1_o,
    18,
    2,
    (void *) &acl_2_ecc_func
};
```

Source file 103 acl_p_ecc_chk.c

```
// returns TRUE if affine point is on curve, FALSE otherwise

// the routine also calculates:
// t1 = tmp + 2*len = right side of equation (x^3 + ax + b)
// t2 = tmp + 3*len = left side of equation (y^2)
// this "feature" is used by the point decompression routine acl_p_ecc_str2p

// a - pointer to ecc point in affine coordinates (x, y)
// tmp - pointer to storage space for 4*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"
#include "..\acl_config.h"

bool_t acl_p_ecc_chk(vect2 a, vect4 tmp, ecc_t *c)
{
```

```
    vect m, t1, t2, yy, fr; uint len;
    // tmp = tmp tmp t1 t2

    m = c->m; len = c->l; fr = c->fr;
    yy = xx + len; t1 = tmp + 2*len; t2 = t1 + len;

#if ACL_CHK_INF_ON_CURVE
    if(acl_zero(a, 2*len)) return TRUE;
#endif
    acl_p_sqr_fr(t1, xx);           // t1 = x^2
    if((int) c->a == -3)            // t1 = x^2 + a
        acl_p_mod_sub32(t1, t1, 3, m, len);
    else if(c->a)
        acl_p_mod_add(t1, t1, c->a, m, len);
    acl_p_mul_fr(t1, t1, xx);       // t1 = x^3 + ax
    if((int) c->b <= ACL_MAX_B)     // t1 = x^3 + ax + b
        acl_p_mod_add32(t1, t1, (int) c->b, m, len);
    else
        acl_p_mod_add(t1, t1, c->b, m, len);
    acl_p_sqr_fr(t2, yy);           // t2 = y^2
    if(c->t & ECC_A_MASK) {
        acl_mov(tmp, t1, len); acl_p_mod(t1, tmp, len, m + len, len);
        acl_mov(tmp, t2, len); acl_p_mod(t2, tmp, len, m + len, len);
    }
    return !acl_cmp(t1, t2, len);
}
```

Source file 104 acl_p_ecc_dbl.c

```
// point doubling with fast reduction (Jacobian <= 2 * Jacobian)
// taken directly from
//      D. Hankerson, A. Menezes, S.A. Vanstone:
//      Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004
// algortihm 3.21, p. 91

// a - pointer to ecc point in projective coordinates (x, y, z)
// tmp - pointer to storage space for 4*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_p_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c)
{
    vect m, t1, t2, yy, zz, fr; uint len;
    // tmp = tmp tmp t1 t2

    m = c->m; len = c->l; fr = c->fr;
    yy = xx + len; zz = yy + len; t1 = tmp + 2*len; t2 = t1 + len;

    if(!acl_zero(zz, len)) {    // 2 * inf == inf
        if(!c->a) {
            acl_p_sqr_fr(t1, xx);               // t1 = xx^2
            acl_p_mod_add(t2, t1, t1, m, len);  // t2 = 2 * t1
        } else if((int) c->a == -3) {
            acl_p_sqr_fr(t1, zz);               // 2 t1 = zz^2
            acl_p_mod_sub(t2, xx, t1, m, len);  // 3 t2 = xx - t1
```

```
            acl_p_mod_add(t1, t1, xx, m, len);  // 4 t1 = t1 + xx
            acl_p_mul_fr(t1, t1, t2);           // 5 t1 = t2 * t1
            acl_p_mod_add(t2, t1, t1, m, len);  // t2 = 2 * t1
        } else {
            acl_p_sqr_fr(t2, zz);               // t2 = zz^2
            acl_p_sqr_fr(t2, t2);               // t2 = zz^4
            acl_p_mul_fr(t2, t2, c->a);         // t2 = t2 * a
            acl_p_sqr_fr(t1, xx);               // t1 = xx^2
            acl_p_mod_add(t2, t2, t1, m, len);  // t2 = t2 + xx^2
            acl_p_mod_dbl(t1, 1, m, len);       // t1 = 2 * t1
        }
        acl_p_mod_add(t2, t2, t1, m, len);  // 6 t2 = t2 + t1 == D
        acl_p_mod_dbl(yy, 1, m, len);       // 7 yy = 2 * yy
        acl_p_mul_fr(zz, zz, yy);           // 8 zz = zz * yy == new zz
        acl_p_sqr_fr(yy, yy);               // 9 yy = yy^2 == 4 A
        acl_p_mul_fr(t1, xx, yy);           // 10 t1 = yy * xx == B
        acl_p_sqr_fr(yy, yy);               // 11 yy = yy^2 == 16 A^2
        acl_p_mod_hlv(yy, 1, m, len);       // 12 yy = yy/2 == C
        acl_p_sqr_fr(xx, t2);               // 13 xx = t2^2 == D^2
        acl_p_mod_sub(xx, xx, t1, m, len);
        acl_p_mod_sub(xx, xx, t1, m, len);  // 15 xx = xx - 2 * t1 == new xx
        acl_p_mod_sub(t1, t1, xx, m, len);  // 16 t1 = t1 - xx
        acl_p_mul_fr(t1, t1, t2);           // 17 t1 = t1 * t2
        acl_p_mod_sub(yy, t1, yy, m, len);  // 18 yy = t1 - yy == new yy
    }
}
```

Source file 105 acl_p_ecc_add.c

```
// point addition with fast reduction  (Jacobian <= Jacobian + Affine)
// taken directly from
//      D. Hankerson, A. Menezes, S.A. Vanstone:
//      Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004
// algortihm 3.22, pp. 91-92

// a - pointer to ecc point in projective coordinates (x, y, z)
// b - pointer to ecc point in affine coordinates (x, y)
// tmp - pointer to storage space for 5*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_p_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c)
{
    vect m, t1, t2, t3, yy1, zz1, yy2, fr; uint len;
    // tmp = tmp tmp t1 t2 t3

    m = c->m; len = c->l; fr = c->fr;
    yy2 = b + len; yy1 = xx1 + len; zz1 = yy1 + len;
    t1 = tmp + 2*len; t2 = t1 + len; t3 = t2 + len;

    if(!acl_zero(b, 2*len)) {       // if b == inf then ret a
        if(acl_zero(zz1, len))      // if a == inf then ret b
            acl_ecc_pro(a, b, len);
        else {
            acl_p_sqr_fr(t1, zz1);              // 3 t1 = zz1^2 == A
```

```
            acl_p_mul_fr(t2, t1, zz1);          // 4 t2 = t1 * zz1 == B
            acl_p_mul_fr(t1, t1, xx2);          // 5 t1 = t1 * xx2 == C
            acl_p_mul_fr(t2, t2, yy2);          // 6 t2 = t2 * yy2 == D
            acl_p_mod_sub(t1, t1, xx1, m, len); // 7 t1 = t1 - xx1 == E
            acl_p_mod_sub(t2, t2, yy1, m, len); // 8 t2 = t2 - yy1 == F
            if(acl_zero(t1, len))
                if(acl_zero(t2, len)) acl_ecc_dbl(a, tmp, c);
                else acl_mov32(zz1, 0, len);
            else {
                acl_p_mul_fr(zz1, zz1, t1);     // 10 zz1 = zz1 * t1 -> zz
                acl_p_sqr_fr(t3, t1);           // 11 t3 = t1^2 == G
                acl_p_mul_fr(t1, t3, t1);       // 12 t1 = t3 * t1 == H
                acl_p_mul_fr(t3, t3, xx1);      // 13 t3 = t3 * xx1 == I
                acl_p_sqr_fr(xx1, t2);          // 15 xx1 = t2^2
                acl_p_mod_sub(xx1, xx1, t3, m, len);
                acl_p_mod_sub(xx1, xx1, t3, m, len); // 16 xx1 = xx1 - 2 * t3
                acl_p_mod_sub(xx1, xx1, t1, m, len); // 17 xx1 = xx1 - t1 -> xx
                acl_p_mod_sub(t3, t3, xx1, m, len);  // 18 t3 = t3 - xx1
                acl_p_mul_fr(t3, t3, t2);            // 19 t3 = t3 * t2
                acl_p_mul_fr(t1, t1, yy1);           // 20 t1 = t1 * yy1
                acl_p_mod_sub(yy1, t3, t1, m, len);  // 21 yy1 = t3 - t1 -> yy
            }
        }
    }
}
```

Source file 106 acl_p_ecc_aff.c

```
// convert projective to affine coordinates (x, y, z) -> (x', y', ??)
// where x' and y' are the affine coordinates (z is corrupted)

// a - pointer to ecc point in projective coordinates (x, y, z)
// tmp - pointer to storage space for 4*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_p_ecc_aff(vect3 a, vect4 tmp, ecc_t *c)
{
    vect m, t1, yy, zz, fr; uint len;
    // tmp = tmp tmp tmp t1

    m = c->m; len = c->l; fr = c->fr;
    yy = xx + len; zz = yy + len; t1 = tmp + 3*len;

    if(c->t & ECC_A_MASK) {
        m = m + len;
        acl_mov(t1, zz, len); acl_p_mod(zz, t1, len, m, len);
    }
    if(acl_zero(zz, len))
        acl_mov32(a, 0, 2*len);
    else {
        acl_p_mod_inv(t1, zz, 0, m, tmp, len);
        acl_p_mul_fr(yy, yy, t1);
        acl_p_sqr_fr(t1, t1);
        acl_p_mul_fr(yy, yy, t1);
```

```
        acl_p_mul_fr(xx, xx, t1);
        acl_mov(t1, xx, len); acl_p_mod(xx, t1, len, m, len);
        acl_mov(t1, yy, len); acl_p_mod(yy, t1, len, m, len);
    }
}
```

Source file 107 acl_p_ecc_p2str.c

```
// convert point a(x,y) to string (with or without compression)
// str - pointer to free space for resulting string
// a - pointer to ecc point in affine coordinates (x, y)
// comp - TRUE: use compression, FALSE: don't use compression
// tmp - unused, for compatibility with acl_2_ecc_p2str (set to zero)
// c - pointer to elliptic curve

// for exact description see SEC 1: Elliptic Curve Cryptography, p. 11
// if (x,y) == point at infinity then return "00"
// if comp == TRUE
//   if y mod 2 == 0 return "02xxxxxxxxxx..."
//   else            return "03xxxxxxxxxx..."
// else
//   return "04xxxxxxxxxx...yyyyyyyyy..."

#include "..\acl.h"

void acl_p_ecc_p2str(bytes str, vect2 a, bool_t comp, vect tmp, ecc_t *c)
{
    vect m; uint len, len_m;

    m = c->m; len = c->l;

    *str++ = '0';
    if(acl_zero(a, 2*len)) {
        *str++ = '0';
    } else {
        if(c->t & ECC_A_MASK) m = m + len;
        len_m = 4*len;
        while(((bytes) m)[len_m - 1] == 0) len_m--;
        if(comp) {
            if(a[len] & 1) *str++ = '3';
            else           *str++ = '2';
            acl_hex2str_le(str, a, 2*len_m);
            str += 2*len_m;
        } else {
            *str++ = '4';
            acl_hex2str_le(str, a, 2*len_m);
            str += 2*len_m;
            acl_hex2str_le(str, a + len, 2*len_m);
            str += 2*len_m;
        }
    }
    *str = 0;
}
```

Source file 108 acl_p_ecc_str2p.c

```
// convert string (with or without compression) to point a(x,y)
// a - resulting point in affine coordinates (x, y)
// str - string representation of point
// tmp - pointer to storage space for 9*len ints
// c - pointer to elliptic curve
// returns TRUE if the point is valid, FALSE otherwise

// for exact description see SEC 1: Elliptic Curve Cryptography, p. 12

// str can be one of the following:
// "00"
// "02xxxxxxxx..."
// "03xxxxxxxx..."
// "04xxxxxxxx...yyyyyyyyyyy..."

