Firmware Deployment of Strong Encryption: An Investigation and Implementation

John Ronan

12th July 2001

Waterford Institute of Technology
Department of Physical and Quantitative Sciences

Firmware Deployment of Strong Encryption: An Investigation and Implementation

A thesis
submitted for the
Degree of Master of Science to
the National Council of Educational Awards
by

John Ronan

To my parents, grandparents and Deirdre.

Acknowledgements: Eamonn de Leastar, Dr Willie Donnelly, Dr Martin Hayes, Michael O'Foghlu, Jimmy McGibney, Michael McCarthy, Kristian Walsh, and my colleagues in the Telecommunications Software Systems Group.

Abstract

Software encipherment and an equivalent firmware implementation is the general scope of this work. Current cryptographic implementations rely on software running under general purpose, often multi-user, operating systems alongside a horde of untrusted and possibly malicious applications. Additionally there are other threats to security, such as that posed by "crackers" or government agencies listening in to network traffic. This work addresses one method for minimising these risks.

A framework is presented for implementation of a cryptographic coprocessor, capable of securely performing encryption, decryption and key management. To achieve maximum performance and security the algorithm is instantiated in firmware. This achieves superior performance to pure software implementations.

This work also examines various issues related to the choice of algorithms out "in the wild" today, how they operate, and how they can be used for different purposes. It shows how a software algorithm can be brought into the hardware/firmware domain and deployed as effectively therein. The framework implemented retains all the functionality of the pure software solution while gaining significantly in performance. This approach is also significantly more secure, as a firmware implementation is not open to the standard security workarounds and breaches commonly applied to software solutions.

As part of the project a corresponding software implementation has been verified against the firmware equivalent, and an assessment made on the relative merits of both approaches with respect to speed, security, and ease of implementation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction & Background

Computer Security

.

Cryptography

.

Thesis Structure

1.1 Computer Security

"We already are seeing the first wave of deliberate cyber attacks — hackers break into government and business computers, stealing and destroying information, raiding bank accounts, running up credit card charges, extorting money by threats to unleash computer viruses." — President Bill Clinton.[Gol99]

While it is very laudable for the President of the U.S.A. to say these words, the reality of the situation is that computer security is woefully neglected in the general computing world (the damage that the "I Love U" virus was able to do being a case in point). Today's computer environment consists of physically distributed personal computers and workstations, connected by networks. Such an environment is inherently difficult to protect.

Arbaugh et al. have pointed out that the integrity of the operating system's kernel cannot be trusted because malicious code can be injected in the bootstrap process[AFS97]. For example, typical PCs can be booted from CDROM, floppy disks or over the network, thus allowing arbitrary code to be run. Some do allow an administrator to set a BIOS (Basic Input Output System) password thus allowing the boot process to continue only if the correct password is entered. However it is not to difficult to remove the password by resetting the BIOS[FW99].

The advantage of software only implementations is that they are inexpensive and easy to deploy. The disadvantage of these implementations is that they provide a very low level of protection for cryptographic variables and, unfortunately, this low level of security is unlikely to change in the future. Vulnerable software ranges from the applications right down to the operating system. Some examples:

1. Operating systems
   - scanning memory for encryption keys in Windows NT[SS98].
   - Windows NT jump table patching[RC97].
   - erroneous permissions of DLL cache on Windows NT 4.0[dil99].
2. Basic system programs
   - intercepting shell messages[Hei98].

- buffer overflow in `df`, `eject`, `login`, etc., in IRIX[CER97].

3. Daemons

   - buffer overflow in wu-ftpd[CER99c].
   - buffer overflow in IIS web server[CER99b].
   - bugs in sample files in IIS[wel99].

4. Applications

   - buffer overflow in sendmail[CER99a]

5. Network protocols

   - flaws in ICMP Router Discover Protocol allowing man-in-the-middle attack[sil99a]

6. Security software

   - poor encrypting of shellock[ml99].
   - password appraiser sending Windows passwords in the clear on the Internet[mud99]

Such vulnerabilities can be quite serious. They may yield control of the machine to an opponent, crash the system or leak critical information (cryptographic keys). Unfortunately, the main problem is that for most people there is no problem:

> "If only more people were aware of the problem. Then I wouldn't have to start off by having to convince them they needed to do something - we could get right down to solving problems"[Ran99]

Users see it as normal for software to contain bugs. Since program correctness is difficult to achieve[Wie93], and as long as features are the major selling point for products, buggy and insecure systems will be the normal state of affairs[CF98]. The simple fact is that no mainstream operating system has the features required to implement proper security policies[LSM$^+$98], and the work involved in performing the necessary redesign from scratch with an operating system containing millions of lines of code would make it prohibitive[Gut00].

This seems to be painting a very bleak picture for someone trying to provide "secure" services to users. How do we address this problem? One approach is to move the cryptography away from the insecurity. In other words, though a user may be running the latest up to date operating system with all the latest bugs and un-patched buffer overflows, none of these can subvert the cryptography. Hence we move the crypto operations and the critical cryptographic keys into hardware.

In [Gut00] Gutmann does this by using an embedded Linux solution running on a RISC processor. At the opposite end of the spectrum, Itoi [Ito00] uses IBM's 4758 cryptographic co-processor [chp] to secure a Kerberos Key Distribution centre. Not one of these approaches use a firmware implementation, they involve downloading a conventional program into an on-board RISC or CISC processor.

1.1.1 Goals

The goals of this work are:

- Investigate the area of cryptographic algorithms, and decide on one algorithm for further investigation.
- Implement this chosen algorithm both in software and firmware.
- Compare the differences in implementation and performance, of the different domains.
- Suggest possible improvements of the different implementations based on experience gained.

1.2 Cryptography

1.2.1 A Brief History

"On a day nearly 4000 years ago, in a town near the Nile, a master scribe sketched out the hieroglyphs that told the story of his lord's life — and in doing he opened the recorded history of cryptology". His was not a system of secret writing as the modern world knows, he merely used some unusual hieroglyphics here and there in place of more ordinary ones. The intention was not to make it hard to read the text, but to impart a dignity and authority to it, perhaps in the same way that a government proclamation will spell out "In the year of Our Lord Nineteen Hundred and Ninety Nine" instead of just "1999." The anonymous scribe may also have been demonstrating his knowledge for posterity. Thus the inscription was not secret writing, but it incorporated one of the essential elements of cryptography: a deliberate transformation of the writing. It is the oldest text known to do so.

In its first 4000 years, Cryptology did not grow steadily, it rose and grew independently in many places, and in most of them it also died with its civilisations. In other places, it survived, embedded in a literature, and from this the next generation could climb to higher levels. But progress was slow and inconsistent. More was lost than retained. Much of the history of cryptology of this time is a patchwork of unrelated items, sprouting, flourishing, withering. Only toward the Western Renaissance does the accreting of knowledge begin to build up a momentum. Modern cryptologywas born among the Arabs. They were the first to discover and write down the methods of breaking codes and ciphers, the word "cipher" itself, comes from the Arabic word for mathematics[Kah96].

1.2.2 The Fundamental Idea of Cryptography

It is possible to transfer or encipher a message or *plaintext* into "an intermediate form" or *ciphertext* in which the information is *present* but *hidden.* Then we can release the transformed message (the ciphertext) without exposing the information it represents.

By using different transformations, we can create many different ciphertexts for the exact same message. So if we select a particular transformation "at random," we can hope that anyone wishing to expose the message ("break" the cipher) can do no better than simply trying all available transformations (or on average half) one-by-one. This is a brute force attack.

A cryptographic "key" is an item of data which is shared among the communicating parties to facilitate cryptographic techniques. This data may include public or secret keys, initialisation values, and additional non-secret parameters. The are generally classified according to Table 1.1. Hence a "symmetric" algorithm uses a single key for encryption and decryption, and similarly, an "asymmetric" algorithm uses a public and private key for decryption and encryption respectively.

The difference between intermediate forms is the *interpretation* of the ciphertext data. Different ciphers and different keys will produce different inter-

Term	Meaning
private key, public key	paired keys in an asymmetric cryptographic system
symmetric key	key in a symmetric (single-key) cryptographic system
secret	adjective used to describe private or symmetric key

Table 1.1: Key Classification

pretations (different plain-texts) for the exact same ciphertext. The uncertainty of how to interpret any particular ciphertext is how information is "hidden."

Naturally, the intended recipient needs to know how to decipher the intermediate form back into the original message, and needs to receive the decryption key in a secure manner. The latter is the key distribution problem.

As an example, while attending Secondary School is was common practice to encipher messages to pass to friends in "study". Key Distribution was easy: drop the key into the other party's desk while entering study. Unfortunately, in a real system, key distribution is not so such an easy task, and will be dealt with later.

By itself, ciphertext is literally *meaningless*, in the sense of having no one clear interpretation. In so-called perfect ciphers, *any* ciphertext (of appropriate size) can be interpreted as *any* message, just by selecting an appropriate key. In fact, any number of *different* messages can produce *exactly the same ciphertext*, by using the appropriate keys. In other ciphers, this may not always be possible, but it must always be considered. To attack and break a cipher, it is necessary to somehow confirm that the message we generate from ciphertext is the message which was originally sent (i.e., we know the plaintext).

1.2.2.1 The Single Transformation

The single transformation cipher is a simple substitution cipher: a streaming or repeated letter-by-letter application of the same transformation. That "transformation" is the particular arrangement of letters in the second column, for example, a simple permutation of the alphabet. There can be *many* such arrangements. But in this case the key *is* that particular arrangement. We can copy it and give it to someone and then send secret messages to them. But if that sheet is acquired – or even copied – by someone else, the enciphered messages would be exposed. This means that we have to keep the transformation secret.

One of the earliest descriptions of this method appears in the *Kāma-sūtra*, a text written in the fourth century AD by the Brahmin scholar Vātsyāyana, based on manuscripts dating back to the fourth century BC. The *Kāma-sūtra* recommends that women should study 64 arts, such as cooking, dressing, massage and the preparation of perfumes. Number 45 on the list is *mlecchita-vikalpā*, the art of secret writing, advocated in order to help women conceal the details of their liaisons. One of the first documented uses of a substitution cipher for military purposes appears in Julius Caesar's *Gallic Wars*. The substitution replaced Roman letters with Greek letters, rendering the message unintelligible to the enemy ([Sin99, pp9-10]).

1.2.2.2 Many Transformations

Now suppose we have a full notebook of lined pages, each of which contains a different arrangement in the second column and each page is numbered. Now we just pick a number and encipher the message using that particular page. That number thus becomes our key, which is now a sort of numeric shorthand for the full transformation. So even if the notebook is exposed, someone who wishes to expose our message must try about half of the transformations in the book before finding the right one. Since exposing the notebook does not immediately expose our messages, maybe we can leave the notebook unprotected. We also can use the same notebook for messages to different people, and each of them can use the exact same notebook for their own messages to each other. Different people can use the same notebook and yet still cipher messages which are difficult to expose without knowing the right key. Note that there is some potential for confusion in first calling the transformation a key, and then calling the number which selects that transformation *also* a key. But both of these act to select a particular ciphertext construction from among the many. There are only two of the various kinds of "key" in cryptography.

1.2.2.3 Weak and Strong Transformations

The simple substitution used in the above cipher is very weak, because it "leaks" information: the more often a particular plaintext letter is used, the more often the associated ciphertext letter appears. And since language uses some letters more than others, simply by counting the number of times each ciphertext letter occurs we can make a good guess about which plaintext letter it represents. Then we can try our guess and see if it produces something we can understand. It usually does not take too long before we can break the cipher, even without having the key. In fact, we develop the ultimate key (the enciphering transformation) to break the cipher.

A "real" cipher will have a far more complex transformation. For example, the usual 64-bit block cipher will encipher 8 plaintext letters at the same time, and a change in any one of those letters will change all 8 letters of the resulting ciphertext. This is still simple substitution, but with a huge alphabet. Instead of using 26 letters, a 64-bit block cipher views each of 2^{64}different block values as a separate letter, which is approximately 1.8×10^{19} "letters."

1.2.2.4 Key-space

Suppose we have 256 (0-255) pages of transformations in the notebook; this means there are exactly 256 different keys we can select from. If we write the number 256 in binary we get "100000000"; here the leftmost 1 represents 1 count of 2^8. Or, we can compute the base 2 logarithm by first taking the natural log of 255 (about 5.545) and dividing that by the natural log of two (about 0.693); this result is also 8. So we say that having 256 key possibilities is an "8 bit" key-space. If we choose one of the 256 key values at random, and use that transformation to encipher a message, someone wishing to break our cipher should have to try about 128 deciphering operations before happening upon the correct one. The effort involved in trying, on average, 128 deciphering operations (a brute force attack) before finding the right one, is the design strength of the cipher.

If our notebook had 65,536 pages or keys (instead of just 256), we would have a "16 bit" key-space. Notice that this number of key possibilities is 256 *times* that of an "8 bit" key-space, while the key itself only has 8 bits *more* than the "8 bit" cipher. The strength of the "16 bit" cipher is the effort involved in trying, on average, 32,768 deciphering operations before finding the right one.

The idea is the same for a modern cipher: We have a machine which can produce a huge number of transformations between plaintext and ciphertext, and we select one of those transformations with a key value. Since there are many, many possible keys, it is difficult to expose a message, even though the machine itself is not secret. And many people can use the exact same machine for their own secrets, *without* revealing those secrets to everyone who has such a machine.

1.2.2.5 Digital Electronic Ciphering

One of the consequences of having a computer for ciphering is that it operates very, very fast. This means that someone trying to break the cipher (the "cryptanalyst"), can try a lot more possibilities than they could with a pen and paper.

Assume that we have a perfect (symmetric key) algorithm. By perfect, we assume that there is no better way to break the crypto-system other than trying every possible key in a brute-force attack.

To launch this attack, a cryptanalyst needs a small amount of ciphertext and the corresponding plaintext; a brute-force attack is a known-plaintext attack[1]. For a block cipher (see §1.2.11), the cryptanalyst would need a block of ciphertext and corresponding plaintext: generally 64 bits. Assuming a computer can try a million keys a second, it will take 2285 years to find the correct key[Sch96, p151]. If the key is 64 bits long, then it will take the same computer about 585,000 years to find the correct key among the 2^{64} possible keys. If the key is 128 bits long, it will take 10^{25} years. The universe is only 10^{10} years old, so 10^{25} years is a long time. With a 2048-bit key, a million million-attempts-per-second computers working in parallel will spend 10^{597}years finding the key. By that time the universe will have long collapsed or expanded into nothingness.

A brute- force attack is tailor made for parallel processors. Each processor can test a subset of the key-space. The processors do not have to communicate among themselves; the only communication required at all is a single message signifying success. There are no shared memory requirements. It is easy to design a machine with a million parallel processors, each working independent of the others.

In 1994, Michael Wiener decided to design[Wei93][Wei94] a brute-force cracking machine. He designed the machine for the Data Encryption Standard (DES)[2][oST93] algorithm, but the analysis holds for almost any symmetrical key algorithm. He designed specialised chips, boards, and racks. He estimated prices and discovered that for $1 million, someone could build a machine that could crack a 56-bit DES key in an average of 3.5 hours (results guaranteed in 7 hours). And that the price/speed ratio is linear. Table 1.2 generalises these

[1]By trying all possible keys, the ciphertext will eventually be decoded to the plaintext. Thus the confidentiality of all transmissions hinges on the choosing a key length that can be "cracked" in an "impossibly" long time.

[2]Known at the Data Encryption Algorithm by ANSI and DEA-1 by the ISO, has been a worldwide standard for 25 years. The standard UNIX password system uses DES with variations intended to discourage brute-force cracking. See [Sch96, pp265-294] for details.

Cost	40	56	64	80	128
\$100K	2 seconds	35 hours	1 year	70,000 years	10^{19} years
\$1M	.2 seconds	3.5 hours	37 days	7000 years	10^{18} years
\$10M	.02 seconds	21 minutes	4 days	700 years	10^{17}years
\$100M	2 milliseconds	2 minutes	9 hours	70 years	10^{16}years
\$1G	.2 milliseconds	13 seconds	1 hour	7 years	10^{15}years
\$10G	.02 milliseconds	1 second	5.4 minutes	245 days	10^{14}years
\$100G	2 microseconds	.1 second	32 seconds	24 days	10^{13}years
\$1T	.2 microseconds	.01 second	3 seconds	2.4 days	10^{12}years
\$10T	.02 microseconds	1 millisecond	.3 second	6 hours	10^{11}years

Table 1.2: Average Time Estimates for a Hardware Brute -Force Attack in 1995

numbers to a variety of key lengths. Moore's Law, however states: Computer processing power doubles approximately every 18 months.[3] This means that the costs go down roughly by a factor of 10 every 5 years; what cost \$1 million in 1995 will cost a mere \$100,000 today. Pipelined computers might do even better[HA94]. For 56-bit keys, these numbers are within the budgets of most large companies and many criminal organisations. The military budgets of most industrialised nations can afford to break 64 bit keys. Breaking an 80 bit key is still beyond the realm of possibility, but if trends continue, that will change within the next 25 years. In cryptography is is wise to be pessimistic.

Without special-purpose hardware and massively parallel machines, brute-force attacks are significantly harder. A software attack is about a thousand times slower than a hardware attack and is generally measured in MIPS-years: a million-instructions-per-second processor running for one year, which is about $3 \star 10^{13}$ instructions executed. A 200-MHz Pentium is about a 50-MIPS machine. Unfortunately the "MIPS-year" is often both miscalculated and misused. Silverman tries to clear up this confusion [Sil99b], demonstrating how the "MIPS-Year" can be applied as a measurement of the amount of effort required to break, and compare the "strength" of, cryptographic keys.

The real threat of a software-based brute-force attack is not that it is certain, but that it is "free". It costs nothing to set up a microcomputer to test possible keys whenever it is idle (In a typical educational institution for example all the student-accessible PCs are guaranteed to be idle for at least 10 hours day). If it finds the key — great; if it doesn't, then nothing is lost (arguments about the electricity usage not withstanding). It costs nothing to set up an entire network

[3]This increase in computing speed results mainly from the increasing miniaturisation of components.

to do that. In 1991 an experiment with DES used the collective idle time of 40 workstations to test 2^{34} keys in a single day[GO91]. At this speed it would take 4 million days to test all keys. If enough people try attacks like this, then someone somewhere will get lucky.

1.2.3 What Cryptography *Can* do

Potentially, cryptography can hide information while it is in transit or storage[4]. In general, cryptography can:

- Provide Secrecy
- Authenticate that a message has not changed in transit
- Implicitly authenticate the sender

Cryptography hides *words*: At most it can only hide *talking about* contraband or illegal actions. In countries with "freedom of speech", we normally expect crimes to be more than just "talk."

Cryptography can kill. Not in the same way as a knife can kill, but as part of a system or process. It could be argued that the Japanese Pacific fleet lost the battle at Midway because of cryptography. They assumed that their crypto-systems were unbreakable, which was not the case, as the Americans were reading their traffic. The American cryptanalysts were almost certain that the main attack would come at Midway (they had deciphered messages relating to "AF" which they were pretty sure was an indicator for Midway on a partially solved map grid) but they needed proof. So they sent out a plain language message saying that the fresh-water distillation plant on Midway had failed. Several days later the American cryptanalysts deciphered a Japanese intercept stating that "AF" was short of fresh water. Now they were sure where the attack would come [Kah96, p569].

Cryptography, in general, is defensive, and can *protect* ordinary commerce and ordinary people.

> In the face of the snowballing bigness of the institutions of globalised human life, we must reserve privacy rights explicitly so that we may misrepresent ourselves to those against whom we have no

[4]No matter how secure an algorithm, if it is part of a badly implemented "crypto-system" then it does nothing more than give the user a deluded sense of security.

other defence, against those for whom our name is but a label on data collected without our consent[Gee99].

Cryptography can hide *secrets*, either from others, or during communication. There are many good and non-criminal reasons to have secrets: certainly those engaged in commercial research and development have "secrets" they must keep. Professors and writers may want to keep their work private, until an appropriate time. Negotiations for new jobs are generally private, at least we might prefer that detailed discussions not be exposed. One possible application for cryptography is to secure on-line communications between work and home, perhaps leading to a society-wide reduction in commuting.

1.2.4 What Cryptography Can *Not* do

Cryptography can only hide information after it is encrypted and while it remains encrypted. But secret information generally does not start out encrypted, so there is normally an original period during which the secret is not protected. And secret information generally is not used in encrypted form[5], so it is again outside the cryptographic envelope every time the secret is used.

Secrets are often related to public information, and subsequent activities based on the secret can indicate what the secret is.

Cryptography simply cannot protect against:

- Informants.
- Undercover Spying.
- Bugs (in both system implementation and eavesdropping devices).
- Photographic evidence.
- Testimony.

It is a mistake to imagine that cryptography alone could protect most information against "Big Brother" or potentially of more grave concern, many "little brothers"[Gee99]. Cryptography is only a small part of the protection needed for "absolute" secrecy.

[5]P.G.P. the email encryption package has a mode of operation whereby the received, encrypted, email can be displayed on the screen, without being stored on disk in its un-encrypted form.

1.2.5 Problems with Keys

The physical key model reminds us of various things that can go wrong with keys:

- We can lose our keys.
- We can forget which key is which.
- We can give a key to the wrong person.
- Somebody can steal a key.
- Somebody can pick the lock.
- Somebody can go through a window.
- Somebody can break down the door.
- Somebody can ask for entry, and unwisely be let in.
- Somebody can get a warrant, then legally do whatever is required.
- Somebody can burn down the house, thus making the key irrelevant.

Even absolutely perfect keys cannot solve all problems, nor can they guarantee privacy. And how do we get a key to the correct place/person, are we sure that the person is who they say they are? Indeed, when cryptography is used for communications, generally at least two people know what is being communicated. So either party could reveal a secret:

- By accident.
- To someone else.
- Through third-party eavesdropping.
- As revenge, for actions real or imagined.
- For payment.
- Under duress.
- In testimony.

In summary, when it is substantially less costly to acquire the secret by means other than a technical attack on the cipher, cryptography has pretty much succeeded in doing what it can do.

1.2.6 Strength

Key-space alone only sets an *upper limit* to cipher strength; a cipher can be *much weaker* than it appears. An in-depth understanding or *analysis* of the design may lead to "shortcuts" in the solution. Perhaps a few tests can be designed, each of which eliminates vast numbers of keys[6], quite possibly reducing the effort required by the attacker (called the Opponent) by orders of magnitude such that a brute-force attack is now feasible. This process is called cryptanalysis.

We understand strength as the ability to *resist* cryptanalysis. But this makes "strength" a negative quality (the *lack* of any practical attack), which we cannot measure. We can *infer* the "strength" of a cipher from the best *known* attack. We can only hope that the Opponent does not know of something better.

