



VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Rafiat Sanni

ARM BASED UART DATA
TRANSMISSION WITH ASYMMETRIC
KEY ENCRYPTION USING RSA
ALGORITHM

Technology and Communications
2011

TIIVISTELMÄ

Tekijä	Rafiat Sanni
Opinnäytetyön nimi	ARM based UART Data Transmission with Asymmetric Key Encryption Using RSA Algorithm.
Vuosi	2011
Kieli	English
Sivumäärä	63 + 18 liitettä
Ohjaaja	Yang Liu

Tietoturva on hyvin olennainen osa tämän päivän maailmaa, ja siksi eri alojen ammattilaiset ovat tehneet valtavan määrän työtä ja tutkimusta tiedon turvassa pysymisen varmistamiseksi. Tämän projekti pohjautuu yksinomaan siihen. Myös sellaisen teknologian laajentuminen kattamaan muita laitteita, jotka saattavat olla tekemisissä datan lähettämisen kanssa, oli toinen merkittävä tekijän tällaisen päätöksen tekemisessä.

Tässä lopputyössä pyrittiin toteuttamaan tietoturvaratkaisu mikro-ohjainlaitteella (ARM). Tämä toteutettiin käyttämällä RS-232-standardiin perustuvaa sarjayhteyttä. UART:ta käytettiin datan siirtämiseen. UART, joka on asynkroninen lähetystapa, pohjautuu säädettävään datasiirtonopeuteen ja dataformaattiin. Myös pääte-emulaattoria käytettiin testaamiseen, ja se auttaa mikrokontrollerin sarjamuotoisen datan seuraamista, ja myös sen lähettämistä mikro-ohjaimelle testitarkoituksessa.

Tiedon salaamista käytettiin keinona sen suojaamiseen. Datakryptografia jakautuu kahteen kategoriaan, jotka ovat symmetrisen ja asymmetrisen avaimen suojaus. RSA-algoritmia, joka kuuluu asymmetristen avainten ryhmään, käytettiin tiedon salaamiseen ja purkamiseen. Se käyttää kahta avainparia, yhtä julkista ja yhtä yksityistä. 512 bitin avainpituutta käytettiin avainpareihin. Vaikkei se olekaan paras tämänhetkisistä standardeista, se on edelleen toimiva siinä käytössä, johon sitä tässä projektissa tarvittiin. Avainten pituudet tulisi pääsääntöisesti valita salattavan tiedon tyyppin perusteella.

Tiedon salaaminen ja sitä seuraava salauksen purkaminen osoittautuivat toimiviksi UC:lla, jota käytettiin työn alustana. Onnistunut [implementaatio] osoittaa, että tämän tietojensuojausmenetelmän siirtäminen muille alustoille on mahdollista. Toivon voivani esitellä sellaisen algoritmin mobiililaitteille tärkeiksi luokiteltujen viestien lähettämiseen ja vastaanottamiseen.

ABSTRACT

Author	Rafiat Sanni
Title	ARM based UART Data Transmission with Asymmetric Key Encryption Using RSA Algorithm.
Year	2011
Language	English
Pages	63 + 18 Appendices
Name of Supervisor	Liu Yang

Data security is something essential in the world today and as such, there has been a tremendous amount of research and work carried out by people from various professional realms so as to ensure data is always secure. The basis of this project work is solely based on this reason. Also, the expansion of such technology so as to encompass other devices that may deal with data transmission played another part for making such a decision.

For the purpose of this thesis, an attempt to implement data security on a microcontroller device (ARM) was carried out. This was conducted using a serial connection based on RS-232 standard with the use of UART for the transmission of such data. UART, which is an asynchronous means of transmission, is based on adjustable data transmission speed and data format. A terminal emulator was also used for testing purposes and it helps to view serial data from the microcontroller and also helps in transmitting to the microcontroller for testing purposes.

Data encryption was employed as a means of securing data. Data cryptography is divided into two categories which includes symmetric and asymmetric key cryptography. RSA algorithm was used for data encryption and decryption and it falls under the asymmetric key group. It works by using two pairs of keys, one pair public and the other pair private. 512-bits key length was used for the key pairs, while not being the best standard currently; it still serves the purpose it was intended for in this project. Key lengths should be chosen based on the type of data being secured as a general rule.

The encryptions and subsequent decryption of data turned out to be successful on the microcontroller which was the platform of implementation. The successful implementation shows that it is possible to port this method of data security to other platforms. In the future, I hope to be able to introduce such algorithm to mobile devices in send and receiving messages deemed important.

ACKNOWLEDGEMENT

To God, who makes going on easier to bear. To my mother from whom I learnt hard work never hurt anyone and for her endless support for all my decisions. My gratitude goes to my father for trusting me with the responsibility of making my decisions early in life. To all my siblings, for encouraging me to forge ahead when things were not so easy. To my good friends, though not many, but more than a million friends anyone could have in life; thanks for the tremendous show of trust and support.

To Liu Yang, my supervisor, instructor and to whom I personally refer to as my greatest challenger; this thesis work would have been incomplete but for your continuous support and challenge. To Johan Dams for making everything seems cool. To Seppo Mäkinen for showing me physics really can be fun! To Chao Gao for being an exceptional instructor. Thank you all for making my time at VAMK a great one.

CONTENTS

TIIVISTELMÄ

ABSTRACT

1	INTRODUCTION	8
1.1	Thesis objective	10
1.2	Introduction of the Hardware	10
1.2.1	Overview of the MCU	11
1.3	Introduction of the Software	12
1.3.1	WinARM	12
1.3.2	Bray's Terminal	13
2	CRYPTOGRAPHY	13
2.1	Components of cryptography	14
2.1.1	Plaintext and Ciphertext /1/	14
2.1.2	Key	15
2.1.3	Alice, Bob and Eve	15
2.2	Categories of cryptography	15
2.2.1	Symmetric key (secret-key) cryptography	15
2.2.2	Asymmetric key (public-key) cryptography	17
2.3	Types of keys	17
3	CRC ALGORITHM AND IMPLEMENTATION	18
3.1	Redundancy	19
3.2	Polynomials	19
3.2.1	Addition and subtraction of polynomials	20
3.2.2	Multiplying polynomials	20
3.2.3	Dividing polynomials	20
3.3	Cyclic code	21
3.4	Algorithm and implementation	22
3.4.1	Implementation	24
4	RIVEST, SHAMIR, ADLEMAN (RSA) ALGORITHM	26
4.1	Key generation	27

4.2	Encryption.....	28
4.3	Decryption.....	28
4.4	Example using RSA algorithm /1/	30
4.5	Security of RSA	30
4.6	RSA encryption implementation notes	32
5	RSA ALGORITHM ON ARM	33
5.1	ARM microcontroller hardware setup	33
5.1.1	Software setup for microcontroller programming.....	33
5.1.2	Start up function for the microcontroller	35
5.2	UART configuration function.....	37
5.2.1	UART0 Transmission and Receiving functions	38
5.3	RSA implementation.....	39
5.3.1	Multiple-precision integer arithmetic.....	41
5.3.2	Left-to-right binary exponentiation algorithm	45
5.4	Compilation and code download	47
6	TESTING FOR RSA AND PROBLEMS	50
6.1	Testing.....	50
6.2	Problems	54
7	CONCLUSION AND SUGGESTIONS	56
7.1	Conclusion for RSA.....	56
7.2	Conclusion for CRC.....	57
7.3	Suggestions and future developments.....	58
	REFERENCES.....	59
	APPENDIX.....	ERROR! BOOKMARK NOT DEFINED.
	APPENDICES	

LIST OF FIGURES AND TABLES

Figure 1.	Image of OLIMEX MCU LPC–H2129 /3/	14
Figure 2.	An overview of cryptography /1/	17
Figure 3.	Symmetric key cryptography /1/	19
Figure 4.	Asymmetric key cryptography/1/	20
Figure 5.	Input data request before CRC calculation.	28
Figure 6.	CRC value returned for input data	29
Figure 7.	An image of programmer’s notepad	38
Figure 8.	MCU setup code snippet	40
Figure 9.	UART0 setup function	41
Figure 10.	UART0 receiving functions	42
Figure 11.	UART0 transmitting functions	43
Figure 12.	Calling encryption and decryption sequence	52
Figure 13.	Programming the MCU	52
Figure 14.	BrayTerm with data transfer settings	55
Figure 15.	BrayTerm connected to the MCU	56
Figure 16.	Code running on the MCU	57
Figure 17.	Ciphertext for ‘hello’ after encryption	57
Figure 18.	Decrypted ciphertext for ‘hello’	58

Table 1.	Public-key encryption schemes and the related computational problems upon which their security is based/5/.	30
Table 2.	Function names and their functionalities.	50

LIST OF APPENDICES

APPENDIX 1. References

APPENDIX 2. Source codes

ABBREVIATIONS

MCU	Microcontroller Unit
UART	Universal Asynchronous Receiver/Transmitter
RS-232	Recommended Standard 232
PC	Personal Computer
USB	Universal Serial Bus
RAM	Random Access Memory
RTC	Real-Time Clock
ADC	Analogue to Digital Converter
CAN	Controller Area Network
I ² C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
CCR	Condition Code Register (Status Register)
PWM	Pulse Width Modulation
WDT	Watch Dog Timer
I/O	Input/Output
RSA	Rivest, Shamir, Adleman
LED	Light Emitting Diode
PLL	Phase Locked Loop
CRC	Cyclic Redundancy Check

1 INTRODUCTION

The world today can be described as modernized, yet there are still elements which remind us that modernization is based solely on the development of old

methods and materials which in essence has given way to the creation of products which either work based on the improvement of old methods and devices or the introduction, due to extensive research based on old products, of a new method or device. This project shows how this has been manifested in the use of an algorithm which was introduced decades ago to manage data that become ever more important as time passes by.

Cryptography, a word with Greek origins, means “secret writing” /1/. It is a way by which messages are hidden from third parties or unintended recipients in order to guarantee its authenticity. There are several means by which this can be carried out but this project focuses mainly on using asymmetric key cryptography to carry this out. Cryptography is achieved through a set of mathematical calculations which has been used to develop several algorithms leading to different methods of encryption.

One of such methods is the RSA algorithm developed, and named after, the inventors of public key cryptography: Ron Rivest, Adi Shamir and Leonard Adleman /2/. Their innovation solved a daunting challenge in network security: how to enable secure yet transparent exchange of encrypted communications between users and enterprises that are strangers to each other /2/. RSA algorithm was used as the method of achieving data encryption for the purpose of this project.

Data error correction is also another important part of data transmission. This takes into account the problems that could occur as a result of loss or the change of data that occurs due to the presence of noise on the line being used to transmit such data. This could also give false positives to encrypted data. This means that the data could be considered false and thereby discarded as a result of problems of noise present on the line being used for sending and receiving data. Cyclic redundancy check (CRC) was introduced for data error checking for the purpose of this project.

1.1 Thesis objective

The purpose of the project is to implement a method of data security known as asymmetric key cryptography based on the RSA algorithm method of encryption. The initial objective of the project was to implement the algorithm on two MCUs but because of insufficient availability of resources, there was only one MCU available. Therefore, in the absence of the second MCU, a terminal emulator, running on a PC, was used for checking the results of both encryption and decryption which ended up being implemented on only one MCU. The mode of connection of the MCU was based on a serial mode of communication (RS-232) between the MCU and the PC with the use of the UART port of the MCU. The system works by connecting the MCU and PC using an RS-232 cable via a UART port on the MCU. A serial-to-USB adapter was used in order to enable the connection to the PC because most PCs hardly ever come manufactured with a serial port any longer. The MCU takes in an input from the terminal emulator and encrypts the data input based on the RSA algorithm and then transmits the encrypted text to through the UART back to the terminal which in this case also serves as the second MCU. The encrypted text then gets decrypted by the MCU and then displays the encrypted text in plaintext format.

Data error detection was also considered at the start of the project. The method considered falls under the category of block coding schemes. There are two schemes and the other scheme is known as convolution coding scheme. Block codes are further divided into two categories and this includes, linear block coding and cyclic block coding. The method of cyclic coding used in this project was CRC. This method of data error check, and its implementation will be further detailed later in this report.

1.2 Introduction of the Hardware

The hardware used for the purpose of this project includes the MCU, the serial cable (RS-232), a serial-to-USB adapter, a terminal emulator and a PC. The most

important of this is the MCU which will be shortly introduced. A serial adapter was necessary because of the absence of a serial port on the PC used for this project.