#include "..\acl.h"

bool_t acl_p_ecc_str2p(vect2 a, bytes str, vect9 tmp, ecc_t *c)
{
    vect m, t1, yy; uint len, len_m, h;

    m = c->m; len = c->l; t1 = tmp + 8*len; yy = a + len;

    if(str[1] == '0')
        acl_mov32(a, 0, 2*len);
    else {
        if(c->t & ECC_A_MASK) m = m + len;
        len_m = 4*len;
        while(((bytes) m)[len_m - 1] == 0) len_m--;
        acl_str2hex_le(a, len, str + 2, 2*len_m);
        if(acl_cmp(a, m, len) >= 0) return FALSE;
        if(str[1] == '4') {
            acl_str2hex_le(yy, len, str + 2 + 2*len_m, 2*len_m);
            if(acl_cmp(yy, m, len) >= 0) return FALSE;
            if(!acl_p_ecc_chk(a, tmp, c)) return FALSE;
        } else {
            h = str[1] - '2';
            if(h & ~1) return FALSE;
            acl_p_ecc_chk(a, tmp, c);   // now tmp+2*len holds (x^3 + ax + b)
            acl_mov(yy, tmp + 2*len, len);
            if(!acl_p_sqrt(t1, yy, m, &acl_prng_lc, tmp, len)) return FALSE;
            acl_mov(yy, t1, len);
            if((h ^ yy[0]) & 1)
                acl_p_mod_sub(yy, m, yy, m, len);
        }
    }
    return TRUE;
}
```

Source file 109   acl_p_ecc_func.c

```
#include "..\acl.h"

const ecc_func_t acl_p_ecc_func = {
    acl_p_ecc_chk,
    acl_p_ecc_dbl,
    acl_p_ecc_add,
```

```
    acl_p_ecc_aff,
    acl_p_ecc_p2str,
    acl_p_ecc_str2p
};
```

Source file 110 acl_2_ecc_chk.c

```
// returns TRUE if affine point is on curve, FALSE otherwise

// a - pointer to ecc point in affine coordinates (x, y)
// tmp - pointer to storage space for 4*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"
#include "..\acl_config.h"

bool_t acl_2_ecc_chk(vect2 a, vect4 tmp, ecc_t *c)
{
    vect m, t1, t2, yy, fr; uint len;
    // tmp = tmp tmp t1 t2

    m = c->m; len = c->l; fr = c->fr;
    yy = xx + len; t1 = tmp + 2*len; t2 = t1 + len;

#if ACL_CHK_INF_ON_CURVE
    if(acl_zero(a, 2*len)) return TRUE;
#endif
    acl_2_sqr_fr(t1, xx);           // t1 = x^2
    if(!c->a) {                     // t1 = x^2 * (x + a)
        acl_2_mul_fr(t1, t1, xx);
    } else if((int) c->a == 1) {
        acl_xor32(t2, xx, 1, len);
        acl_2_mul_fr(t1, t1, t2);
    } else {
        acl_xor(t2, xx, c->a, len);
        acl_2_mul_fr(t1, t1, t2);
    }
    if((int) c->b == 1)             // t1 = x^3 + ax + b
        acl_xor32(t1, t1, 1, len);
    else
        acl_xor(t1, t1, c->b, len);
    acl_xor(t2, xx, yy, len);       // t2 = y + x
    acl_2_mul_fr(t2, t2, yy);       // t2 = y^2 + xy
    return !acl_cmp(t1, t2, len);   // t1 == t2 ?
}
```

Source file 111 acl_2_ecc_dbl.c

```
// point doubling with fast reduction (Lopez-Dahab <= 2 * Lopez-Dahab)
// taken directly from
//      D. Hankerson, A. Menezes, S.A. Vanstone:
//      Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004
// algortihm 3.24, p. 94

// a - pointer to ecc point in projective coordinates (x, y, z)
```

```
// tmp - pointer to storage space for 4*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_2_ecc_dbl(vect3 a, vect4 tmp, ecc_t *c)
{
    vect m, t1, t2, yy, zz, fr; uint len;
    // tmp = tmp tmp t1 t2

    m = c->m; len = c->l; fr = c->fr;
    yy = xx + len; zz = yy + len; t1 = tmp + 2*len; t2 = t1 + len;

    if(!acl_zero(zz, len)) {    // 2 * inf == inf
        acl_2_sqr_fr(t1, zz);           // 2 t1 = zz^2
        acl_2_sqr_fr(t2, xx);           // 3 t2 = xx^2
        acl_2_mul_fr(zz, t1, t2);       // 4 zz = t1 * t2 == new zz
        acl_2_sqr_fr(xx, t2);           // 5 xx = xx^4
        acl_2_sqr_fr(t1, t1);           // 6 t1 = zz^4
        if((int) c->b == 1)             // 7 t2 = b * zz^4
            acl_mov(t2, t1, len);
        else {
            acl_2_mul_fr(t2, t1, c->b);
        }
        acl_xor(xx, xx, t2, len);       // 8 xx = xx + t2 == new xx
        acl_2_sqr_fr(yy, yy);           // 9 yy = yy^2
        if((int) c->a == 1)             // 10 yy = yy + a * zz
            acl_xor(yy, yy, zz, len);
        else if(c->a) {
            acl_2_mul_fr(t1, c->a, zz);
            acl_xor(yy, yy, t1, len);
        }
        acl_xor(yy, yy, t2, len);       // 11 yy = yy + t2
        acl_2_mul_fr(yy, yy, xx);       // 12 yy = yy * xx
        acl_2_mul_fr(t1, zz, t2);       // 13 t1 = zz * t2
        acl_xor(yy, yy, t1, len);       // 14 yy = yy + t1 == new yy
    }
}
```

Source file 112 acl_2_ecc_add.c

```
// point addition with fast reduction  (Lopez-Dahab <= Lopez-Dahab + Affine)
// taken directly from
//      D. Hankerson, A. Menezes, S.A. Vanstone:
//      Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004
// algortihm 3.25, p. 95

// a - pointer to ecc point in projective coordinates (x, y, z)
// b - pointer to ecc point in affine coordinates (x, y)
// tmp - pointer to storage space for 5*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_2_ecc_add(vect3 a, vect2 b, vect5 tmp, ecc_t *c)
```

```
{
    vect m, t1, t2, t3, yy1, zz1, yy2, fr; uint len;
    // tmp = tmp tmp t1 t2 t3

    m = c->m; len = c->l; fr = c->fr;
    yy2 = b + len; yy1 = xx1 + len; zz1 = yy1 + len;
    t1 = tmp + 2*len; t2 = t1 + len; t3 = t2 + len;

    if(!acl_zero(b, 2*len)) {       // if b == inf then ret a
        if(acl_zero(zz1, len))      // if a == inf then ret b
            acl_ecc_pro(a, b, len);
        else {
            acl_2_mul_fr(t1, zz1, xx2);     // 3 t1 = zz1 * xx2
            acl_2_sqr_fr(t2, zz1);          // 4 t2 = zz1^2
            acl_xor(xx1, xx1, t1, len);     // 5 xx1 = xx1 + t1 == B
            acl_2_mul_fr(t1, zz1, xx1);     // 6 t1 = zz1 * B == C
            acl_2_mul_fr(t3, t2, yy2);      // 7 t3 = t2 * yy2
            acl_xor(yy1, yy1, t3, len);     // 8 yy1 = yy1 + t3 == A
            if(acl_zero(xx1, len))
                if(acl_zero(yy1, len)) {
                    acl_ecc_pro(a, b, len);
                    acl_ecc_dbl(a, tmp, c);
                } else
                    acl_mov32(zz1, 0, len);
            else {
                acl_2_sqr_fr(zz1, t1);      // 10 zz1 = C^2 == new zz
                acl_2_mul_fr(t3, t1, yy1);  // 11 t3 = A * C == E
                if((int) c->a == 1)         // 12 t1 = t1 + a * t2
                    acl_xor(t1, t1, t2, len);
                else if(c->a) {
                    acl_2_mul_fr(t2, c->a, t2);
                    acl_xor(t1, t1, t2, len);
                }
                acl_2_sqr_fr(t2, xx1);      // 13 t2 = B^2
                acl_2_mul_fr(xx1, t2, t1);  // 14 xx1 = t2 * t1 == D
                acl_2_sqr_fr(t2, yy1);      // 15 t2 = A^2
                acl_xor(xx1, xx1, t2, len); // 16 xx1 = xx1 + t2
                acl_xor(xx1, xx1, t3, len); // 17 xx1 = xx1 + t3 == new xx
                acl_2_mul_fr(t2, xx2, zz1); // 18 t2 = xx2 * zz1
                acl_xor(t2, t2, xx1, len);  // 19 t2 = t2 + xx1 == F
                acl_2_sqr_fr(t1, zz1);      // 20 t1 = zz1^2
                acl_xor(t3, t3, zz1, len);  // 21 t3 = t3 + zz1
                acl_2_mul_fr(yy1, t3, t2);  // 22 yy1 = t3 * t2
                acl_xor(t2, xx2, yy2, len); // 23 t2 = xx2 + yy2
                acl_2_mul_fr(t3, t1, t2);   // 24 t3 = t1 * t2 == G
                acl_xor(yy1, yy1, t3, len); // 25 yy1 = yy1 + t3 == new yy
            }
        }
    }
}
```

Source file 113 acl_2_ecc_aff.c

```
// convert projective to affine coordinates (x, y, z) -> (x', y', ??)
// where x' and y' are the affine coordinates (z is corrupted)

// a - pointer to ecc point in projective coordinates (x, y, z)
```

```
// tmp - pointer to storage space for 5*len ints
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_2_ecc_aff(vect3 a, vect5 tmp, ecc_t *c)
{
    vect t1, t2, yy, zz, fr; uint len; int k;
    // tmp = tmp tmp tmp t1 t2

    len = c->l; fr = c->fr;
    yy = xx + len; zz = yy + len; t1 = tmp + 3*len; t2 = t1 + len;

    if(acl_zero(zz, len))
        acl_mov32(a, 0, 2*len);
    else {
        acl_mov32(t1, 1, len);      // recover m from fr
        k = 0;
        while(c->fr[k]) {
            acl_bit_set(t1, c->fr[k]);
            k++;
        }
        acl_2_mod_inv(t2, zz, t1, tmp, len);
        acl_2_mul_fr(xx, xx, t2);
        acl_2_sqr_fr(t2, t2);
        acl_2_mul_fr(yy, yy, t2);
    }
}
```

Source file 114 acl_2_ecc_p2str.c

```
// convert point a(x,y) to string (with or without compression)
// str - pointer to free space for resulting string
// a - pointer to ecc point in affine coordinates (x, y)
// comp - TRUE: use compression, FALSE: don't use compression
// tmp - pointer to storage space for 5*len ints
// c - pointer to elliptic curve

// for exact description see SEC 1: Elliptic Curve Cryptography, p. 11
// if (x,y) == point at infinity then return "00"
// if comp == TRUE
//   if x == 0 return "02xxxxxxxxxx..."
//   else
//     if y/x mod z == 0 return "02xxxxxxxxxx..."
//     else              return "03xxxxxxxxxx..."
// if comp == FALSE
//   return "04xxxxxxxxxx...yyyyyyyyy..."