Every user of cryptography should understand that all known ciphers are at least *potentially* vulnerable to some unknown technical attack[7]. And if such a break does occur, there is absolutely no reason that we would find out about it. However, a direct technical attack may be one of the least likely avenues of exposure.

1.2.7 System Design and Strength

Cryptographic design may seem as easy as selecting a cipher from a book of ciphers. But ciphers, are only part of a secure encryption system. It is common for a cipher system to require cryptographic design beyond simply selecting a cipher, and such design is much tricker than it looks.

The use of an unbreakable cipher does *not* mean that the encryption system will be similarly unbreakable. A prime example of this is the man-in-the-middle attack on public-key ciphers (see §1.2.12). Public-key ciphers *require* that one use the correct key for the desired recipient. The correct key must be known to cryptographic levels of assurance, or the key itself becomes the weakest link in the system; Suppose an Opponent can get us to use his key instead of the right one (perhaps sending a faked message saying "Here is my new key"). If he can do this to both ends, and also intercept all messages between them (which is conceivable, since Internet routing is *not* secure), the Opponent can sit "in the middle." He can decipher each message (now in one of his keys), then re-encipher that message in the correct user key, and send it along. So the users

[6]"Idea", the algorithm used in P.G.P., has a number of "weak" keys. [Sch96, p323][DGV94b]

[7]The "One-Time Pad" cipher has as many values in the key as the plaintext. This is the source of its *perfect* security. It is essential that no portion of the key *ever* be reused for another encryption (hence the name), otherwise current cryptanalysis can break the cipher.

communicate, and no cipher has been broken, yet the Opponent is still reading the conversation. Such are the consequences of system design error.

1.2.8 Cryptanalysis versus Subversion

Cryptanalysis is hard; it is often tedious, repetitive, and very expensive. Success is never assured, and resources are always limited. Consequently, other approaches for obtaining the hidden information (or the key) can be more effective.

Approaches other than a direct technical attack on ciphertext include getting the information by cunning, outright theft, bribery, or intimidation. The room or computer could be bugged, network sniffers could be installed on "rooted"[8] computers, secretaries subverted, files stolen, etc. Most information can be obtained in some way other than "breaking" ciphertext.

When the effort required to break the cipher greatly exceeds the effort required to obtain the same information in another way, the cipher is probably strong enough. And the mere fact that information has escaped does not necessarily mean that a cipher has been broken.

It is interesting to note that virtually every crypto-system invented before the 1940s was systematically overcome by applying Shannon's information theory of secrecy systems, first published in 1949[Sha49]. All of the systems had been broken piecemeal before that time, but for the first time, cryptanalysts had a general way to attack all crypto-systems. Only two systems remain impervious to this attack, the "One Time Pad" and Public Key Systems (detailed in §1.2.12).

1.2.9 Secret Ciphers

Although cryptanalysis might succeed even if the ciphering process was unknown, we would certainly expect that this would make the Opponent's job much harder. It can thus be argued that the ciphering process *should* remain secret. Military cipher systems are not actually published, although it would be foolish of the military not to assume that any competent Opponent will obtain this information through other channels. Military Ciphers are still designed with the following ideals which were deduced by Kerchoffs in 1883[Kah96, p235]:

[8]A underground term used by a "cracker" to describe a machine that has been successfully broken in to, and which is now under the control of the cracker and, more than likely, his cohorts.

1. The system should be, if not theoretically unbreakable, unbreakable in practice.
2. Compromise of the system should not inconvenience the correspondents.
3. The key should be rememberable without notes and be easily changeable.
4. The cryptograms should be transportable by telegraph.
5. The apparatus or documents should be portable and operable by a single person.
6. The system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain.

Over time, these requirements have been rephrased, and qualities that lie implicit have been made explicit. But any modern cryptographer would be very happy if any cipher fulfilled all six.

In commercial cryptography we normally assume that the Opponents will know every detail of the cipher (excluding the key, of course). There are several reasons for this:

- It is common for a cipher to have unexpected weaknesses which are not found by its designers. If the cipher design is kept secret, it cannot be examined by other parties, and so weaknesses may not be publicly exposed, and could be exploited in practice.
- If a cipher itself is a secret, then that secret is increasingly compromised by making it available for use. For a cipher to be used, it must be present at various locations, and the more widely it is used, the greater the risk of the secret being exposed. So whatever advantage there may have been in keeping the cipher-mechanism secret will be lost, and the Opponents eventually will have the same advantage as they would have had from public disclosure.[9]

There is another level of secrecy here, and that is the trade secrecy involved with particular software designs. Few large companies are willing to release source code for their products without some serious controls. While the crypto routines themselves presumably might be patented, releasing that code alone

[9]Only now the cipher designers may comfort themselves with the dangerous delusion that they have an advantage over their opponents.

probably would not support a thorough security evaluation as the years of research and conclusions from that research have not been made available for scrutiny. People are fallible, thus conclusions taken from research may be in error. Source code might reasonably be made available to customers under a non-disclosure agreement, but this will not satisfy everyone. And while it might seem nice to have all source code available for free, this will not support an industry of continued cipher design and development. However the Advanced Encryption Standard (AES, §1.2.14) program is hoping to address this issue by standardising on an algorithm for the future that is free, and that has received as much exposure and cryptanalysis as possible. As the minimum key length of AES is 128 bits, it is envisaged that AES will be "Secure Enough"[BDR+96] for everyone's needs, assuming no better attack than brute force (Figure 1.2). But who knows, maybe some of the more powerful security agencies have already compromised all of the candidates.

1.2.10 Hardware vs Software Ciphers

Currently, most ciphers are implemented in software, that is, by a program of instructions executed by a general-purpose computer. Normally, it is cheaper to implement an algorithm in software, but hardware can run faster, and nobody can change its operation through the use of software. Of course, there are levels to hardware, from microprocessors (which thus require significant interface software) to external boxes with communications lines running in and out. Anyone trying to decide whether to deploy a software or hardware implementation has to consider the security risks, their expense along with the following issues (amongst others):

- Software, especially in a multi-user system, is almost completely insecure ([LSM+98]).This may not be an issue for home users, and real solutions here may depend upon secure operating systems.
- Hardware represents a capital expense, and is extremely inflexible.
- Software operates more efficiently on blocks of data than streams of data.
- It is more expensive both in terms of time and monetary value to replace (crypto) hardware than (crypto) software[10].

[10]See http://www.ireland.com/newspaper/finance/2000/0315/fin18.htm for a current example

This rest of this work will take the approach of putting the algorithm, and even more importantly, the key-authentication-key, into firmware. Our primary requirement being the capability to instantiate a secure communications channel between two hosts. This approach gives higher security and speed at a higher (monetary) cost, if, and only if, implemented correctly[LSM+98]. The ideal being a "computational device that can be trusted to execute its programming correctly, despite physical attack"[SW99]

1.2.11 Block & Stream Ciphers

There are two basic types of encryption algorithms: block ciphers and stream ciphers. A block cipher is one in which a block of plaintext is treated as a whole and used to produce a block of ciphertext of equal length. A stream cipher is one that encrypts a digital data stream one bit or byte at a time. While all ciphers are classified in this fashion, in truth most algorithms can be either: blocks can be formed from streams and vice versa [Sch00].

1.2.12 Public Key Ciphers

Public key ciphers (and their related public key exchange algorithms) are generally block ciphers, with the unusual property that one key is used to encipher, while a different and apparently unrelated key is used to decipher a message. So if one of the keys is kept private, the other key (the "public" key) can be released "into the wild", and anyone can use that to encipher a message to us. Then our private key can be used to decipher any such messages. An interesting side effect of this scheme is that someone who enciphers a message and sends it to us cannot decipher their own message even if they want to.

The prototypical public key cipher is RSA, which uses large numeric values as keys. These numbers may contain 1,000 bits or more (over 400 decimal digits), in which each and every bit is significant. The key-space is much smaller, however, because there are very severe constraints on the keys; not just any random value will do. So a 1,000-bit public key may have a brute-force strength similar to a 128-bit secret key cipher.

Because public key ciphers operate on huge values, they are very slow, and so are normally used just to encipher a random message key. The message key is then used by a conventional secret key cipher which actually enciphers the data.

At first glance, public key ciphers apparently solve the key distribution problem (mentioned earlier in §1.2.5 this problem is first acknowledged in literature in [DH76]). But in fact they also open up the new possibility of a man-in-the-middle attack or the user authentication problem.

> How do you know that the key belongs to who you think it does? Still a research problem.[Ros00]

To avoid this, it is necessary to assure that one is using exactly the correct key for the desired user. This requires authentication (validation or certification) via some sort of secure channel, and that can take as much effort as a secure secret key exchange. A man-in-the-middle attack is extremely worrisome, because it does *not* involve breaking any cipher, which means that all the efforts spent in cipher design and analysis and mathematical proofs and public review would be completely irrelevant.

1.2.13 Quantum Leap

Quantum computing is an area where things get either very exciting or very weird, depending on your point of view. It is outside the scope of this work to go into quantum computing in any detail. Suffice it to say that for long term security, the mere existence of quantum computing, means that Public Key Encryption systems are already broken. Currently, with quantum computing, the only type of message that it is known how to share securely, is a completely random string of bits. However, a random string is the perfect key on which to base standard symmetric key cryptography schemes. By using a system in which the integrity and secrecy of the key is guaranteed by the laws of nature we are getting closer to realisation of the "ideal" cipher (§1.2.9). Unfortunately, for the moment it is unlikely this type of cryptography will be practical for the foreseeable future [Sch96][Sin99][SR00].

1.2.14 The Advanced Encryption Standard (AES)

On January 2, 1997[11], National Institute of Standards and Technology (NIST) announced the initiation of the AES development effort and made a formal call for algorithms on September 12, 1997. The call stipulated that the AES would specify an unclassified, publicly disclosed encryption algorithm(s), available royalty-free, worldwide. In addition, the algorithm(s) must implement

[11]This section is an excerpt from http://www.nist.gov/aes

symmetric key cryptography as a block cipher and (at a minimum) support block sizes of 128-bits and key sizes of 128-, 192-, and 256-bits.

On August 20, 1998, NIST announced a group of fifteen AES candidate algorithms at the First AES Candidate Conference (AES1). These algorithms had been submitted by members of the cryptographic community from around the world. At that conference and in a simultaneously published Federal Register notice, NIST solicited public comments on the candidates. A Second AES Candidate Conference (AES2) was held in March 1999 to discuss the results of the analysis conducted by the global cryptographic community on the candidate algorithms. The public comment period on the initial review of the algorithms closed on April 15, 1999. Using the analyses and comments received, NIST selected five algorithms from the fifteen.

The AES finalist candidate algorithms are MARS, RC6, Rijndael, Serpent, and Twofish. NIST has developed a Round 1 Report describing the selection of the finalists.

These finalist algorithms will receive further analysis during a second, more in-depth review period prior to the selection of the final algorithm(s) for the AES Federal Information Processing Standard (FIPS). NIST solicits comments on the remaining algorithms through May 15, 2000. Comments and analysis are actively sought by NIST on any aspect of the candidate algorithms, including, but not limited to, the following topics: cryptanalysis, intellectual property, cross-cutting analyses of all of the AES finalists, overall recommendations and implementation issues. An informal AES discussion forum is also provided by NIST for interested parties to discuss the AES finalists and relevant AES issues.

1.3 Thesis Structure

In Chapter 2, *Encryption Algorithm,* an algorithm is selected after first spending some time on issues such as algorithm type, encryption modes, and their implications. Then the algorithm is examined in detail to try to get a "feel" for the difficulties in the area. As ciphers can be used in different ways, the main "modes" of operation are examined and a decision is made how to deploy the selected algorithm. Without secure keys, the "system" can easily be compromised so key generation issues are investigated. This is a whole area of study in its own right, as generation of cryptographically secure random numbers is inherently more difficult than one would assume. Finally the reference software implementation is examined in detail

In Chapter 3, *Firmware,* the technology and different development environments are introduced and examined. A development environment is chosen. The difference between the software and firmware "domains" are highlighted and explored. A firmware implementation of the algorithm selected in Chapter 2 is presented.

In Chapter 4, *Deployment,* the software implementation is packaged as a "device driver" and uses the reference code implementation covered in Chapter 2. The firmware implementation covered in Chapter 3 is packaged as an equivalent cryptographic device. Significant extra circuitry (scaffolding) is required to actually realise a functioning firmware implementation. Building on the work of the previous two chapters, a stable and functional solution is realised.

Chapter 5, *Results and Conclusions,* details the results of tests performed on the solution, presents a discussion on the achievements of the work and compares the differences between the software and firmware implementations. Some further work outside the scope of this thesis is discussed which would extend the work in interesting and useful directions.

Chapter 2

Encryption Algorithm

Introduction

.

Algorithm Selection

.

3way: The Basic Building Blocks

.

Cipher Modes Explained

.

Reference Code - A short tour

.

Conclusion

2.1 Introduction

In §1.2.11 block ciphers were introduced. The primary motivation for following this path is, in the real world, block ciphers seem to be more general and stream ciphers seem to be easier to analyse mathematically. Otherwise, the differences are in the implementation. Stream ciphers only encrypt and decrypt data one bit at a time and are not really suitable for software implementation. Block ciphers can be easier to implement in software. On the other hand stream ciphers can be more suitable for hardware implementation because they can be implemented very efficiently in silicon. These are important considerations. It makes sense for a hardware encryptor to encrypt each bit on a digital communications channel as it passes. This is what the device sees. On the other hand it makes no sense for a software encryption device to encrypt each individual bit separately. ([Sch96, pp210-211])

As can be seen from §1.2.10. The requirements are to prototype an algorithm in software, then proceed to implement it in hardware. This is to maintain compatibility with the software algorithm and can also give a more a more secure system by virtue of the Key-Encryption-Key [Smi97, pp107-108]being only in silicon and there being no way to retrieve the key other than destroying the chip. Though there are other reasons covered in [LSM$^+$98, Section 3.2] why a hardware cryptographic device may not be as ideal as it sounds. The primary one being, the operating system can be instructed not to use the cryptographic device, thus nullifying all our efforts.

2.2 Algorithm Selection

In choosing an algorithm there are several alternatives:

1. Write our own, based on the belief that our cryptographic ability is second-to-none.[1]

2. Choose a published algorithm, based on the belief that a published algorithm has been scrutinised by many cryptographers; if no one has broken it, then it must be good.

[1]The Japanese PURPLE system is a classic in this regard. Allied cryptographers, never saw a PURPLE machine and yet still managed to break the cipher.

3. Trust a manufacturer, the government, a private consultant to write one.[2]

All of these alternatives are problematic. The second option seems to be the most sensible, given that creating good encryption algorithms is hard[Sch99c] [Cur][Sch99b] and that the last option quite often can prove to be the worst choice:

> It is not unusual for people who sell or promote products to claim that their product is the solution to all of your problems but, unfortunately, in information protection, no product can solve all of your problems.[Coh97, p8]

See [Sch99a] and [Cur] for discussions on reasons why this might be. Though Terry Ritter has a very interesting perspective in [Rit99] and the response it generated in [Rit00]. Of course, the glaring problem in deciding that item 2 above is best, is of course that we do not know the abilities of the various military cryptanalysis organisations and probably never will. They may have several unpublished attacks against some, or indeed all published algorithms. The debate on this precise topic can go on for ever, one can argue that the reason the NSA is so set against export of strong crypto is that they have not yet figured out how to break it (other than brute force), thus using any algorithm they do not allow to be exported is safe. The counter is that they already can break them but just do not want to publicise this fact. For example, at the USENIX Security symposium in August 1999[3] the National Security Agency (NSA)[4] had a minimum of two representatives in the front row at every session. Who knows which session they were actually interested in?[5]

So, we decide to go with well published algorithms for the clear advantages they provide. Now we have to decide to go with a public-key system or a symmetric key system, (there are other systems/algorithms not mentioned here see [Kal93] for a more detailed discussion) both provide "confidentiality" and "Key management" which are our primary requirements. We turn to Schneier [Sch96, page 216] for inspiration:

> Public-key cryptography and symmetric cryptography are different sorts of animals; they solve different sorts of problems. Symmetric

[2]See http://www.ireland.com/newspaper/finance/2000/0315/fin18.htm, the French Banks trusted a manufacturer.

[3]http://www.usenix.org/publications/library/proceedings/sec99/

[4]http://www.nsa.gov

[5]These questions arose from a conversation that took place in the lobby of the Marriot Hotel, Wednesday evening 25'th of August 1999, during the 8th USENIX Security Symposium.

> cryptography is best for encrypting data. It is orders of magnitude faster and is not susceptible to chosen-ciphertext attacks. Public-key cryptography can do things that symmetric cryptography can't; it is best for key management and a myriad of protocols...

The choice seems clear: a symmetric system it is, as our main thrust is data encryption. Again Schneier helps out, with a comprehensive (though maybe a bit dated) analysis of the current range of usable block ciphers. DES was the first candidate for rejection as it is no longer considered "secure enough"[HA94]. A selection of other algorithms were ruled out because of either security concerns, patent restrictions or speed. I was left with the following list:

- LOKI91[BKPS93]
- IDEA[Lai92]
- CAST[Ada94][AT93]
- Blowfish[Sch94a][Sch94b]
- 3-Way[Dae95][DGV94a]

LOKI91 and CAST removed themselves, as they have 64 bit keys and while looking at Table 1.2 and taking into account Moore's Law, $300M would get me a decrypted a message after 1 Hour. This was deemed not to be "Secure Enough". This left IDEA, Blowfish and 3way. Of these three, 3way was chosen[6]. It was considered "Strong Enough" (96 bits), it is of a differing design to either of the other two, Schneier [Sch96, page 354] seems to think it would suffice, and going by Terry Ritters thinking[Rit99], it is probably as good a choice as any of the rest. Also it was designed expressly to be fast in hardware and coming from a programming environment, its explanation as a pseudo "For" loop appealed to the programmer in me.

2.3 3way: The Basic Building Blocks

Here, the main characteristics of the cipher are introduced. All operations will be on binary vectors whose components are indexed starting from 0, e.g. $X = (x_0, x_1, ..., x_{n-1})^T$. The dimension of a vector is by default denoted by n.

[6]During the deployment stage a research paper[KSW] was found in which a successful attack against 3way is detailed.

If a mapping of vectors is specified in terms of its components, the use of the index i implies the range $0 \leq i < n$. Indices consisting of expressions containing i must be reduced modulo n.

2.3.1 μ

If μ(mu) is a bit permutation that inverts the order of components of a vector. For $B = \mu(A)$ we have

$$b_i = a_{n-1-i}$$

Obviously $\mu^{-1} = \mu$. This bit permutation plays an important role in the structure of the cipher. The basic building blocks of the cipher γ and θ have been chosen such that $\gamma^{-1} = \mu \circ \gamma \circ \mu$ and $\theta^{-1} = \mu \circ \theta \circ \mu$.

2.3.2 The Nonlinear substitution γ

The mapping γ (gamma) is defined for vectors whose dimension is a multiple of 3. If $B = \gamma(A)$ and the dimension $n = 3k$ we have

$$b_i = \bar{a}_i \oplus \bar{a}_{i+k}\ \bar{a}_{i+2k}$$

In fact γ is the parallel execution of k substitutions, acting upon 3-bit blocks (called triplets) consisting of bits a_j, a_{j+k} and a_{j+2k}.

2.3.3 The Linear Substitution θ

A vector A can be interpreted as a binary polynomial $a(x) = \sum a_i x^i$. The mapping θ (theta) is defined for vectors whose dimension is a multiple of 12. If $B = \theta(A)$ and the dimension $n = 12h$ we have

$$b(x) = e(x^h)a(x) mod\ (1 + x^{12h})$$

with

$$e(x) = 1 + x + x^2 + x^3 + x^5 + x^6 + x^{10}$$

In fact θ is the parallel execution of h substitutions acting upon 12-bit blocks consisting of bits a_j, a_{j+h}, a_{j+2h},... a_{j+11h}. The linear substitution was chosen such that every output bit depends on 7 input bits. As can be seen in Figure 2.1, theta is a cyclic bit shifting operation.

```
 Each digit represents a nibble.
 Block of 96 bits passed to theta
msb                               lsb
   00000000 00000000 00000001

Result of 96 bits passed to theta
msb                               lsb
   00010000 00010100 01010101
```

Figure 2.1: Theta, bit shifter.

2.3.4 π_1 and π_2

$\pi_1$and π_2 are two bit permutations such that $\pi_1 \circ\ \mu \circ \pi_2 = \mu$, hence the choice of π_1 fixes π_2. For 3-WAY these operations are block-wise rotations of vector sub-blocks of length 32 to facilitate software implementations.

2.3.5 The Structure of the Block Cipher

The encryption process consists of the iterative application of a number of rounds r. One 3-Way round consists of the subsequent application of θ, π_1, γ and π_2 and is denoted by ρ:

$$\rho = \pi_2 \circ \gamma \circ \pi_1 \theta.$$

Before every round, the intermediate result is XOR-ed with a vector (round-constant) that depends on the secret key and the round number. XOR-ing with K_i is denoted by $\delta(K_i)$. The last round is followed by an extra application of δ and θ. We have

$$E_k = \theta \circ \delta(K_r) \circ \rho \circ \delta(K_{r-1}) \circ ... \circ \rho \circ \delta(K_1) \circ \rho \circ \delta(K_0)$$

with E_k denoting the encryption operation under secret key K . The order of the components and their interaction with μ causes decryption to be of a very similar operation to encryption. It can be proven

$$D_K = \mu \circ\ (\theta \circ \delta(K'_0) \circ \rho \circ \delta(K'_1) \circ ... \circ \rho \circ \delta(K'_{r-1}) \circ \rho \circ \delta(K'_r)) \circ \mu$$

with the round keys given by $K'_j = \mu(\theta(K_{r-j}))$.

For efficiency reasons the key schedule is kept as simple as possible. The key length is the block length and every encryption round key is equal to the key global key K XOR-ed with a round constant C_j with small Hamming weight[7]. The decryption round keys can be computed by XOR-ing round constants with the so-called decryption key $K' = \mu(\theta(K))$. In the actual implementation of the algorithm, π_1, γ, and π_2 are always called in the same order so, to reduce the

[7]Number of "1" bits in the binary sequence.

number of function calls, they are implemented as one function "pi_gamma_pi". In pseudo-code, to encipher a plaintext block, x, with n rounds (Daemen in [DGV94a] recommends 11):

Algorithm 1 3way pseudo algorithm.[Sch96, p342]

```
For i = 0 to n -1

    x = x XOR $K_i$
    x = theta(x)
    x = pi_gamma_pi(x)
  x = x XOR $K_n$
  x = theta(x)
```

Where K is a round-constant. Decryption is similar to encryption except that the bits of the input have to be reversed, the bits of the output have to be reversed, the key is different and the round-constants are different.