1.2.1 Overview of the MCU

For the purpose of this project, the MCU used was LPC-H2129 header board, manufactured by OLIMEX [3]. This board houses the LPC2129 ARM7TDMI-S microprocessor, manufactured by NXP Semiconductors, a subsidiary of Philips. The figure below shows an image of the MCU.

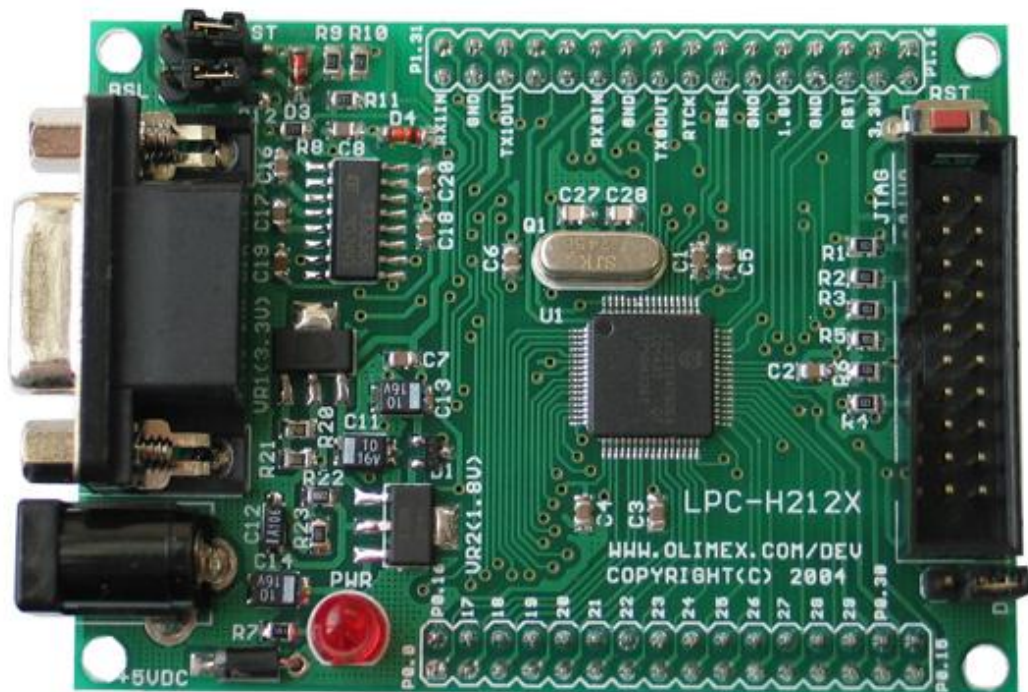


Figure 1. Image of OLIMEX MCU LPC-H2129.

The image above shows the first module produced by OLIMEX but the module used for the purpose of this project does not differ by a lot. The MCU has a lot of features, some of which are listed below [3]:

- MCU: LPC2129 16/32 bit ARM7TDMI-S™ with 256K Bytes Program Flash.
- 16K Bytes RAM, RTC.
- 4 10-bit ADC with 2.44 μ S conversion time.

- In-System/In-Application Programming (ISP/IAP) through on-chip boot loader software. A full erase of the chip or a complete flash of a sector in 100 millisecond and programming up to 256 bytes in one millisecond.
- Two CAN, two UART ports, I²C bus, SPI, two 32-bit TIMERS, seven CCR, six PWM channels, and WDT.
- 5V tolerant I/O, up to 60MHz operation.
- BSL jumper for bootloader enable.
- JNST jumper for enable/disable external RESET control by RS-232.
- Vectored Interrupt Controller.
- Up to 46 General Purpose I/O.

1.3 Introduction of the Software

The software part of this project includes just two which are WinARM and Bray's Terminal (BrayTerm). WinARM was used for compiling the code for this project and also for programming the MCU. BrayTerm is a terminal emulator used because the current version of windows does not come with the traditional terminal, Hyperterminal. BrayTerm serves the same purpose and because it has other functionalities such as its data rate being customizable makes it even better than using Hyperterminal.

1.3.1 WinARM

WinARM is a collection of GNU and other tools to develop software for the ARM-family of controllers/processors on MS-Windows-hosts /4/. WinARM contains all needed tools in its distribution package. It is an open source package that still needs a lot of work for it to be very functional on windows. There are some other platforms, such as Eclipse and Keil, which can also be used for such work but because of the insufficient availability of resources, WinARM was chosen for the project. WinARM was used to program the MCU throughout the duration of the project.

1.3.2 Bray's Terminal

Bray's Terminal (BrayTerm) is a terminal emulator which can be used with various devices with the capability of communicating serially with other devices. One of such devices is the MCU which has been used for this thesis project. The MCU includes two UART ports which can be used for such a purpose. One advantage it has over some other freeware terminal emulators is that its data rates can be customized. Another advantage is that it can be configured to use macros which can come in handy depending on the kind of device it is being used with. The version of BrayTerm used was version v1.9.

2 CRYPTOGRAPHY

Cryptography generally refers to the method of making data invisible to any third party who the availability of such data could be potentially harmful to the original parties the data needs to be available to. It deals mostly with the aspect of information security that includes data integrity, authentication and data confidentiality [5]. This means that the parties exchanging information do not necessarily know each other; they may not even know the location of one another,

but still need to exchange information or data, depending on the reason for communication existing between both parties. One of the applications of cryptography that best describes this scenario is the use of ATM cards. The authenticity of the user may not necessarily be known but by furnishing such cards with the use of a PIN number, the use of such PIN verifies that the user of the card is currently the owner of the card. There are two different categories of cryptography. This includes symmetric and asymmetric key cryptography.

2.1 Components of cryptography

To comprehensively explain the concept of cryptography and its categories there are some components that need to be introduced.

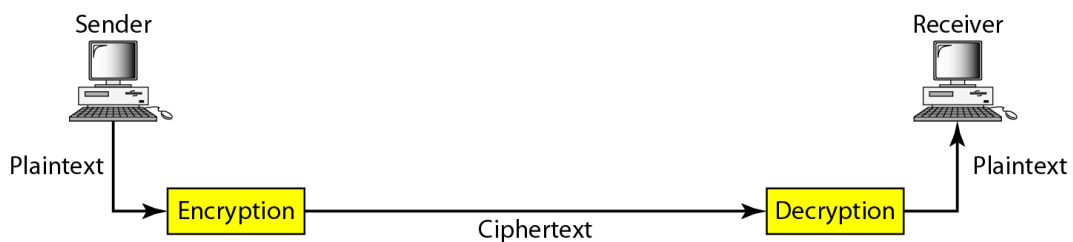


Figure 2. The components of cryptography.

The figure above shows a pictorial representation of the components of cryptography. Some short notes about some of the components are given below.

2.1.1 Plaintext and Ciphertext /1/

Plaintext refers to the original message or data, as the case may be, before being encrypted or changed. The changed message or whatever the message becomes after being encrypted is referred to as the ciphertext. An algorithm that transforms a plaintext into a ciphertext is an encryption algorithm and vice-versa, a decryption algorithm.

2.1.2 Key

The numbers used by an encryption or decryption algorithm to perform the process of transformation is referred to as a Key. Therefore for encryption, three things are needed; a plaintext, an encryption key and the encryption algorithm while for decryption, the ciphertext, the decryption key and the decryption algorithm are needed before the ciphertext can be transformed back to its original format.

2.1.3 Alice, Bob and Eve

In cryptography, there are usually three entities that need to be depicted in an information exchange scenario. There is the entity that needs to send secure messages or data, the other entity is the intended recipient of the message, and the third is the entity always trying to intercept the message and always tries to act as an impostor. Alice represents the sender, Bob the receiver and Eve the impostor.

2.2 Categories of cryptography

As mentioned earlier, there are two categories of cryptography which includes symmetric key, also called secret-key cryptography, and asymmetric key, also known as public-key, cryptography. Both categories are going to be introduced in the following paragraphs; however, public-key cryptography will further be detailed with emphasis on the algorithm of choice (RSA algorithm).

2.2.1 Symmetric key (secret-key) cryptography

In symmetric key cryptography, the most important thing to note is that both communicating parties actually share the same key. The key is only ever known to both the intended communicating parties, hence the name secret-key. Alice uses the key with an encryption algorithm to transform plaintext to ciphertext; Bob uses the same key with a corresponding decryption algorithm to transform ciphertext back to plaintext. The figure below shows the idea behind secret-key cryptography.

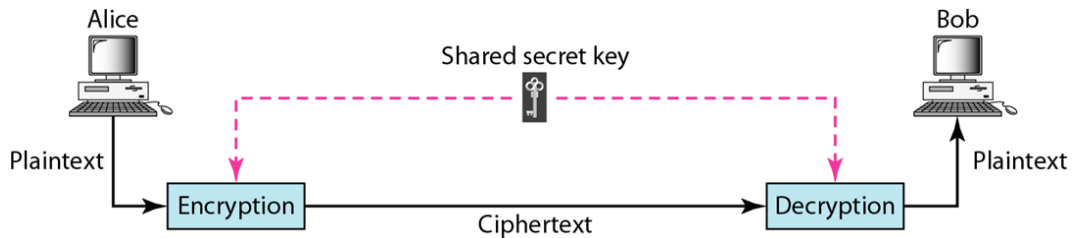


Figure 3. Symmetric key cryptography.

Symmetric-key cryptography is the oldest form of cryptography and has been used for thousands of years. There are the old methods and those have been replaced by more efficient forms of secret-key cryptography. The old methods are known as traditional ciphers. The old algorithms were usually character dependent, while the modern ciphers are bit-oriented. Traditional ciphers include the following, substitution ciphers and transposition ciphers.

Substitution ciphers are further divided into two categories known as monoalphabetic and polyalphabetic. Substitution ciphers work on the principle of substituting an alphabet with another. This is also the origin of its name. An example of a monoalphabetic cipher is the shift cipher, which works based on the assumption that plaintext and ciphertext consists only of uppercase letters. This cipher is also known as the Caesar cipher, and this is because Julius Caesar used this method to communicate with his officers.

Transposition ciphers, instead of substituting alphabets, changes the location of each character based on predefined rules as to which character in the alphabet goes to what location in the positioning of alphabets.

Other categories of symmetric-key cryptography include simple modern ciphers and modern round ciphers. The XOR cipher, rotation cipher, substitution cipher and transposition cipher are examples of simple modern ciphers and the examples

of widely used modern round ciphers includes the data encryption standard (DES) and advanced encryption standard (AES) /5/.

2.2.2 Asymmetric key (public-key) cryptography

In asymmetric key cryptography, there are two keys involved in the communication process. There is a private key, used by the receiver only and therefore kept private. The other is the public key which is announced or publicly available and used by the sender of the message. The figure below shows a scenario using public-key cryptography.

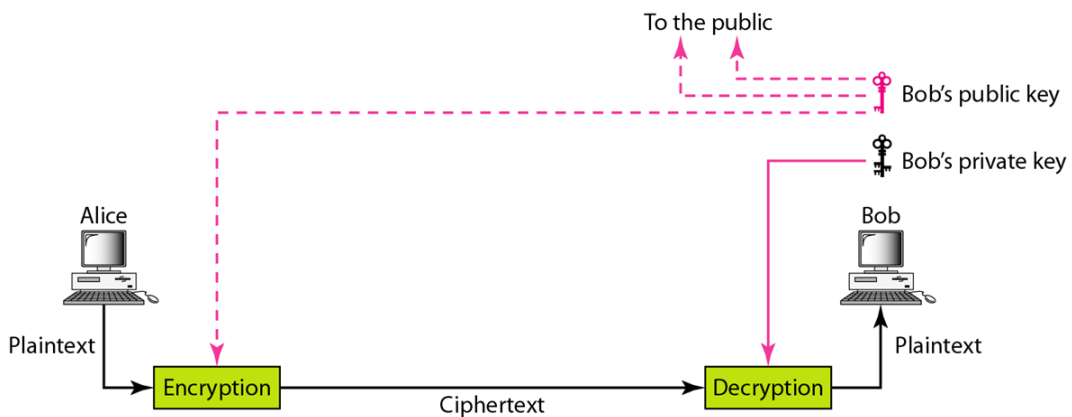


Figure 4. Asymmetric key cryptography.

According to the figure above, Alice intends to send a message to Bob so what she does is that she encrypts the message using Bob's public key with an agreed upon encryption algorithm. As can be seen from the figure, Bob's public key is known to everyone, that is, it is public. Bob, being the owner of the public key, is the only one able to decrypt the message, using a corresponding decryption algorithm, with his private key. This ensures that Bob is the only recipient of such a message.