#include "..\acl.h"
#include "..\acl_int.h"

void acl_2_ecc_p2str(bytes str, vect2 a, bool_t comp, vect5 tmp, ecc_t *c)
{
    vect t1, t2, fr; uint len, len_m; int k;
    // tmp = tmp tmp tmp t1 t2
```

```
    len = c->l; fr = c->fr; t1 = tmp + 3*len; t2 = t1 + len;

    *str++ = '0';
    if(acl_zero(a, 2*len))
        *str++ = '0';
    else {
        acl_mov32(t1, 1, len);      // recover m from fr
        k = 0;
        while(c->fr[k]) {
            acl_bit_set(t1, c->fr[k]);
            k++;
        }
        len_m = 4*len;
        while(((bytes) t1)[len_m - 1] == 0) len_m--;
        if(comp) {
            acl_2_mod_inv(t2, a, t1, tmp, len);
            acl_2_mul_fr(t1, a + len, t2);
            if(t1[0] & 1) *str++ = '3';
            else          *str++ = '2';
            acl_hex2str_le(str, a, 2*len_m);
            str += 2*len_m;
        } else {
            *str++ = '4';
            acl_hex2str_le(str, a, 2*len_m);
            str += 2*len_m;
            acl_hex2str_le(str, a + len, 2*len_m);
            str += 2*len_m;
        }
    }
    *str = 0;
}
```

Source file 115 acl_2_ecc_str2p.c

```
// convert string (with or without compression) to point a(x,y)
// a - resulting point in affine coordinates (x, y)
// str - string representation of point
// tmp - pointer to storage space for 6*len ints
// c - pointer to elliptic curve
// returns TRUE if the point is valid, FALSE otherwise

// for exact description see SEC 1: Elliptic Curve Cryptography, p. 12

// str can be one of the following:
// "00"
// "02xxxxxxxx..."
// "03xxxxxxxx..."
// "04xxxxxxxx...yyyyyyyyyyy..."

// for square root and half-trace calculation details see
// D. Hankerson, A. Menezes, and S.A. Vanstone,
// Guide to Elliptic Curve Cryptography, Springer-Verlag, 2004
// half-trace: p. 132, square root: p. 136

#include "..\acl.h"
#include "..\acl_int.h"
```

```
bool_t acl_2_ecc_str2p(vect2 a, bytes str, vect6 tmp, ecc_t *c)
{
    vect fr, m, t1, t2, yy; uint i, len, len_m;
    // tmp = tmp tmp tmp m t1 t2

    len = c->l; fr = c->fr;
    m = tmp + 3*len; t1 = m + len; t2 = t1 + len; yy = a + len;

    if(str[1] == '0')
        acl_mov32(a, 0, 2*len);
    else {
        acl_mov32(m, 1, len);      // recover m from fr
        i = 0;
        while(fr[i]) {
            acl_bit_set(m, fr[i]);
            i++;
        }
        len_m = 4*len;
        while(((bytes) m)[len_m - 1] == 0) len_m--;
        acl_str2hex_le(a, len, str + 2, 2*len_m);
        for(i = fr[0]; i < 32*len; i++) if(acl_bit(a, i, len)) return FALSE;
        if(str[1] == '4') {
            acl_str2hex_le(yy, len, str + 2 + 2*len_m, 2*len_m);
            for(i = fr[0]; i < 32*len; i++) if(acl_bit(yy, i, len)) return FALSE;
        } else {
            if(acl_zero(a, len)) {
                if((int) c->b == 1)
                    acl_mov32(yy, 1, len);
                else {
                    acl_mov(yy, c->b, len);
                    for(i = 0; i < fr[0] - 1; i++) {
                        acl_2_sqr_fr(yy, yy);       // calculate square root
                    }
                }
            } else {
                acl_2_mod_inv(t2, a, m, tmp, len);
                acl_2_sqr_fr(t2, t2);
                if((int) c->b != 1) { acl_2_mul_fr(t2, t2, c->b); }
                if((int) c->a == 1) {
                    acl_xor32(t2, t2, 1, len);
                } else if(c->a) {
                    acl_xor(t2, t2, c->a, len);
                }
                acl_xor(t2, t2, a, len);
                acl_mov(t1, t2, len);
                for(i = 1; i <= (fr[0] >> 1); i++) {
                    acl_2_sqr_fr(t2, t2);       // calculate half-trace
                    acl_2_sqr_fr(t2, t2);
                    acl_xor(t1, t1, t2, len);
                }
                if(((str[1] - '2') ^ t1[0]) & 1) acl_xor32(t1, t1, 1, len);
                acl_2_mul_fr(yy, a, t1);
            }
        }
        if(!acl_2_ecc_chk(a, tmp, c)) return FALSE;
    }
    return TRUE;
}
```

Source file 116 acl_2_ecc_func.c

```
#include "..\acl.h"

const ecc_func_t acl_2_ecc_func = {
    acl_2_ecc_chk,
    acl_2_ecc_dbl,
    acl_2_ecc_add,
    acl_2_ecc_aff,
    acl_2_ecc_p2str,
    acl_2_ecc_str2p
};
```

Source file 117 acl_ecc_pro.c

```
// b(x,y) -> a(x,y,1)    copy a point in affine to projective coordinates

#include "..\acl.h"

void acl_ecc_pro(vect3 a, vect2 b, size_t len)
{
    acl_mov(a, b, 2*len);
    acl_mov32(a + 2*len, 1, len);
}
```

Source file 118 acl_ecc_neg.c

```
// negate a point in projective or affine coordinates

// a - pointer to ecc point in projective (x, y, z) or affine (x, y) coordinates
// c - pointer to elliptic curve

#include "..\acl.h"

void acl_ecc_neg(vect3 a, ecc_t *c)
{
    if((c->t & ECC_F_MASK) == ECC_2)    // over GF(2)
        acl_xor(a + c->l, a, a + c->l, c->l);
    else                                // over GF(p)
        acl_p_mod_sub(a + c->l, c->m, a + c->l, c->m, c->l);
}
```

Source file 119 acl_ecc_pre.c

```
// pre-computation for ecc point multiplication

// pre - where to store the pre-comp - space for (2^w-1)*len*2 ints
// p - affine point
// w - number of teeth in comb
// s - distance between teeth of comb (bitlength of exponent <= w*s !!!)
// tmp - temporary storage (8*len ints)
// c - pointer to elliptic curve

// examples of width, spacing
```

```
// width    spacing     comment                              table size (ints)
//   1         0        no precomputation                        2*len
//   2      32*len/2                                             2*len*3
//   3      32*len/3+1  to make sure that w*s >= 32*len          2*len*7
//   4      32*len/4                                             2*len*15
//   5      32*len/5+1  same                                     2*len*31
//   6      32*len/6+1                                           2*len*63
//     ...
// actually it shouldn't be +1, but ceiling(32*len/width)


#include "..\acl.h"
#include "..\acl_int.h"

void acl_ecc_pre(vectN pre, vect2 p, uint w, uint s, vect8 tmp, ecc_t *c)
{
    vect zz, t1, base, h; uint len, len2, comb, i, j;
    // tmp[8*len] = x y z t1 t1 t1 t1 t1

    len = c->l; len2 = 2*len; zz = tmp + len2; t1 = tmp + 3*len;

    acl_mov(pre, p, len2);
    base = pre;
    comb = 1;
    for(i = 1; i < w; i++) {
        acl_ecc_pro(tmp, base, len);            // previous base point
        for(j = 0; j < s; j++) acl_ecc_dbl(tmp, t1, c); // 2^s * base
        acl_ecc_aff(tmp, t1, c);
        base += comb * len2;                    // new base point
        acl_mov(base, tmp, len2);
        comb <<= 1;

        for(j = 1; j < comb; j++) {             // the in betweens
            acl_ecc_pro(tmp, base, len);
            h = pre + (j - 1) * len2;           // already done
            acl_ecc_add(tmp, h, t1, c);
            acl_ecc_aff(tmp, t1, c);
            h = base + j * len2;                // destination
            acl_mov(h, tmp, len2);
        }
    }
}
```

Source file 120 acl_ecc_mul.c

```
// ecc point multiplication   res = k * p + l * q

// res - result in affine coordinates (but must have space for projective)
// p - pointer to first point or its pre-computation (affine)
// q - pointer to second point or its pre-computation (affine)
// w - number of teeth in comb (1 -> no pre-computation)
// s - distance between teeth of comb (if pre-computation used)
// k - pointer to number multiplying p
// l - pointer to number multiplying q
// len_kl - length of k, l in 32-bit words
// tmp - temporary storage (5*len ints)
// c - pointer to elliptic curve
```

```
#include "..\acl.h"
#include "..\acl_int.h"

void acl_ecc_mul(vect3 res, vect p, vect q, uint w, uint s, vect k, vect l, \
                 size_t len_kl, vect5 tmp, ecc_t *c)
{
    uint len2, i, j, hk, hl;

    len2 = 2 * c->l;

    acl_mov32(res + len2, 0, c->l);
    if(w == 1) {
        for(i = 32 * len_kl; i; i--) {
            acl_ecc_dbl(res, tmp, c);
            if(p && acl_bit(k, i - 1, len_kl)) acl_ecc_add(res, p, tmp, c);
            if(q && acl_bit(l, i - 1, len_kl)) acl_ecc_add(res, q, tmp, c);
        }
    } else {
        for(i = s; i; i--) {
            acl_ecc_dbl(res, tmp, c);
            hk = 0; hl = 0;
            for(j = w * s; j; j -= s) {
                if(p) hk = (hk << 1) + acl_bit(k, i - 1 + j - s, len_kl);
                if(q) hl = (hl << 1) + acl_bit(l, i - 1 + j - s, len_kl);
            }
            if(hk) acl_ecc_add(res, p + (hk - 1) * len2, tmp, c);
            if(hl) acl_ecc_add(res, q + (hl - 1) * len2, tmp, c);
        }
    }
    acl_ecc_aff(res, tmp, c);
}
```

Source file 121 acl_ecdsa_gen.c

```
// ecdsa signature generation; the length of all arrays is len = c->ln
// (the length of the order of the base point)
// except for e (the hash) whose length is "len_e"

// r, s - resulting signature
// e - hash
// e_len - length of hash in 32-bit words
// dA - private key
// base - the curve's base-point or its pre-computation table
// wi - width of comb
// sp - spacing of comb (ignored if wi == 1)
// rnd_strong - random number generator
// tmp - temporary storage (9*len ints)
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

void acl_ecdsa_gen(vect r, vect s, vect e, size_t len_e, vect dA, \
                   vectN base, uint wi, uint sp, \
                   prng rnd_strong, vect9 tmp, ecc_t *c)
{
```

```
    uint len, m_inv; vect t1, t2, a, k, m;
    // tmp = tmp tmp t1 t2 tmp x y z k

    m = c->n; len = c->ln; a = tmp + 5*len; k = a + 3*len;
    t1 = tmp + 2*len; t2 = t1 + len;

aeg_again:
    rnd_strong(t1, len);
    acl_p_mod(k, t1, len, m, len);          // k = rnd mod n
    if(acl_zero(k, len)) goto aeg_again;
    acl_ecc_mul(a, base, 0, wi, sp, k, 0, len, tmp, c);  // a = k * G
    acl_p_mod(r, a, c->l, m, len);          // r = x1 mod n
    if(acl_zero(r, len)) goto aeg_again;
    acl_p_mont_pre(0, t1, &m_inv, m, len);
    acl_p_mul_mont(t1, r, t1);              // t1 = r * R
    acl_p_mul_mont(t1, t1, dA);             // t1 = r * dA
    acl_p_mod(t2, e, len_e, m, len);        // t2 = e
    acl_p_mod_add(t1, t1, t2, m, len);      // t1 = e + r * dA
    acl_p_mod_inv(t2, k, 32*len, m, a, len);    // t2 = k^(-1) * R
    acl_p_mul_mont(s, t1, t2);              // s = k^(-1) * (e + r * dA)
    if(acl_zero(s, len)) goto aeg_again;
}
```

Source file 122 acl_ecdsa_ver.c

```
// ecdsa signature verification; the length of all arrays is len = c->ln
// (the length of the order of the base point)
// except for e (the hash) whose length is "len_e"

// r, s - signature to verify
// e - hash
// e_len - length of hash in 32-bit words
// qA - public key (ec point) or its pre-computation table
// base - the curve's base point or its pre-computation table
// wi - width of comb
// sp - spacing of comb (ignored if wi == 1)
// tmp - temporary storage (10*len ints)
// c - pointer to elliptic curve