2.4 Cipher Modes Explained

In §1.2.11 Block & Stream Ciphers were introduced. It was stated that block ciphers always encrypt the same plaintext block to the same ciphertext block, given the same key, this *electronic-codebook mode* (ECB) has disadvantages in most applications, which motivates the implementation of other methods of employing block ciphers (modes of operation) on larger messages. The four most common modes are Electronic Codebook mode (ECB), Cipher-Block Chaining mode (CBC), Cipher Feedback mode (CFB) and Output Feedback mode (OFB).

2.4.1 ECB mode.

Properties of ECB (electronic codebook mode) of operation:

1. Identical plaintext blocks (under the same key) result in identical ciphertext
2. Blocks are enciphered independently of other blocks. Re-ordering ciphertext blocks results in correspondingly re-ordered plaintext blocks.
3. One or more bit errors in a single ciphertext block affect decipherment of that block only. For typical ciphers, decryption of such a block is then random (with about 50% or the recovered bits in error).

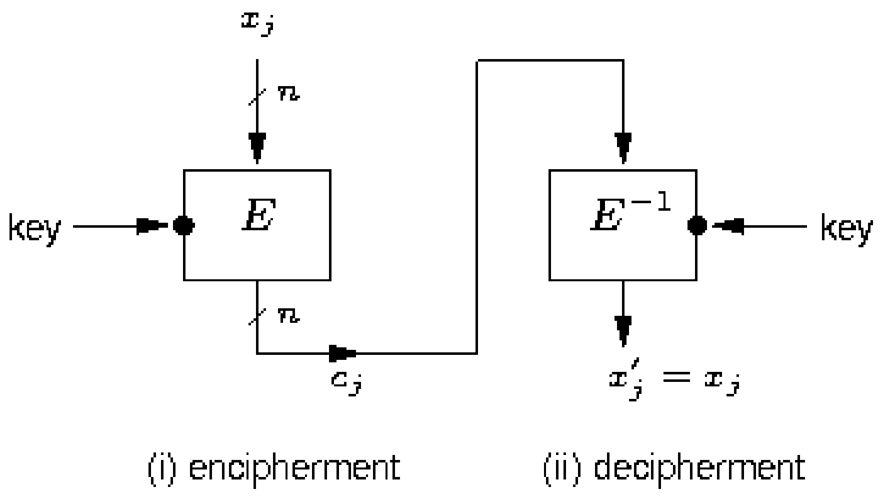

Figure 2.2: ECB Mode

This mode is self-synchronising in the sense of recovery from bit errors, recovery from "lost" bits causes errors in block boundaries. Since ciphertext blocks are independent, someone replacing (accidentally or maliciously) ECB blocks does not affect the decryption of adjacent blocks. Furthermore, this mode does not hide data patterns - identical ciphertext blocks imply identical plaintext blocks. For this reason, the ECB mode is not normally recommended for messages longer than one block, or if keys are reused for more than a single one-block message. Security may be improved by the inclusion of random padding bits in each block (termed an Initialisation Vector).

2.4.2 CBC mode

The *cipher-block chaining* (CBC) mode of operation involves the use of a an Initialisation Vector (IV) which is equal to the block length.

Properties of CBC mode of operation:

1. Identical ciphertext blocks result when the same plaintext is enciphered under the same Key and IV. Changing the IV, key, or first plaintext block (e.g., using a counter or a random field) results in different cipher-texts.

2. The chaining mechanism causes ciphertext to depend on all preceding

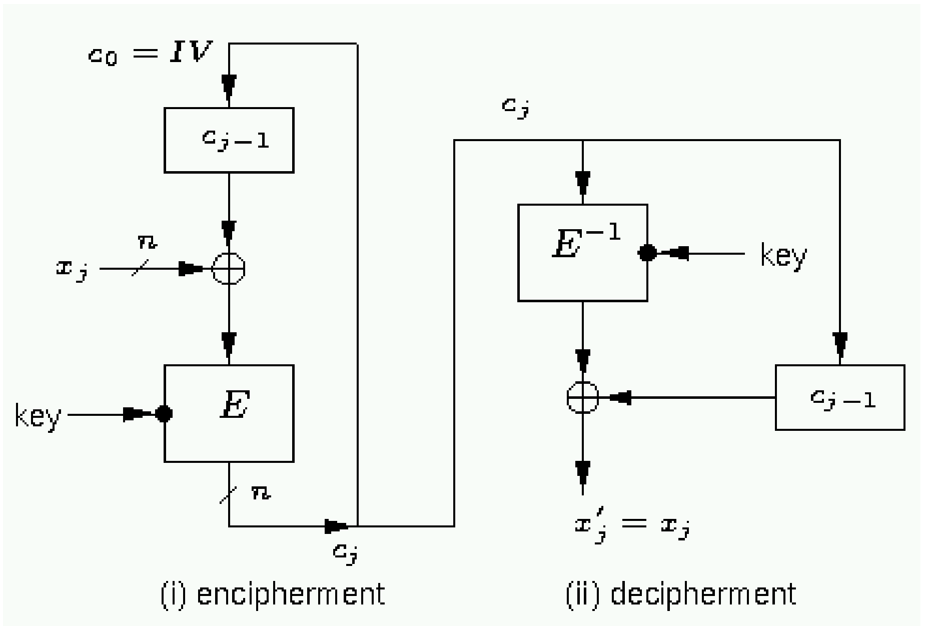

Figure 2.3: CBC Mode

plaintext blocks. Thus re-arranging the order of ciphertext blocks affects decryption. Proper decryption of a correct ciphertext block requires a correct preceding ciphertext block.

3. A single bit error in a ciphertext block c_j affects decipherment of that and all subsequent blocks. Plaintext block p_j recovered from c_j is typically random (50% in error), while the recovered plaintext p_{j+1} has bit errors precisely where c_j did. Thus an opponent can cause predictable bit changes in p_{j+i} by altering corresponding bits of c_j

4. The CBC mode is ***self-synchronising*** in the sense that if an error occurs in c_j but not c_{j+1}, c_{j+2} is correctly decrypted to p_{j+2}.

Although CBC mode decryption recovers from errors in ciphertext blocks, modifications from a plaintext block during encryption alters all subsequent ciphertext blocks. This impacts the usability of chaining modes for applications requiring random read/write access to encrypted data. ECB mode is as alternative (but see §2.4.1).

Although self-synchronising in recovering from bit errors, recovery from "lost" bits causing errors in block boundaries is not possible in the CBC or

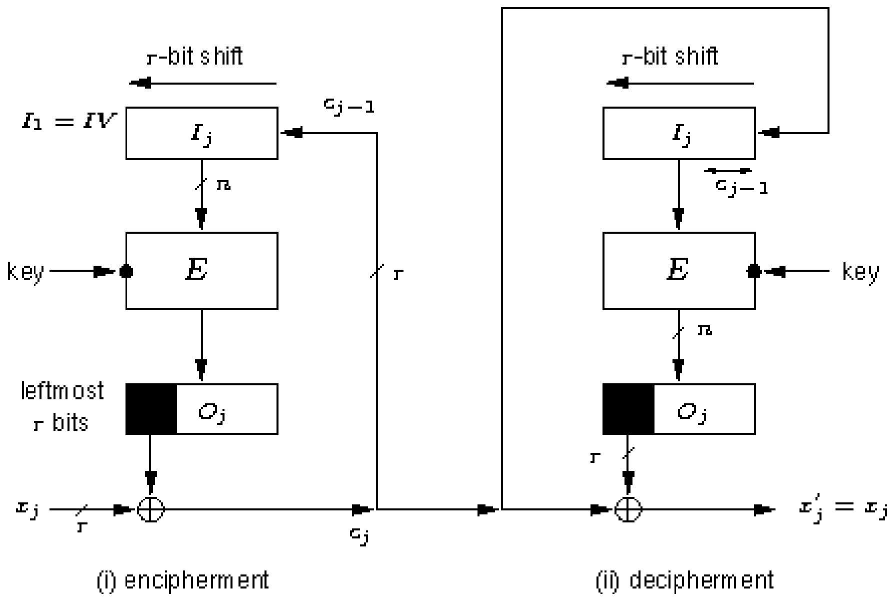

Figure 2.4: CFB Mode

other modes.

The integrity of the IV (Initialisation Vector) in the CBC mode must be maintained, since a malicious opponent could modify the IV and make predictable bit changes to the first plaintext block recovered. Some texts recommend that the IV be kept secret, Schneier[Sch96, page 194] makes a good argument that this need not be the case. If message integrity is required, an appropriate mechanism should be used; encryption mechanisms guarantee confidentiality only.

2.4.3 CFB Mode

While the CBC mode processes plaintext n bits at a time (using an n - bit block cipher), some applications require that r - bit plaintext units be encrypted, for some $r < n$. In this case, the *cipher feedback mode* (CFB) may be used.

Properties of the CFB mode of operation:

1. As per CBC encryption, changing the IV results in the same plaintext input being enciphered to a different output. The IV need not be kept

secret (though unpredictability in the IV is a desirable feature in any chaining mode).

2. Similar to CBC encryption, the chaining mechanism causes ciphertext block c_j to depend on both p_j and preceding plaintext blocks; consequently, re-ordering ciphertext blocks affects decryption. Proper decryption of a correct ciphertext block requires the preceding ciphertext blocks to be correct (so that the shift register contains the proper value)

3. One or more bit errors in any single r - bit ciphertext block c_j affects the decipherment of that and next ciphertext blocks until the error block has shifted entirely out of the shift register. The recovered plaintext will differ from the actual plaintext precisely in the bit positions c_j was in error; the other incorrectly recovered plaintext blocks will typically have 50% of bits in error. Thus an opponent may cause predictable bit changes in the recovered plaintext by altering corresponding bits in the ciphertext.

4. The CFB mode is self-synchronising similar to CBC, but in n - bit CFB a single ciphertext error will affect the decryption of the current and the following $m/n - 1$ blocks, where m is the blocks size.

5. For $r < n$, throughput is decreased by a factor of n/r (vs. CBC) in that each encryption yields only r bits of ciphertext output.

Since the encryption function is used for both CFB encryption and decryption, the CFB mode must not be used if the block cipher is a public-key algorithm: instead, the CBC mode should be used.

2.4.4 OFB Mode

The *output feedback mode* (OFB) of operation may be used for applications in which error propagation must be avoided. It is similar to CFB, and allows encryption of various block sizes (characters), but differs in that the output of the encryption block function (rather than the ciphertext) serves as the feedback.

Properties of the OFB mode of operation:

1. As per CBC and CFB modes, changing the IV results in the same plaintext being enciphered to a different output.

2. The key-stream is plaintext-independent.

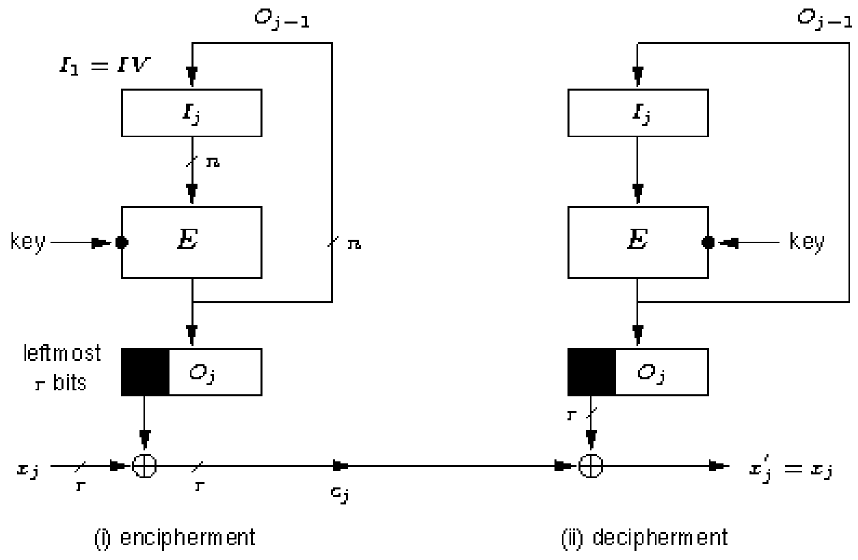

Figure 2.5: OFB Mode

3. One or more bit errors in any ciphertext character c_j affects the decipherment of only that character, in the precise bit position(s) in error, causing the corresponding recovered plaintext bit(s) to be complemented.

4. The OFB mode recovers from ciphertext bit errors, but cannot resynchronise itself after loss of ciphertext bits, which destroys alignment of the decrypting key-stream (in which case explicit re-synchronisation is required)

5. For $r < n$, throughput is decreased as per the CFB mode. However, since the key-stream is independent of plaintext of ciphertext, it may be pre-computed (given the key and IV).

The IV, which need not be secret, must be changed if an OFB key K is re-used. Otherwise an identical key-stream results, thus allowing an opponent to recover the plaintext of the current message if he has already intercepted the previous ciphertext. As per CFB, the remark above on public-key ciphers applies to OFB mode as well as CFB.

Mode	Description	Typical Application
ECB	Each block of X plaintext bits is encoded independently using the same key.	Secure transmission of a single value (e.g., an encryption key).
CBC	The input to the encryption algorithm is the XOR of the next X bits of plaintext and the preceeding X bits of ciphertext.	General purpose block orientated transmission.
CFB	Input is processed J bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XOR-ed with plaintext to produce the next unit of ciphertext.	Authentication.
OFB	Similar to CFB, except that the input to the encryption algorithm is the preceding algorithm's output.	Stream orientated transmission over noisy channels (e.g., satellite communication)

Table 2.1: Modes of Operation

2.4.5 Choosing a cipher mode

Having chosen an algorithm and having covered the main encryption modes, the correct modes for this application must be decided. It was decided to use 3way in ECB mode for en/decrypting Key-encryption-Keys, and CBC mode for block encryption of data. The reasons for both these decisions are covered in this chapter and quite succinctly summarised in Table 2.1[Sta98] above.

2.4.6 Key Generation

> "Anyone attempting to generate random numbers by deterministic means is, of course, living in a state of sin."
> — John Von Neumann

A secure crypto system needs keys that cannot be guessed. Any deviation from this statement means that your crypto system has a vulnerable point of attack. A few points from Richard Smith[Smi97, pages 88-89]:

- The data is not secret unless the key is secret.
- The more random your key, the harder it will be to guess.
- Randomness really does not come easily, especially to computers.

- The more a key is used, the easier it is to crack.

A good generator will produce keys that cannot be guessed even if attackers know how the generator works (see §1.2.13). To do this it must generate numbers that are practically impossible to predict. Computers by themselves are poor sources of unpredictable numbers; they are essentially deterministic machines that are designed to be predictable. Good random key generation is at the heart of every strong crypto-system. For our purposes what we need is a good pseudorandom number generator, as it happens we are fortunate in that some enlightened minds have already implemented two useful pseudo-devices for GNU-Linux systems (which is the development platform) called */dev/random* and */dev/urandom*[8]. Reading from /dev/random yields a small pool of random bits obtained from internal system state. If one observes this device while typing on the keyboard, bits will be produced which will be more and more random. Disk drive accesses, IRQ timings, and key presses; all of this gets hashed into a small pool of entropy[9] that can be accessed directly from /dev/random. /dev/urandom is a stream that hashes /dev/random, and gives the hash value; then it hashes the last hash and the pool forever. Both give a good source of random bits ([Men98]). By default, /dev/urandom uses the Secure Hash Algorithm (SHA). For more discussion on keys, key security and pseudo-random number generation[10] see [Sch99a][Smi97, Chapter 4][Sch96, Chapter 16]. For this application, where we need a "random" key and Initialisation Vector, these devices will suffice.[11]

2.5 Reference Code - A short tour

We begin with a structure chart (Figure 2.6[PJ98]) showing the caller/callee relationships based on the code in appendix A, first presented in [DGV94a] , it also clearly identifies commonality between the two main functions in the algorithm. Clearly, *mu()* is the main difference between encryption and decryption.

[8]Purists would argue that this is not random enough and would insist on "bleaching" the random source[Men98]. Recently, I followed a heated discussion on the ***linux-kernel*** mailing list (majordomo@vger.rutgers.edu) which concluded that the best source of "randomness" would be to amplify the ambient noise in a resistor.

[9]Entropy is a measure of uncertainty. The larger volume of "randomness" accumulated the higher the entropy.

[10]Also see RFC 1750, http://www.kobira.co.jp/document/rfc/RFC1750.txt

[11]In fact, when in kernel space, the kernel function get_random_bytes() is used.

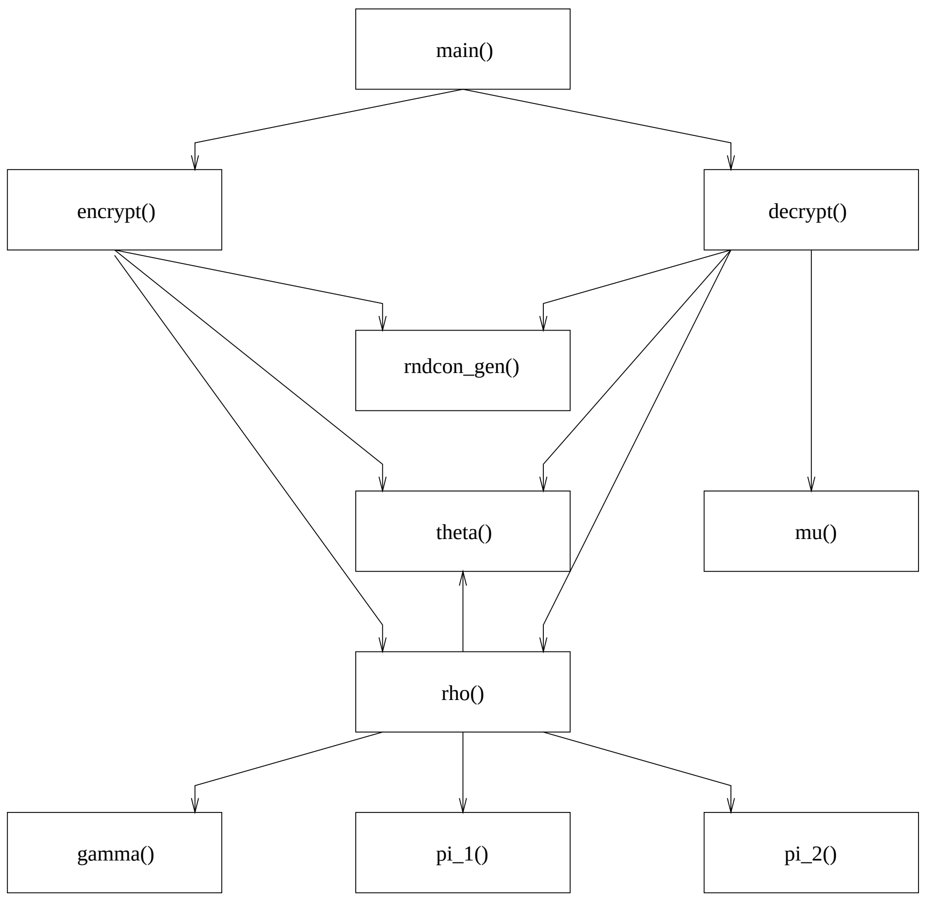

Figure 2.6: Structure Chart of Reference Code.

2.5.1 main

Algorithm 2 3way, main().

```
main()
{
  word32 a[3], k[3];

  scanf("%x %x %x %x %x %x",a+2,a+1,a,k+2,k+1,k) ;
  printf("key : ") ; printvec(k) ;
  printf("plaintext : ");  printvec(a) ; encrypt(a,k) ;
  printf("ciphertext : "); printvec(a) ; decrypt(a,k) ;
}
```

We begin with main, which is primarily a text harness for the core algorithm. When the program is run, it waits for the user to enter six values, the first three are assigned to the plaintext, the second three to the key. Then it prints the results of encryption. The reference document[DGV94a] has values that the designers have tested and are used as test cases.

2.5.2 encrypt

Algorithm 3 3way, encrypt()

```
void encrypt(word32 *a, word32 *k)
{
  int i ;
  word32 rcon[NMBR+1] ;

  rndcon_gen(STRT_E,rcon) ;
  for( i=0 ; i<NMBR ; i++ )
    {
      a[0] ^= k[0] ^ (rcon[i]<<16) ;
      a[1] ^= k[1] ;
      a[2] ^= k[2] ^ rcon[i] ;
      rho(a) ;
    }
  a[0] ^= k[0] ^ (rcon[NMBR]<<16) ;
  a[1] ^= k[1] ;
  a[2] ^= k[2] ^ rcon[NMBR] ;
  theta(a) ;
}
```

This function is the first relevant function in the execution path. It takes two parameters, a pointer to the plaintext (soon to be ciphertext) a and a pointer to the encryption key k. The result of this call is that the data pointed to by a has been transformed to ciphertext. Now we will examine each function of the cipher in turn.

2.5.3 rndcon_gen

Algorithm 4 3way, rndcon_gen()

```
void rndcon_gen(word32 strt,word32 *rtab)
{ /* generates the round constants */
  int i ;


  for(i=0 ; i<=NMBR ; i++ )
    {
      rtab[i] = strt ;
      strt <<= 1 ;
      if( strt&0x10000 ) strt ^= 0x11011 ;
    }
}
```

The first function called upon is rndcon_gen() . This function generates a table of *constants* which are XOR-ed with the key and plaintext. The purpose of these *constants* are to remove all exploitable symmetrical properties of the structure of the cipher. The difference between the round *constants* of two subsequent encryption or decryption rounds is different for all cases. The function itself takes a *seed* value and a pointer to the data structure that will hold the result, i.e. the constants themselves. The same function can be used to generate both encryption and decryption round constants, it is a simple matter of using a different (but fixed, see Appendix A) *seed* value. Also, it should be noted that as the table is so small, the constants can be pre-generated, and hard coded, thus it becomes a trivial matter of looking up the value in a table.

2.5.4 rho

Algorithm 5 3way, rho()

```
void rho(word32 *a) /* the round function */
{
  theta(a) ;
  pi_1(a) ;
  gamma(a) ;
  pi_2(a) ;
}
```

Rho is short for round, thus each application of rho is a equivalent to a single round of the 3way algorithm (see §2.3.5 above). Its primary function is to encapsulate invocations to several other functions. It is not absolutely necessary.