2.3 Types of keys

There are three types of keys being used in cryptography. These include the secret key, the private key and the public key. The first key (secret key), is only ever

used in symmetric-key cryptography. The other two keys (private and public keys) are used only for asymmetric-key cryptography.

Examples of asymmetric-key cryptography include RSA algorithm and Diffie-Hellman. This report will focus mainly on the RSA algorithm as it is the algorithm used to complete this project.

3 CRC ALGORITHM AND IMPLEMENTATION

Cyclic redundancy check is an important cyclic method of checking data widely known in telecommunications. It is especially useful in the detection of burst errors. Burst errors can best be described as errors that occur contiguously in any data stream. The rate at which data error is measured varies depending on the type of data transmission method being used. The method of transmission could be synchronous or asynchronous. Asynchronous means of transmission usually involves errors of burst type. One could say that this perhaps is one reason for the continuous use of the method of data check. Cyclic redundancy check can also be used to detect single bit errors, double bits errors as well as an odd number of errors.

Cyclic methods should be better explained before one embarks on describing the method of cyclic redundancy check. A better way of understanding cyclic codes can be described as; the change of one stream of data bit cyclically into another stream with the new one forming another code in its form. As an example, one can consider a stream of data 110011 in its original format being modified cyclically to become 100111. The modified code can be considered as another code itself because it can be translated to mean something under the data translation method being used. Cyclic codes have been developed based on this kind of system of data modification.

3.1 Redundancy

This concept is central to error detection and correction. This is so because, before any data stream can be checked or corrected, some extra bits have to be added to the original data stream. These added bits are added by the sender before transmission and removed by the receiver before using the data. The use of these bits makes it possible for the receiver to detect and correct the corrupted part of data.

3.2 Polynomials

The theory behind the use of cyclic redundancy check can best be explained with the use of polynomials. This is important because of the concept of burst error. Burst errors have the nature of involving a long length of data stream. Since all data transmission deals with numbers of modulo-2 base, that is 1s and 0s, burst errors are usually represented when using cyclic redundancy check as polynomials whose highest degree is of the form $n - 1$, where n is the number of data bits being transmitted. The data stream can be represented by polynomials when the power of the polynomial reflects the position of the bit and the coefficient used to represent the value of the bit, either 1 or 0. As an example, considering the data stream used earlier, 110011, a polynomial representation would be written as $1x^5 + 1x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$. An advantage of polynomial representation is that it can be easily reduced to a simple single term, just as is the case with normal polynomial representation. The above polynomial representation can be reduced

to $x^5 + x^4 + x + 1$. The last bit is represented as a 1 because the power to zero of any term of a polynomial is 1. The last bit becomes unrepresented only if the value of the bit itself is 0. The following paragraphs discuss shortly about different polynomial arithmetic.

3.2.1 Addition and subtraction of polynomials

Addition and subtraction of polynomials normally is performed by the addition or subtraction as the case may be, of the coefficients of terms with the same power. The coefficients in modular arithmetic only have the value 0 or 1. This implies two things as a result, addition and subtraction of modulo-2 arithmetic yields the same results. The other is that the two polynomials being added or subtracted are only merged and terms with the same power become removed. For the second condition, this only applies to even number of times of occurrence of terms with the same power, if such a term were to exist for an odd number of times; there is only one representative of such a term in the resulting polynomial. One thing to be noted in polynomial addition or subtraction is that all terms that are non-zero are only represented once in any modulo-2 polynomial.

3.2.2 Multiplying polynomials

Polynomial multiplication is carried out per term. This means that each term in each polynomial expression is used by turn to multiply the terms of the other polynomial. Multiplication of terms occurs by adding the powers of both terms being multiplied. The resulting expression is added after all terms have been multiplied and based on the condition of addition, the terms occurring an even number of times becomes deleted.

3.2.3 Dividing polynomials

Polynomial division is very similar to the regular long division. The only difference is that for any polynomial division, the divisor is not subtracted but a xor operation is performed upon it and the dividend. This is done over and over

again until all the terms of the quotient are completely exhausted and the highest power of the remainder is less than the power of the dividend. The remainder can then be used as appropriate for CRC implementation.

3.3 Cyclic code

Cyclic codes are analysed with the help of polynomials. Cyclic code can best be analysed by first defining some terms. These terms are introduced below:

- **Data-word:** the original data to be transmitted, $d(x)$.
- **Code-word:** the data-word with redundant bits added, $c(x)$.
- **Polynomial generator:** this is the divisor of the polynomial, $g(x)$.
- **Syndrome:** the results from the division of the dividend, which contains the code-word, $s(x)$.
- **Error:** the error discovered from the code-word, $e(x)$.

There are a few things that can be used to ascertain that any form of data does not contain any error. Some of those are described below.

1. For a case where $s(x) \neq 0$, a certain fact is that 1 or more bits are corrupted.
2. For a case where $s(x) = 0$, two facts are established.
 - a. Either no bits are corrupted; or
 - b. Some bits are corrupted but were not discovered by the receiver.

As a result of these two cases, one discovers that it is also important to choose a very good generator to divide the code that will not allow any error, 1 or more, to occur at any point in time on a line. The received code-word contains $c(x)$ and $e(x)$. The receiver divides the received code-word received by the same generator and this can either mean a 0 error transmission or that the errors could not be detected. As can be seen in the equation below, if the error bits, $e(x)$ are completely divisible by the generator, $g(x)$, then, there will always be a false positive result of the polynomial calculation, which means errors will become undetected.

$$\frac{\text{Code - word}}{g(x)} = \frac{c(x)}{g(x)} + \frac{e(x)}{g(x)}$$

Based on the type of generator polynomial chosen, the errors that can be detected include, single bit errors, two-isolated single bit errors, and burst errors. The following list includes the conditions that a good generator has to meet for it to catch the errors mentioned earlier.

1. The generator should have a minimum of two terms.
2. The coefficient of the last term needs to be 1.
3. The generator should not be able to divide $x^t + 1$, for values of t ranging from 2 to $n - 1$, where n is the number terms of the code-word polynomial.
4. The generator should have the factor $x + 1$ common to its polynomial.

Over the years, the choice of generators has been standardized based on different researches carried out and different polynomial values used. The most popular protocols for CRC generators include the following:

1. CRC-8: $x^8 + x^2 + x + 1$. This generator is used in ATM headers.
2. CRC-10: $x^{10} + x^9 + x^5 + x^4 + x^2 + 1$. Used for ATM AAL.
3. CRC-16: $x^{16} + x^{12} + x^5 + 1$. This is used for HDLC.
4. CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. This polynomial has its use in LANs.

For the purpose of this project, CRC-32 was implemented. One of the advantages of using CRC is that it can detect errors of various positions and of different types. It is also data transfer rate independent.

3.4 Algorithm and implementation

CRC-32 has about four different types of generator polynomial standardized for it; however, the most used out of all four is listed above. There are three different ways whereby it is represented in binary or hexadecimal format. Depending on this method of expression, the generator polynomial can appear to be different.

However, all representations are equal to one another. The main reason for such representations takes into consideration, which of the bit is being transmitted first on the line. There is the LSB first style and the MSB first style. For this project the LSB first style was used. The polynomial for this representation is 0xEBD8888320. This only means that there is the assumption that the LSB digits get transmitted first before the MSB digits. The generator polynomial is normally 1 bit more than the actual name indicates. So in this case, there are 33 generator bits in total. CRC algorithm uses two readily defined facts to its advantage; the first is that the MSB of generator polynomials is always 1. The other reason is that the MSB of the xor operation for modulo-2 arithmetic is always zero and therefore, eventually becomes shifted out of the remainder at any point in time. This means that one need not worry about any carry values since the first bit is always know to be 1. The generator polynomial which is 33 bits in length can then be stored using a 32-bit register. The algorithm uses two functions which will be described here. The first function `init_crc32_tab()` works as to fill an array for computing CRC, with values. The algorithm is outline below:

INPUT: nothing.

OUTPUT: nothing.

1. Initialize `crc_tab32_init` to false.
2. For i count (from 0 to max), max is the size of the array; create max number of instances in memory for crc.
3. For j count (from 0 to b), b represents a byte;
 - a. If `crc = 0`; perform $(\text{crc} \gg 1) \text{ xor } g(x)$; else do $(\text{crc} \gg 1)$.
4. Do `crc_table[i] = crc`;
5. Set `crc_tab32_init` to true.

The second function is `update_crc_32()` and it serves the purpose of updating the CRC table with the computation of CRC-32 for the previous byte and the next byte of data being computed. The algorithm is outlined below:

INPUT: data byte to be transformed, current table values.

OUTPUT: current crc value.

1. Find all 1 bits in each byte of data.
2. If `crc_tab32_init = true`; do `init_crc32_tab()`;
3. Flip all high bits of data byte and store in `tmp`;
4. Do `crc = (crc right-shift 8 bits) xor crc_tab32[high bits in tmp]`;
5. Return remainder value (`crc`).

3.4.1 Implementation

Implementing this on hardware requires some initialisations. This will be detailed in the next chapter as this is not the main project work. The `main()` function for implementing CRC-32 takes in data stream from the user and stores it in an array of 256 characters each 8-bytes in size. For doing this, all CRC-32 implementations are initialized as 1s. There is a check for carriage return and newline command the table is updated with the input data and the high initialized bits. Then it counts for each byte of data. The return remainder value is the flipped again as this uses the reversed generator polynomial, that is LSB first.

After running the code and connecting the MCU to BrayTerm, the following figures show the different inputs and the subsequent CRC-32 values it returns.

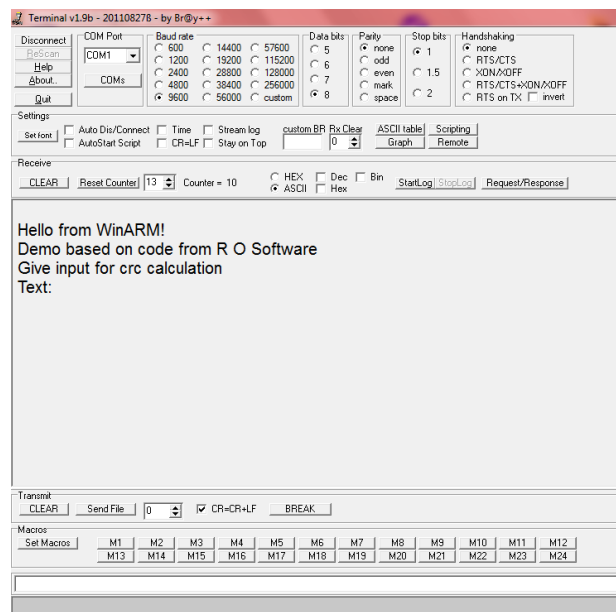


Figure 5. Input data request before CRC calculation.

Figure 6 below shows the CRC-32 value that was returned after the algorithm was performed on the input data.

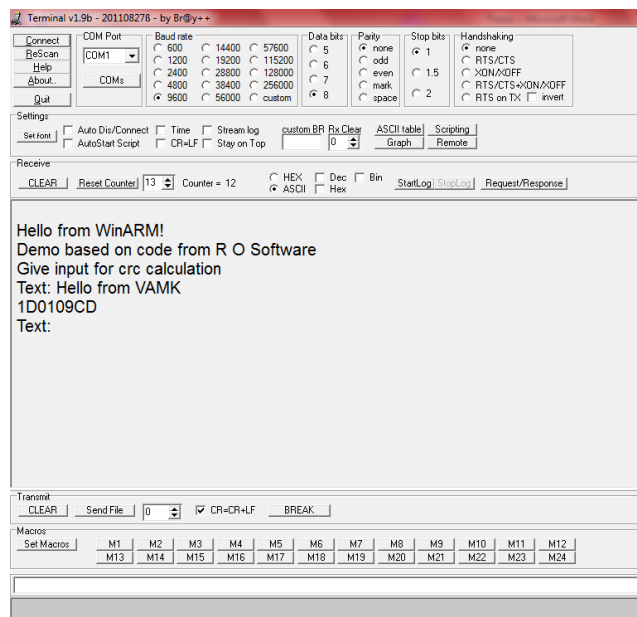


Figure 6. CRC value returned for input data.