#include "..\acl.h"
#include "..\acl_int.h"

bool_t acl_ecdsa_ver(vect r, vect s, vect e, size_t len_e, vectN qA, \
                     vectN base, uint wi, uint sp, vect10 tmp, ecc_t *c)
{
    uint len, m_inv; vect a, k, l, m;
    // tmp = tmp tmp tmp tmp tmp x y z k l

    m = c->n; len = c->ln; a = tmp + 5*len; k = a + 3*len; l = k + len;

    if(acl_zero(r, len)) return FALSE;
    if(acl_zero(s, len)) return FALSE;
    if(acl_cmp(r, m, len) > 0) return FALSE;
    if(acl_cmp(s, m, len) > 0) return FALSE;
    acl_p_mod(k, e, len_e, m, len);         // k = e
    acl_p_mod_inv(l, s, 32*len, m, a, len); // l = s^(-1) * R
    m_inv = acl_p_mont_m_inv(m);
```

```
    acl_p_mul_mont(k, k, l);                // k = e * s^(-1)
    acl_p_mul_mont(l, l, r);                // l = r * s^(-1)
    acl_ecc_mul(a, base, qA, wi, sp, k, l, len, tmp, c);    // a = k*G + l*qA
    acl_p_mod(k, a, c->l, m, len);          // k = x1 mod n
    if(acl_cmp(r, k, len)) return FALSE;    // k != r ?
    return TRUE;
}
```

Source file 123 magma.txt

```
# this is a magma script. magma can be found here:
# http://magma.maths.usyd.edu.au/magma/

# double the base point on the nist curve p-192

p192 := 2^192-2^64-1;
a := -3;
b := 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1;

K := GF(p192);
a := K!a;
b := K!b;
E := EllipticCurve([a, b]);

gx := 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012;
gy := 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811;

G := E![gx, gy, 1];
P := 2*G;

n := Integers()!P[1]; rr := 2^32;
for i:=1 to 6 do
    x := n mod rr;
    x:Hex;
    n := n div rr;
end for;
```

Source file 124 system.h

```
#ifndef SYSTEM_H
#define SYSTEM_H

#define FREQUENCY 12000000

#define UNIT_CYCLES 0
#define UNIT_MICROSECONDS 1
#define UNIT_MILLISECONDS 2

#include "..\acl.h"

void init_serial(void);
void put_str(char *p);
void put_char(int ch);
int  get_char(void);
void put_hex(unsigned int hex);
void put_int(unsigned int x);
```

```
void put_vect(unsigned int *p, unsigned int len);
void put_str(char *p);

void init_timers(int unit);
void restart_timer(int timer);
void start_timer(int timer);
uint stop_timer(int timer);
void put_val(char *p, uint value);

#endif
```

Source file 125 timing.c

```
#include "lpc213x.h"
#include "system.h"
#include "..\acl.h"

int our_unit;

void init_timers(int unit) {
    int tmp;

    our_unit = unit;
    if(unit == UNIT_CYCLES) tmp = 1;
    else {
        if(unit == UNIT_MICROSECONDS) tmp = FREQUENCY / 1000000;
        if(unit == UNIT_MILLISECONDS) tmp = FREQUENCY / 1000;
        tmp *= (PLLSTAT & 0x1F) + 1;
        if((VPBDIV & 3) == 0) tmp >>= 2;
        if((VPBDIV & 3) == 2) tmp >>= 1;
    }

    T1TCR = 0; T1PC = 0; T1TC = 0; T1PR = tmp - 1;

    T0TCR = 0; T0PC = 0; T0TC = 0; T0PR = tmp - 1;
}

void restart_timer(int timer) {
    if(timer) {
        T1TCR = 0; T1PC = 0; T1TC = 0; T1TCR = 1;
    } else {
        T0TCR = 0; T0PC = 0; T0TC = 0; T0TCR = 1;
    }
}

void start_timer(int timer) {
    if(timer) T1TCR = 1;
    else      T0TCR = 1;
}

uint stop_timer(int timer) {
    int tmp;

    if(timer) {
        T1TCR = 0; tmp = T1TC;
    } else {
        T0TCR = 0; tmp = T0TC;
```

```
    }

    if(our_unit == UNIT_CYCLES) {
        if((VPBDIV & 3) == 0) tmp *= 4;
        if((VPBDIV & 3) == 2) tmp *= 2;
    }
    return tmp;
}
```

Source file 126 serial.c

```
/* Parts taken from KEIL ARM development tools libraries */

#include "lpc213x.h"                   /* LPC213x definitions               */
#include "system.h"
#include "..\acl.h"

#define CR 0x0D                        /* carriage return character         */
#define BAUD_RATE 9600
#define WIDTH 80                       /* max chars in line                 */

int no_chars = 0;

void init_serial(void) {               /* Initialize Serial Interface       */
    int tmp;

    PINSEL0 = 0x00050000;              /* Enable RxD1 and TxD1              */
    U1LCR = 0x83;                      /* 8 bits, no Parity, 1 Stop bit     */
    tmp = FREQUENCY/BAUD_RATE;
    tmp *= (PLLSTAT & 0x1F) + 1;
    if((VPBDIV & 3) == 0) tmp >>= 2;
    if((VPBDIV & 3) == 2) tmp >>= 1;
    tmp >>= 4;
    U1DLL = tmp & 0xFF;
    U1DLM = (tmp >> 8) & 0xFF;
    U1LCR = 0x03;                      /* DLAB = 0                          */
}

void put_char(int ch) {                /* Write character to Serial Port    */
    if((no_chars == WIDTH) || (ch == '\n')) {
        no_chars = 0;
        while(!(U1LSR & 0x20));
        U1THR = CR;
        while(!(U1LSR & 0x20));
        U1THR = '\n';
    } else {
        no_chars++;
        while(!(U1LSR & 0x20));
        U1THR = ch;
    }
}

int get_char(void) {                   /* Read character from Serial Port   */
    while(!(U1LSR & 0x01));
    return(U1RBR);
}
```

```
void put_hex(unsigned int hex) {        /* Write Hex Digit to Serial Port  */
    int tmp;

    tmp = hex - 10;
    if(tmp >= 0) put_char('A' + tmp); else put_char('0' + hex);
}

void put_int(unsigned int x) {
    int i;

    for(i=0; i<8; i++) {
        put_hex((x >> 28) & 0x0F);
        x <<= 4;
    }
}

void put_vect(unsigned int *p, unsigned int len) {
    uint i;

    for(i=len; i; i--) put_int(p[i-1]);
}

void put_str(char *p) {                 /* Write string */
    while(*p) put_char(*p++);
}

char data[11];

void put_val(char *p, uint value) {
    put_str(p);
    acl_hex2str_dec(data, 10, &value, 1);
    data[10] = '\0';
    put_str(data);
}
```

Source file 127 main.c

```
#include "system.h"
#include "..\acl.h"

bool_t test_aes(void);
bool_t test_sha(void);
bool_t test_rsa(void);
bool_t test_ecc(void);
bool_t test_ecdsa(void);


static void acl_error(char *p) {
    put_str("\nerror: "); put_str(p);
    while(1) ;
}

static void init_random(void) {
    int res;

    put_str("\nEnter 4 chars: ");
    res = get_char();
```

```
    res = (res << 8) | get_char();
    res = (res << 8) | get_char();
    res = (res << 8) | get_char();
    acl_prng_lc_init(res);
}

int main(void)
{
    init_serial();
    init_random();

    init_timers(UNIT_CYCLES);
    if(test_aes()) acl_error("AES");

    init_timers(UNIT_MICROSECONDS);
    if(test_sha()) acl_error("SHA");
    if(test_rsa()) acl_error("RSA");
    if(test_ecc()) stop_error("ECC");
    if(test_ecdsa()) stop_error("ECDSA");

    put_str("\n\na-ok");    // if program gets here, the tests have been passed
    while(1) ;
}
```

Source file 128 test_aes.c

```
// perform monte carlo tests of the aes implementation
// the files referenced here can be found for example at
//      www.gnu.org/software/gnu-crypto/vectors/
// the tables go all the way to 400 iterations, but we only
// go to 2 for time's sake
// for more rigorous tests, change the number of iterations
// and the known answer

#include "..\acl.h"
#include "system.h"

uint key[8];
uint key_exp[60];  /* (nk+7)*4, biggest: nk = 8 for aes-256 */
uint pt[4];
uint ct[4];
uint oct[4];
uint iv[4];
uint tmp[4];
uint i, j, k, h, len_aes;

const char *aes_ecb_en_results[] = {
    "0AC15A9AFBB24D54AD99E987208272E2",
    "77BA00ED5412DFF27C8ED91F3C376172",
    "C737317FE0846F132B23C8C2A672CE22"
};

const char *aes_ecb_de_results[] = {
    "E3FD51123B48A2E2AB1DB29894202222",
    "CC01684BE9B29ED01EA7923E7D2380AA",
    "15173A0EB65F5CC05E704EFE61D9E346"
};
```

```
const char *aes_cbc_en_results[] = {
    "983BF6F5A6DFBCDAA19370666E83A99A",
    "C6FB25A188CF7F3F24B07896C0C76D90",
    "81EA5BA46945C1705F6F89778868CC67"
};

const char *aes_cbc_de_results[] = {
    "F5372F9735C5685F1DA362AF6ECB2940",
    "F9604074F8FA45AC71959888DD056F9F",
    "D36C27EBB8FA0BC9FA368DF850FD45FB"
};

static void zero_all(void) {
    acl_mov32(key, 0, 8);
    acl_mov32(pt, 0, 4);
    acl_mov32(ct, 0, 4);
    acl_mov32(oct, 0, 4);
    acl_mov32(iv, 0, 4);
    restart_timer(0);
}

bool_t test_aes_ecb_en(void) {

    /* 128-bit ecb monte carlo encryption test */
    put_str("\naes ecb en 128");
    len_aes = ACL_128;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(ct, pt, 4);
        for(j=0; j<10000; j++) acl_aes_ecb_en(ct, ct, key_exp, len_aes);
        /* compare with ecb_e_m.txt 128 bits */
        acl_mov(pt, ct, 4);
        acl_xor(key, key, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_en_results[0], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;

    /* 192-bit ecb monte carlo encryption test */
    put_str("\naes ecb en 192");
    len_aes = ACL_192;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(ct, pt, 4);
        for(j=0; j<10000-2; j++) acl_aes_ecb_en(ct, ct, key_exp, len_aes);
        acl_aes_ecb_en(oct, ct, key_exp, len_aes);
        acl_aes_ecb_en(ct, oct, key_exp, len_aes);
        /* compare with ecb_e_m.txt 192 bits */
        acl_mov(pt, ct, 4);
        acl_xor(key, key, oct+2, 2);
        acl_xor(key+2, key+2, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_en_results[1], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;
```

```
    /* 256-bit ecb monte carlo encryption test */
    put_str("\naes ecb en 256");
    len_aes = ACL_256;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(ct, pt, 4);
        for(j=0; j<10000-2; j++) acl_aes_ecb_en(ct, ct, key_exp, len_aes);
        acl_aes_ecb_en(oct, ct, key_exp, len_aes);
        acl_aes_ecb_en(ct, oct, key_exp, len_aes);
        /* compare with ecb_e_m.txt 256 bits */
        acl_mov(pt, ct, 4);
        acl_xor(key, key, oct, 4);
        acl_xor(key+4, key+4, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_en_results[2], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;

    return FALSE;
}

bool_t test_aes_ecb_de(void) {

    /* 128-bit ecb monte carlo decryption test */
    put_str("\naes ecb de 128");
    len_aes = ACL_128;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000; j++) acl_aes_ecb_de(pt, pt, key_exp, len_aes);
        /* compare with ecb_d_m.txt 128 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_de_results[0], 4);
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    /* 192-bit ecb monte carlo decryption test */
    put_str("\naes ecb de 192");
    len_aes = ACL_192;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000-2; j++) acl_aes_ecb_de(pt, pt, key_exp, len_aes);
        acl_aes_ecb_de(oct, pt, key_exp, len_aes);
        acl_aes_ecb_de(pt, oct, key_exp, len_aes);
        /* compare with ecb_d_m.txt 192 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, oct+2, 2);
        acl_xor(key+2, key+2, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_de_results[1], 4);
```