2.5.5 theta

Algorithm 6 3way, theta()

```
void theta(word32 *a) /* the linear step */
{
  word32 b[3];

  b[0] = a[0] ^ (a[0]>>16) ^ (a[1]<<16) ^ (a[1]>>16) ^ (a[2]<<16) ^
               (a[1]>>24) ^ (a[2]<<8) ^ (a[2]>>8) ^ (a[0]<<24) ^
               (a[2]>>16) ^ (a[0]<<16) ^ (a[2]>>24) ^ (a[0]<<8) ;
  b[1] = a[1] ^ (a[1]>>16) ^ (a[2]<<16) ^ (a[2]>>16) ^ (a[0]<<16) ^
               (a[2]>>24) ^ (a[0]<<8) ^ (a[0]>>8) ^ (a[1]<<24) ^
               (a[0]>>16) ^ (a[1]<<16) ^ (a[0]>>24) ^ (a[1]<<8) ;
  b[2] = a[2] ^ (a[2]>>16) ^ (a[0]<<16) ^ (a[0]>>16) ^ (a[1]<<16) ^
               (a[0]>>24) ^ (a[1]<<8) ^ (a[1]>>8) ^ (a[2]<<24) ^
               (a[1]>>16) ^ (a[2]<<16) ^ (a[1]>>24) ^ (a[2]<<8) ;

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}
```

This function is gruesome to look at. Its purpose, along with pi_1 and pi_2 below, is diffusion. Each output bit is dependent on 7 input bits, thus each input bit contributes to the output state of 7 bits. The data is passed in, gets contorted and then passed back. It is basically a cyclical shift operation over the whole 96 bits. (See Figure 2.1 for another look at its operation)

2.5.6 pi_1 & pi_2

Algorithm 7 3way, pi_1 & pi_2

```
void pi_1(word32 *a)
{
  a[0] = (a[0]>>10) ^ (a[0]<<22);
  a[2] = (a[2]<<1) ^ (a[2]>>31);
}

void pi_2(word32 *a)
{
  a[0] = (a[0]<<1) ^ (a[0]>>31);
  a[2] = (a[2]>>10) ^ (a[2]<<22);
}
```

These two functions compliment each other (pi_2 reverses the action of pi_1). They are designed to frustrate both *linear* and *differential* cryptanalysis. They cannot do this on their own however, but as a small part of the larger whole. Put simply, they rotate the data (as opposed to key) bits.

2.5.7 mu

Algorithm 8 3way, mu()

```
void mu(word32 *a) /* inverts the order of the bits of a */
{
  int i ;
  word32 b[3] ;

  b[0] = b[1] = b[2] = 0 ;
  for( i=0 ; i<32 ; i++ ){
    b[0] <<= 1 ; b[1] <<= 1 ; b[2] <<= 1 ;
    if(a[0]&1) b[2] |= 1 ;
    if(a[1]&1) b[1] |= 1 ;
    if(a[2]&1) b[0] |= 1 ;
    a[0] >>= 1 ; a[1] >>= 1 ; a[2] >>= 1 ;
  }

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}
```

This function is the most unremarkable of all of the functions, yet this is the main software decryption bottleneck. Modern microprocessors are not designed for bit manipulation. Mu reverses the bits in a 96 bit block (previously mentioned in §2.3.1). When the firmware design is covered it will be shown that this function is trivial to implement. This is hugely significant as bit operations are quite expensive on block orientated processors (see [DeH00] for an in depth discussion), whereas in firmware the same function is achieved by a simple cross connection. This will be borne out by empirical measurements.

2.5.8 gamma

This function implements the distributed nonlinearity mentioned previously in §2.3.2. Its sole purpose is “confusion” as dictated by Shannon[Sha49]. This function gives the 3way algorithm its name (parallel execution of substitutions acting on 3-bit blocks)

Algorithm 9 3way, gamma()

```
void gamma(word32 *a) /* the nonlinear step */
{
  word32 b[3] ;

  b[0] = a[0] ^ (a[1]|(~a[2])) ;
  b[1] = a[1] ^ (a[2]|(~a[0])) ;
  b[2] = a[2] ^ (a[0]|(~a[1])) ;

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}
```

Algorithm 10 3way, decrypt()

```
void decrypt(word32 *a, word32 *k)
{
  int i ;
  word32 ki[3] ; /* the ‘inverse’ key */
  word32 rcon[NMBR+1] ; /* the ‘inverse’ round constants */

  ki[0] = k[0] ;
  ki[1] = k[1] ;
  ki[2] = k[2] ;
  theta(ki) ;
  mu(ki) ;

  rndcon_gen(STRT_D,rcon) ;
  mu(a) ;
  for( i=0 ; i<NMBR ; i++ )
    {
      a[0] ^= ki[0] ^ (rcon[i]<<16) ;
      a[1] ^= ki[1] ;
      a[2] ^= ki[2] ^ rcon[i] ;
      rho(a) ;
    }
  a[0] ^= ki[0] ^ (rcon[NMBR]<<16) ;
  a[1] ^= ki[1] ;
  a[2] ^= ki[2] ^ rcon[NMBR] ;
  theta(a) ;
  mu(a) ;
}
```

2.5.9 decrypt

Now that we have covered all the constructs used to implement the encryption, we now have to reverse the process. The structure is almost the same. There are two major differences.

- The "key" is the encryption key modified by the application of the functions theta() and mu()
- There is a call to mu() to reverse the bits of the data before the "decryption" takes place, and another call to mu() after it completes.

2.5.10 CBC Mode Implementation

Though not part of the reference code, CBC (Cipher Block Chaining) mode was initially implemented by wrapping the reference code functions (see §2.4.2 and §2.4.5 to recap on why this mode is needed). Once a functioning CBC mode implementation was complete, work could continue on the rest of the project. The final CBC mode implementation is contained in function swWrite() in §C.5.

2.6 Conclusion

Having covered issues related to algorithm choice, modes of encryption, and the implicit implications of choosing one over another. The chosen algorithm is then adapted for use and the source code is explored in detail. The next chapter will go on to assess firmware technologies, identify candidate platforms and their associated development environments. This will lead to a chosen development platform, environment, and an implementation.

Chapter 3

Firmware

Introduction

.

Altera & MAX+PLUS II

.

3way: The Basic Building Blocks

.

Conclusion

3.1 Introduction

Programmable logic devices (PLDs) are standard, off-the-shelf user-configurable integrated circuits (ICs) used to implement custom logic functions. In the early 1980s, simple PLDs were typically used to integrate multiple discrete logic devices and designs were typically expressed using Boolean equations.

Prompted by the development of new types of field programmable devices [BFRV92][OD95] (FPDs), the process of designing digital hardware has changed dramatically over the past few years[BR00]. Unlike previous generations of technology, in which board-level designs included large numbers of Small Scale Integrated (SSI) chips containing basic gates, virtually every digital design produced today consists mostly of high-density devices. For these reasons, most prototypes, and also many production designs are now built using Field-Programmable Devices (FPDs). The most compelling advantages of FPDs are instant manufacturing turnaround, low start up costs, low financial risk and (since programming is done by the end user) ease of design changes.

3.1.1 Definitions of Relevant Terminology

- Field-Programmable Device *(FPD)* — A general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs.

- *PLA* — a Programmable Logic Array (PLA) is relatively small FPD that contains two levels of logic, and AND-plane and an OR-plane, where both levels are programmable.

- *PAL*[1] — a Programmable Array Logic (PAL) is a relatively small FPD that has a programmable AND-plane followed by a fixed OR-plane.

- *SPLD* — refers to any type of simple PLD, usually either a PLA or PAL.

- *CPLD* — a more Complex PLD that consists of an arrangement of multiple SPLD-like blocks on a single chip.

- *FPGA* — a Field-Programmable Gate Array is an FPD featuring a general structure that allows very high logic capacity. Whereas CPLDs feature logic resources with a wide number of inputs (AND planes), FPGAs offer

[1]PAL is a trademark of Advanced Micro Devices

more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than do CPLDs.

- *Interconnect* — the wiring resources in an FPD.
- *Programmable Switch* — a user-programmable switch that can connect a logic element to an interconnect wire, or one interconnect wire to another.
- *Logic Block* — a relatively small circuit block that is replicated in a array in an FPD. When a circuit is implemented in an FPD, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block.
- *Logic Capacity* — the amount of digital logic that can be mapped into a single FPD. This is usually measured in units of "equivalent number of gates in a traditional gate array"
- *Logic Density* — the amount of logic per unit area in an FPD.
- *Speed-Performance* — measures the maximum operable speed of the circuit when implemented in an FPD. For combinational circuits, it is set by the longest delay through any path, and for sequential circuits it is the maximum clock frequency for which the circuit functions properly.

3.1.2 Evolution of Programmable Logic Devices

The first type of user programmable chip that could implement logic circuits was the Programmable Read-Only-Memory (PROM), in which address lines can be used as logic circuit inputs and data lines as outputs. Logic functions, however, rarely require more than a few product terms, and a PROM contains a full decoder for its address inputs. PROMs are thus an inefficient architecture for realizing logic circuits, and so are rarely used in practice for that purpose. The first device developed specifically for implementing logic circuits was the Field-Programmable Logic Array (FPLA), or simply PLA for short.

When PLAs were introduced by Philips, their main drawback was that they were expensive to manufacture had a very poor price/performance ratio. Both disadvantages were due to the two levels of configurable logic, because programmable logic planes were difficult to manufacture and introduced significant propagation delays. To overcome these weaknesses, Programmable Array Logic (PAL) devices were developed. To compensate for lack of generality incurred because the OR-plane is fixed (finite number of OR gates), several variants of

PALs are produced, with different numbers of inputs and outputs, and various sizes of OR-gates. PALs usually contain flip-flops connected to the OR-gate outputs so that sequential circuits can be realized. All small PLDs, including PLAs, PALs, and PAL-like devices are grouped into a single category called Simple PLDs (SPLDs), their important characteristics are low cost and very high pin-to-pin speed-performance.

The difficulty with increasing capacity of a strict SPLD architecture is that the structure of the programmable logic-planes grow too quickly in size as the number of inputs is increased. The only feasible way to provide large capacity devices based on SPLD architectures is then to integrate multiple SPLDs onto a single chip and provide interconnects to programmably connect the SPLD blocks together. Many commercial FPD products exist on the market today with this basic structure, and are collectively referred to as Complex PLDs (CPLDs).

CPLDs were first pioneered by Altera[2], first in their family of chips called Classic EPLDs, and then in three additional series, called MAX 5000, MAX 7000 and MAX 9000. Because of a rapidly growing market for large FPDs, other manufacturers developed devices in the CPLD category and there are now many choices available. CPLDs provide logic capacity up to the equivalent of about 50 typical SPLD devices, but it is difficult to extend these architectures to higher densities.

FPGAs however consist of an array of uncommitted circuit elements, called *logic blocks*, and interconnect resources, but FPGA configuration is performed through programming by the end user. As the only type of FPD that supports very high capacity, FPGAs have been responsible for a major shift in the way digital circuits are designed.

3.1.3 Computer Aided Design (CAD) Flow for FPDs

When designing circuits for implementation in FPDs, it is essential to employ Computer-Aided Design (CAD) programs. CAD tools are important not only for complex devices like CPLDs and FPGAs, but also for SPLDs. A typical CAD system would involve software for the following tasks: initial design entry, logic optimisation, device fitting, simulation, and configuration. This design flow is illustrated in Figure 3.1, which also indicates how some stages feed back to others. Design entry may be done either by creating a schematic diagram with a graphical CAD tool, by using a text based system to describe a design

[2]http://www.altera.com

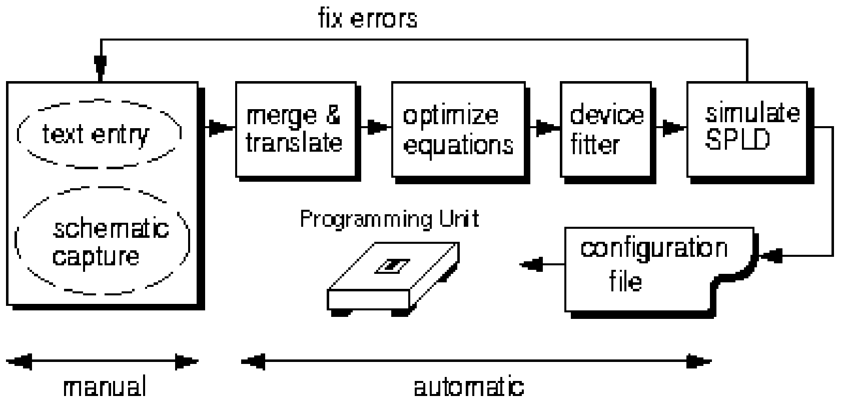

Figure 3.1: CAD Design Flow for SPLDs[BR00]

in a simple hardware description language, or with a mixture of entry methods. Since initial logic entry is not usually in an optimised form, algorithms are employed to optimise the circuits, after which additional algorithms analyse the resulting logic equations and "fit" them into the SPLD. Simulation is used to verify correct operation, and the user would return to the design entry step to fix errors. When a design simulates correctly it can be loaded into a programming unit and used to configure a SPLD.

The steps involved for implementing circuits in CPLDs are similar to those for SPLDs, but the tools themselves are more sophisticated. FPGAs, because of their increased complexity, require additional tools. The major difference is in the "device fitter" step that comes after logic optimisation and before simulation, where FPGAs require at least three steps: a technology mapper to map from basic logic gates into the FPGAs logic blocks, placement to choose which specific logic blocks to use in the FPGA, and a router to allocate the wire segments in the FPGA to interconnect the logic blocks. With this added complexity, the CAD tools might require a fairly long period of time to complete their tasks. (God Bless Moore's law!)

3.1.4 Commercially Available FPGAs

As one of the largest growing segments of the semiconductor industry, the FPGA market-place is volatile. As such, the pool of companies involved changes rapidly

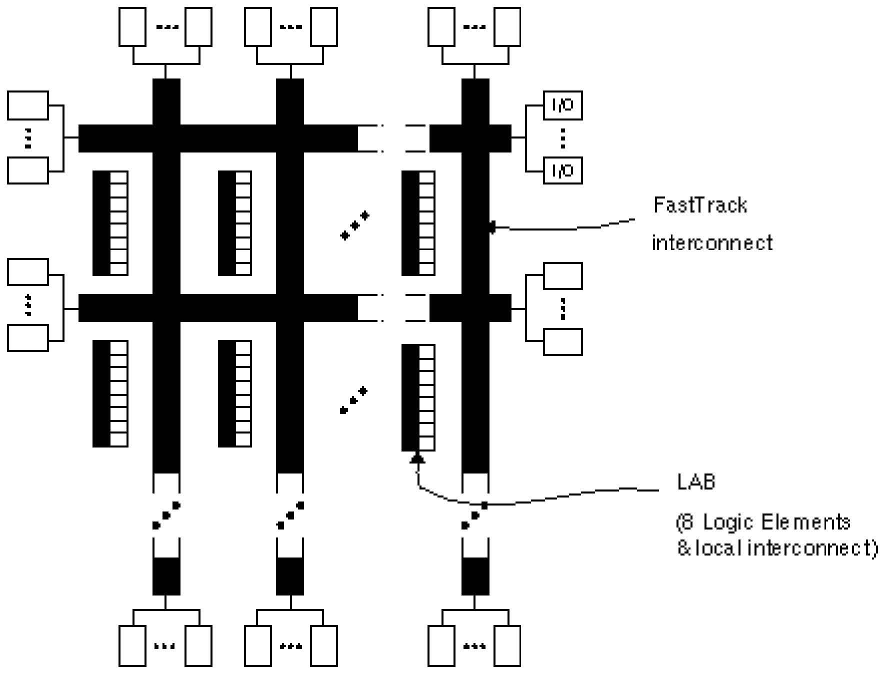

Figure 3.2: Flex 10K Architecture

and it is somewhat difficult to say which products will be the most significant when the industry reaches a stable state.

There are two basic categories of FPGAs on the market today:

1. SRAM-based FPGAs[Tri94].
2. Antifuse-based[3] FPGAs[H+88].

In the first category, Xilinx and Altera are the leading manufacturers in terms of number of users, with the major competitor being AT&T. For antifuse-based products. Actel, Quicklogic, Cypress, and Xilinx offer competing products.

In general, FPGAs have gained rapid acceptance and growth over the past decade because they can be applied to a very wide range of applications. A typical list includes: random logic, integrating multiple SPLDs, device controllers, communication encoding (the application investigated by this work) and filtering, brute-force cracking of symmetrical crypto keys and many more.

[3]Originally open-circuits and take on low resistance only when programmed.

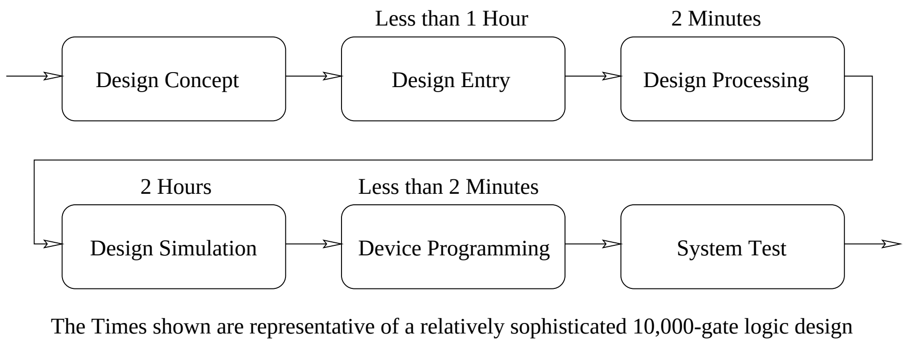

Figure 3.3: Altera PLD development cycle using the Quartus & MAX+PLUS II Software

Other interesting applications of FPGAs are prototyping of designs later to be implemented in gate arrays, and also emulation of entire large hardware systems. The former might be possible using only a single large FPGA, and the latter would entail many FPGAs connected by some sort of interconnect.

Another promising area for FPGA application, which is only beginning to be developed, is the usage of FPGAs as custom computing machines. This involves using the programmable parts to "execute" software, rather than compiling the software for execution on a regular CPU.

3.2 Altera & MAX+PLUS II

Altera Corporation offers a complete solution to a systems designer. The solution combines programmable logic devices (PLDs) and advanced development tools supporting faster design cycles. To further increase efficiency, Altera offers a full range of (what Altera terms) mega-functions (similar to library functions in C/C++) to eliminate common programming tasks, allowing designers to focus on task-specific device functions.

When investigation was first begun in this area, it was realised that there was significant local experience with MAX+PLUS II. It seemed prudent to continue to utilise this resource. With this in mind, and from a detailed examination at MAX+PLUS II[Cora][Corb] (Altera's Programmable Logic Development System) and AHDL[Cor95]. Altera's MAX+PLUS II was selected.

3.2.1 AHDL Design Language

The Altera Hardware Description Language (AHDL) is a high-level, modular language which is completely integrated into the MAX+PLUS II Application. One can use any editor to create AHDL Text Design Files (.tdf). Then they are compiled to create output files for simulation, timing analysis, and device programming.

AHDL statements can be used to design complex combinatorial logic, group operations, state machines, truth tables and parameterised logic. One can create entire hierarchical projects with AHDL, or mix AHDL TDFs with other types of design files in a hierarchical design (a "project" in MAX+PLUS II). One can use all custom functions or incorporate any of the Altera-provided mega-functions and macro-functions — including a Library of Parameterised Modules (LPM) functions — into any TDF by automatically creating an Include File (.inc) in the Text Editor.

3.2.2 Text Design File Sections

A Text Design File (.tdf) is an ASCII text file, written in AHDL, that can be entered with the MAX+PLUS II Text Editor or any standard text editor. The following AHDL sections and statements are listed in the order they can appear in a TDF [Cor95].

- (Optional) Title Statement - provides comments for the Report File (.rpt) generated bye the MAX+PLUS II Compiler .
- (Optional) Include Statement - specifies an Include File that replaces the Include Statement in the TDF.
- (Optional) Constant Statement - specifies a symbolic name that can be substituted for a constant.
- (Optional) Define Statement - defines an evaluated function, which is a mathematical function that returns a value that is based on optional arguments.
- (Optional) Parameters Statement - declares one or more parameters that control the implementation of a parameterised mega-function or macro-function. A default value can be specified for each parameter.

- (Optional) Function Prototype Statement - declares the ports of a logic function and the order in which those ports must be declared in an in-line reference. In parameterised functions, it also declares the parameters used by the function.
- (Optional) Options Statement - sets the default bit-ordering for the file, or for the project if the file is a top-level TDF.
- (Optional) Assert Statement - allows you to test the validity of an arbitrary expression and report the results.
- (Required) Subdesign Section - declares the input, output and bidirectional ports of an AHDL TDF.
- (Optional) Variable Section - declares variables that represent and hold internal information. Variables can be declared for ordinary or tri-state nodes, primitives, mega-functions, macro-functions and state machines. Variables can also be generated conditionally with an "If Generate" Statement. The Variable Section can include any of the following constructs:
  - Instance Declaration
  - Node Declaration
  - Register Declaration
  - Machine Alias Declaration
  - If Generate Statement
- (Required) Logic Section - Defines the logical operations of the file. The Logic Section can define logic with Boolean equations, conditional logic, and truth tables. It also supports conditional and iterative logic generation, and the capability to test the validity of an arbritrary expression and report the results. The Logic Section can include any of the following constructs:
  - Defaults Statement
  - Assert Statement
  - Boolean Equations
  - Boolean Control Equations
  - Case Statement

- For Generate Statement
- If Generate Statement
- If Then Statement
- In-Line Logic Function Reference
- Truth Table Statement

AHDL is a concurrent language. All behaviour specified in the logic section of a TDF is evaluated at the same time rather than sequentially. Equations that assign multiple values to the same AHDL node or variable are logically connected (OR-ed if the node or variable is active high, AND-ed if it is active low). A TDF must contain a Subdesign section and a Logic section. The last entries in a TDF are the Subdesign Section, Variable Section (Optional), and Logic Section, which together contain the behavioural description of the TDF.

Files in a project hierarchy can be TDFs, GDFs, WDFs, ADFs, SMFs, EDIF Input Files OrCAD Schematic Files, AHDL Design Files, or Xilinx Netlist format files. Each logic function is connected through its input and output ports to the design file at the next higher level.

An include file is an ASCII text file (with the extension .inc) that can be imported into a TDF with an AHDL include statement. The contents of the include file replace the Include Statement that calls the file. Include files can contain Function Prototypes, Constant, Define, and Parameters Statements.

3.2.3 Firmware & Software: Differences.

Though AHDL looks quite similar to other "programming" languages, one has to be mindful of the differences. What one tends to forget as a programmer is that the design will eventually be on silicon, and will be subject to all the laws of physics, i.e. propagation delays etc. Take for example a "for" loop in a normal programming language. There is no equivalent in AHDL, it has to be constructed from latches and XOR gates.

3.3 3way: The Basic Building Blocks

Initially, the decision was made to try to keep the same "block" structure in order that test and comparison could more easily be accomplished between the firmware and software blocks on completion of the project. This probably makes comparison easier, but utilises more "real-estate" on the firmware device and thus is not as "space-efficient" as it could be. Through this section, we will break the basic firmware blocks into the same constituent parts as in §2.3 above. All the AHDL code is contained is Appendix B.