4 RIVEST, SHAMIR, ADLEMAN (RSA) ALGORITHM

Asymmetric-key (public-key) cryptography was developed much later compared to symmetric-key cryptography. It works based on the computational complexity of difficult problems, usually from number theories. One of the earliest public-key algorithms is the Diffie-Hellman algorithm, which was proposed by, and named after, Whitfield Diffie and Martin Hellman in 1976. In 1978, another algorithm, which has become the basis for most public-key cryptography, was invented by three men. They are Ronald Rivest, Adi Shamir, and Len Adleman. Their invention was named after them and is now known as the RSA algorithm. The different variations of the public-key cryptographies that are based on the number-theoretic computational problems as the basis of security are shown in the table below.

Table 1. Public-key encryption schemes and corresponding computational problems which their security is based upon /5/.

Public-key Encryption Scheme	Computational Problem
RSA	Integer factorization problem
Rabin	Integer factorization problem; square roots modulo composite n

ElGamal	Discrete logarithm problem; Diffie-Hellman problem
Generalized ElGamal	Generalized discrete logarithm problem; generalized Diffie-Hellman problem
McEliece	Linear code decoding problem
Merkle-Hellman knapsack	Subset sum problem
Chor-Rivest knapsack	Subset sum problem
Goldwasser-Micali probabilistic	Quadratic residuosity problem.
Blum-Goldwasser probabilistic	Integer factorization problem; Rabin problem

As can be seen from the table, RSA has its origins from integer factorization problem. RSA mode of encryption has three main steps to complete a full message cycle. These steps are listed below:

- Key generation.
- Plaintext encryption.
- Ciphertext decryption.

These three parts are elaborated upon in the following sub-chapters.

4.1 Key generation

The most important part of RSA algorithm is the generation of the right keys. This is so because when keys are not generated correctly, such keys are not strong enough to completely encrypt any message as it becomes relatively easier to decipher the message by the third party. Illustrated below are a set of rules to take into account when attempting to generate public and private keys for RSA encryption. Each entity, in this case Bob, generates a public key and a

corresponding private key. To generate a strong pair of keys, Bob should take the following steps /1/ /5/:

1. Chooses a pair of large random primes x and y , about the same size each.
2. Calculates $n = xy$ and $\phi = (x - 1)(y - 1)$.
3. Selects a random integer i , where $1 < i < \phi$, where the $\gcd(i, \phi) = 1$. This means that i and ϕ only have the number 1 as their greatest common divisor (gcd).
4. Then calculates a unique integer d , $1 < d < \phi$, such that $id \equiv 1 \pmod{\phi}$.
5. Bob's public key is (n, i) which he announces to the public; Bob's private key is d . Bob keeps d and ϕ secret.

Both integers i and d , are referred to as the **encryption exponent** and **decryption exponent** respectively and the value n is the **modulus**.

4.2 Encryption

Anyone that wishes to send a message to Bob can use n and e /1/. If Alice wishes to send a message to Bob, the following steps should be followed to achieve this.

1. Alice changes the plaintext to integer m , m must satisfy the condition $0 < m < (n - 1)$ /1/ /5/.
2. Alice then calculates $c = m^e \pmod{n}$, where c is the ciphertext /1/ /5/.
3. Alice then sends the ciphertext c , to Bob.

By going through these outlines steps, Alice can encrypt the message she wishes to send to Bob without having to worry about it getting into the wrong hands.

4.3 Decryption

To decrypt Alice's message, Bob uses his private key d . To get this done, Bob only needs to do the following.

1. Bob uses his private key d to decrypt the message by: $m = c^d \pmod{n}$.

m is the plaintext in integer; all Bob needs to do now is to convert it back to its original format.

The following steps will prove that decryption works using the private key d .

Proof:

$$\text{Since } ed \equiv 1 \pmod{\phi}, \quad (3.1)$$

Then there exists an integer k such that

$$ed = 1 + k\phi. \quad (3.2)$$

Now, if

$$\gcd(m, p) = 1 \quad (3.3)$$

Then; by Fermat's theorem /5/

$$m^{p-1} \equiv 1 \pmod{p}. \quad (3.4)$$

Raising both sides in equation (4) to the power $k(q-1)$ and then multiplying both sides by m yields:

$$m^{1+k(p-1)(q-1)} \equiv m \pmod{p}. \quad (3.5)$$

On the other hand, if $\gcd(m, p) = p$, then this last congruence is valid since each side is congruent to $0 \pmod{p}$. Hence, in all cases

$$m^{ed} \equiv m \pmod{p}. \quad (3.6)$$

By the same argument,

$$m^{ed} \equiv m \pmod{q}. \quad (3.7)$$

Finally, since p and q are distinct primes, it follows that

$$m^{ed} \equiv m \pmod{n}; \quad (3.8)$$

and, hence,

$$c^d \equiv (m^e)^d \equiv m \pmod{n}. \quad (3.9).$$

4.4 Example using RSA algorithm /1/

Bob chooses 7 and 11 as p and q and calculates $n = 7 * 11 = 77$. The values of $\phi = (7 - 1) (11 - 1)$ or 60. Now he chooses two keys, e and d . If he chooses e to be 13, then d is 37. Now imagine Alice sends plaintext 5 to Bob. She uses the public key 13 to encrypt 5. This is shown in the following steps:

Plaintext: 5

$$C = 5^{13} = 26 \pmod{77}$$

Ciphertext: 26

From these calculations, Bob receives the ciphertext 26 and uses the private key 37 to decipher the ciphertext according to the steps below:

Ciphertext: 26

$$m = 26^{37} = 5 \pmod{77}$$

Plaintext: 5

The plaintext 5 sent by Alice is received by Bob as plaintext 5.

4.5 Security of RSA

There exists several security issues related to RSA encryption. The following sections will talk about some of the known issues and subsequent solutions to deal with these problems.

1. **Factoring:** In the generation of RSA public and private keys, it is very important that both prime x and y chosen be in such a way that the

factoring of the product of both primes n is computationally very tasking. This means in essence that if the probability of factoring both primes is high, then it automatically becomes easy for a third party, in this case Eve, to successfully intercept and decrypt a message intended for Bob. Therefore, great attention must be paid to ensure that the primes are very distinct and are not close to each other numerically.

2. **Encryption component e , size:** it is usually the case that the size of the encryption component e be considerably smaller than the decryption component d , and *modulo*, n for efficient encryption. A problem arises when the size of e is too small. This is because it becomes easier for Eve to decrypt the message send to several people using a small sized e as the encryption pattern is very similar and therefore it infers that their moduli are relatively prime to one another. The easiest way to solve this is by salting the plaintext before encryption /5/. Salting is addition of bit-string of zeros to the original plaintext before encryption.
3. **Forward search attack:** this refers to a small or predictable message size. This means Eve can decrypt the encrypted plaintext by just encrypting all possible plaintext message of the same size until she gets a similar ciphertext. Salting the plaintext can also help prevent this from occurring.
4. **Adaptive chosen ciphertext attack:** this is a problem because of the multiplicative properties, otherwise known as the homomorphic property of RSA algorithm. This property is shown below:

For two plaintexts p_1 and p_2 , and a respective cipher text t_1 and t_2 ; then it can be shown that:

$$(p_1 p_2)^e \equiv p_1^e p_2^e \equiv t_1 t_2 \pmod{n}. \quad (3.10)$$

This means that the ciphertext whose plaintext is:

$$p = p_1 p_2 \pmod{n}, \text{ is}$$

$$t = t_1 t_2 \pmod{n}.$$

Due to this property, Eve only needs to send an encrypted ciphertext t' , to Bob using Bob's public key. This is because Bob will not decrypt a message sent from Eve. Once Bob has decrypted this ciphertext to its corresponding plaintext p' , Eve only needs to calculate p such that:

$$p = p'x^{-1} \text{ mod } n.$$

Since

$$p' \equiv (t')^d \equiv t^d (x^e)^d \equiv px \text{ (mod } n).$$

The way to solve this problem is by padding such a message before it is being encrypted. This gives the plaintext a certain structure that has been agreed between both Alice and Bob. Eve's encrypted text will always be discarded as it will not fit into this structure.

5. **Message concealing:** this problem occurs when a plaintext message is said to be unconcealed. This happens when the plaintext encrypts to itself. This generally does not pose a threat as the number of unconcealed messages remains always negligibly small.

4.6 RSA encryption implementation notes

Over the years there have been great improvements in speeding up the implementations of RSA algorithm, both encryption and decryption, in software and hardware. A few of the methods being used include, fast modular multiplication, fast modular exponentiation, and by using Chinese remainder theorem for faster decryption. Despite all these improvements, RSA algorithm is still relatively very slow when compared with symmetric key cryptography. This limits the use of RSA algorithm in applications that require speed. Another reason RSA encryption and decryption cannot totally be improved is because of the fact that, longer key sizes increase the overall decryption time of RSA algorithm. The use of this algorithm is therefore limited to applications that require a short message length. An example of this is digital signatures.

Digital signature is very similar to physically signing a document as it is a definitive proof of identity of the signer of such a document. However, it is more secure as such a signature cannot be forged.

5 RSA ALGORITHM ON ARM

The previous chapter has been used to thoroughly introduce the concept of RSA algorithm, how it works, its pros and cons. In this chapter, the implementation of the algorithm on the hardware used for this project will be detailed below.

The implementation started with the setup of the MCU and subsequent activation of other parts necessary for the completion of this project. Configuring the hardware for the using the UART port, which is necessary for serial data transmission to and from the MCU. The system setup will be explained in the subsequent subheadings, followed only by the implementation and testing of the software. Results will then be discussed and possible improvements proposed.

5.1 ARM microcontroller hardware setup

The MCU used for the purpose of this project has been introduced at the start of this report. The hardware setup required to program the device is not a lot as it is a very compact device. When setting up the MCU for programming, there are several ways to achieve this, depending on the available means of programming the hardware. For the purpose of this project, the only software used to program the microprocessor is WinARM, which uses Programmer's Notepad as its compiler.

5.1.1 Software setup for microcontroller programming

WinARM happened to be the cheapest solution to programming, successfully, the MCU. The only piece of hardware needed to communicate with it is an RS-232 cable. The MCU has two UART ports, one of which is used by the bootloader to program the MCU flash memory in the absence of an external programmer. This happens to be the case regarding this project, therefore UART channel 0 (UART

0) was used for programming and also testing the performance of the MCU according to the programmable instructions.

The bootloader is enabled when the BSL jumper on the MCU is shortened at the time of power up. For programming the MCU, the JRST and BSL jumper have to be shortened at time of power up. However, when running the code, both jumpers should be left open and the manual reset button on the MCU pushed to enable to microcontroller to exit bootloader mode.

The LED on the board has to be shortened before it can be activated. This can be done at any time while running the code or before the MCU has been programmed for whatever function.

The figure below shows the image of the interface that was used to compile and program the MCU.

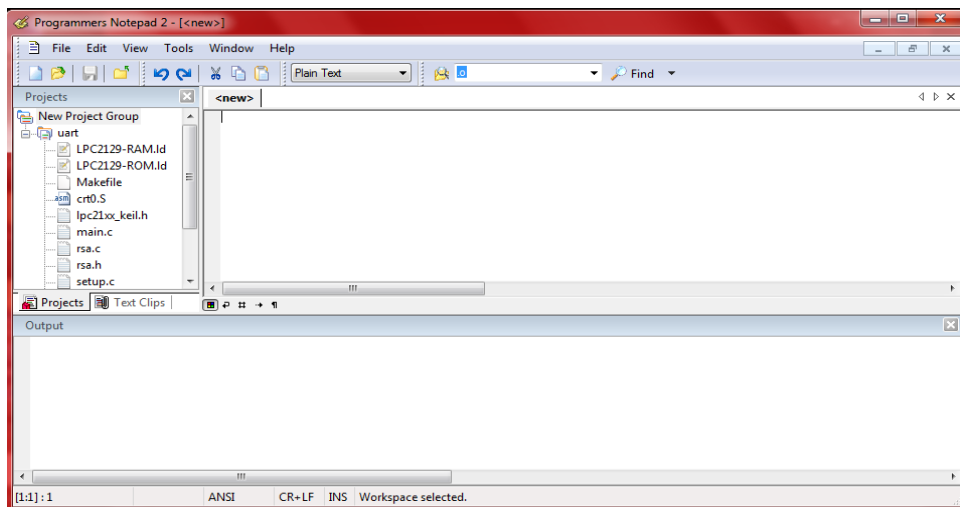


Figure 7. An image of programmer's notepad.

The platform is a very basic one, but very efficient as it has a customizable makefile and linker. This makes for having an efficient run-time environment.