```
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    /* 256-bit ecb monte carlo decryption test */
    put_str("\naes ecb de 256");
    len_aes = ACL_256;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000-2; j++) acl_aes_ecb_de(pt, pt, key_exp, len_aes);
        acl_aes_ecb_de(oct, pt, key_exp, len_aes);
        acl_aes_ecb_de(pt, oct, key_exp, len_aes);
        /* compare with ecb_d_m.txt 256 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, oct, 4);
        acl_xor(key+4, key+4, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_ecb_de_results[2], 4);
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    return FALSE;
}

bool_t test_aes_cbc_en(void) {

    /* 128-bit cbc monte carlo encryption test */
    put_str("\naes cbc en 128");
    len_aes = ACL_128;
    zero_all();
    for(i=0; i<2; i++) {
        /* compare with cbc_e_m.txt 128 bits */
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(oct, iv, 4);
        for(j=0; j<10000; j++) {
            acl_aes_cbc_en(ct, pt, key_exp, len_aes, iv);
            acl_mov(pt, oct, 4);
            acl_mov(oct, ct, 4);
        }
        acl_xor(key, key, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_en_results[0], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;

    /* 192-bit cbc monte carlo encryption test */
    put_str("\naes cbc en 192");
    len_aes = ACL_192;
    zero_all();
    for(i=0; i<2; i++) {
        /* compare with cbc_e_m.txt 192 bits */
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(oct, iv, 4);
        for(j=0; j<10000; j++) {
            acl_aes_cbc_en(ct, pt, key_exp, len_aes, iv);
            acl_mov(pt, oct, 4);
            acl_mov(oct, ct, 4);
        }
```

```
        acl_xor(key, key, pt+2, 2);
        acl_xor(key+2, key+2, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_en_results[1], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;

    /* 256-bit cbc monte carlo encryption test */
    put_str("\naes cbc en 256");
    len_aes = ACL_256;
    zero_all();
    for(i=0; i<2; i++) {
        /* compare with cbc_e_m.txt 256 bits */
        acl_aes_key_en(key_exp, key, len_aes);
        acl_mov(oct, iv, 4);
        for(j=0; j<10000; j++) {
            acl_aes_cbc_en(ct, pt, key_exp, len_aes, iv);
            acl_mov(pt, oct, 4);
            acl_mov(oct, ct, 4);
        }
        acl_xor(key, key, pt, 4);
        acl_xor(key+4, key+4, ct, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_en_results[2], 4);
    if(acl_cmp(pt, tmp, 4)) return TRUE;

    return FALSE;
}

bool_t test_aes_cbc_de(void) {

    /* 128-bit cbc monte carlo decryption test */
    put_str("\naes cbc de 128");
    len_aes = ACL_128;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000; j++) acl_aes_cbc_de(pt, pt, key_exp, len_aes, iv);
        /* compare with cbc_d_m.txt 128 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_de_results[0], 4);
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    /* 192-bit cbc monte carlo decryption test */
    put_str("\naes cbc de 192");
    len_aes = ACL_192;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000-2; j++) acl_aes_cbc_de(pt, pt, key_exp, len_aes, iv);
        acl_aes_cbc_de(oct, pt, key_exp, len_aes, iv);
        acl_aes_cbc_de(pt, oct, key_exp, len_aes, iv);
```

```
        /* compare with cbc_d_m.txt 192 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, oct+2, 2);
        acl_xor(key+2, key+2, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_de_results[1], 4);
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    /* 256-bit cbc monte carlo decryption test */
    put_str("\naes cbc de 256");
    len_aes = ACL_256;
    zero_all();
    for(i=0; i<2; i++) {
        acl_aes_key_de(key_exp, key, len_aes);
        acl_mov(pt, ct, 4);
        for(j=0; j<10000-2; j++) acl_aes_cbc_de(pt, pt, key_exp, len_aes, iv);
        acl_aes_cbc_de(oct, pt, key_exp, len_aes, iv);
        acl_aes_cbc_de(pt, oct, key_exp, len_aes, iv);
        /* compare with cbc_d_m.txt 256 bits */
        acl_mov(ct, pt, 4);
        acl_xor(key, key, oct, 4);
        acl_xor(key+4, key+4, pt, 4);
    }
    h = stop_timer(0); put_val("  20000 = ", h);
    acl_str2bytes(tmp, (bytes) aes_cbc_de_results[2], 4);
    if(acl_cmp(ct, tmp, 4)) return TRUE;

    return FALSE;
}

bool_t test_aes(void) {

    if(test_aes_ecb_en()) return TRUE;
    if(test_aes_ecb_de()) return TRUE;
    if(test_aes_cbc_en()) return TRUE;
    if(test_aes_cbc_de()) return TRUE;

    return FALSE;
}
```

Source file 129 test_sha.c

```
// perform known answer tests of the sha implementations

#include "..\acl.h"
#include "system.h"

uint state[68];  // sha-1: 23, sha-256: 26, sha-512: 68
uint i, h;
uint tmp[16];

byte sha_test_str1[] = "abc";
byte sha_test_str2[] = "abcdbcdecdefdefgefghfghighijhijkijkljklm" \
                       "klmnlmnomnopnopq";
byte sha_test_str3[] = "abcdefghbcdefghicdefghijdefghijkefghijkl" \
                       "fghijklmghijklmnhijklmnoijklmnopjklmnopq" \
```

```
                       "klmnopqrlmnopqrsmnopqrstnopqrstu";

const char *sha1_results[] = {
    "a9993e364706816aba3e25717850c26c9cd0d89d",
    "84983e441c3bd26ebaae4aa1f95129e5e54670f1",
    "34aa973cd4c4daa4f61eeb2bdbad27316534016f"
};

const char *sha224_results[] = {
    "23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7",
    "75388b16512776cc5dba5da1fd890150b0c6455cb4f58b1952522525",
    "20794655980c91d8bbb4c1ea97618a4bf03f42581948b2ee4ee7ad67"
};

const char *sha256_results[] = {
    "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad",
    "248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1",
    "cdc76e5c9914fb9281a1c7e284d73e67f1809a48a497200e046d39ccc7112cd0"
};

const char *sha384_results[] = {
    "cb00753f45a35e8bb5a03d699ac65007272c32ab0eded163"
    "1a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7",

    "09330c33f71147e83d192fc782cd1b4753111b173b3b05d2"
    "2fa08086e3b0f712fcc7c71a557e2db966c3e9fa91746039",

    "9d0e1809716474cb086e834e310a4a1ced149e9c00f24852"
    "7972cec5704c2a5b07b8b3dc38ecc4ebae97ddd87f3d8985"
};

const char *sha512_results[] = {
    "ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a"
    "2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f",

    "8e959b75dae313da8cf4f72814fc143f8f7779c6eb9f7fa17299aeadb6889018"
    "501d289e4900f7e4331b99dec4b5433ac7d329eeb6dd26545e96e55b874be909",

    "e718483d0ce769644e2e42c7bc15b4638e1f98b13b2044285632a803afa973eb"
    "de0ff244877ea60a4cb0432ce577c31beb009c5c2c49aa2e4eadb217ad8cc09b"
};

static void sha1_str(bytes p) {
    acl_sha1_init(state);
    while(*p) acl_sha1(state, *p++);
    acl_sha1_done(state);
}

static void sha224_str(bytes p) {
    acl_sha224_init(state);
    while(*p) acl_sha256(state, *p++);
    acl_sha256_done(state);
}

static void sha256_str(bytes p) {
    acl_sha256_init(state);
    while(*p) acl_sha256(state, *p++);
    acl_sha256_done(state);
```

```
}

static void sha384_str(bytes p) {
    acl_sha384_init(state);
    while(*p) acl_sha512(state, *p++);
    acl_sha512_done(state);
}

static void sha512_str(bytes p) {
    acl_sha512_init(state);
    while(*p) acl_sha512(state, *p++);
    acl_sha512_done(state);
}

bool_t test_sha1(void) {

    put_str("\n\nsha-1");

    sha1_str(sha_test_str1);
    acl_str2hex_be(tmp, (bytes) sha1_results[0], 5);
    if(acl_cmp(state, tmp, 5)) return TRUE;

    sha1_str(sha_test_str2);
    acl_str2hex_be(tmp, (bytes) sha1_results[1], 5);
    if(acl_cmp(state, tmp, 5)) return TRUE;

    restart_timer(0);
    acl_sha1_init(state);
    for(i=0; i<1000000; i++) acl_sha1(state, 'a');
    acl_sha1_done(state);
    h = stop_timer(0); put_val("\n1000000 = ", h);
    acl_str2hex_be(tmp, (bytes) sha1_results[2], 5);
    if(acl_cmp(state, tmp, 5)) return TRUE;

    return FALSE;
}

bool_t test_sha224(void) {

    put_str("\n\nsha-224");

    sha224_str(sha_test_str1);
    acl_str2hex_be(tmp, (bytes) sha224_results[0], 7);
    if(acl_cmp(state, tmp, 7)) return TRUE;

    sha224_str(sha_test_str2);
    acl_str2hex_be(tmp, (bytes) sha224_results[1], 7);
    if(acl_cmp(state, tmp, 7)) return TRUE;

    restart_timer(0);
    acl_sha224_init(state);
    for(i=0; i<1000000; i++) acl_sha256(state, 'a');
    acl_sha256_done(state);
    h = stop_timer(0); put_val("\n1000000 = ", h);
    acl_str2hex_be(tmp, (bytes) sha224_results[2], 7);
    if(acl_cmp(state, tmp, 7)) return TRUE;

    return FALSE;
```

```
}

bool_t test_sha256(void) {

    put_str("\n\nsha-256");

    sha256_str(sha_test_str1);
    acl_str2hex_be(tmp, (bytes) sha256_results[0], 8);
    if(acl_cmp(state, tmp, 8)) return TRUE;

    sha256_str(sha_test_str2);
    acl_str2hex_be(tmp, (bytes) sha256_results[1], 8);
    if(acl_cmp(state, tmp, 8)) return TRUE;

    restart_timer(0);
    acl_sha256_init(state);
    for(i=0; i<1000000; i++) acl_sha256(state, 'a');
    acl_sha256_done(state);
    h = stop_timer(0); put_val("\n1000000 = ", h);
    acl_str2hex_be(tmp, (bytes) sha256_results[2], 8);
    if(acl_cmp(state, tmp, 8)) return TRUE;

    return FALSE;
}

bool_t test_sha384(void) {

    put_str("\n\nsha-384");

    sha384_str(sha_test_str1);
    acl_str2hex_be(tmp, (bytes) sha384_results[0], 12);
    if(acl_cmp(state, tmp, 12)) return TRUE;

    sha384_str(sha_test_str3);
    acl_str2hex_be(tmp, (bytes) sha384_results[1], 12);
    if(acl_cmp(state, tmp, 12)) return TRUE;

    restart_timer(0);
    acl_sha384_init(state);
    for(i=0; i<1000000; i++) acl_sha512(state, 'a');
    acl_sha512_done(state);
    h = stop_timer(0); put_val("\n1000000 = ", h);
    acl_str2hex_be(tmp, (bytes) sha384_results[2], 12);
    if(acl_cmp(state, tmp, 12)) return TRUE;

    return FALSE;
}

bool_t test_sha512(void) {

    put_str("\n\nsha-512");

    sha512_str(sha_test_str1);
    acl_str2hex_be(tmp, (bytes) sha512_results[0], 16);
    if(acl_cmp(state, tmp, 16)) return TRUE;

    sha512_str(sha_test_str3);
    acl_str2hex_be(tmp, (bytes) sha512_results[1], 16);
```