One of the first differences: in software, the round constants have to be generated every time the program is run (see Figure 2.5.3), whereas for firmware, they can be pre-calculated and written into the silicon. Thus there is no firmware equivalent of the *rndcon_gen()* function. For comparison between the reference software implementation and firmware implementation see figures 2.6 (software) and 3.4 (firmware). As can be seen from Figure 3.4, there is some scope for optimisation of the whole design, which is discussed later in §5.

It is worth pointing out at this point that this is a totally different domain to operate in. It requires different thought processes in order to get the same functionality that can quite easily done in software with little or no regard as to how it is actually implemented. At this level one either uses an Altera provided megafunction or else design, test and verify it completely yourself. Thus implementing functions at this level generally requires much more thought and effort than is required to implement the software counterpart.

3.3.1 cipher

This function encapsulates the encrypt/decrypt functions. A control line (E_D, see Figure 3.5, which is a graphical representation of the "subdesign" segment of the code in §B.1) selects between them. It also takes the encryption key and feeds it to both the encipher and decipher circuits, though the key is first passed through theta and mu in the case of the deciphering circuit. All the other lines in the figure are there to control the action of the circuit, and have the following action:

clock clock input into the circuit to drive all sequential logic.

e_d en/decipher, controls which half of the device the data passes through.

load loads the data on the *datain* lines into the device and begins the encryption/decryption operation.

counter_en this controls whether the circuit does any en/decryption by enabling the counter which counts through the requisite 11 rounds of the 3way function.

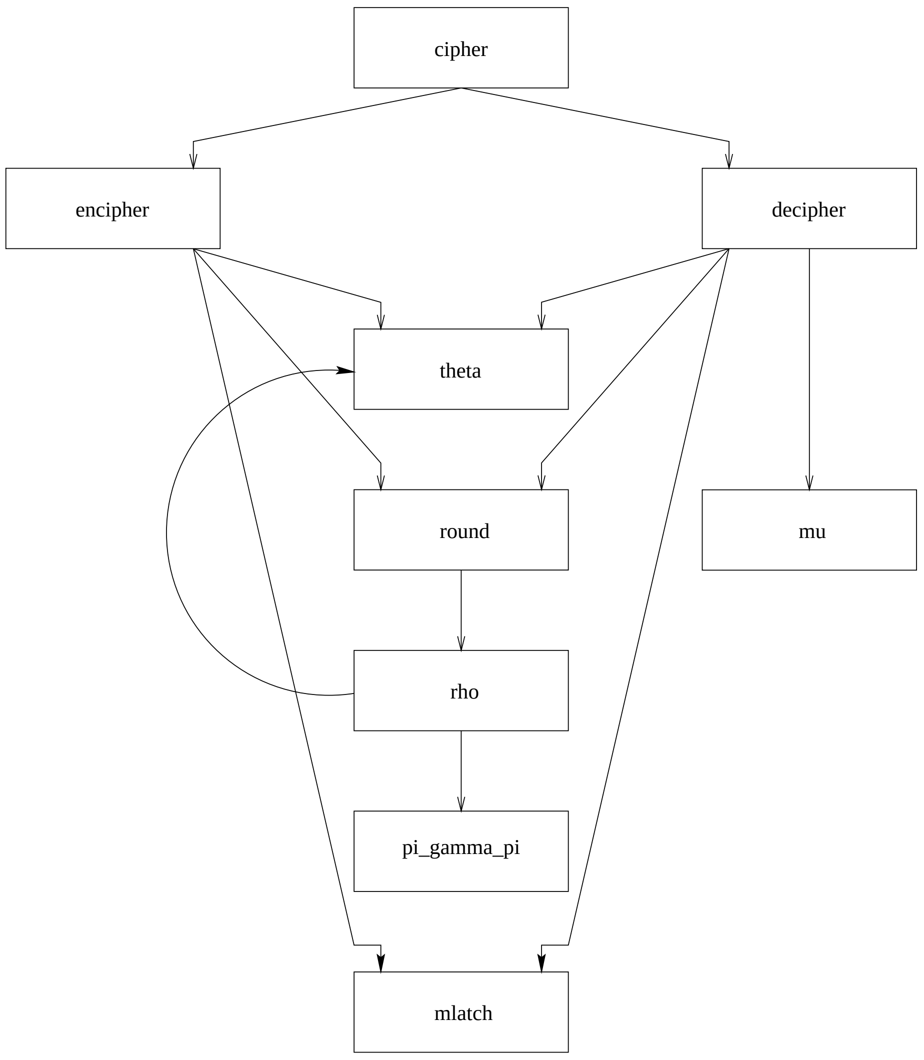

Figure 3.4: Firmware block diagram

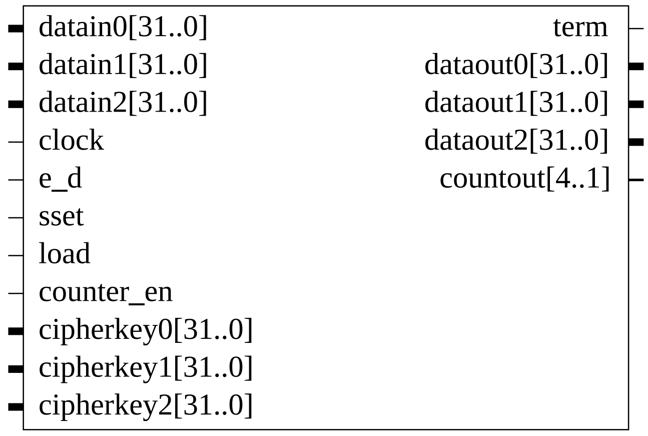

Figure 3.5: Cipher

term this signals the termination of the count. I.e. the encipherment operation is complete.

countout this is a diagnostic output, which shows what state the internal counter is in.

3.3.2 encipher

Encipher (§B.2) takes as input the encryption key and a block of data, the round constants are hard coded into it, and outputs the encrypted block of data.

3.3.3 mlatch

The mlatch block (§B.3) takes two separate blocks of input data and, depending on the control signals, selects one or the other of them to be latched and output. The selection is between *datain*, or data from the output of the circuit, this is how data is routed from the output of the circuit, back into the input, thus providing a “loop” for the data to travel around.

3.3.4 round

This (§B.4) is the firmware version of the “for” loop implemented previously in software (§2.5.4). It takes a block of data, the encryption key, and the round constant as its input and returns a suitable processed block of data.

When foundation work was done on this block, a significant effort was made to design completely in combinatorial logic. This would have meant no need for a loop and no requirement for a latch. The design itself seemed comparatively straightforward. Unfortunately, MAX+PLUS II was unable to resolve the "design" into something that could be programmed into silicon. After some iterations through varying designs, it eventually compiled when a latch was inserted between every two "rounds". The reason MAX+PLUS II could not get a fit was the huge amount of interconnections required by the combinatorial design. In the end a latch was inserted after every round to simplify the overall design. This highlights one of the problems with this domain. One has to work within the limitations of the device, more memory or CPU power cannot be "thrown" at the problem.

3.3.5 rho

This block (§B.5) encapsulates the "theta" and "pi_gamma_pi" firmware functions to maintain the same structure as the software version. Packaging it similarly to the software version also makes it easier to verify correct operation. It takes a block of data and returns a block of (different) data.

3.3.6 pi_gamma_pi

This (§B.6) is a the concatenation of pi_1, gamma, and pi_2 (see §2.5.6 and §2.5.8 for explanation) software functions implemented in firmware. They have been combined like this as they are never used separately.

3.3.7 theta

This (§B.7) is a slightly different implementation of theta than is in the reference code (see §2.5.5 for explanation). This implementation is similar in structure to the actual software implementation. Either way, it makes no difference to the firmware implementation as it uses the same number of Logic Cells (192).

3.3.8 decipher

This function (§B.8) is almost identical to encrypt. The difference is that the data is passed through *mu* first on the way in, and again as the last operation on the way out.

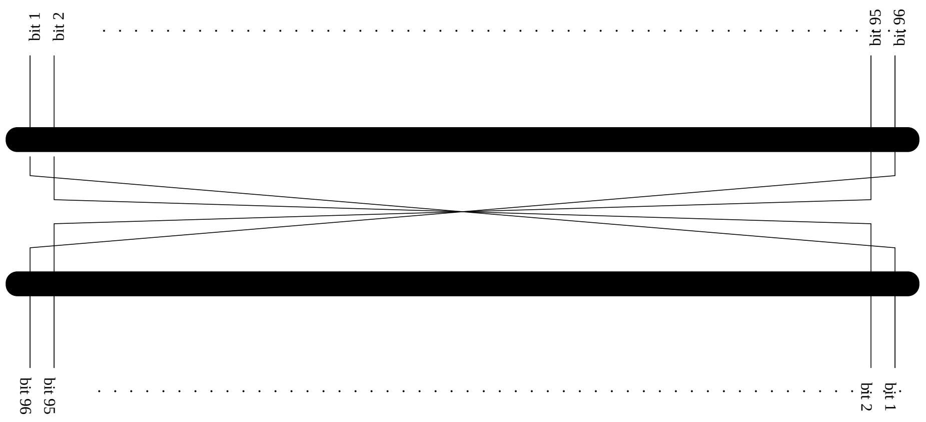

Figure 3.6: Firmware Mu.

3.3.9 mu

Mu (§B.9) has the exact same effect as its software counterpart (see §2.5.7). It reverses the bits in each of the three blocks of data (see Figure 3.6). It is very interesting to note that this function accounts for the difference in speed between the software encrypt and decrypt functions (detailed in §5.1), whereas in firmware it is just a trivial cross-connection of wires which has negligible delay.

3.4 Conclusion

Having selected and thoroughly examined an algorithm in Chapter 2, this chapter has investigated firmware development environments, chosen one, and implemented a firmware version of the algorithm. Unfortunately, neither the software nor firmware solutions are particularly useful at this stage. The software version would have to be adapted into any application in which we wished to use it, and the firmware version is only usable in MAX+PLUS II development environment. In the next chapter we move to deploy these implementations in a more useful fashion.

Chapter 4

Deployment

Introduction

.

Device Driver Overview

.

Software - Pseudo device driver

.

Firmware

.

Conclusion

4.1 Introduction

First, lets review the work and take a look back at what has been covered. In Chapter 2, the algorithm and the reference code that implements it were introduced. In Chapter 3 the firmware structure and implementation were introduced.

At this stage we have a reference software implementation, and a firmware based implementation, at the moment neither of these are particularly useful. The software version is limited in that it has to be compiled into every application that uses it, or alternatively it could be compiled into a shared library and then we could use it by including a header file and calling its functions. The firmware solution is even less useful as we have no way of getting the data into the device.

Ultimately we want a solution whereby we can use an application using encryption, on a machine running GNU-Linux (hence referred to as Linux[1]) doesn't know if it is using firmware or software encryption. To talk to any firmware device will require a set of functions explicitly designed to facilitate this. As the application doesn't need to know whether there is a software or an actual firmware implementation behind it, but our requirement is that the functionality should be identical. The ideal place to put this set of software functions (including the ones that invoke the firmware) is in a software entity called a "Device Driver" or "Device Handler" [Ben96][Ste92][Lew91].

4.2 Device Driver Overview.

To put it simply, a device driver is nothing more than a piece of software that takes data from the user, converts it into a format suitable for the device, sends it to the device, takes the result, puts it back into the format the user requires[2]. Under Linux, a device driver can operate out of one of two sections of memory; kernel space, or user space (i.e. what *malloc()* returns). Kernel-space drivers have the most power, but they are difficult to understand and develop. User-space drivers are easier to develop, and they can draw on many functions and procedures in C libraries[KR98], but they have limits as to what they can to with system hardware. Drivers that run in kernel space have several advantages over

[1]Linux is interchangeably used to refer to the whole system or just the Kernel, for the rest of this chapter it is used to refer to the Kernel

[2]Actually users normally talk to devices through a standard library and thus restricted in the format they can send and receive.

those in user space. They are fast because they do not invoke context switches, they have direct access to interrupts, they can support block-transfer devices, they are available until one unloads them, and if properly written they can be re-entrant. Linux loads kernel-space drivers and retains them in memory until you specifically unload them (for modules) or until the computer is shut-down (statically compiled into the kernel). Properly written kernel-space drivers make their device look like a file. Thus normal file-system system calls such as open, close, read and write can be used to send and retrieve data from the device.

A program that crashes in user-space will not halt the system, unless of course you are trying to control hardware that locks up the computer on its own. User-space device access requires a context switch, so if Linux has swapped the device driver from user-space, response time suffers; Linux must retrieve the driver before it can use it. During a "swap" , the Linux kernel can "move" the contents of memory into a swap space or onto hard disk. User space drivers run as a single thread, they serialise read/write accesses and make simultaneous I/O accesses difficult. In contrast, correctly implemented kernel-space drivers are re-entrant, so even if a process is locked whilst waiting for an external event, another device can use the corresponding driver without compromising data integrity.

In general, writing a kernel-space driver calls for more advanced C programming skills. To write one properly, one requires knowledge of such issues as kernel-dependent version control, memory management, and resource control. All without recourse to any C libraries. And if the driver doesn't work correctly, it will crash your computer, requiring a hard reboot. Debugging a kernel-space driver requires more work and will cause more headaches (See [Rub98, Chapter 4] for a more in-depth discussion.). As an example of the difference between user-space and kernel-space, a kernel version of the hopefully familiar "hello world" program. Firstly, note that the module cannot link with standard C libraries, thus we cannot use stdio.h or any other libraries (and it is compiled into an object file not a normal executable, see [SM98][W$^+$95][Pom]). Kernel-level drivers do not contain a "main()" program entry point. Testing this driver requires using the Linux `insmod` and `rmmod` commands on the command line. The insmod command registers the module's capabilities with the Linux kernel and the `rmmod` command removes and "cleans-up" the driver's registration. Execution of the final print statement in the example simply confirms that the module got terminated properly.

To run this module one has to have "root" privileges (the highest privilege)

Algorithm 11 Kernel version of "Hello World"

```
#define MODULE // must define before including module.h
#include <linux/module.h> // include module library functions

int
init_module(void) // registers module with kernel
{
printk("Salut Mundi!"); // call kernel level print function
return 0;
}

void cleanup_module(void) // undoes init_modules kernel registration
{
printk("bye cruel world"); // prints exit message, look in /var/log/messages
}
```

because it operates in kernel space. So after logging in as the root user, type `/sbin/insmod hello_k.o` (hello_k.o being the name of the compiled binary). In general Linux systems divert all kernel messages to the file */var/log/messages* so the output should appear in this file.

Drivers make devices look like files to software. So calling a kernel-level driver from user space is the job of file operations. Setting up a kernel-space driver so that an application can call it requires steps that identify the driver to the system and other applications. This requires the establishment of the file operations read, write, close and any other operations that you want the driver to perform. Finally, you must add the driver to the Linux file-operations table so that application programs can treat the driver as if it controls a file-orientated device. One can determine which kernel drivers, or modules, Linux has loaded by typing `cat /proc/modules` at the command line. The resulting list shows active modules along with a usage count. To remove the driver from kernel space, type `/sbin/rmmod hello_k`. The kernel placed the driver-exit message "bye" in the system log files. To make matters worse, Linux is a moving target. The kernel calls that provide modules with services can change, thus the module no longer even compiles, never mind loads. This means that while one is kernel programming, one has to keep an eye on "latest developments" [Goo] as they will have an effect on your driver in the future.

For the "pseudo" driver a user-space driver would be more than adequate, but as we will need the more powerful facilities of a kernel-space driver eventually we will work towards developing a complete kernel space driver. In the following sections, we will first introduce the software "pseudo" device driver then introduce the development board, and "scaffolding" to get the data to and

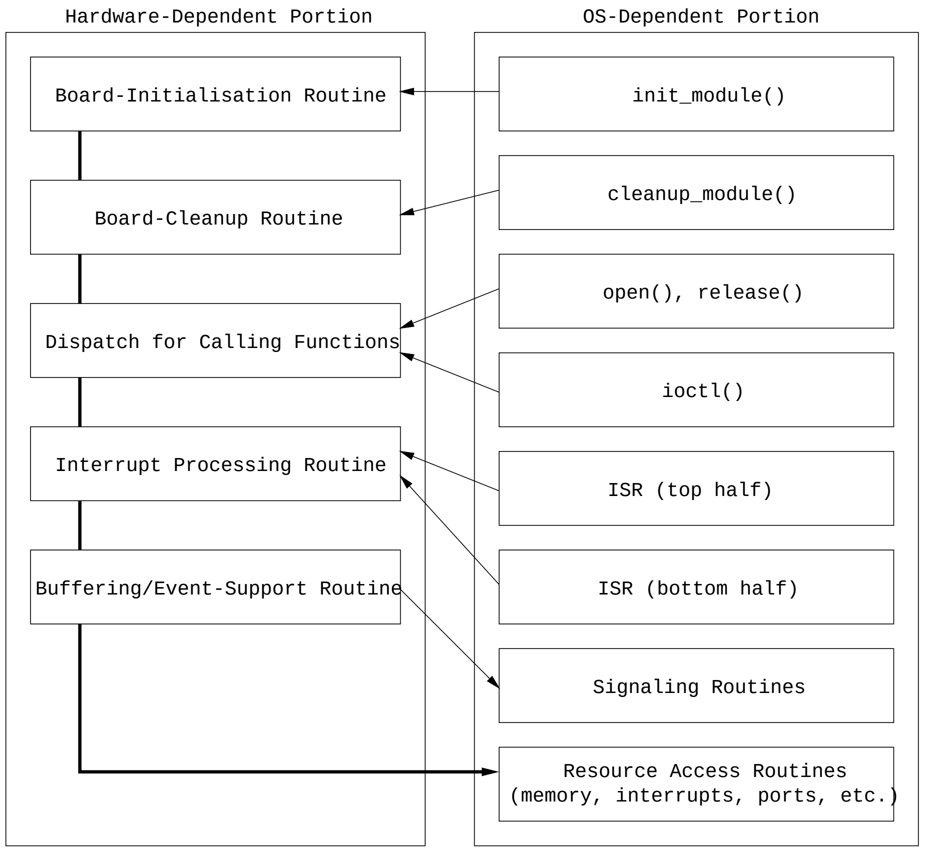
Hardware-Dependent Portion
OS-Dependent Portion
Board-Initialisation Routine
Board-Cleanup Routine
Dispatch for Calling Functions
Interrupt Processing Routine
Buffering/Event-Support Routine
init_module()
cleanup_module()
open(), release()
ioctl()
ISR (top half)
ISR (bottom half)
Signaling Routines
Resource Access Routines
(memory, interrupts, ports, etc.)

Figure 4.1: Structural diagram of pseudo device driver

from the cryptographic core, and finally the device driver proper, that talks to the development board.

4.3 Software - Pseudo device driver

Now that we have decided on a device driver interface, we first concentrate on the software “pseudo” device driver. First lets have a quick look at a structural diagram of what this will look like (Figure 4.1). It should be apparent from Figure 4.2 that all the work is done in the `write()` system call, `read()` just retrieves the data. All functions below the dotted arrows are the exact same as in Figure 2.6.

We use the term “pseudo” because there is no physical device behind the

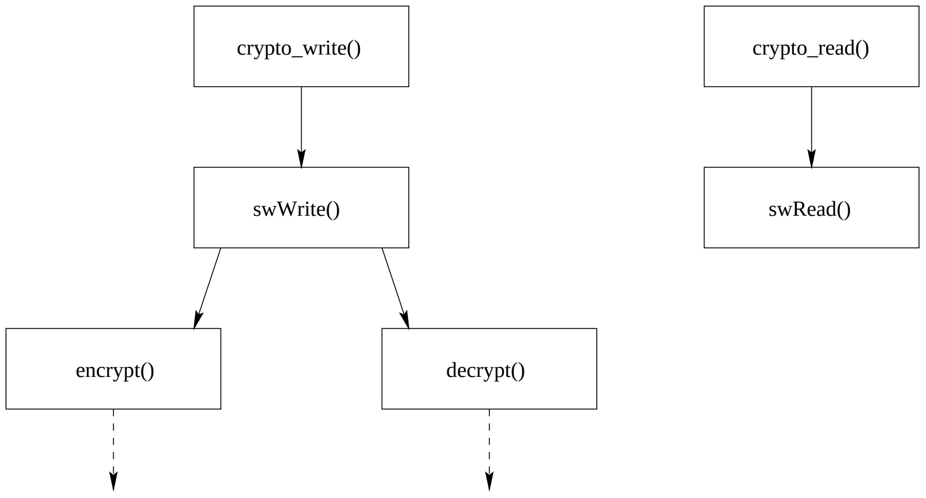

Figure 4.2: Device Driver Architecture

driver, only software emulation of what the firmware version will do, though its function obviously has to be identical. Now, let us re-examine the architecture of a "real" device driver (Figure 4.1) and consider how to set up the user and application interfaces for the driver. Linux gives access to devices as if they were files. Linux users are accustomed to controlling a driver through shell commands and scripts. Therefore, the driver should include a minimal set of functions accessible using read() and write() operations at the Linux shell command prompt.

Linux device drivers exist as files in a directory called */dev.* When you create a driver file, you register the device's name with the Operating System (OS). As part of the registration process, Linux identifies drivers by integers. Each driver has one major number and can have several minor numbers[3]. The OS cares about major numbers only; the driver keeps track of opening and closing of minor numbers. Driver complexity can be minimised by denying device sharing. Write the driver so it opens a minor number and stores the process ID (PID) of the process that opened it. If another process tries to gain access to the same minor number, the driver denies it access.[4]

Otherwise, if the driver code lets several processes have access to the same

[3]The major number indicates the device type, the minor number the subsystem of that device.

[4]Allesandro Rubini, the author of [Rub98], recommended that I use this approach with my driver in some email correspondences I had with him.

board subsystem, the driver would have to stop any ongoing operation (encryption), reconfigure the board for another process, run the new operation, and then reset the board to its previous state. This sequence might not present difficulties if two applications with low throughput need to share the board, but for anything else, this arrangement could present throughput problems.

As part of the driver design, it should be decided if access to the driver is required from the Linux command line. Providing such access lets applications programmer and systems integrator confirm that their hardware is working before compiling and running any code. If one is familiar with Linux shell programming then the driver can be used directly from the command line using the Linux shell commands. Generally it is a good idea to provide enough access to allow board installers to test the board before they write applications.

`Read()` and `write()` function calls can be used for simple devices. When dealing with a complex device that incorporates many functions, however, implementing reads and writes with a command language can become confusing for users, and the driver must take steps to properly parse the command line. So, although these two commands are useful for accessing driver functions from the Linux shell, the `ioctl()`[5] command should be used when accessing the driver using an applications program.

The `ioctl()` command presents a different entry point into the same driver code. Instead of requiring applications programmers to include every I/O and driver parameter in the calling function, programmers can use a pointer to a buffer that contains that information.

In order for the driver to be flexible enough to work with new hardware or to be easily ported to another operating system, one should split the driver into OS-dependent and hardware-dependent parts (Figure 4.1). By doing this, you only need to replace the affected driver portions when adding support for a new card or OS.