5.1.2 Start up function for the microcontroller

The microcontroller start up uses the PLL present on the MCU for getting the microcontroller to run at different configurable speeds. PLL is able to boost the system clock up to 60 MHz, depending on the input frequency. The system clock runs between 10 – 25 MHz normally. This PLL clock is used to provide the on-chip clock for the ARM microprocessor. This allows the LPC 2129 run at its maximum configurable frequency with a low value oscillator, thus minimizing the EMC emissions of the device [6]. The PLL consists of two values, which include the multiplexer and the divider. Cclk, which is the CPU clock of the MCU, is the output of the PLL after the fundamental crystal frequency has been multiplied [7]. There is a current controlled frequency on the path of the PLL and it must be between the frequency ranges of 156–320 MHz. The PLL parameters are shown below.

$$Cclk = M \times Fosc;$$

$$Fcco = Cclk \times 2 \times P;$$

$$156 \leq Fcco \leq 320 \text{ MHz.}$$

From the equations above, Fcco can be simplified to become:

$$Fcco = Fosc \times M \times 2 \times P;$$

The MCU has an external frequency of 14567 KHz; the runtime clock of the system was configured to run at the maximum PLL speed. This means that the values of M and P have to be deduced based on the equations above. Therefore:

$$M = \frac{Cclk}{Fosc} = \frac{6000000}{14567000} = 4;$$

For the value of P, we have:

$$156 \leq 14567000 \times 4 \times 2 \times P \leq 320;$$

Therefore; P = 2.

The values of P and M are the ones that are used to initialize the register. After configuring the register responsible for PLL output frequency, PLL is first disconnected, while PLLFEED register is updated according to the sequence required for the activation of the MCU. After the PLLFEED sequence, there is a condition to check for PLL to set and lock at the required frequency. The connecting register for PLL is then set to activate the preset frequency. The PLLFEED sequence is again updated after which comes the activation of the peripheral clock. The register for setting the peripheral clock is the VPBDIV. The equation below shows how the register value is calculated.

$$VPBDIV = \frac{Cclk}{Pclk}$$

The register value for the peripheral clock was set to 1, this ensures that the peripheral clock runs at the same frequency as the MCU frequency of 60MHz. The last part to be set is the activation of memory accelerator module (MAM) registers which help to latch the next set of ARM instructions so as to avoid the system stalling. The figure below shows a snapshot of the function for activating the MCU.

```

/*Code for initializing the controller*/
void Initialize(void){
    PLLCFG = 0x23; /*MSEL = 3 (M = 1); PSEL = 2*/
    /*FOSC = 14745600; CCLK = 60MHZ;
    FCCO = 240MHZ*/
    PLLCON = 0x01;
    PLLFEED = 0xAA;
    PLLFEED = 0x55;

    while (!(PLLSTAT & 0x0400)){

    PLLCON = 0x03; /*Activate and connect PLL*/
    PLLFEED = 0xAA;
    PLLFEED = 0x55;

    VPBDIV = 0x01; /*Peripheral clock same as system clock*/

    MAMCR = 0; /*Disable Memory Accelerator*/
    MAMTIM = 0x03; /*Choose 3 fetch cycles cclk > 40MHz*/
    MAMCR = 0x02; /*Memory Acc fully enabled*/

    #if defined(RAM_RUN)
    MEMMAP = MEMMAP_USER_RAM_MODE;
    #elif defined(ROM_RUN)
    MEMMAP = MEMMAP_USER_FLASH_MODE;
    #else
    #error RUN_MODE not defined!
    #endif
    }
}

```


Figure 8. Code snippet showing MCU setup.

5.2 UART configuration function

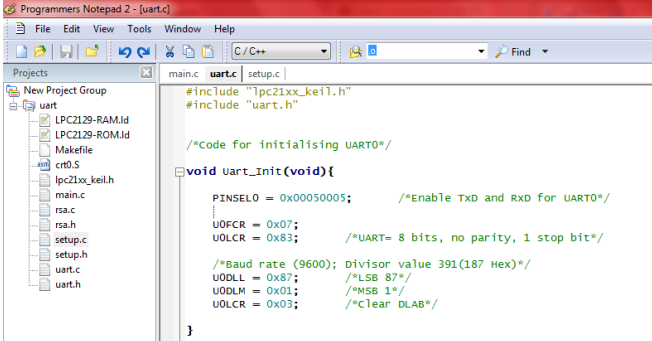
For the purpose of this project, only one UART channel (UART0) was used. This is the same UART channel used for programming the MCU. The function mainly deals with the setup of the UART0 port for transmission and reception of data. Also set in the function is the data rate at which the MCU will be using to communicate with other devices, in this case, the terminal emulator. The port is setup by activating the corresponding input for the PINSEL register. The data rate used for this project is 9600 bps (bits per second). The data format was set for no start bit, 8 data bits, 1 stop bit, and no parity bit. The same settings were applied to BrayTerm which is the emulator that was used. A formula for the determination of the data rate or baud rate is shown below:

$$Divisor = \frac{Pclk}{16 \times Baud};$$

For an expected baud rate of 9600, the equation becomes:

$$Divisor = \frac{60000000}{16 \times 9600} = 391;$$

This value is equivalent to 187_{hex}. This provides the value used for setting the UODLL and UODLM registers. The code used for the configuration of UART0 for this project is shown in the figure below.



```
main.c  uart.c  setup.c
#include "lpc21xx_keil.h"
#include "uart.h"

/*Code for initialising UART0*/
void uart_Init(void){
    PINSEL0 = 0x00050005;    /*Enable TXD and RxD for UART0*/
    UOPCR = 0x07;
    UODLL = 0x85;          /*UART= 8 bits, no parity, 1 stop bit*/

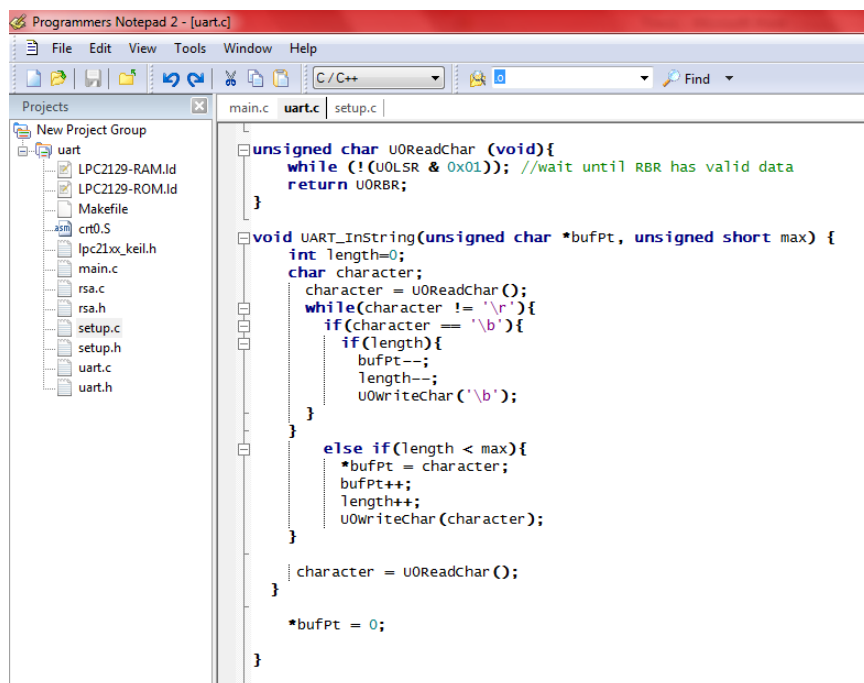
    /*Baud rate (9600); Divisor value 391(187 Hex)*/
    UODLL = 0x87;         /*LSB 87*/
    UODLM = 0x01;        /*MSB 1*/
    UOLCR = 0x03;        /*Clear DLAB*/
}
```

Figure 9. UART0 setup function.

5.2.1 UART0 Transmission and Receiving functions

This project was implemented basically to be able to transmit data serially; therefore, there are a few functions which have been used to configure the UART0 port for receiving and transmitting data. The configuration supports the reception of data in ASCII (American Standard Code for Information Interchange) format. Some further notes are given below.

1. **Receiving functions:** there are two functions programmed to perform this action. One receives data character by character while the other receives a data string all at once. `UART_Instring` is the function receiving a string of data at once. This function calls the first function and stores each value in a buffer to be closed once the return key has been pressed. Below is a code snippet showing both functions.



```

unsigned char U0ReadChar (void){
    while (!(U0LSR & 0x01)); //wait until RBR has valid data
    return U0RBR;
}

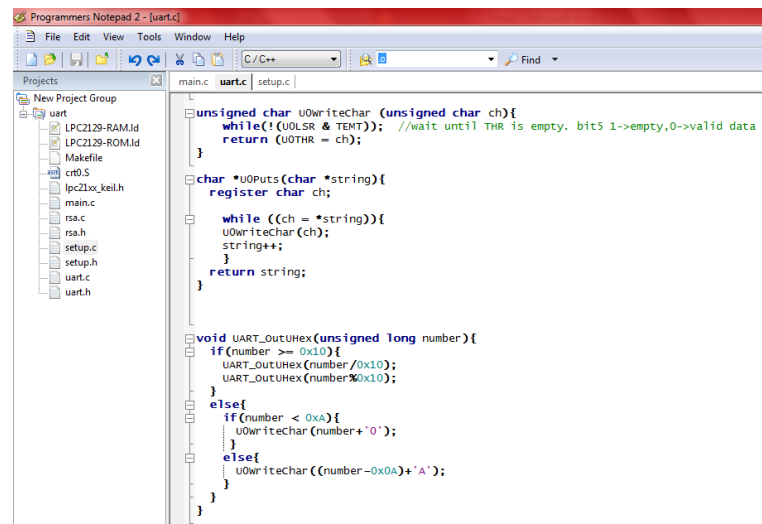
void UART_Instring(unsigned char *bufPt, unsigned short max) {
    int length=0;
    char character;
    character = U0ReadChar();
    while(character != '\r'){
        if(character == '\b'){
            if(length){
                bufPt--;
                length--;
                U0writeChar('\b');
            }
        }
        else if(length < max){
            *bufPt = character;
            bufPt++;
            length++;
            U0writeChar(character);
        }
    }
    character = U0ReadChar();
    *bufPt = 0;
}

```

Figure 10. UART0 receiving functions.

2. **Transmitting functions:** there are three functions written for this part of the implementation so as to enable all output as expected from the project. One of the functions transmits data, one character at a time. The second one transmits data character strings. The third one actually formats data. It takes in an integer and formats it into hexadecimal format before transmitting the converted data through UART0.

All three functions helped make the communication of the MCU work seamlessly, without any problem at any point in time. The code snippet below shows the functions.



```
unsigned char UOwritechar (unsigned char ch){
    while(!(UOLSR & TEMT)); //wait until THR is empty, bits 1->empty,0->valid data
    return (UOTHR = ch);
}

char *UOPuts(char *string){
    register char ch;

    while ((ch = *string)){
        UOwritechar(ch);
        string++;
    }

    return string;
}

void UART_OUTUHex(unsigned long number){
    if(number >= 0x10){
        UART_OUTUHex(number/0x10);
        UART_OUTUHex(number%0x10);
    }
    else{
        if(number < 0xA){
            UOwritechar(number+'0');
        }
        else{
            UOwritechar((number-0xA)+'A');
        }
    }
}
```

Figure 11. UART0 transmitting functions.

5.3 RSA implementation

Implementing RSA algorithm efficiently is essential in applications. This is so because the algorithm is slow compared to the symmetric means of encryption and decryption. Due to this reason, implementing it in an efficient manner is very important to any application in which it is being used. There are a few algorithms that help to make this possible; among them is the Chinese remainder theorem which improves the processing time for decryption and generation of signature by using modular representation /5 Page 612/. Also the Garner's algorithm, which has its efficiency in the calculation of RSA moduli and due to its exponentiation

factors, results in the computation of decryption being four times faster. Exponentiation is another well-known method of implementing RSA algorithm. It is regarded as one of the most important arithmetic operations for public-key cryptography/5 Page 613/. The RSA scheme requires exponentiation for a positive integer. An efficient method for multiplying two elements in a group of finite elements is essential to performing efficient exponentiation. For cryptographic applications, the order of the group finite group exceeds 2^{160} , and these days, it exceeds 2^{1024} for RSA algorithm.