```
    if(acl_cmp(state, tmp, 16)) return TRUE;

    restart_timer(0);
    acl_sha512_init(state);
    for(i=0; i<1000000; i++) acl_sha512(state, 'a');
    acl_sha512_done(state);
    h = stop_timer(0); put_val("\n1000000 = ", h);
    acl_str2hex_be(tmp, (bytes) sha512_results[2], 16);
    if(acl_cmp(state, tmp, 16)) return TRUE;

    return FALSE;
}

bool_t test_sha(void) {

    if(test_sha1()) return TRUE;
    if(test_sha224()) return TRUE;
    if(test_sha256()) return TRUE;
    if(test_sha384()) return TRUE;
    if(test_sha512()) return TRUE;

    return FALSE;
}
```

Source file 130 test_rsa.c

```
// test prime generation and RSA algorithm

#include "..\acl.h"
#include "system.h"

#define NO_TESTS 100    // number of times to run the main loop
#define NO_TESTS2 1     // number of fermat tests to try for prime p
#define RM_K 8          // k-parameter for rabin miller test
#define PRNG 2          // 0: AES, 1: SHA, 2: BBS

#define LEN 8           // length of primes to generate in 32-bit words
uint p[LEN], q[LEN], r2_mod_p[LEN], r2_mod_q[LEN];
uint dmp1[LEN], dmq1[LEN], iqmp[LEN];
uint n[2*LEN], e[2*LEN], d[2*LEN];
uint dt[2*LEN], pt[2*LEN], ct[2*LEN], r_mod_m[2*LEN], r2_mod_m[2*LEN];
uint tmp[7*LEN];

bool_t test_rsa(void) {

    uint i, j, k, h, avg1, avg2, avg3, m_inv, p_inv, q_inv;
    size_t len = LEN;
    prng rnd_strong;

    put_str("\nrnd");

    // choose a strong pseudo-random number generator
#if PRNG == 0
    rnd_strong = &acl_prng_aes;
    acl_prng_aes_init(&acl_prng_lc);
#elif PRNG == 1
    rnd_strong = &acl_prng_sha;
```

```
    acl_prng_sha_init(&acl_prng_lc);
#else
    rnd_strong = &acl_prng_bbs;
    acl_prng_bbs_init(&acl_prng_lc, &acl_prng_lc, tmp);
#endif

    // rnd_strong(p, 8);      // use to measure throughput of prngs

    put_str("\nrsa");
    for(i = 0; i < NO_TESTS; i++) {
        // generate two primes p and q (with the 2 most significant bits set)
rsa_again:
        restart_timer(0);
        acl_p_rnd_prime(p, tmp, RM_K, 32*len-2, &acl_prng_lc, rnd_strong, len);
        h = stop_timer(0); put_val("\ngen = ", h);
        //put_str("\np = 0x"); put_vect(p, len);  // print it

        restart_timer(0);
        acl_p_rnd_prime(q, tmp, RM_K, 32*len-2, &acl_prng_lc, rnd_strong, len);
        h = stop_timer(0); put_val("\ngen = ", h);
        //put_str("\nq = 0x"); put_vect(q, len);  // print it

        // test fermat's little theorem
        acl_p_mont_pre(0, r2_mod_m, &m_inv, p, len); // montgomery
        for(j = 0; j < NO_TESTS2; j++) {
            acl_prng_lc(tmp, len);          // choose random number
            acl_p_mod(pt, tmp, len, p, len); // make sure it's smaller than p
            acl_p_mont_exp(ct, pt, p, len, p, tmp, m_inv, r2_mod_m, len);
                                        // raise it to the power of p
            if(acl_cmp(pt, ct, len)) {  // should be the same as before
                put_str("\np not prime"); return TRUE;
            }
        }

        // e = 65537
        acl_mov32(e, 0x10001, 2*len);       // expand e to 2*len

        // precomputation for RSA / CRT
        if(!acl_rsa_pre(n, d, dmp1, dmq1, iqmp, e, p, q, tmp, len))
            goto rsa_again;
        acl_p_mont_pre(0, r2_mod_p, &p_inv, p, len);
        acl_p_mont_pre(0, r2_mod_q, &q_inv, q, len);
        //put_str("\nn = 0x"); put_vect(n, 2*len);  // print it
        //put_str("\nd = 0x"); put_vect(d, 2*len);  // print it

        avg1 = 0; avg2 = 0; avg3 = 0;
        for(k = 0; k < 8; k++) {
            // choose plaintext
            acl_p_mont_pre(r_mod_m, r2_mod_m, &m_inv, n, 2*len); // montgomery
            acl_prng_lc(tmp, 2*len);             // choose random number
            acl_p_mod(pt, tmp, 2*len, n, 2*len); // make sure it's < n
            //put_str("\npt = 0x"); put_vect(pt, 2*len);  // print it

            // RSA encryption
            restart_timer(0);
            acl_p_mont_exp(ct, pt, e, 2*len, n, tmp, m_inv, r2_mod_m, 2*len);
                                                // encode plaintext
            h = stop_timer(0); avg1 += h; put_val("\nenc = ", h);
```

```
            //put_str("\nct = 0x"); put_vect(ct, 2*len);  // print it

            // RSA decryption (long)
            restart_timer(0);
            acl_p_mont_exp(dt, ct, d, 2*len, n, tmp, m_inv, r2_mod_m, 2*len);
            h = stop_timer(0); avg2 += h; put_val("  dec = ", h);
            //put_str("\ndt = 0x"); put_vect(dt, 2*len);  // print it

            // compare decrypted ciphertext with plaintext
            if(acl_cmp(pt, dt, 2*len)) {
                put_str("\nerror: long"); return TRUE;
            }

            // RSA decryption (CRT)
            restart_timer(0);
            acl_rsa_crt(dt, ct, p, r2_mod_p, p_inv, q, r2_mod_q, q_inv, \
                        dmp1, dmq1, iqmp, tmp, len);
            h = stop_timer(0); avg3 += h; put_val("  crt = ", h);
            //put_str("\ndt = 0x"); put_vect(dt, 2*len);  // print it

            // compare decrypted ciphertext with plaintext
            if(acl_cmp(pt, dt, 2*len)) {
                put_str("\nerror: crt"); return TRUE;
            }
        }
        put_val("\navg = ", avg1 >> 3);
        put_val("  avg = ", avg2 >> 3);
        put_val("  avg = ", avg3 >> 3);
    }
    return FALSE;
}
```

Source file 131 test_ecc.c

```
// test some ECC functions

#include "system.h"
#include "..\acl.h"

#define PART 0      // 0 - GF(p) part 1 - uVision3 won't simulate a target >16kB
                    // 1 - GF(p) part 2
                    // 2 - GF(2) curves
#define LEN 18      // the biggest curve (acl_sect571r1) needs 18 32-bit words
                    // of storage for each field element

uint tmp[(LEN+1)*10];
uint a[LEN*3];
uint b[LEN*3];
uint d[LEN*3];
uint dd[LEN+1];
uint pre[2*LEN*16];
char str[320];

// for each SECG curve, the following "ecc_comp_list" arrays contain compressed
//   base point representations
//   (taken from SEC 2: Recommended Elliptic Curve Domain Parameters)
// the library contains its own representation of the base points
```

```
// the library base points are compressed and compared to the official ones
// also, the official compressed points are decompressed
//   and compared with the library ones
// this way, both the compression and decompression routines are tested

#if PART == 0

#define CURVES 9
const ecc_t *ecc_list[] = {
                    &acl_secp112r1, &acl_secp112r2,
                    &acl_secp128r1, &acl_secp128r2,
    &acl_secp160k1, &acl_secp160r1, &acl_secp160r2,
    &acl_secp192k1, &acl_secp192r1
};

const char *ecc_comp_list[] = {
    "0209487239995A5EE76B55F9C2F098",                       // acl_secp112r1
    "034BA30AB5E892B4E1649DD0928643",                       // acl_secp112r2
    "03161FF7528B899B2D0C28607CA52C5B86",                   // acl_secp128r1
    "027B6AA5D85E572983E6FB32A7CDEBC140",                   // acl_secp128r2
    "023B4C382CE37AA192A4019E763036F4F5DD4D7EBB",           // acl_secp160k1
    "024A96B5688EF573284664698968C38BB913CBFC82",           // acl_secp160r1
    "0252DCB034293A117E1F4FF11B30F7199D3144CE6D",           // acl_secp160r2
    "03DB4FF10EC057E9AE26B07D0280B7F4341DA5D1B1EAE06C7D",   // acl_secp192k1
    "03188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012"    // acl_secp192r1
};
#elif PART == 1

#define CURVES 6
const ecc_t *ecc_list[] = {
    &acl_secp224k1, &acl_secp224r1,
    &acl_secp256k1, &acl_secp256r1,
                    &acl_secp384r1,
                    &acl_secp521r1
};

const char *ecc_comp_list[] = {
    "03A1455B334DF099DF30FC28A169A467E9E47075A90F7E650E" \
      "B6B7A45C",                                           // acl_secp224k1,
    "02B70E0CBD6BB4BF7F321390B94A03C1D356C21122343280D6" \
      "115C1D21",                                           // acl_secp224r1
    "0279BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D9" \
      "59F2815B16F81798",                                   // acl_secp256k1
    "036B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0" \
      "F4A13945D898C296",                                   // acl_secp256r1
    "03AA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B98" \
      "59F741E082542A385502F25DBF55296C3A545E3872760AB7",   // acl_secp384r1
    "0200C6858E06B70404E9CD9E3ECB662395B4429C648139053F" \
      "B521F828AF606B4D3DBAA14B5E77EFE75928FE1DC127A2FF" \
      "A8DE3348B3C1856A429BF97E7E31C2E5BD66"                // acl_secp521r1
};
#else

#define CURVES 18
const ecc_t *ecc_list[] = {
                    &acl_sect113r1, &acl_sect113r2,
                    &acl_sect131r1, &acl_sect131r2,
    &acl_sect163k1, &acl_sect163r1, &acl_sect163r2,
```

```
                    &acl_sect193r1, &acl_sect193r2,
    &acl_sect233k1, &acl_sect233r1,
    &acl_sect239k1,
    &acl_sect283k1, &acl_sect283r1,
    &acl_sect409k1, &acl_sect409r1,
    &acl_sect571k1, &acl_sect571r1
};

const char *ecc_comp_list[] = {
    "03009D73616F35F4AB1407D73562C10F",                     // acl_sect113r1
    "0301A57A6A7B26CA5EF52FCDB8164797",                     // acl_sect113r2
    "030081BAF91FDF9833C40F9C181343638399",                 // acl_sect131r1
    "030356DCD8F2F95031AD652D23951BB366A8",                 // acl_sect131r2
    "0302FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8",         // acl_sect163k1
    "030369979697AB43897789566789567F787A7876A654",         // acl_sect163r1
    "0303F0EBA16286A2D57EA0991168D4994637E8343E36",         // acl_sect163r2
    "0301F481BC5F0FF84A74AD6CDF6FDEF4BF6179625372D8C0C5E1", // acl_sect193r1
    "0300D9B67D192E0367C803F39E1A7E82CA14A651350AAE617E8F", // acl_sect193r2
    "02017232BA853A7E731AF129F22FF4149563A419C26BF50A4C" \
    "9D6EEFAD6126",                                         // acl_sect233k1
    "0300FAC9DFCBAC8313BB2139F1BB755FEF65BC391F8B36F8F8" \
    "EB7371FD558B",                                         // acl_sect233r1
    "0329A0B6A887A983E9730988A68727A8B2D126C44CC2CC7B2A" \
    "6555193035DC",                                         // acl_sect239k1
    "020503213F78CA44883F1A3B8162F188E553CD265F23C1567A" \
    "16876913B0C2AC2458492836",                             //acl_sect283k1
    "0305F939258DB7DD90E1934F8C70B0DFEC2EED25B8557EAC9C" \
    "80E2E198F8CDBECD86B12053",                             //acl_sect283r1
    "030060F05F658F49C1AD3AB1890F7184210EFD0987E307C84C" \
    "27ACCFB8F9F67CC2C460189EB5AAAA62EE222EB1B35540CFE9" \
    "023746",                                               //acl_sect409k1
    "03015D4860D088DDB3496B0C6064756260441CDE4AF1771D4D" \
    "B01FFE5B34E59703DC255A868A1180515603AEAB60794E54BB" \
    "7996A7",                                               //acl_sect409r1
    "02026EB7A859923FBC82189631F8103FE4AC9CA2970012D5D4" \
    "6024804801841CA44370958493B205E647DA304DB4CEB08CBB" \
    "D1BA39494776FB988B47174DCA88C7E2945283A01C8972",       //acl_sect571k1
    "030303001D34B856296C16C0D40D3CD7750A93D1D2955FA80A" \
    "A5F40FC8DB7B2ABDBDE53950F4C0D293CDD711A35B67FB1499" \
    "AE60038614F1394ABFA3B4C850D927E1E7769C8EEC2D19"        //acl_sect571r1
};
#endif

static bool_t str_cmp(char *str1, char *str2) {
    while((*str1) && (*str2)) if(*str1++ != *str2++) return TRUE;
    return FALSE;
}

bool_t test_ecc(void) {

    ecc_t *c; int i, j, k; //uint h, avg1, avg2, avg3;

    for(j=0; j<100; j++) {
        for(i=0; i<CURVES; i++) {
            c = (ecc_t *) ecc_list[i];