4.3.1 Device driver initialisation sequence

1. Make sure the driver supports the device found. Each manufacturer of PCI cards has a unique ID, as does each family of PCI cards. The driver initialisation routine reads from the PCI configuration space information about an installed card. The driver can then check whether these values match those of the cards its designed to handle.

[5] Ioctl provides a "catch-all" entry point for device specific commands.

- Not relevant for a "pseudo" driver.

2. Allocate room for a structure that contains all the device information needed to work with — initialisation settings, status, and runtime parameters within the driver's memory space. Ideally, to support multiple boards, you should create this structure using an array of pointers. For this application a pointer reference was kept to the structure. The kernel function, `kmalloc()`[6] should be used to allocate memory space for structures. Later, in `cleanup_module()`, release the memory space with `kfree()`. Note, however, that `kmalloc()` does not fill the allocated memory with zeroes. In addition, kmalloc allocates memory by pages (4Kbytes/page on x86, 8Kbytes/page on Alpha), so you can efficiently allocate memory by creating memory blocks for device structures. Finally, remember that initialisation is not time critical. Thus, the priority given to `kmalloc()` can be set to GFP_KERNEL, which means the kernel can wait for sufficient memory to become available as other processes free it. To do this we have a static allocation of a structure which then maintains a pointer to device specific data in memory allocated on the heap.

Algorithm 12 Extract from driver.h, §C.6

```
typedef struct Crypto_Dev
    {
      u16 deviceOpen;
      uid_t deviceOwner;
      u8 encDec;               // 0 for pass through 1 for encrypt 2 for decrypt
      u32 inputBufferLength;   // this may be redundant
      u32 outputBufferLength;
      u8 dataPending;          // data left in the output fifo.
      u8 encUsedIV;            // Have we used the IV in the encryption stream
      u8 decUsedIV;            // Ditto for decryption stream.
      void *iFifoBuf;
      void *oFifoBuf;
      struct pci_dev *dev;
      struct wait_queue *readq, *writeq, *openq;
      struct software_key softwareKeys;
      struct tq_struct crypto_queue;
    }
    CryptoDev;
```

3. Some boards require the host computer to download and start executing on-board firmware. This is the point at which this should now be done.

[6] Only as long as that structure is smaller than PAGE_SIZE, which is platform dependent. See [Rub98] for more details.

- Not relevant for a "pseudo" driver.

4. Register all `read()`, `write()`, and `ioctl()` routines with the kernel using the `register_chrdev()` function as shown in the code fragment below. Generally you should write a separate dispatch routine for each type of board the driver supports. This approach eliminates the need for the driver to perform an extra checking step, simplifying driver development. Thus you write, register, and later call just one routine for each card and its subsystem.

Algorithm 13 Extract from driver.c, §C.7.

```
// init_module. NULL is for unimplemented functions.
struct file_operations crypto_Fops = {
  crypto_lseek,                // seek
  crypto_read,
  crypto_write,
  NULL,      // readdir
  NULL,                        // select
  crypto_ioctl,                // ioctl
  NULL,                        // mmap
  crypto_open,
  NULL,                        // flush
  crypto_release               // a.k.a. close
};

// Initialize the module - Register the character device
int
init_module ()
{
.
.


  // Register the character device (atleast try)

  ret_val =
    ret_val + module_register_chrdev(MAJOR_NUM, DEVICE_NAME, &crypto_Fops);
.
.
  // Negative values signify an error
  if (ret_val < 0)
    {
      PINFO ("%s failed with %d\n",
             "Sorry, registering the character device ", ret_val);
      return 1;
    }
.
.
  return 0;
}
```

5. Most modern PCI-boards support PCI bus mastering or direct memory access (DMA), so you must allocate memory pages for these operations. It is generally recommended to use the kernel functions `get_free_page()` and `_get_dma_pages()` for this type of memory allocation. While critical in a real-time data-acquisition board driver, PCI bus mastering and DMA place extra demands on memory, so one has to be realistic about requests for memory space. The kernel will try and satisfy the allocation request, especially if you set the priority to GPF_KERNEL by swapping out as many pages as possible. This swapping can dramatically degrade system performance. In this case, to make things more convenient, the memory used for both the pseudo driver and the actual driver are allocated in the same way.

Algorithm 14 Extract from driver.c, §C.7.

```
AllocateDmaBuffers (void)
{
.
.
  cryptoDevice.oFifoBuf = (void *) __get_free_pages (GFP_DMA, order);
  cryptoDevice.iFifoBuf = (void *) __get_free_pages (GFP_DMA, order);

  if (!cryptoDevice.oFifoBuf || !cryptoDevice.iFifoBuf)
    {
      PINFO("Error allocating DMA pages %s(%d)\n",
__FILE__, __LINE__);
      return 1;
    }
.
.
  return SUCCESS;
}
```

6. Next, if required, an interrupt service routine (ISR) should be registered for the board, assuming convention is followed the ISR is split into two halves, the driver sets the address for the service routines top half.
   - Not relevant for a “pseudo” driver.

7. As a final step, run a hardware initialisation routine to set the board to a known or desired state. If the driver has to deal with multiple cards, it would be normal practice to now increment a “card_installed” counter.

Now we have a functioning device driver which simulates the presence of an actual device. If a user opens the “device” and tries to write to it the operating system calls the `crypto_write()` function which we have registered above.

Algorithm 15 Extract from driver.c, §C.7.

```
static ssize_t
crypto_write (struct file *file,
              const char *buffer, size_t length, loff_t * offset)
{
.
.
  // if the input buffer is not empty then put the caller to sleep.
  while (!isDeviceReadyForData (cDev))
    {
      // If the device was opened in nonblocking mode,try again
      if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;
      interruptible_sleep_on (&cDev->writeq);
      if (signal_pending (current)) // a signal arrived
        return -ERESTARTSYS;
    }
.
.
      retval = swWrite (file, buffer, length, offset);
  return retval;
}
```

And similarly for all the other registered functions. This function just does some sanity checks and then passes the data down to the function that does the real work swWrite() (§C.5). Now, with this and all the other functions we have written, we have a fully functioning "pseudo" driver. It is clear however, that the two main firmware advantages of speed, and absolute confidentiality of the key-authentication-key, are both badly compromised by using the "pseudo" driver.

4.4 Firmware

Having a functional hardware design is one thing, getting data to it over the PCI (Peripheral Component Interconnect [Tom95])bus is another matter. This poses a problem: how to get data transferred to the firmware "encryptor" (the cipher block in §3.3.1 above), have the data encrypted or decrypted, and then returned to the running program. Initially it was envisaged that a PCI or ISA (Industry Standard Architecture) interface board would have to be constructed from scratch, and as the project progressed it seemed that this would not be possible. PCI was selected as the most appropriate interface as this has now become the standard for interfacing peripheral devices directly to the micro-processor. This also allowed the use of a PCI Development board from PLD Applications,[7] which was available.

4.4.1 PCI10K-PROD Board

The PLDA PCI10K-PROD board is a 32/64bit capable PCI board which can operate at both 33MHz and 66MHz PCI bus speeds. It is both 5-volt and 3.3-volt capable, and with it comes PLD applications 32-bit and 64-bit PCI target and master/target controllers. It is based on Altera FLEX 10K technology and has a EPF10K200SFC484-1 on-board. The board also has two chained EPC2 PROMs which automatically program the 10K200 on power up. More detailed information can be found in [Appb][Corb].

4.4.2 PCI-Core and Interface Circuitry.

The PLDA PCI Core[Appa] implements all the functionality required of a PCI device and thus relieves us of this burden. What had yet to be implemented was the interface from the PCI core to the "encryptor". As the development computer only a 32 bit PCI Data path, two problems presented themselves:

1. To get the data to and from my "encryptor"
2. To convert the data from 32 bits to the 96 bits the "encryptor" requires and back again to 32 bits.

PLDA themselves partially solved the first problem, as one of the samples they provided with the board has DMA channel 0 connected to the input of a First In First Out register (FIFO) and DMA Channel 1 connected to the output of the same FIFO. They also provide a 32 bit Input/Output Register which can be used to exchange data with the back end hardware. A 32 to 96 bit de-multiplexer circuit to convert between the input FIFO and my "encryptor" was required, the output of my "encryptor" was then connected to a 96 to 32 bit multiplexer and then to a second FIFO. So now the complete design looked like Figure 4.3. (the "encryptor block at the centre of the figure is the cipher block as documented in Figure 3.5)

Once again the design was broken down into its constituent blocks, and all of them, except the Input/Output Register (IOR) Interface and the PCI Core were written in AHDL. The blocks are connected together differently depending on

[7]http://www.plda.com

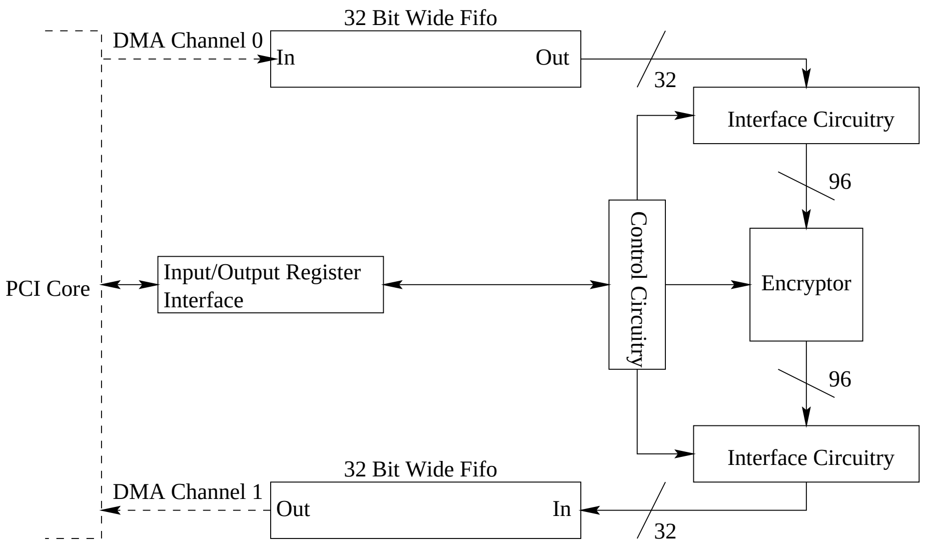

Figure 4.3: PCI Core & Interface Circuitry

the inputs from the control circuitry[8]. See figures 4.4, 4.5 and 4.6 for flowcharts that describe the action of the circuitry.

The Input/Output Register (IOR)is adapted from the supplied PLDA design to incorporate the required design changes. It is written in VHDL. The PCI core is the PLDA implementation used verbatim. In the following sections, we will go through the different blocks used to complete the interface to both FIFOs and the IOR.

4.4.2.1 inputlogic / outputlogic

The Input Logic block (§B.10) simply acts as a de-multiplexer, distributing the 32 bit input across the 96 bits required by the cipher block. Thus the first 32 bit block becomes the bottom 32 bits (a0), the second becomes the middle(a1) and the third becomes the top(a2). An Altera LPM function could have been used, but as these were two of the first circuits designed, it was decided to build our own in the interests of gaining as much knowledge as possible. The Output Logic block (§B.13) is a multiplexer, again an Altera LPM (§3.2.1) could have been used.

[8]ECB Mode Encryption and Decryption do not use the CBC Feedback Latch. So they are functionally Identical.

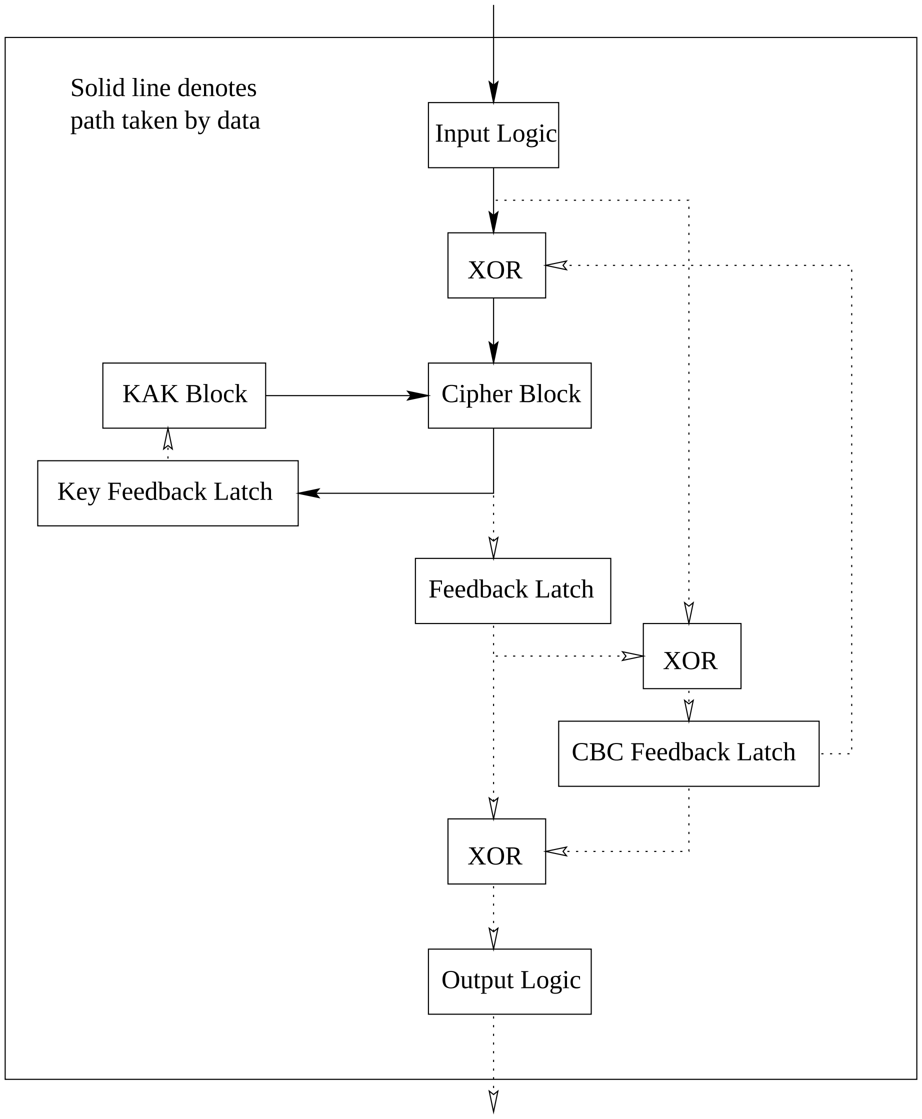

Figure 4.4: Flowchart of Session Key Decryption. (ECB Mode, §2.4.1)

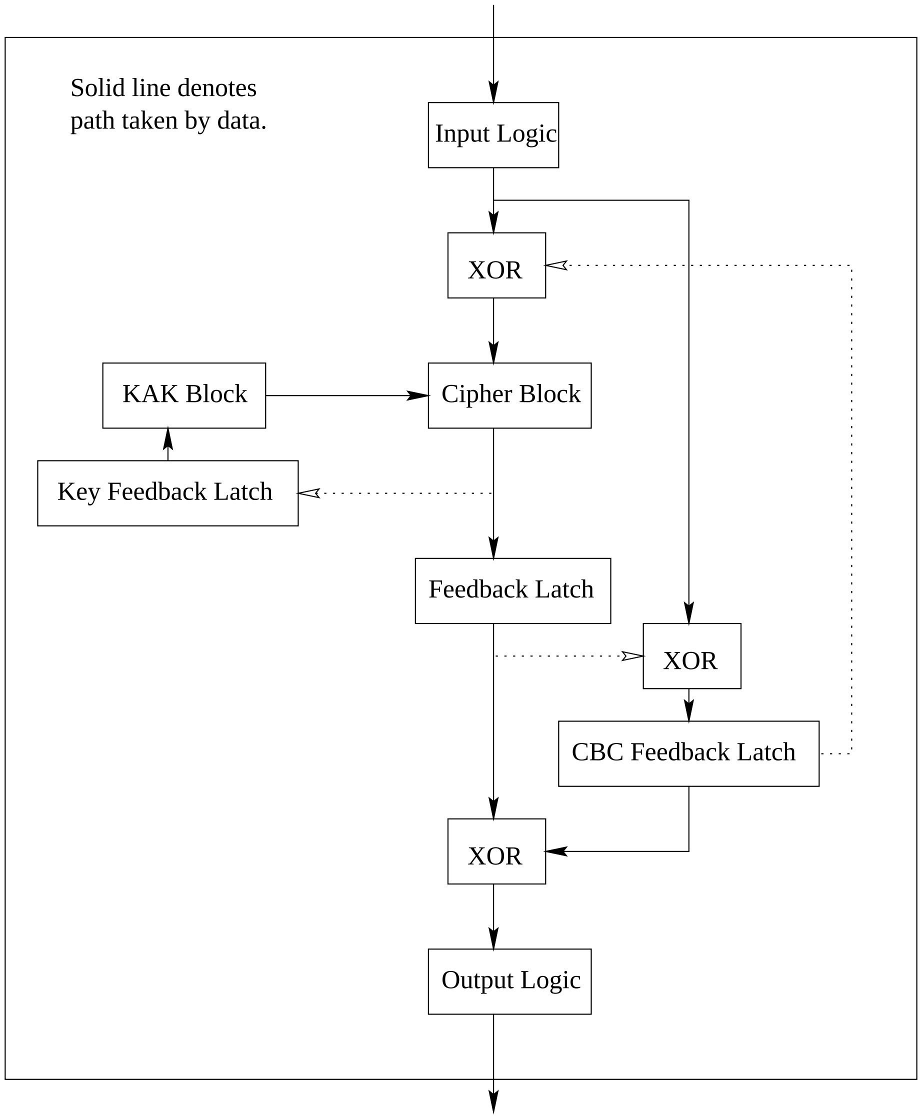

Figure 4.5: Flowchart of CBC Mode (§2.4.2) Decryption.

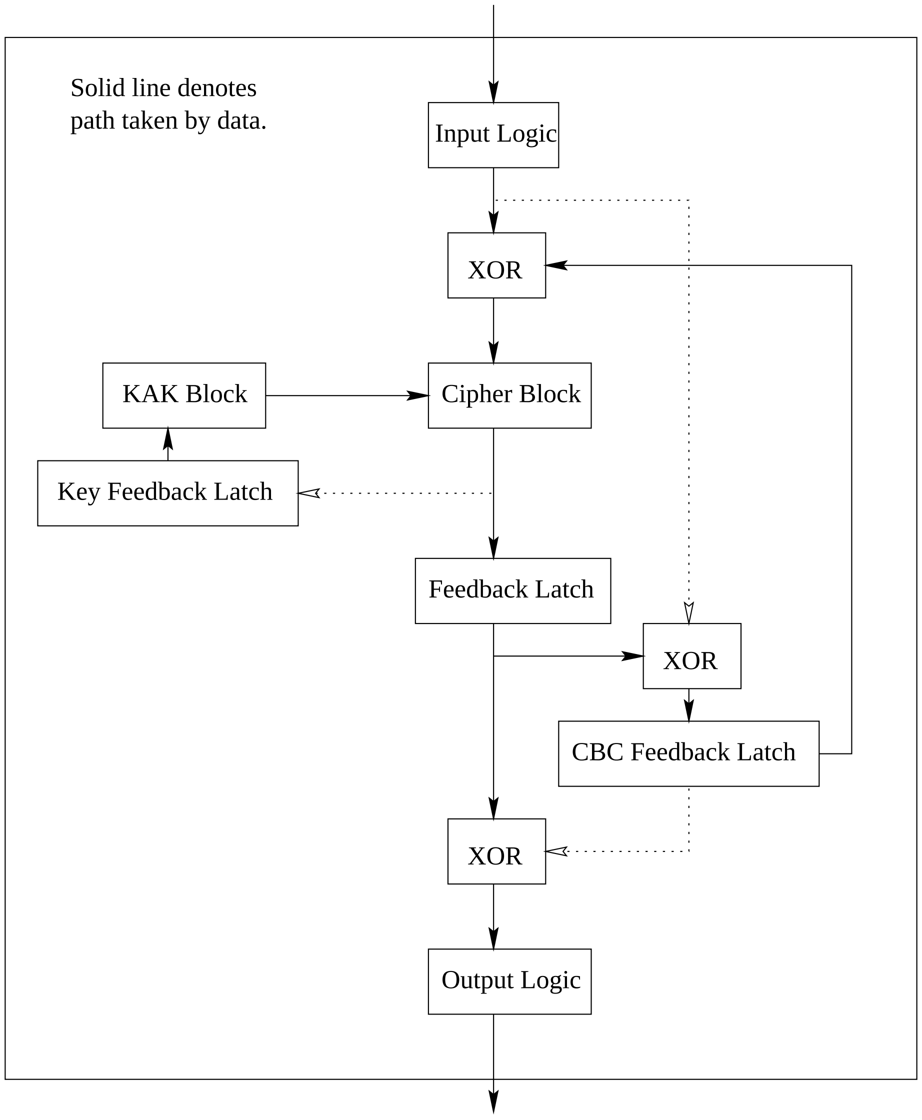

Figure 4.6: Flowchart of CBC Mode (§2.4.2) Encryption.

4.4.2.2 kak

This block (§B.11) stores the Key Authentication Key (KAK) and has an input from the Session Key Latch. Depending on the mode selected the output is either the KAK or the session key.

4.4.2.3 feedbacklogic

This circuit (§B.12)is just a simple 96 bit latch. It is used at the output of the Cipher block to re-time the output signals. As the Key Feedback Latch, it is used to store the session keys for the decryption and encryption circuitry. And as the CBC Feedback latch, it is used to store the Feedback value the next operation.

4.4.2.4 cbcreset

Cbcreset (§B.14) provides a pulse to reset the CBC Feedback latch when:

- The Encryption Key changes.
- The Device is changed from encryption to decryption mode.

It also passed through the CBCRESET signal from the IOR, which is already a pulse. This signal is under device driver control.

4.4.2.5 core

This is block (§B.15) which encapsulates all of the above functions in which all the interconnections are made.

4.4.2.6 controlncore

This is a state machine (§B.16) that controls the operation of the device. The operation is simple, if there is data in the input FIFO, process it. If the output FIFO is full, stop processing. These checks are made before new data is taken in, so there is never data "stuck" in the circuit. Also, if an interrupt is to be sent, then this is the point at which the interrupt will be generated. There are two cases.

1. The output FIFO becomes full (a rising edge on ofifofull). In this case the operation of the encryptor is stalled and needs to be cleared.

2. The input FIFO becomes empty. 18 clock cycles[9] later (excluding the case above) the data is processed and needs to be cleared from the output FIFO.

Case 1 is dealt with in "crypto" below.