Two ways exist for the reduction of time required to perform exponentiation. One is to decrease the time to multiply two elements; the other is to reduce the number of multiplications used to calculate the exponent of a number. In ideal applications, both would be implemented. There are three types of exponentiation algorithms that can be considered when referring to RSA algorithm and its variants. A brief description is given below for all three.

1. **Basic techniques for exponentiation:** this uses arbitrary choices for the base and exponent.
2. **Fixed-exponent exponentiation algorithms:** as the name implies, the exponent is fixed, that is, unchanging and arbitrary values of the base are allowed. This are the algorithms RSA encryption and decryption are most reliable upon.
3. **Fixed-base exponentiation algorithms:** here the base is fixed and arbitrary choices of the exponent can be chosen. Variations of RSA such as ElGamal and other public-key systems such as Diffie-Hellman key agreement favour this kind of techniques.

Out of all three the first one will be explained further as this was the method employed for the algorithm in this project. The reason for the use of this method of implementation was the availability of resources which helped to make clearer the method of implementation. Another reason, which was also important, was the ability to implement methods of abstraction to the data being used for the

algorithm. The manipulation of large numbers in software level can be very difficult to perform. The use of basic arithmetic operations of addition, subtraction, multiplication, squaring, and division for multiple-precision integers had to be introduced because of the large key sizes used by RSA. The C programming language has its limitations whenever the integer being computed exceeds 64-bits. As was discussed earlier, RSA key-lengths are of the minimum these days of 1024-bits. Handling that size of integer poses a challenge already before the introduction of the exponentiation to compute RSA encryption and subsequent decryption.

As a result, multiple-precision integer arithmetic has to be discussed in order to fully understand the method of implementing RSA encryption and decryption.

5.3.1 Multiple-precision integer arithmetic

Multiple-precision integer can be explained to be calculations that are performed on numbers whose digits of precision are limited only by the available memory of the host system. In C language, multiple-precision integers are handled by using variable-length arrays of digits. There are several libraries provided for such data manipulation readily available for use with the C language. A few of these are available for free for non-commercial purposes. One of such library was used for this project. The only other option would have been to write all functions necessary for such integer arithmetic, which in itself would have greatly extended the period of time necessary to complete this project.

The library used for this project is the BigDigits library, available for free use, provided by D.I. Management Services Pty Limited. The use of this library made it easier to deal with the data being used RSA algorithm.

This library deals with multiple precision arithmetic parts of implementing the RSA algorithm. There are several functions to handle single and multiple precision numbers, however, the algorithm for handle multiple precision numbers will be detailed below.

1. **Addition and subtraction for multiple precision integers:** this type of arithmetic is performed on two integers with the same number of digits. The numbers must be of a similar base before such operations can be performed. If one of the numbers is of a different base, then such number will need to be converted to the required base number. Concerning a condition where one of the numbers has a different length, the shorter number needs to be padded with 0s on the left, that is, the most significant bit position. The algorithm for performing the addition arithmetic is outlined below:

INPUT: takes in positive integers a and b , each with $n + 1$ base x digits.

OUTPUT: returns the sum $a + b = (w_{n+1}, w_n \dots w_1, w_0)_b$ in x representation.

1. $k \leftarrow 0$ where k is the carry digit.
2. For j count (from 0 up to n); n is the number of digits.
 - a. $w_i \leftarrow (a_i + b_i + k) \bmod b$.
 - b. Check if $(a_i + b_i + k) < b$, then $k \leftarrow 0$; else $k \leftarrow 1$.
3. $w_{n+1} \leftarrow k$.
4. Return $((w_{n+1}, w_n \dots w_1, w_0))$.

The algorithm for calculating multiple precision subtractions is outlined below:

INPUT: takes positive integers a and b , with $n + 1$ with base x digits, while $a \geq b$.

OUTPUT: difference $a - b = (w_n, w_{n-1} \dots w_1, w_0)_x$ in b representation.

1. $k \leftarrow 0$.
2. While counting down from n to 0; n is the number of digits.
 - a. Check if $a < b$, return -1.

Check if $a > b$, return 1.

2. **Multiplication for multiple precision integers:** this type of arithmetic will have the length of $n + t + 1$, where n and t are the number of digits of each of the integer to be multiplied, at the most. The algorithm is a modification of the standard method used in schools. The algorithm is outlined below:

INPUT: positive integers a and b with $n + 1$ and $t + 1$ same base digits respectively.

OUTPUT: the product $ab = (w_{n+t+1} \dots w_1 w_0)_x$ in base x representation.

1. For i count (from 0 to $n + t + 1$); initialise w_i to 0.
2. For j count (from 0 to t);
 - a. $k \leftarrow 0$.
 - b. For i count (from 0 to n):
 - i. Calculate $(uv)_b = w_{i+j} + a_i b_j + k$, and set $w_{i+j} \leftarrow v$, $k \leftarrow u$.
 - c. $w_{i+n+1} \leftarrow u$.
3. Return $((w_{n+t+1} \dots w_1 w_0))$.

Calculating $(uv)_b$ is known as the inner-product method. Since w_{i+j} , a_i , b_j and k are all of the same base x , the result of the operation is at the most $(x - 1) + (x - 1)^2 + (x - 1) = x^2 - 1$, and therefore it can be represented by to base x digits. The outline above requires $(n + 1)(t + 1)$ single precision multiplications.

3. **Squaring for multiple precision integers:** from the previous algorithm one will notice that $(uv)_b$ has both u and v as single precision integers. In this section, u and is used as a double precision integer in such a way that $0 \leq u \leq 2(x - 1)$. The value v still remains a single precision digit.

The algorithm for this is outlined below:

INPUT: positive integer $y = (y_{t-1} y_{t-2} \dots y_1 y_0)_b$.

OUTPUT: $(y) (y) = y^2$ in base b .

1. For i count (from 0 to $(2t - 1)$); $w_i \leftarrow 0$.
2. For i count (from 0 to $(t - 1)$);
 - a. $(uv)_b \leftarrow w_{2i} + y_i y_i$, $w_{2i} \leftarrow v$, $k \leftarrow u$.
 - b. For j from $(i + 1)$ to $(t - 1)$;

$$(uv)_b \leftarrow w_{i+j} + 2y_j y_i + k$$
, $w_{i+j} \leftarrow v$, $k \leftarrow u$.
 - c. $w_{i+j} \leftarrow u$.
3. Return $((w_{2t-1}, w_{2t-2} \dots w_i w_0)_b)$.

From the step 2a, there comes up a situation where u can become larger than a single precision number. This situation is shown below:

Since w_{i+j} takes the value of v , $w_{i+j} \leq b - 1$.

If $k \leq 2(b - 1)$, then $w_{i+j} + 2y_j y_i + k \leq (b - 1) + 2(b - 1)^2 + 2(b - 1) = (b - 1)(2b + 1)$; which implies $0 \leq u \leq (2b - 1)$. The value of u in this case can exceed single precision. This had to be handled in the algorithm.

4. **Division for multiple precision integers:** the division operation is the most complicated out of all the arithmetic for multiple precision integers. The algorithm below calculates the quotient q and remainder in base b representation when u and v are divided where v is the denominator. The algorithm is outlined below.

INPUT: positive integers $u = (u_n \dots u_1 u_0)_b$, $v = (v_t \dots v_1 v_0)_b$ where $n \geq t \geq 1$, $v_t \neq 0$.

OUTPUT: the quotient, $q = (q_{n-t} \dots q_1 q_0)$ and remainder $r = (r_t \dots r_1 r_0)_b$ so that $u = qv + r$, $0 \leq r \leq v$.

1. For j count (from 0 to $(n - t)$); $q_j \leftarrow 0$.
2. While $(u \geq vb^{n-t})$; do $q_{n-t} \leftarrow q_{n-t} + 1$, $u \leftarrow u - vb^{n-t}$.

3. For i count (n down to $(t + 1)$); do
 - a. If $u_i = v_t$, set $q_{i-t-1} \leftarrow b - 1$; else, set $q_{i-t-1} \leftarrow \lfloor (u_i b + u_{i-1}) / v_i \rfloor$.
 - b. Test the condition $(q_{i-t-1}(v_i b + v_{i-1}) > u_i b^2 + u_{i-1} b + u_{i-2})$; perform $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
 - c. $u \leftarrow u - q_{i-t-1} v b^{i-t-1}$.
 - d. If $u < 0$, set $u \leftarrow u - v b^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
4. $r \leftarrow u$.
5. Return (q, r) .

This algorithm was implemented using two different functions to make it more efficient. The first function, `spDivide()`, returns the high digit of the quotient, q . The second function, `mpShortDiv()`, returns the value of the remainder, r . Normalization is ensured in binary by left-shifting the binary form of u and v .

5.3.2 Left-to-right binary exponentiation algorithm

This method is one of the basic techniques used for exponentiation. The general idea behind the algorithm and a walk-through for it are discussed next. The function performing this logic is outlined below:

INPUT: g (a member of finite elements) and a positive integer $e = (e_t e_{t-1} \dots e_1 e_0)_2$.

OUTPUT: g^e .

1. $A \leftarrow 1$.
2. For i count (from t down to 0):
 - a. $A \leftarrow A \times A$.
 - b. If $e_i = 1$, then $A \leftarrow A \times g$.
3. Return (A) .

This can be better understood when thinking about the computational efficiency of the algorithm. If one would consider the bit-length of the binary representation of e , and the number of 1s in e would also be noted, one would discover that the

algorithm performs the squaring of g the bit-length number of times and multiplication of g onetime less than the number of 1s available in e . The advantage of using this method is that multiplication is always with the fixed value g . For a g with a special structure, multiplication becomes easier than multiplying two arbitrary elements.

Other functions that were used and their functionalities are listed in table 2 below.

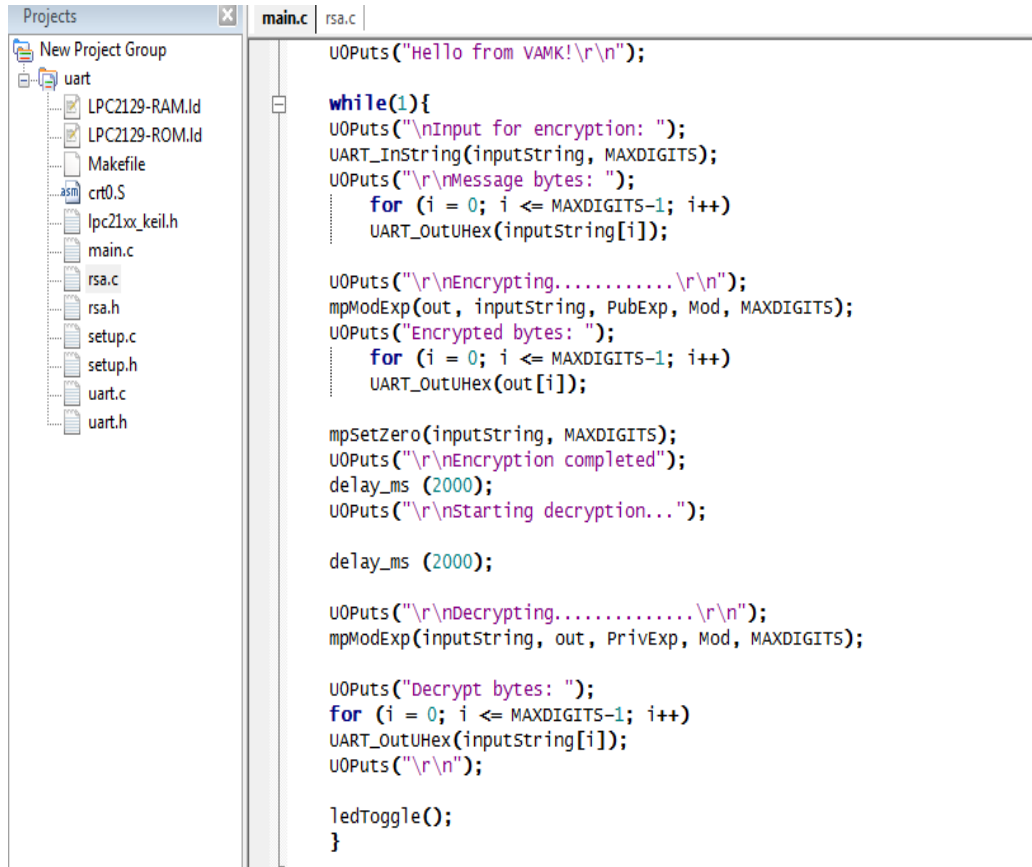
Table 2. Function names and their functionalities.