            // print name of curve
            put_str("\n\n");
```

```
            put_str((char *) c->s);

#if 1       // test basic ecc operations

            // make sure that the base point lies on the curve
            if(!acl_ecc_chk(c->g, tmp, c)) {
                put_str(" chk"); return TRUE;
            }

            // generate random point
            acl_prng_lc(tmp, c->ln);
            acl_p_mod(dd, tmp, c->ln, c->n, c->ln);
            acl_ecc_mul(d, c->g, 0, 1, 0, dd, 0, c->ln, tmp, c);

            // basic operations
            acl_mov32(a + 2*c->l, 0, c->l);         // a = point at infinity
            for(k=0; k<8; k++) acl_ecc_add(a, d, tmp, c);
            acl_ecc_aff(a, tmp, c);                 // a = a+d+d+...+d = 8d

            acl_ecc_pro(b, d, c->l);                // b = d
            for(k=0; k<3; k++) acl_ecc_dbl(b, tmp, c);
            acl_ecc_aff(b, tmp, c);                 // b = 2*2*2*b = 8d

            if(acl_cmp(a, b, 2*c->l)) {             // is a == b ?
                put_str(" add/mul"); return TRUE;
            }

            // point to string with compression (base point)
            acl_ecc_p2str(str, c->g, 1, tmp, c);
            put_str("\nG = "); put_str(str);
            if(str_cmp(str, (char *) ecc_comp_list[i])) {
                put_str(" p2str w/ comp"); return TRUE;
            }

            // string to point with compression (base point)
            if(!acl_ecc_str2p(a, str, tmp, c)) put_str(" invalid");
            if(acl_cmp(a, c->g, 2*c->l)) {
                put_str(" str2p w/ decomp"); return TRUE;
            }

            // point to string conversion without compression (base point)
            acl_ecc_p2str(str, c->g, 0, tmp, c);
            put_str("\nG = "); put_str(str);

            // string to point conversion without compression (base point)
            if(!acl_ecc_str2p(a, str, tmp, c)) put_str(" invalid");
            if(acl_cmp(a, c->g, 2*c->l)) {
                put_str(" str2p w/o comp"); return TRUE;
            }

            // point to string with compression (random point)
            acl_ecc_p2str(str, d, 1, tmp, c);
            put_str("\nD = "); put_str(str);

            // string to point with compression (random point)
            if(!acl_ecc_str2p(a, str, tmp, c)) put_str(" invalid");
            if(acl_cmp(a, d, 2*c->l)) {
                put_str(" str2p w/ comp"); return TRUE;
```

```
            }

            // point to string conversion without compression (random point)
            acl_ecc_p2str(str, d, 0, tmp, c);
            put_str("\nD = "); put_str(str);

            // string to point conversion without compression (random point)
            if(!acl_ecc_str2p(a, str, tmp, c)) put_str(" invalid");
            if(acl_cmp(a, d, 2*c->l)) {
                put_str(" str2p w/o comp"); return TRUE;
            }

#else   // this code was used to generate a table of field operation timings
        // (multiplication, fast reduction, inversion)

            //acl_mov32(dd, 1, c->l);      // recover m from fr
            //k = 0;
            //while(c->fr[k]) { acl_bit_set(dd, c->fr[k]); k++; }

            avg1 = 0; avg2 = 0; avg3 = 0;
            for(k = 0; k < 16; k++) {
                acl_prng_lc(a, c->l);
                acl_prng_lc(b, c->l);
                acl_prng_lc(d, c->l);

                restart_timer(0);
                acl_p_mul(tmp, a, b, c->l);
                //acl_2_mul(tmp, a, b, c->l);
                h = stop_timer(0); avg1 += h; //put_val("\nm=", h);

                restart_timer(0);
                acl_p_fr(b, tmp, c->fr, c->l);
                //acl_2_fr(b, tmp, c->fr, c->l);
                h = stop_timer(0); avg2 += h; //put_val(" f=", h);

                restart_timer(0);
                acl_p_mod_inv(b, a, 0, c->m, tmp, c->l);
                //acl_2_mod_inv(b, a, dd, tmp, c->l);
                h = stop_timer(0); avg3 += h; //put_val(" i=", h);
            }
            put_val("\nm=", avg1 >> 4);
            put_val(" f=", avg2 >> 4);
            put_val(" i=", avg3 >> 4);
#endif
        }
    }
    return FALSE;
}
```

Source file 132 test_ecdsa.c

```
// test ECDSA operation

#include "system.h"
#include "..\acl.h"

#define PART 0      // 0-5 (uVision3 won't simulate a target > 16kB)
```

```
#define LEN 18      // the biggest curve (acl_sect571r1) needs 18 32-bit words
                    // of storage for each field element
#define WIDTH 4     // width of comb used with pre-computation
                    // memory required grows exponentially:
                    // 8*LEN*((2^WIDTH)-1) bytes for each ECC point
                    // 1: no pre-computation
                    // 2 - 5: realistic values
                    // 6: requires 2kB for smallest, 9kB for biggest curve

uint tmp[(LEN+1)*10];
uint a[LEN*3];
uint pre1[2*LEN*((1<<WIDTH)-1)];
uint pre2[2*LEN*((1<<WIDTH)-1)];
uint hash[5] = { 0x12452643, 0xabcda431, 0xff509ac8, 0xb909cd90, 0x5329cb0a };
uint r[LEN+1];
uint s[LEN+1];
uint dA[LEN+1];
uint qA[LEN*3];

#if PART == 0
    #define CURVES 5
    const ecc_t *ecc_list[] = {
                        &acl_secp112r1, &acl_secp112r2,
                        &acl_secp128r1, &acl_secp128r2,
        &acl_secp160k1,
    };
#elif PART == 1
    #define CURVES 5
    const ecc_t *ecc_list[] = {
                        &acl_secp160r1, &acl_secp160r2,
        &acl_secp192k1, &acl_secp192r1,
        &acl_secp224k1
    };
#elif PART == 2
    #define CURVES 3
    const ecc_t *ecc_list[] = {
                        &acl_secp224r1,
        &acl_secp256k1, &acl_secp256r1,
    };
#elif PART == 3
    #define CURVES 2
    const ecc_t *ecc_list[] = {
                        &acl_secp384r1,
                        &acl_secp521r1
    };
#elif PART == 4
    #define CURVES 11
    const ecc_t *ecc_list[] = {
                        &acl_sect113r1, &acl_sect113r2,
                        &acl_sect131r1, &acl_sect131r2,
        &acl_sect163k1, &acl_sect163r1, &acl_sect163r2,
                        &acl_sect193r1, &acl_sect193r2,
        &acl_sect233k1, &acl_sect233r1
    };
#elif PART == 5
    #define CURVES 7
    const ecc_t *ecc_list[] = {
        &acl_sect239k1,
```

```
        &acl_sect283k1, &acl_sect283r1,
        &acl_sect409k1, &acl_sect409r1,
        &acl_sect571k1, &acl_sect571r1
    };
#endif

bool_t test_ecdsa(void) {

    ecc_t *c; int i, j, k, wi, sp; bool_t res; uint h, avg1, avg2, avg3, avg4;

    for(j=0; j<100; j++) {
        for(i=0; i<CURVES; i++) {
            c = (ecc_t *) ecc_list[i];

            // print name of curve
            put_str("\n\n");
            put_str((char *) c->s);

            // width and spacing of comb
            wi = WIDTH; // number of teeth of the comb
            sp = 1;  // spacing = how many bits apart the teeth of the comb are
            while(sp * wi < 32 * c->ln) sp++;  // sp >= 32*len / width  !!!

            // generate private key for ecdsa
            acl_prng_lc(tmp, c->ln);
            acl_p_mod(dA, tmp, c->ln, c->n, c->ln);

            // generate public key for ecdsa
            acl_ecc_mul(qA, c->g, 0, 1, 0, dA, 0, c->ln, tmp, c);

            // generate pre-computation for base point
            acl_ecc_pre(pre1, c->g, wi, sp, tmp, c);

            // generate pre-computation for qA
            acl_ecc_pre(pre2, qA, wi, sp, tmp, c);

            avg1 = 0; avg2 = 0; avg3 = 0; avg4 = 0;
            for(k = 0; k < 4; k++) {
                // generate ecdsa signature without pre-computation
                restart_timer(0);
                acl_ecdsa_gen(r, s, hash, 5, dA, pre1, 1, 0, \
                              &acl_prng_lc, tmp, c);
                h = stop_timer(0); avg1 += h; put_val("\ngen =", h);

                // verify ecdsa signature without pre-computation
                restart_timer(0);
                res = acl_ecdsa_ver(r, s, hash, 5, pre2, pre1, 1, 0, tmp, c);
                h = stop_timer(0); avg2 += h; put_val(" ver =", h);
                if(!res) { put_str(" ecdsa"); return TRUE; }

                // generate ecdsa signature with pre-computation
                restart_timer(0);
                acl_ecdsa_gen(r, s, hash, 5, dA, pre1, wi, sp, \
                              &acl_prng_lc, tmp, c);
                h = stop_timer(0); avg3 += h; put_val(" gen pre =", h);

                // verify ecdsa signature with pre-computation
                restart_timer(0);
```

```
                res = acl_ecdsa_ver(r, s, hash, 5, pre2, pre1, wi, sp, tmp, c);
                h = stop_timer(0); avg4 += h; put_val(" ver pre =", h);
                if(!res) { put_str(" ecdsa"); return TRUE; }
            }
            put_str("\naverages\n");
            put_val("gen =", avg1 >> 2);
            put_val(" ver =", avg2 >> 2);
            put_val(" gen pre =", avg3 >> 2);
            put_val(" ver pre =", avg4 >> 2);
        }
    }
    return FALSE;
}
```

Source file 133   makefile

```
# makefile for ARM cryptographic library
# has dependency tracking, but uses perl

.LIBPATTERNS =

AR = arm-none-eabi-ar
AS = arm-none-eabi-as
CC = arm-none-eabi-gcc
RANLIB = arm-none-eabi-ranlib

CCFLAGS = -c -mcpu=arm7tdmi -mthumb -MD -Wall -Os -mapcs-frame \
    -mthumb-interwork -Wa,-alhms=Lst/$*.lst -o Obj/$*.o
ASFLAGS = -mcpu=arm7tdmi -mthumb-interwork --MD Obj/$*.d -alhms=Lst/$*.lst \
    -o Obj/$*.o