4.4.2.7 crypto

Encapsulates (§B.17) both "core" and "controlncore" above, also takes into account the rising edge of OutputFifoFull (case 1 above) and uses it to generate an interrupt. The interconnections between Core and Controlncore are completed in this block. Crypto clock is now finished. All that needs to be connected are the two FIFOs, and all of the requisite control lines.

4.4.2.8 display_control.vhd

This block (§B.18) is a highly modified version of the PLDA example. The new design manages the Input/Output Register, Interrupts and the control of the on-board LEDs for diagnostic and debugging purposes.

4.4.3 Device driver initialisation sequence

Now that we have the "scaffolding" in place, we can go on to implement a full device driver. Now we shall revisit §4.3.1 and briefly cover some of the more important functions required to implement a fully functional device driver.

1. Make sure the driver supports the device found. Each manufacturer of PCI cards has a unique ID, as does each family of PCI cards. The driver code reads from the PCI configuration space information about an installed card. The driver can then check whether these values match those of the cards its designed to handle.

2. Allocate room for a structure that contains all the device information you need to work with:

   - This process is identical to what was done previously in §4.3.1.

3. Some boards require the host computer to download and start executing on-board firmware. This is the point at which this should now be done.

[9]This is the number of cycles required to process a block of data.

Algorithm 16 Extract from hardware.c, §C.9.

```
int pldInit(CryptoDev* const cDev)
{
  int retVal = 0;
.
.
      cDev->dev =
        pci_find_device(PLDA_VENDOR_ID_PLD, PLDA_CRYPTO_ID, cDev->dev);
      if (cDev->dev)
        {
.
.
          // Success
          retVal = 0;
        }
      else
        {
          // Failure
          retVal = 1;
        }
    }
  return retVal;
}
```

   - Not relevant at the moment, but I will discuss cases where this may be relevant in the next chapter.

4. Register all `read()`, `write()`, and `ioctl()` routines with the kernel using the `register_chrdev()` function.

   - This process is identical to what was done previously in §4.3.1.

5. Most modern PCI-boards support PCI bus mastering or direct memory access (DMA), so you must allocate memory pages for these operations.

   - Again, this process is identical to what was done previously in §4.3.1.

6. Next, if required, an interrupt service routine (ISR) should be registered for the board, assuming convention is followed the ISR is split into two halves, the driver sets the address for the service routines top half.

7. As a final step, run a hardware initialisation routine to set the board to a known or desired state. If the driver has to deal with multiple cards, it would be normal practice to now increment a `card_installed` counter.

Now we have a dual purpose device driver which can both simulate the presence of an actual device or talk to the device, depending on parameters we pass to the

Algorithm 17 Extract from hardware.c, §C.9.

```
.
.
          // Fill the task structure, used for the bottom half handler
          cDev->crypto_queue.routine = pldBhInterrupt;
          cDev->crypto_queue.data = cDev;
.
.
```

Algorithm 18 Extract from hardware.c, §C.9.

```
void
pldDeviceReset (struct pci_dev *const pciDev)
{
  u32 ior;
  PDEBUG ("Resetting Back end\n");
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  ior = ior & PLDA_RESET;
  pci_write_config_dword (pciDev, IOR_REG, ior);
}
```

driver when we are loading it into the kernel. Now let us re-visit the `write()` system call which we covered previously:

This time we now have two write calls `pldWrite()`[10] is our new function that talks to the hardware. And based upon the value of the variable ***usingHardware*** (§C.7), the driver either dispatches the data to the hardware to be processed or processes it in software instead. The default action of the driver is that if there is no user intervention and if finds the hardware board, then it will use it. If the user requests, then the software functions will be used instead. This change can only be made when the driver is loaded into the kernel, and to change it requires the driver to be unloaded and re-loaded again.

4.5 Conclusion

Now we have added the "scaffolding" to the software and firmware implementations. This is in the form of a Linux "device driver" architecture and (for the firmware implementation) a development board and extra circuitry that allows the device driver to communicate with the firmware cryptographic device. This gives a useful implementation which can be easily configured to use either the software or firmware version to encrypt data. Now we go on to bench test the two versions and discuss the implications.

[10]PLDA Device Write.

Algorithm 19 Extract from driver.c, §C.7

```
// This function is called when somebody tries to
// write into our device file.
static ssize_t
crypto_write (struct file *file,
              const char *buffer, size_t length,
loff_t  *offset
)
{
.
.
  // if the input buffer is not empty then put the caller to sleep.
  while (!isDeviceReadyForData (cDev))
    {
      // If the device was opened in nonblocking mode, try again
      if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;
      interruptible_sleep_on (&cDev->writeq);
      if (signal_pending (current))     // a signal arrived
        return -ERESTARTSYS;
    }

  if (usingHardware == 1)
    {
      retval = pldWrite (file, buffer, length, offset);
    }
  else
    {
      retval = swWrite (file, buffer, length, offset);
    }
  return retval;

}
```

Chapter 5

Results and Conclusions

Benchtest Results

.

Conclusion & Future Work

5.1 Benchtest Results

In order to benchmark the device, a small program, *mode* (§C.10), was written which can be used to transfer data to and from the device, as well as change keys, change IV, reset, change mode and change stream mode. If the program is invoked as *pld_read* it assumes the user wishes to read data from the device and requires a file name and a length (in bytes). If the program is invoked as *pld_write* it assumes the user wishes to write data to the device and requires a file name to read from. As there are two separate operations involved, two simple shell scripts were written to make calls to the device (§C.12 and C.13). It was decided to test the device on several size files, a file size of 6000 bytes was initially chosen and this was doubled in size until 49152000 bytes was reached. Three encryptions and three decryptions were done, for each size of file, on two different hardware (x86 and Alpha) platforms were and plotted the results. The GNU-Linux *time* command was used in the following manner to get the results:

```
rm 6000enc; /usr/bin/time /testenc.sh 6000
```

This command removes the output from the previous encryption and then does an encryption of the 6000 byte size file (which is named 6000) and then prints its results to standard output. All these results were placed into a text file for retrieval and plotting. Later, the resulting data was plotted with the *gnuplot* application.

5.1.1 x86

The machine used to generate these results was an Intel Pentium II 266 with 256MB of RAM, SuSE Linux 6.4, kernel version 2.2.16 and an Ultra Wide SCSI Hard Drive

5.1.1.1 Comparison

As can be seen in Figure 5.7 there is a visible difference in speed between hardware and software almost immediately. Leading to a difference of over 15 seconds between hardware encryption and software encryption, and a difference of over 50 seconds between hardware decryption and software decryption for a file size of > 48 MB

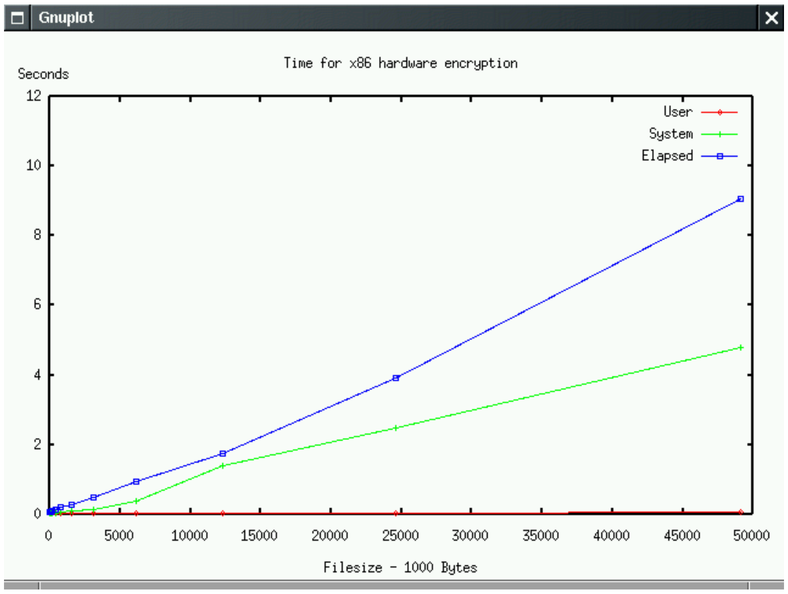

Figure 5.1: Elapsed time for x86 hardware encryption.

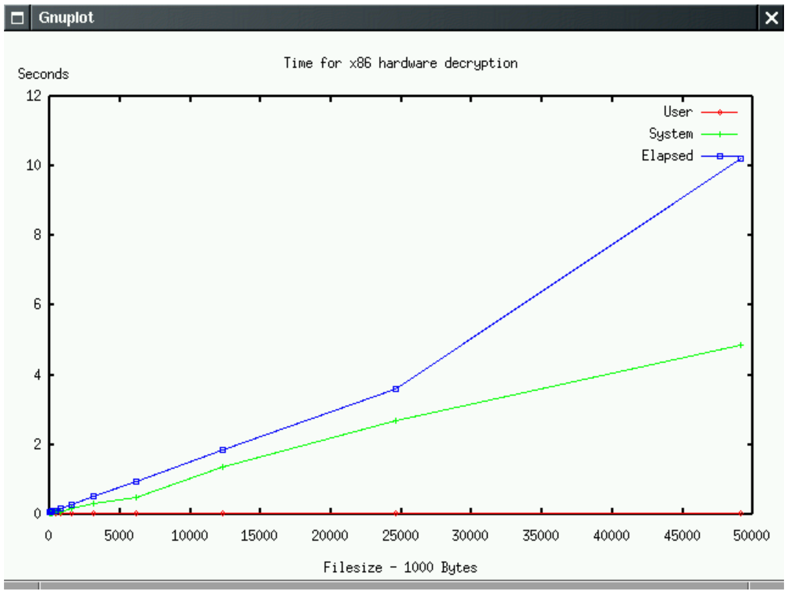

Figure 5.2: Elapsed time for x86 hardware decryption.

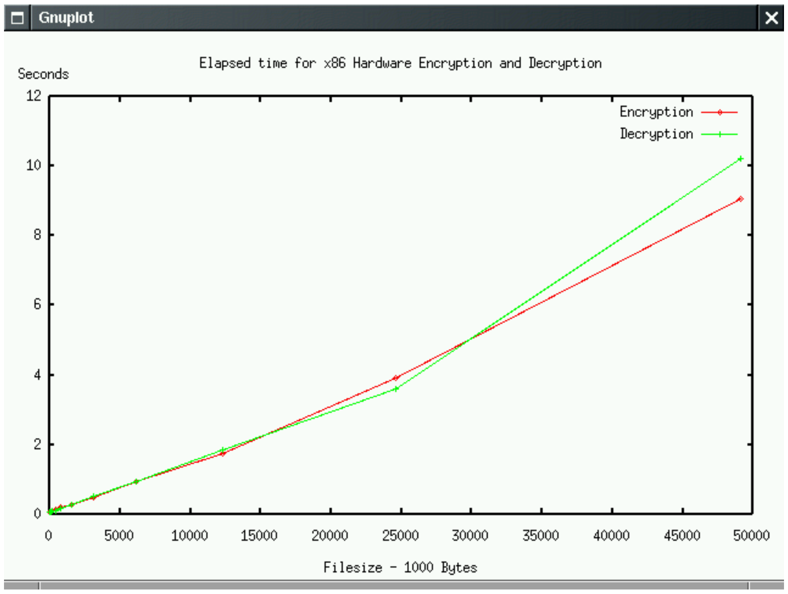

Figure 5.3: Comparison of x86 hardware encryption vs decryption.

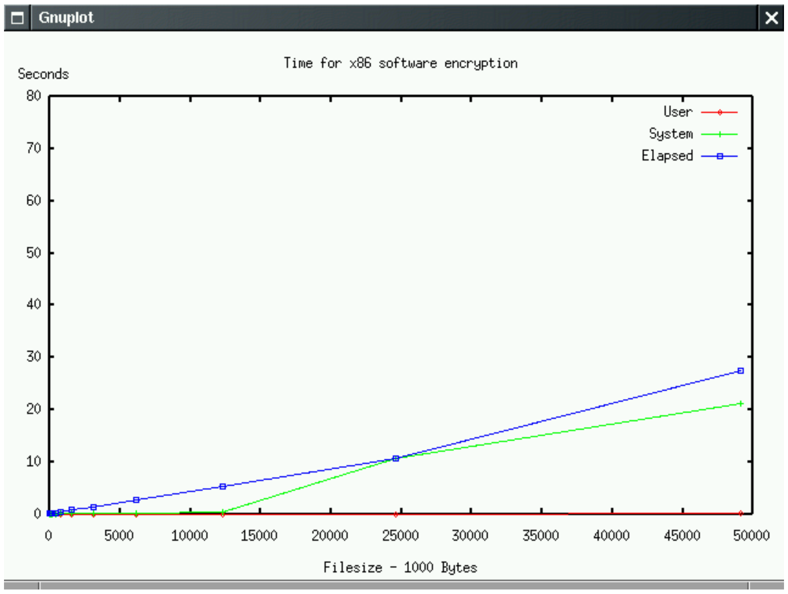

Figure 5.4: Elapsed time for x86 software encryption.

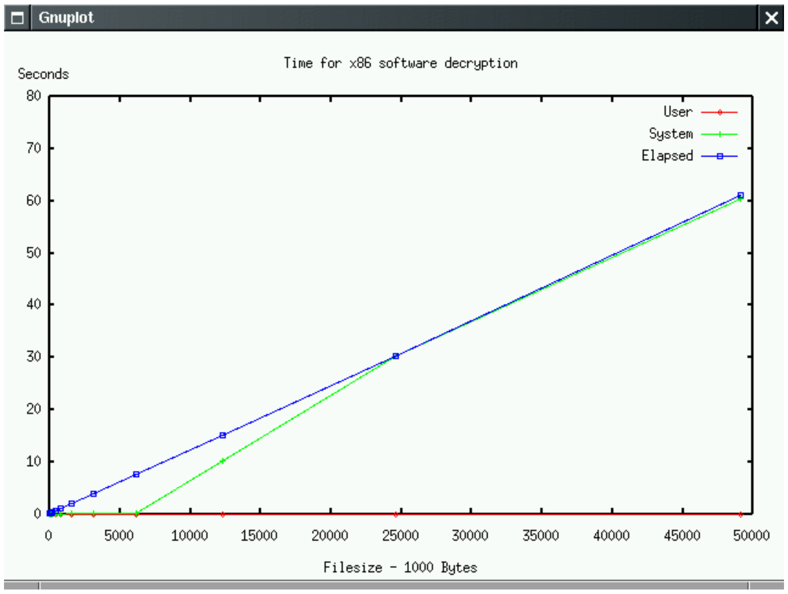

Figure 5.5: Elapsed time for x86 software decryption.

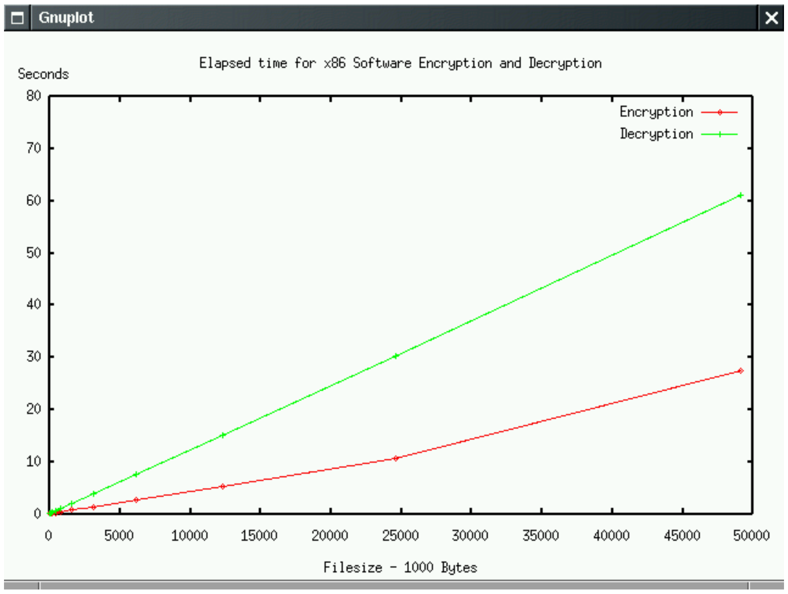

Figure 5.6: Comparison of x86 software encryption vs decryption.

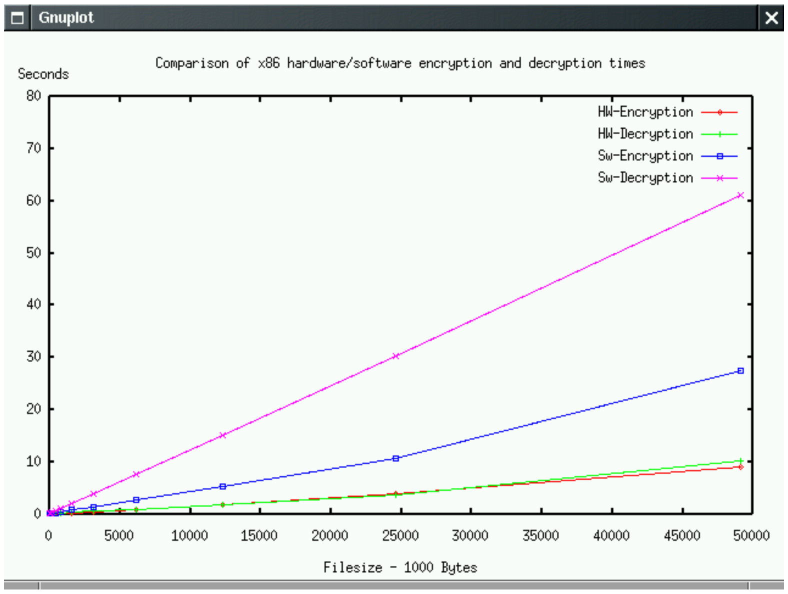

Figure 5.7: Comparison of x86 software vs hardware implementation.

5.1.2 Alpha

The machine used to generate these results was a Quant-X Alpha 21164A with 512MB of RAM, SuSE Linux 6.4, kernel version 2.2.16 and an Ultra Wide SCSI Hard Drive.

5.1.2.1 Comparison

Two different processor architectures were used to aid in eliminating any obscure bugs and to generate comparative test results. The underlying IO subsystems are hopefully similar enough that the results graphed above are more a reflection on the processor and memory throughput than anything else.

As the alpha is closer to the speed of the firmware board we shall use this for comparison. There is a discernable difference between hardware encryption/decryption and software decryption on a file size of 48000 bytes, it does not become clear for software encryption until the file size reaches 192000 bytes. The speed difference between hardware encryption/decryption and software decryption exceeds a factor of two when the file size reaches 1536000 bytes. Based on the graph and estimation, the hardware encryption/decryption would become twice as fast as the software encryption for a file size of approximately

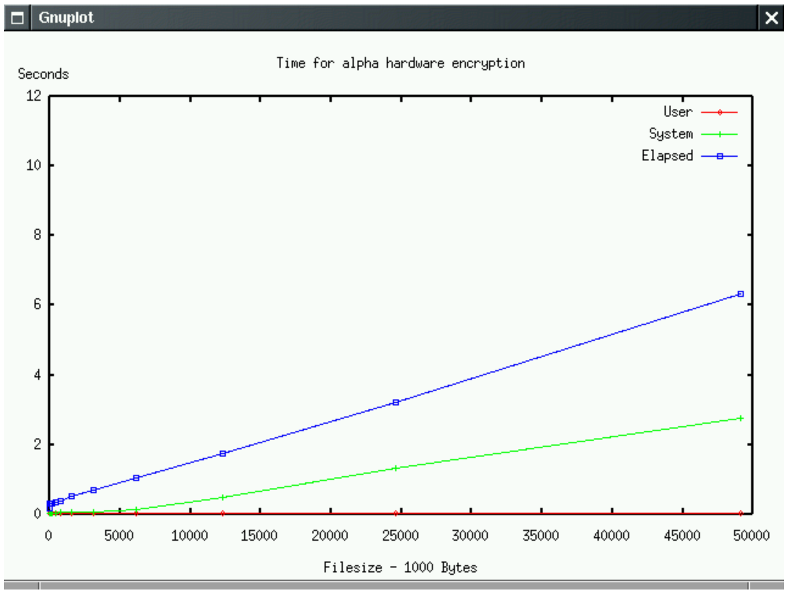

Figure 5.8: Elapsed time for Alpha hardware encryption.

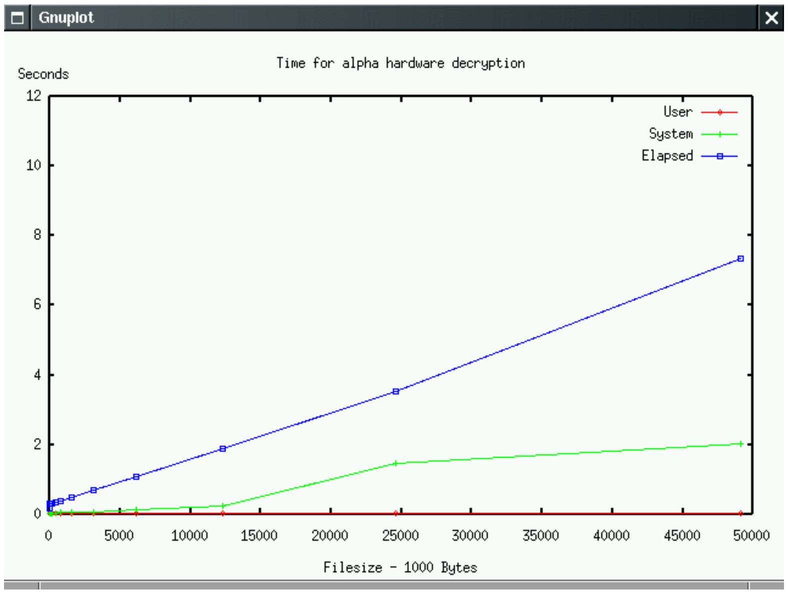

Figure 5.9: Elapsed time for alpha hardware decryption.

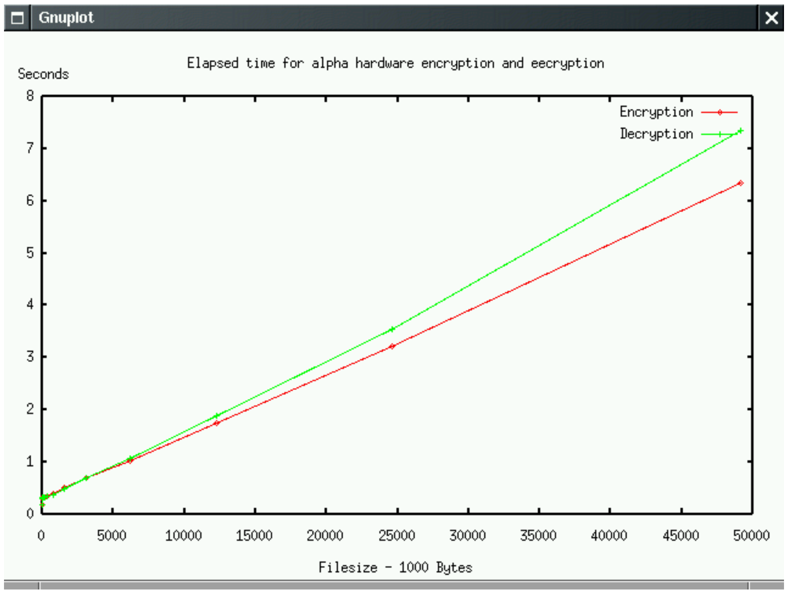

Figure 5.10: Comparison of Alpha hardware encryption vs decryption.