FUNCTION NAMES	FUNCTIONALITY
spMultiply()	Optimizing for 8 bits architecture.
mpSetEqual()	Equalizes the sizes of two arrays.
mpSetZero()	Initializes all array members to zero.
mpSetDigit()	Sets a multiple precision digit to single precision digit.
mpSizeof()	Returns the size of significant bits in an array.
mpShiftLeft()	Left-shifts a binary number by required spaces left and pads with 0s.
mpShiftRight()	Right-shifts a binary number by required spaces left and pads with 0s.
moduloTemp()	Returns remainder r of modular arithmetic.

5.4 Compilation and code download

For the function used in implementing this project, the encryption process works by taking in the public-key e , modulo number n , the number of bits and creating also a space for handling the output ciphertext. Calling this function returns the ciphertext, while a call to this function with the ciphertext and plaintext position reversed, decrypts the data. Figure 10 shows a code snippet in the `main()` function for encrypting and decrypting. Figure 11 shows the microcontroller being programmed using WinARM. For the implementation on this MCU, 512-bits key-length was used. The key size was chosen considering the type of device available for this project. Decryption time was one of the considerations taken into account be the key-length size was decided. While 512-bits key-length is not considered very safe these days, for this project this sufficiently addresses the purpose of this project. The requirement was to be able to communicate between two MCUs with the data encrypted on one side and decryption of encrypted data on the other side.

Encryption and decryption sequence calls the function `mpModExp()`, which implements the binary left-to-right exponentiation algorithm for calculating RSA encryption. This is logical because both sequence use the same algorithm. The first difference is that encryption takes in the plaintext and returns the ciphertext, while decryption takes in the encrypted ciphertext and returns the decrypted plaintext. The second difference, which really affects the time of execution of both sequence is that, encryption uses the public key e , which is of a shorter length while decryption sequence uses the private key d , which is of a similar length with the modulo n . the effects are easily noticeable in the execution time of both sequence.



```

Projects | main.c | rsa.c |
└─ New Project Group
   └─ uart
      ├── LPC2129-RAM.ld
      ├── LPC2129-ROM.ld
      ├── Makefile
      ├── crt0.S
      ├── lpc21xx_keil.h
      ├── main.c
      ├── rsa.c
      ├── rsa.h
      ├── setup.c
      ├── setup.h
      ├── uart.c
      └── uart.h

main.c
UOPuts("Hello from VAMK!\r\n");

while(1){
    UOPuts("\nInput for encryption: ");
    UART_InString(inputString, MAXDIGITS);
    UOPuts("\r\nMessage bytes: ");
    for (i = 0; i <= MAXDIGITS-1; i++)
        UART_OutUHex(inputString[i]);

    UOPuts("\r\nEncrypting.....\r\n");
    mpModExp(out, inputString, PubExp, Mod, MAXDIGITS);
    UOPuts("Encrypted bytes: ");
    for (i = 0; i <= MAXDIGITS-1; i++)
        UART_OutUHex(out[i]);

    mpSetZero(inputString, MAXDIGITS);
    UOPuts("\r\nEncryption completed");
    delay_ms (2000);
    UOPuts("\r\nStarting decryption...");

    delay_ms (2000);

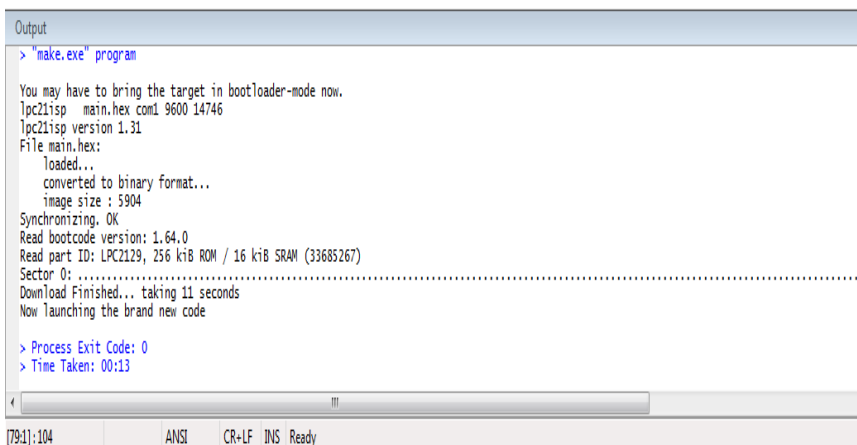
    UOPuts("\r\nDecrypting.....\r\n");
    mpModExp(inputString, out, PrivExp, Mod, MAXDIGITS);

    UOPuts("Decrypt bytes: ");
    for (i = 0; i <= MAXDIGITS-1; i++)
        UART_OutUHex(inputString[i]);
    UOPuts("\r\n");

    TedToggle();
}

```

Figure 12. Calling encryption and decryption sequence.



```

Output
> "make.exe" program

You may have to bring the target in bootloader-mode now.
lpc21isp main.hex com1 9600 14746
lpc21isp version 1.31
File main.hex:
  loaded...
  converted to binary format...
  image size : 5904
Synchronizing. OK
Read bootcode version: 1.64,0
Read part ID: LPC2129, 256 kiB ROM / 16 kiB SRAM (33685267)
Sector 0: .....
Download Finished... taking 11 seconds
Now launching the brand new code

> Process Exit Code: 0
> Time Taken: 00:13

[79:1]:104  ANSI  CR+LF  INS  Ready

```

Figure 13. Programming the MCU.

Programming the MCU with the project took about 15 seconds in total which already shows that the overhead of the algorithm is quite time consuming.

Running the algorithm for testing clearly shows the effect of the difference in key-sizes between encryption and decryption. As mentioned earlier, the private key type being the same size as the modulo number makes the computing of the ciphertext to that power quite time consuming for any type of computer architecture. In the next section, the result of the implementation will be discussed.

6 TESTING FOR RSA AND PROBLEMS

The project was completed successfully with both encryption and decryption being implemented. The only drawback perhaps would be the availability of only one MCU for the whole project. The unavailability of the other MCU, however, did not affect the ability to test the performance of the system. This was made possible with the use of brayterm which serves as a means of viewing the encrypted and corresponding decrypted data or plaintext, however the case may be.

6.1 Testing

Testing of the project was carried out after first programming the MCU and then changing it from the bootloader mode by disconnecting the jumper for the J_RST and BSL pins on the device. A thing to note about the device is that it is 16-bits based. This has also contributed to help speed up the process. The output of the program can be viewed only after connecting it to BrayTerm. With BrayTerm, the user interface is very friendly and as such easily configurable. The connection parameters were chosen based on previously configured data rate parameters when activating the UART0 channel on the MCU. The preconfigured values are 9600 bps for the data rate and for the data format, 8-N-1, which indicates, no start bits, 8 data bits, no parity bit and 1 stop bit.

The figure below shows the setup of BrayTerm showing selected parameters used for testing the project.

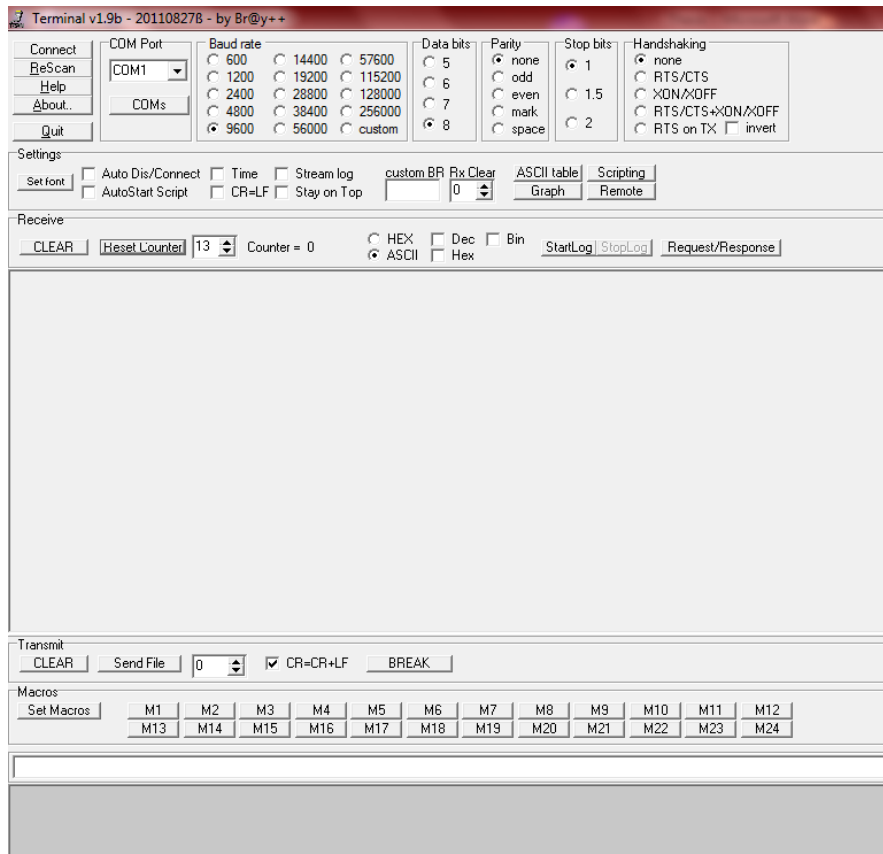


Figure 14. BrayTerm with data transfer settings.

As can be seen from the figure, the COM port used was COM1, the baud rate was set to 9600 bps. Data bits, 8, no parity bit selected, 1 stop bit and no hand shaking measure between the MCU and BrayTerm was used.

Connecting the MCU unit can be achieved by just clicking the connect button on the interface. This can be seen from the first figure shown below. This can be confirmed from the connected text at the bottom left corner of the interface. In addition, the connect button has been replaced with the disconnect button. Testing the project requires that it be connected and the run for execution to commence. This device can be put in run mode by pressing the reset button on the device. The LED on the board was programmed to show each encryption and decryption completion by alternating between switching on and off. A delay function was also used to slow down the execution so that the start of the decryption. After the program on the device has started to run, it transmits a welcome message and then requires a user input, which in this case serves as the plaintext being encrypted. This is shown in the other figure below.

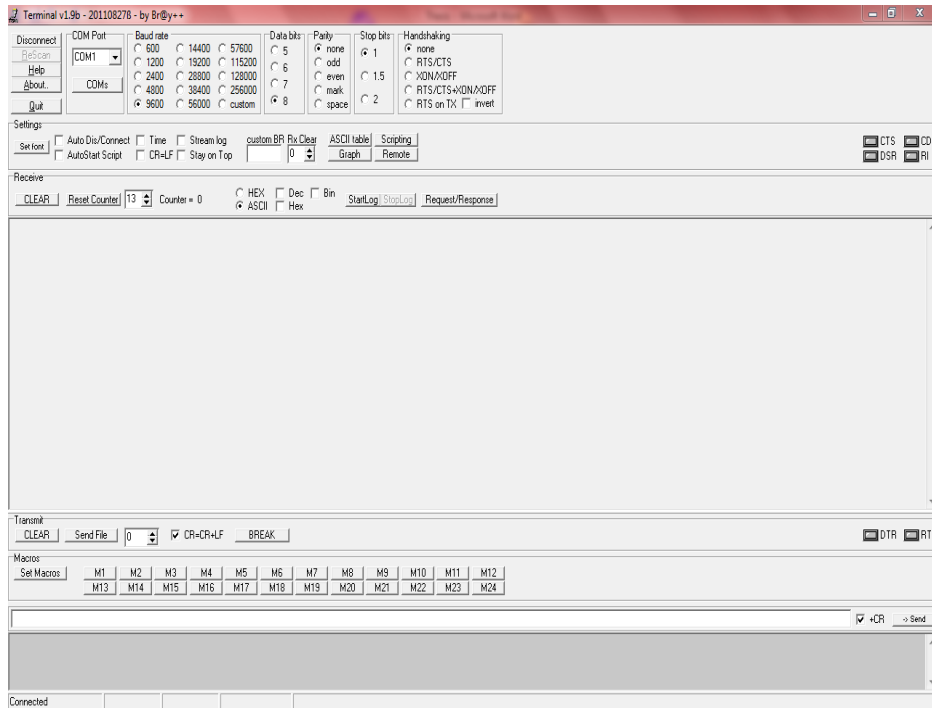


Figure 15. BrayTerm connected to the MCU.

The next figure shows the program running on the MCU. In this figure, the user input is being awaited.