VPATH = Obj:AES:Common:Curves:ECC:GF_2:GF_p:Primes:PRNG:RSA:SHA

AES = acl_aes_cbc_de.s acl_aes_cbc_en.s acl_aes_cntr.s acl_aes_de.s \
    acl_aes_ecb_de.s acl_aes_ecb_en.s acl_aes_en.s acl_aes_key_de.s \
    acl_aes_key_en.s acl_aes_tables.s
SHA = acl_sha1.s acl_sha256.s acl_sha512.s
COMMON = acl_bit.s acl_bit_clr.s acl_bit_set.s acl_cmp.s acl_ctz.s \
    acl_hex2str_dec.s acl_hex2str_le.s acl_log2.s acl_mov.s acl_mov32.s \
    acl_rev.s acl_rsh.s acl_str2bytes.s acl_str2hex_be.s acl_str2hex_le.s \
    acl_xor.s acl_xor32.s acl_zero.s
GF_P = acl_p_coprime.s acl_p_div.c acl_p_fr.s acl_p_mod.c acl_p_mod_add.s \
    acl_p_mod_dbl.s acl_p_mod_hlv.s acl_p_mod_inv.c acl_p_mod_sub.s \
    acl_p_mont_exp.c acl_p_mont_inv.s acl_p_mont_m_inv.s acl_p_mont_pre.c \
    acl_p_mont_red.s acl_p_mul.s acl_p_sqr.s acl_p_sqrt.c
PRIMES = acl_p_rm_test.c acl_p_rm_test2.c acl_p_rnd_prime.c acl_p_tables.s
PRNG = acl_prng_lc_.s acl_prng_lc.c acl_prng_aes.c acl_prng_sha.c acl_prng_bbs.c
RSA = acl_rsa_pre.c acl_rsa_crt.c
GF_2 = acl_2_fr.s acl_2_mod_hlv.s acl_2_mod_inv.c acl_2_mont_inv.s \
    acl_2_mul.s acl_2_sqr.s
CURVES =                acl_secp112r1.c     acl_secp112r2.c \
                        acl_secp128r1.c     acl_secp128r2.c \
    acl_secp160k1.c     acl_secp160r1.c     acl_secp160r2.c \
    acl_secp192k1.c     acl_secp192r1.c                     \
    acl_secp224k1.c     acl_secp224r1.c                     \
    acl_secp256k1.c     acl_secp256r1.c                     \
                        acl_secp384r1.c                     \
```

```
                    acl_secp521r1.c                     \
    \
                    acl_sect113r1.c     acl_sect113r2.c \
                    acl_sect131r1.c     acl_sect131r2.c \
    acl_sect163k1.c     acl_sect163r1.c     acl_sect163r2.c \
                    acl_sect193r1.c     acl_sect193r2.c \
    acl_sect233k1.c     acl_sect233r1.c                     \
    acl_sect239k1.c                                         \
    acl_sect283k1.c     acl_sect283r1.c                     \
    acl_sect409k1.c     acl_sect409r1.c                     \
    acl_sect571k1.c     acl_sect571r1.c
ECC = acl_p_ecc_add.c acl_p_ecc_aff.c acl_p_ecc_chk.c acl_p_ecc_dbl.c \
    acl_p_ecc_func.c acl_p_ecc_p2str.c acl_p_ecc_str2p.c acl_2_ecc_add.c \
    acl_2_ecc_aff.c acl_2_ecc_chk.c acl_2_ecc_dbl.c acl_2_ecc_func.c \
    acl_2_ecc_p2str.c acl_2_ecc_str2p.c acl_ecc_mul.c acl_ecc_pre.c \
    acl_ecc_pro.c
ECDSA = acl_ecdsa_gen.c acl_ecdsa_ver.c
SRC := $(AES) $(SHA) $(COMMON) $(GF_P) $(PRIMES) $(PRNG) $(RSA) $(GF_2) \
    $(CURVES) $(ECC) $(ECDSA)
OBJ := $(subst .c,.o,$(SRC))
OBJ := $(subst .s,.o,$(OBJ))
OBJ_PRE := $(addprefix Obj/,$(OBJ))
DEPS := $(subst .o,.d,$(OBJ))
PERL = perl -p -e "s{[^\.]+\.o[ :]*}{$*\.o $*\.d : }g;s{/}{\\}g;"

.PHONY: all
all: acl.a

acl.a: $(OBJ)
	$(AR) rvu $@ $(OBJ_PRE)
	$(RANLIB) $@

include $(DEPS)

%.d : %.c
	-del Obj\$@
	$(CC) -c -M $< > Obj\$@.orig
	$(PERL) < Obj\$@.orig > Obj\$@
	-del Obj\$@.orig

%.d : %.s
	-del Obj\$@
	$(AS) $(ASFLAGS) $<
	copy Obj\$@ Obj\$@.orig
	$(PERL) < Obj\$@.orig > Obj\$@
	-del Obj\$@.orig

%.o : %.c
	$(CC) $(CCFLAGS) $<
	copy Obj\$*.d Obj\$*.d.orig
	$(PERL) < Obj\$*.d.orig > Obj\$*.d
	-del Obj\$*.d.orig

%.o : %.s
	$(AS) $(ASFLAGS) $<
	copy Obj\$*.d Obj\$*.d.orig
	$(PERL) < Obj\$*.d.orig > Obj\$*.d
	-del Obj\$*.d.orig
```

Table of contents for Appendix B

Source file 1	acl.h	75
Source file 2	acl_config.h	81
Source file 3	acl_int.h	81
Source file 4	acl_gen_tabs.m	82
Source file 5	acl_aes_tables.s	83
Source file 6	acl_aes_key_en.s	83
Source file 7	acl_aes_key_de.s	85
Source file 8	acl_aes_en.s	86
Source file 9	acl_aes_de.s	89
Source file 10	acl_aes_ecb_en.s	92
Source file 11	acl_aes_ecb_de.s	93
Source file 12	acl_aes_cbc_en.s	93
Source file 13	acl_aes_cbc_de.s	94
Source file 14	acl_aes_cntr.s	95
Source file 15	acl_sha1.s	96
Source file 16	acl_sha256.s	101
Source file 17	acl_sha512.s	105
Source file 18	acl_mov.s	112
Source file 19	acl_mov32.s	112
Source file 20	acl_bit.s	113
Source file 21	acl_bit_clr.s	113
Source file 22	acl_bit_set.s	114
Source file 23	acl_cmp.s	114
Source file 24	acl_zero.s	115
Source file 25	acl_xor.s	115
Source file 26	acl_xor32.s	116
Source file 27	acl_log2.s	117
Source file 28	acl_ctz.s	117
Source file 29	acl_rev.s	118
Source file 30	acl_rsh.s	118
Source file 31	acl_hex2str_dec.s	119
Source file 32	acl_hex2str_le.s	121
Source file 33	acl_str2bytes.s	122
Source file 34	acl_str2hex_be.s	123

Source file 35 acl_str2hex_le.s ... 124
Source file 36 acl_p_mod_add.s ... 125
Source file 37 acl_p_mod_dbl.s ... 127
Source file 38 acl_p_mod_sub.s ... 128
Source file 39 acl_p_mod_hlv.s ... 129
Source file 40 acl_p_mul.s ... 130
Source file 41 acl_p_sqr.s ... 132
Source file 42 acl_p_mont_red.s ... 135
Source file 43 acl_p_mont_m_inv.s ... 138
Source file 44 acl_p_mont_pre.c ... 139
Source file 45 acl_p_mont_exp.c ... 139
Source file 46 acl_p_coprime.s ... 140
Source file 47 acl_p_mont_inv.s ... 142
Source file 48 acl_p_mod_inv.c ... 145
Source file 49 acl_p_mod.c ... 146
Source file 50 acl_p_div.c ... 146
Source file 51 acl_p_sqrt.c ... 147
Source file 52 acl_p_fr.s ... 148
Source file 53 acl_prng_lc.s ... 154
Source file 54 acl_prng_aes.c ... 155
Source file 55 acl_prng_sha.c ... 156
Source file 56 acl_prng_bbs.c ... 156
Source file 57 gen_primes.txt ... 157
Source file 58 acl_p_tables.s ... 158
Source file 59 acl_p_rm_test2.c ... 158
Source file 60 acl_p_rm_test.c ... 159
Source file 61 acl_p_rnd_prime.c ... 160
Source file 62 acl_rsa_pre.c ... 161
Source file 63 acl_rsa_crt.c ... 161
Source file 64 acl_2_mod_hlv.s ... 162
Source file 65 acl_2_mul.s ... 163
Source file 66 acl_2_sqr.s ... 166
Source file 67 acl_2_mont_inv.s ... 167
Source file 68 acl_2_mod_inv.c ... 170
Source file 69 acl_2_fr.s ... 170
Source file 70 acl_secp112r1.c ... 173

Source file 71 acl_secp112r2.c ... 173
Source file 72 acl_secp128r1.c ... 174
Source file 73 acl_secp128r2.c ... 175
Source file 74 acl_secp160k1.c ... 176
Source file 75 acl_secp160r1.c ... 176
Source file 76 acl_secp160r2.c ... 177
Source file 77 acl_secp192k1.c ... 178
Source file 78 acl_secp192r1.c ... 178
Source file 79 acl_secp224k1.c ... 179
Source file 80 acl_secp224r1.c ... 180
Source file 81 acl_secp256k1.c ... 180
Source file 82 acl_secp256r1.c ... 181
Source file 83 acl_secp384r1.c ... 182
Source file 84 acl_secp521r1.c ... 182
Source file 85 acl_sect113r1.c ... 183
Source file 86 acl_sect113r2.c ... 184
Source file 87 acl_sect131r1.c ... 185
Source file 88 acl_sect131r2.c ... 185
Source file 89 acl_sect163k1.c ... 186
Source file 90 acl_sect163r1.c ... 187
Source file 91 acl_sect163r2.c ... 187
Source file 92 acl_sect193r1.c ... 188
Source file 93 acl_sect193r2.c ... 189
Source file 94 acl_sect233k1.c ... 189
Source file 95 acl_sect233r1.c ... 190
Source file 96 acl_sect239k1.c ... 191
Source file 97 acl_sect283k1.c ... 191
Source file 98 acl_sect283r1.c ... 192
Source file 99 acl_sect409k1.c ... 192
Source file 100 acl_sect409r1.c ... 193
Source file 101 acl_sect571k1.c ... 194
Source file 102 acl_sect571r1.c ... 194
Source file 103 acl_p_ecc_chk.c ... 195
Source file 104 acl_p_ecc_dbl.c ... 196
Source file 105 acl_p_ecc_add.c ... 197
Source file 106 acl_p_ecc_aff.c ... 198

Source file 107 acl_p_ecc_p2str.c ... 199
Source file 108 acl_p_ecc_str2p.c ... 199
Source file 109 acl_p_ecc_func.c ... 200
Source file 110 acl_2_ecc_chk.c ... 201
Source file 111 acl_2_ecc_dbl.c ... 201
Source file 112 acl_2_ecc_add.c ... 202
Source file 113 acl_2_ecc_aff.c ... 203
Source file 114 acl_2_ecc_p2str.c ... 204
Source file 115 acl_2_ecc_str2p.c ... 205
Source file 116 acl_2_ecc_func.c ... 207
Source file 117 acl_ecc_pro.c ... 207
Source file 118 acl_ecc_neg.c ... 207
Source file 119 acl_ecc_pre.c ... 207
Source file 120 acl_ecc_mul.c ... 208
Source file 121 acl_ecdsa_gen.c ... 209
Source file 122 acl_ecdsa_ver.c ... 210
Source file 123 magma.txt ... 211
Source file 124 system.h ... 211
Source file 125 timing.c ... 212
Source file 126 serial.c ... 213
Source file 127 main.c ... 214
Source file 128 test_aes.c ... 215
Source file 129 test_sha.c ... 220
Source file 130 test_rsa.c ... 224
Source file 131 test_ecc.c ... 226
Source file 132 test_ecdsa.c ... 230
Source file 133 makefile ... 233