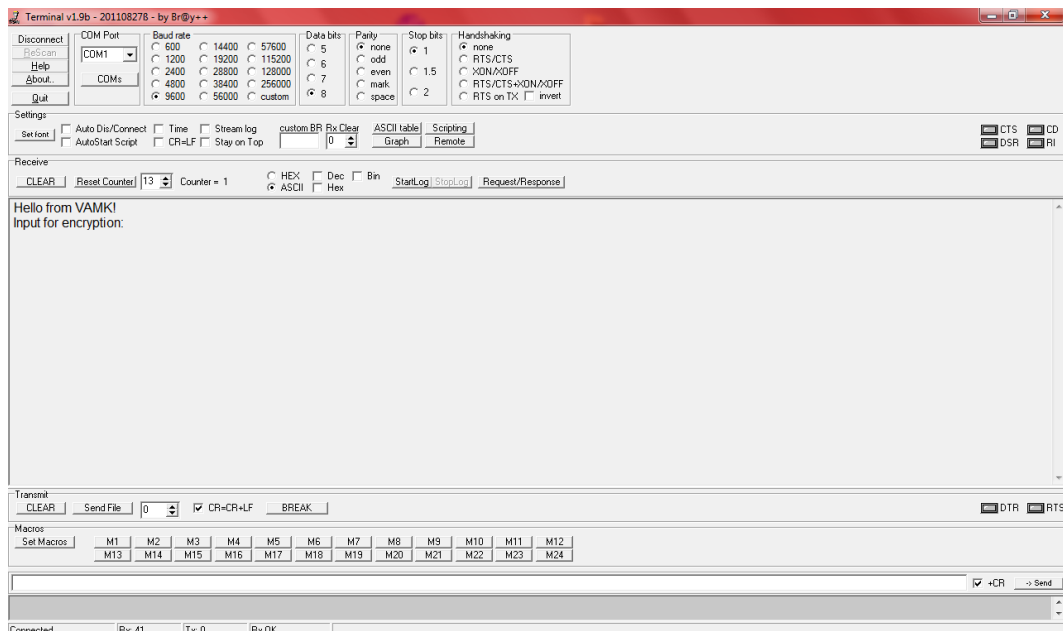


Figure 16. Code running on the MCU.

After the user input is confirmed by hitting the return key, the MCU then starts to encrypt the input plaintext into a cipher text. This is seen from the input word 'hello'. This is shown in the next figure.

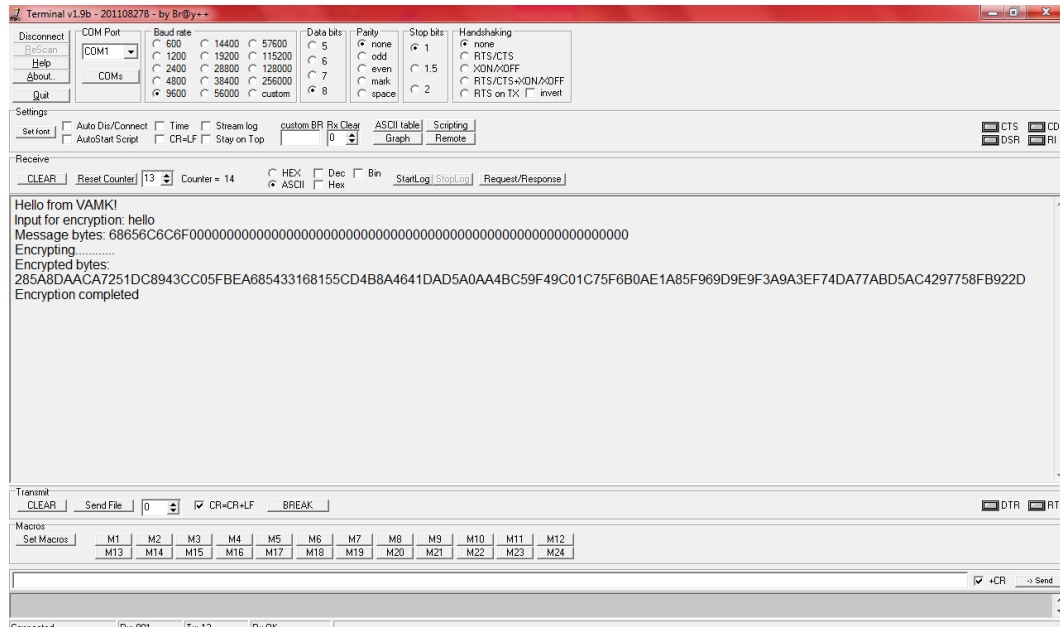


Figure 17. Ciphertext for 'hello' after encryption.

The decryption is done using the ciphertext derived from encryption based on the RSA algorithm. As can be seen, the ciphertext length is up to 512-bits used for the value of modulo n . Something that should be noted about decryption is that it deals in whole with 512-bits of ciphertext and then uses a modulo and the private exponent d , with the same bit length. That considerably slows down the speed of computing the plaintext from ciphertext. Figure 15 below shows decryption part of the project.

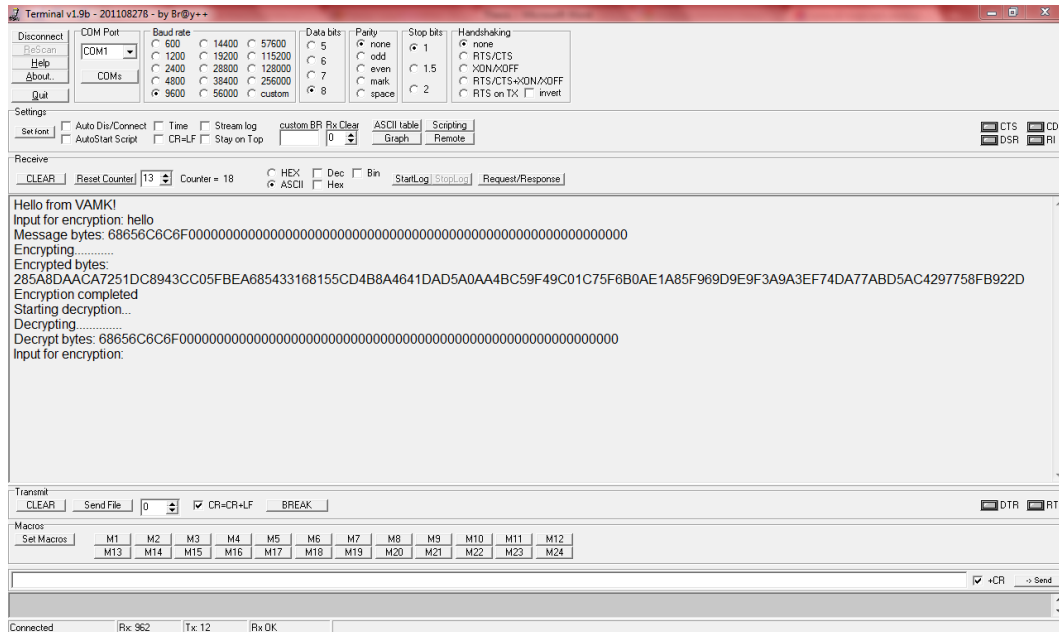


Figure 18. Decrypted ciphertext for ‘hello’.

From the figure above it can be seen that the decrypted data corresponds to the plaintext hello. The decrypted text is displayed in hexadecimal format using the ASCII conversion method. The decryption time was at an average of 9 seconds each time, which is about 4 times the time required to encrypt the same text or data. The reason for this has been discussed in detail in the previous chapter.

6.2 Problems

The problems encountered during the course of completing this project were mostly based on the device of choice. The main reason for this is the lack of experience with the use of the platform. A lot of time was spent just trying to figure out how to make the device work and generally just testing that all part of it are still fully functional. A lot of research time was lost based on this reason.

The compiler choice also did not help as it is, in my opinion, very primitive. It has to be totally configured to work manually and anyone without any prior knowledge of compilers may find it almost impossible to work with. The support for the compiler was also very limited, perhaps because it was developed by

enthusiasts working on the ARM processor at the time. The choice was perhaps mostly because other components such as a JTAG connector would have been required for connecting to other platforms.

Another challenge faced was in being able to adapt the multiple-precision arithmetic library so as to work with the MCU of choice. This also took some time for research but eventually was worth it.

7 CONCLUSION AND SUGGESTIONS

7.1 Conclusion for RSA

After the completion of the project, various concepts of security, under the specific field of data integrity, authentication and data confidentiality have been extensively discussed. In particular, asymmetric-key (public-key) cryptography, specifically RSA algorithm has been delved into. The known issues, advantages, implementation, and drawbacks as a result of its algorithm have also been discussed. As a result of this, one can deduce the importance of the RSA scheme in daily applications.

One major advantage that the keeps the algorithm relatively safe is its basis on factoring of big numbers which is still something very difficult to achieve, even with the availability of processing power needed to achieve it. An RSA scheme is said to be relatively safe if the factoring of the algorithm modulo is infeasible. The last key-length that was broken was the 512-bits key-length. Even that took about 7 months and a lot in terms of finances and processor power before this was achieved. So one may say that it is still relatively safe to use the key-length (512-bits), but that will have to depend on the type of data to be kept hidden and for how long it is going to be kept hidden.

However, currently 1024-bits key-length is the minimum advisable key-length to be used. If one would take for example a bank that needs to authenticate its user with the use of passcodes so as to be able to confirm his or her identity. The waiting time for authentication to be concluded is clearly a worthy price to pay for the verification of user data. Talks are currently going on about when the new minimum key-length will be permanently changed from the current length 1024-bit to 2048-bits key-length.

As a result, RSA algorithm is still a relevant protocol in the security field and will remain so for a long time to come. It still boasts its (almost) irreversible prime

number factorization. While this is true, it does not however, help with the timing for the decryption part of the process. If the applications using RSA algorithm continue to require an authentication or data integrity as one of its important feature, then it is still a rather acceptable price to pay to enable such operations to be properly functional. There is an Eve in every part of the world waiting at any opportunity to make a move. In my opinion, therefore one should be able to draw their conclusions based on the type of data security feature required for such an application to be secure.

7.2 Conclusion for CRC

The need for detection of errors dates back a very long time and as such several methods of detection has been designed. CRC-32 is one of the cyclic codes that were designed to help solve this kind of problem that may arise in data transmission and reception. CRC-32 has proven to be quite efficient in detecting errors of different types and under different data transfer rates. It is known to detect single-bit errors, odd numbered errors, two isolated single-bit error types and burst error types. Based on this feature the only problem with the use of CRC-32 is getting the right generator polynomial. This is of utmost importance as it helps to ensure that all error in transmission is detected and subsequently retransmitted or handled as desired.

As a result, CRC-32 can be said to be considerably reliable when compare with other methods of data error detection algorithms. An added advantage with the use of CRC-32 is that it comes with most applications and the results can be checked almost immediately as there are several means of doing so available, free to use, on the internet. Another great advantage it possesses is the fact that it can be easily implemented in both hardware and software compared with other protocols. Generally, cyclic codes are exceptionally fast in hardware implementation.

In conclusion, one can understand why cyclic codes are used extensively in many networks.

7.3 Suggestions and future developments

The consideration that data can be of different format and transferred on various devices, some wired, others wireless makes it possible to suggest expanding its use to other devices such as the mobile telephone and perhaps someday, it could help secure wireless data between communicating computers. Although this may be challenging, I do believe that it is possible. This is so because new discoveries are being made from various kinds of researches currently going on around the world.

REFERENCES

/1/ Forouzan, Behrouz A. (2007). Data Communications and Networking. Fourth edition. Singapore. McGraw-Hill.

/2/ RSA website. Accessed 14.11.2011.
<http://www.rsa.com/node.aspx?id=2760>

/3/ Olimex Ltd website. Accessed 14.11.2011. Available under development and tools. <http://www.olimex.com/dev/index.html>

/4/ ARM-Projects website. Accessed 14.11.2011.
http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/

/5/ Menezes, Alfred J., van Oorschot, Paul C. and Vanstone, Scott A. (1996). Handbook of Applied Cryptography (Discrete Mathematics and Its Applications). First edition. CRC Press.

/6/ NXP semiconductors website. LPC2000 User Manual. Accessed 10.10.2011
http://www.nxp.com/acrobat_download/usermanuals/UM_LPC21XX_LPC22XX_2.pdf, Page 79

/7/ Initialization code/hints for the LPC2000 family. Application note for LPC2000 family.