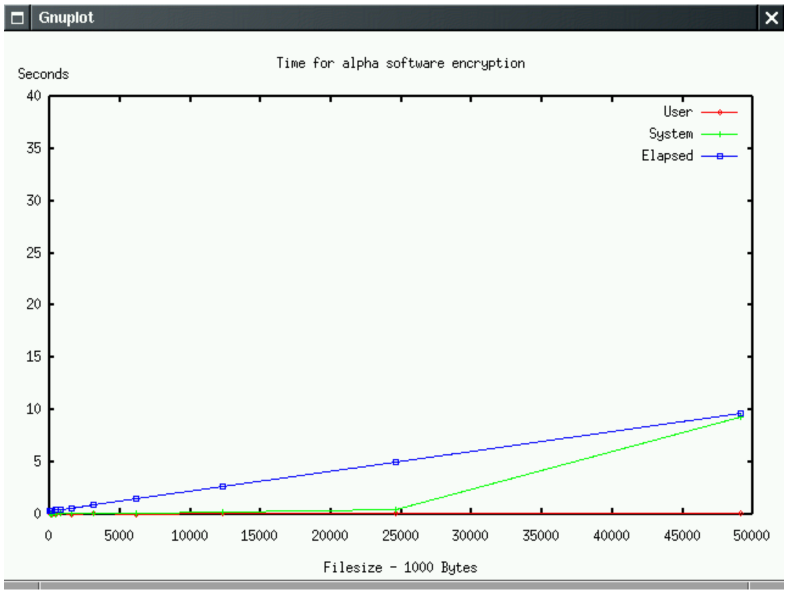

Figure 5.11: Elapsed time for Alpha software encryption.

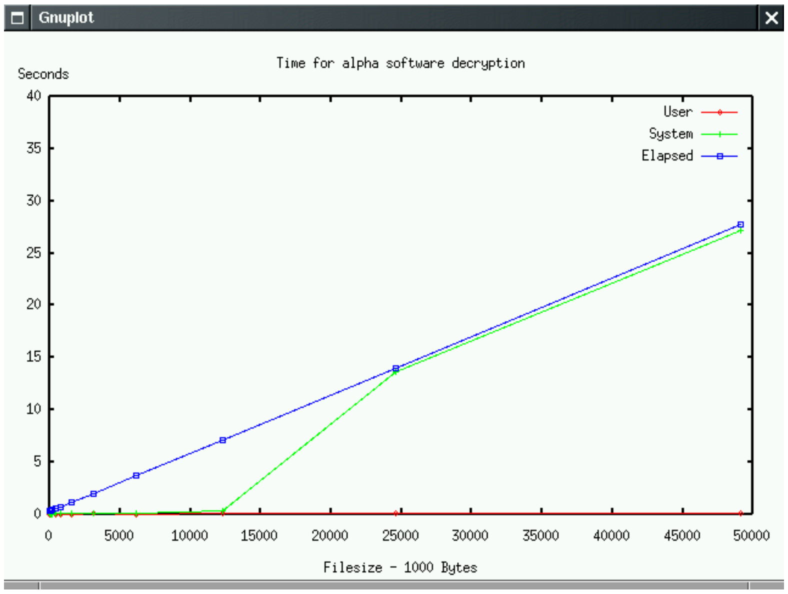

Figure 5.12: Elapsed time for Alpha software decryption.

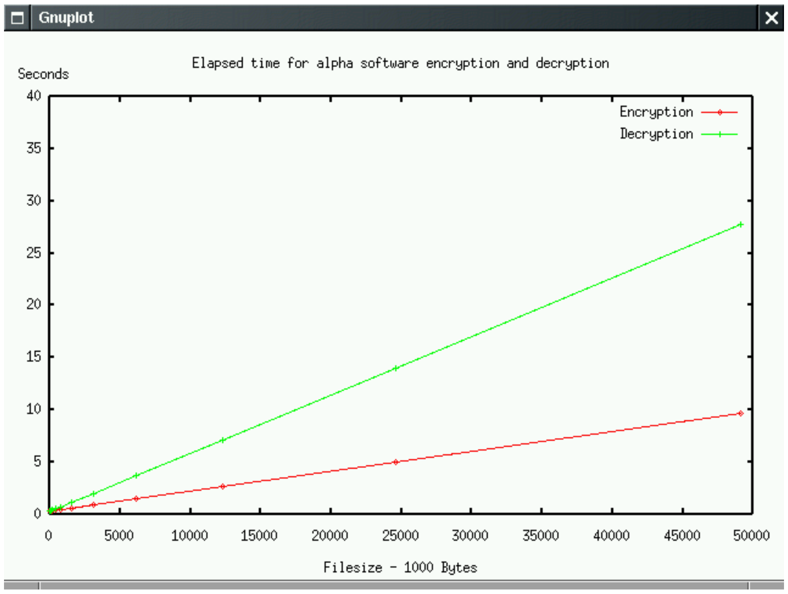

Figure 5.13: Comparison of Alpha software encryption vs decryption

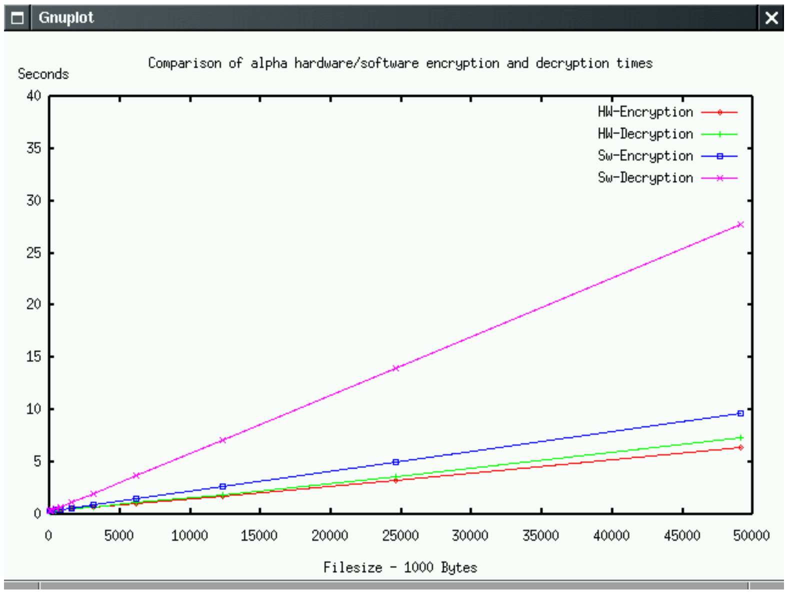

Figure 5.14: Comparison of Alpha software vs hardware implementation

100MB.

Now there is a question raised by the graphs in figures 5.3 and 5.10 which is apparent again in figures 5.7 and 5.14. Why does the board seem faster at encryption than decryption? Intuitively, it should take the same time. This caused some initial concern and investigation was warranted. This is an artificial difference which exposes:

1. A problem with the benchmarking method.
2. The consequence of the Buffer Cache[Bac86].

This came about as the data was generated using the sequence "decipher, encipher, decipher, encipher, decipher, encipher". The file was originally generated by writing a file full of zeros to the disk, in order to make the test realistic, it was first "deciphered" to give a random looking, un-compressible block of data, which was then enciphered, deciphered etc etc. The buffer cache has the following effect. The first time the file is requested, it is read in from the hard-disk, for subsequent requests, it is already in memory, and thus the overhead of reading it in from disk is removed which accounts for the above anomaly. This could have been avoided by pre-reading all files before conducting the test, thus negating the effect of the buffer cache.

5.2 Conclusions and Future Work

This thesis set out to examine various issues related to cryptographic algorithms, and then examine the deployment of a selected algorithm in a real-world setting (in the form of a POSIX-style device driver). The firmware implementation took the form of a cryptographic coprocessor, capable of securely performing encryption, decryption and key protection, while also providing a higher level of performance than a functionally similar software implementation.

From the results shown in the previous section, it is clear that there is a significant performance benefit arising from deploying certain core functions of a cryptographic algorithm in firmware, as opposed to using a purely software-based implementation of the same algorithm.

As further evidence of this benefit, current research and industrial efforts are moving towards cryptographic coprocessors [Gut00][Ito00][chp][Spy][Ci][3co] in order to protect the cryptographic keys (the operating system cannot be subverted into leaking cryptographic keys), and also to relieve the host CPU of the not inconsiderable computational burden of current cryptographic algorithms.

It should also be clear that the field of cryptography is not one to be entered into lightly: there are always performance, security, and flexibility issues to be addressed. These issues were investigated in Chapter 1.

Chapter 2 examined the 3way encryption algorithm in detail in order to gain a better insight to cryptography and cryptographic algorithms. Chapter 2 also investigated the difficulties associated with deploying a cryptographic algorithm. Here, key generation issues were discussed, because without secure keys, any cryptographic system can easily be compromised. However, the treatment of this topic is by no means exhaustive, as key generation is a whole area of study in its own right: generation of cryptographically-secure random numbers is inherently more difficult than it would initially appear.

Following the investigation of the fundamental issues associated with cryptographic systems, a software based implementation of the chosen 3way algorithm was undertaken. With the experience gained from this, work began on a firmware implementation of the same algorithm. The differences between developing in the firmware domain as opposed to software domain were introduced and explored. To conclude, a firmware instantiation of the chosen 3way algorithm was presented.

In order fulfil the final goal of this project, the two implementations (software and hardware) were packaged up as a POSIX-style device driver in order to investigate their performance characteristics.

In Chapter 4, the software implementation was packaged as a "device driver" and used the reference-code implementation described in Chapter 2. The firmware implementation covered in Chapter 3 was packaged as an equivalent cryptographic device, identical in function to the software version. Significant extra circuitry (scaffolding) was required to realise a functioning firmware implementation: specifically, interfacing to the PCI bus and providing input and output data buffers (FIFOs).

Examination of the final, firmware, implementation has suggested further work which could augment the solution:

- Make the FIFOs bigger than PAGE_SIZE (the default size of a memory page of the CPU's memory manager). This is currently an obvious inefficiency in the design, because all UNIX-based programs access files through the standard I/O library. This change could reduce system overhead significantly as the calling program would be put to sleep less often (the reduction would of course depend on the processor architecture and paging system, if any, being used).

- Optimise firmware description (in AHDL) for increased speed. The first improvement would be to increase the operating clock speed of the core block (§4.4.2.5), with the aim of reducing the overall encryption time. The duration of the *CENABLE* signal (see Figure B.10) shows the length of time that the core spends processing data. Currently this time is twelve clock cycles, one to load the data and eleven (§2.3.5) to process it. If this could be reduced, significant performance gains could be achieved.

- Optimise device driver code. To make the driver as efficient as possible, the structure of the software in the driver could be altered slightly to make more use of automatic compiler optimisations (e.g. native word alignment, instruction scheduling, register variables, etc.).

- Reduce the silicon "footprint" (the number of logic elements required) of the firmware without impeding speed, thus allowing room for more functionality, for example: to use more than one cryptographic core. Also, it is clear from Figure 3.4 that there is significant commonality amongst blocks in 3way. If a different design approach were taken, and aggressive inter-block optimisations pursued, it may be possible to reduce the "footprint".

- Add a second, or maybe even a third, cryptographic core with a different algorithm and keys. (e.g. a Public Key Algorithm or the recently-announced AES §1.2.14 selection). This extra functionality would require changes to the device-driver software.
- Add a "daughter card" which could be dynamically reprogrammed (by the device driver) with the correct firmware algorithm depending on the application requirements (this was touched in §4.4.3). Similar concepts are discussed in [CHW00] and [GSB+00]. To preserve the security of such a system, the firmware algorithms themselves would be stored in a read-only memory: the device driver would only instruct the card to load a particular algorithm, and would not supply that algorithm directly.
- Port device-driver to other Operating Systems: the PCI bus is used on non-POSIX software platforms, such as Windows and Mac-OS.
- Integrate into a middle-ware platform. For instance, appropriate hooks are available in CORBA[1] to facilitate interception and encryption/decryption operations. The solution presented here, which uses a standard Unix device driver, would make integration relatively straightforward.

Many of these projects would be substantial design efforts in their own right. This thesis lays a solid foundation, demonstrates that this is a valid approach to take, and points the way for future enhancements.

[1]Common Object Request Broker Architecture - http://www.omg.org

Appendix A

3way Reference Code

This code is taken from [DGV94a] and is the reference for all my work. It is included here for completeness.

```
#define STRT_E 0x0b0b /* round constant of first encryption round */
#define STRT_D 0xb1b1 /* round constant of first decryption round */
#define NMBR 11 /* number of rounds is 11 */

typedef unsigned long int word32 ;

/* the program only works correctly if long = 32bits */

void mu(word32 *a) /* inverts the order of the bits of a */
{
  int i ;
  word32 b[3] ;

  b[0] = b[1] = b[2] = 0 ;
  for( i=0 ; i<32 ; i++ ){
    b[0] <<= 1 ; b[1] <<= 1 ; b[2] <<= 1 ;
    if(a[0]&1) b[2] |= 1 ;
    if(a[1]&1) b[1] |= 1 ;
    if(a[2]&1) b[0] |= 1 ;
    a[0] >>= 1 ; a[1] >>= 1 ; a[2] >>= 1 ;
  }

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}

void gamma(word32 *a) /* the nonlinear step */
{
  word32 b[3] ;

  b[0] = a[0] ^ (a[1]|(~a[2])) ;
  b[1] = a[1] ^ (a[2]|(~a[0])) ;
  b[2] = a[2] ^ (a[0]|(~a[1])) ;

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}

void theta(word32 *a) /* the linear step */
{
  word32 b[3];

  b[0] = a[0] ^ (a[0]>>16) ^ (a[1]<<16) ^ (a[1]>>16) ^ (a[2]<<16) ^
              (a[1]>>24) ^ (a[2]<<8) ^ (a[2]>>8) ^ (a[0]<<24) ^
              (a[2]>>16) ^ (a[0]<<16) ^ (a[2]>>24) ^ (a[0]<<8) ;
  b[1] = a[1] ^ (a[1]>>16) ^ (a[2]<<16) ^ (a[2]>>16) ^ (a[0]<<16) ^
              (a[2]>>24) ^ (a[0]<<8) ^ (a[0]>>8) ^ (a[1]<<24) ^
              (a[0]>>16) ^ (a[1]<<16) ^ (a[0]>>24) ^ (a[1]<<8) ;
  b[2] = a[2] ^ (a[2]>>16) ^ (a[0]<<16) ^ (a[0]>>16) ^ (a[1]<<16) ^
              (a[0]>>24) ^ (a[1]<<8) ^ (a[1]>>8) ^ (a[2]<<24) ^
              (a[1]>>16) ^ (a[2]<<16) ^ (a[1]>>24) ^ (a[2]<<8) ;

  a[0] = b[0] ; a[1] = b[1] ; a[2] = b[2] ;
}

void pi_1(word32 *a)
{
  a[0] = (a[0]>>10) ^ (a[0]<<22);
  a[2] = (a[2]<<1) ^ (a[2]>>31);
}

void pi_2(word32 *a)
{
  a[0] = (a[0]<<1) ^ (a[0]>>31);
  a[2] = (a[2]>>10) ^ (a[2]<<22);
}

void rho(word32 *a) /* the round function */
{
  theta(a) ;
  pi_1(a) ;
  gamma(a) ;
  pi_2(a) ;
}

void rndcon_gen(word32 strt,word32 *rtab)
{ /* generates the round constants */
  int i ;


  for(i=0 ; i<=NMBR ; i++ )
    {
      rtab[i] = strt ;
      strt <<= 1 ;
      if( strt&0x10000 ) strt ^= 0x11011 ;
    }
}
```

```
void encrypt(word32 *a, word32 *k)
{
  int i ;
  word32 rcon[NMBR+1] ;

  rndcon_gen(STRT_E,rcon) ;
  for( i=0 ; i<NMBR ; i++ )
    {
      a[0] ^= k[0] ^ (rcon[i]<<16) ;
      a[1] ^= k[1] ;
      a[2] ^= k[2] ^ rcon[i] ;
      rho(a) ;
    }
  a[0] ^= k[0] ^ (rcon[NMBR]<<16) ;
  a[1] ^= k[1] ;
  a[2] ^= k[2] ^ rcon[NMBR] ;
  theta(a) ;
}

void decrypt(word32 *a, word32 *k)
{
  int i ;
  word32 ki[3] ; /* the ‘inverse’ key */
  word32 rcon[NMBR+1] ; /* the ‘inverse’ round constants */

  ki[0] = k[0] ;
  ki[1] = k[1] ;
  ki[2] = k[2] ;
  theta(ki) ;
  mu(ki) ;

  rndcon_gen(STRT_D,rcon) ;
  mu(a) ;
  for( i=0 ; i<NMBR ; i++ )
    {
      a[0] ^= ki[0] ^ (rcon[i]<<16) ;
      a[1] ^= ki[1] ;
      a[2] ^= ki[2] ^ rcon[i] ;
      rho(a) ;
    }
  a[0] ^= ki[0] ^ (rcon[NMBR]<<16) ;
  a[1] ^= ki[1] ;
  a[2] ^= ki[2] ^ rcon[NMBR] ;
  theta(a) ;
  mu(a) ;
}

//Testprogram
#include !stdio.h? #include !stdlib.h? #include "threewayref.c"

void printvec(word32 *a)
{
printf("%08x %08x %08x\n",a[2],a[1],a[0]) ;
}
```

```
main()
{
  word32 a[3], k[3];

  scanf("%x %x %x %x %x %x",a+2,a+1,a,k+2,k+1,k) ;
  printf("key : ") ; printvec(k) ;
  printf("plaintext : ");  printvec(a) ; encrypt(a,k) ;
  printf("ciphertext : "); printvec(a) ; decrypt(a,k) ;
  /*printf("checking : ") ; printvec(a) ;  */
}
/*
Testvalues key : 00000000 00000000 00000000
plaintext : 00000001 00000001 00000001
ciphertext : ad21ecf7 83ae9dc4 4059c76e

key : 00000004 00000005 00000006
plaintext : 00000001 00000002 00000003
ciphertext : cab920cd d6144138 d2f05b5e

key : bcdef012 456789ab def01234
plaintext : 01234567 9abcdef0 23456789
ciphertext : 7cdb76b2 9cdddb6d 0aa55dbb

key : cab920cd d6144138 d2f05b5e
plaintext : ad21ecf7 83ae9dc4 4059c76e
ciphertext : 15b155ed 6b13f17c 478ea871
*/
```

Appendix B

Altera Text Design Files

B.1 Cipher

```
-- cipher.tdf
-- This function encapsulates the encrypt/decrypt functions and selects between them.
-- It also takes the encryption key and feeds it to both the encrypt and decrypt
-- functions in the correct format
-- John Ronan, 20000707

INCLUDE "lpm_dff.inc";
INCLUDE "lpm_counter.inc";
INCLUDE "lpm_ff.inc";
INCLUDE "encipher.inc";
INCLUDE "decipher.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "theta.inc";
INCLUDE "mu.inc";

CONSTANT REG_LENGTH        = H"20"; -- 32bit Registers
CONSTANT COUNT_LENGTH      = H"4";  -- number of bits required to count to 11
CONSTANT COUNTER_TERMINATE = H"C";  -- counter needs a terminal value for test, gets
                                    -- reset at 11 anyway.

SUBDESIGN cipher
(
        DataIN0[REG_LENGTH - 1..0]      : INPUT;
        DataIN1[REG_LENGTH - 1..0]      : INPUT;
        DataIN2[REG_LENGTH - 1..0]      : INPUT;
        CLOCK                           : INPUT;
        E_D                             : INPUT; -- Encrypt = 0, Decrypt = 1
        SSET                            : INPUT; -- Reset (Synchronous SET) line
        LOAD                            : INPUT;
        COUNTER_EN                      : INPUT; -- Enable the counter to count
                                                 -- ciphering operation to take place
        CIPHERKEY0[REG_LENGTH - 1..0]   : INPUT;
        CIPHERKEY1[REG_LENGTH - 1..0]   : INPUT; -- Encryption Key inputs
        CIPHERKEY2[REG_LENGTH - 1..0]   : INPUT;
        TERM                            : OUTPUT; -- Test Point, Terminate Count
        DataOUT0[REG_LENGTH - 1..0]     : OUTPUT;
        DataOUT1[REG_LENGTH - 1..0]     : OUTPUT;
        DataOUT2[REG_LENGTH - 1..0]     : OUTPUT;
        COUNTOUT[COUNT_LENGTH -1..0]    : OUTPUT; -- Test Point, Value of Counter
)

VARIABLE
        enc      :encipher;
        dec      :decipher;
        t        :theta;
        m        :mu;
        count    :lpm_counter WITH(LPM_WIDTH=COUNT_LENGTH,
                                LPM_MODULUS=COUNTER_TERMINATE,
                                LPM_SVALUE=H"000000");
        out_mux0 :lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");
        out_mux1 :lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");
        out_mux2 :lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");
BEGIN
        out_mux0.sel[0] = E_D;
        out_mux1.sel[0] = E_D;
        out_mux2.sel[0] = E_D;

        count.clock = clock;
        count.cnt_en = COUNTER_EN;
        count.sset = SSET;

        enc.clock = CLOCK;
        dec.clock = CLOCK;

        enc.load = LOAD;
        dec.load = LOAD;

        enc.a0[] = DataIN0[];
        dec.a0[] = DataIN0[];
        enc.a1[] = DataIN1[];
        dec.a1[] = DataIN1[];
        enc.a2[] = DataIN2[];
        dec.a2[] = DataIN2[];

        -- Read in the encryption key from outside
        enc.cipherkey0[] = CIPHERKEY0[];
        enc.cipherkey1[] = CIPHERKEY1[];
        enc.cipherkey2[] = CIPHERKEY2[];

        -- put it through theta and mu to get the decryption key
        t.a0[] = CIPHERKEY0[];
        t.a1[] = CIPHERKEY1[];
        t.a2[] = CIPHERKEY2[];

        m.a0[] = t.q0[];
        m.a1[] = t.q1[];
        m.a2[] = t.q2[];
        -- feed the decryption key into the decryption circuit
        dec.cipherkey0[] = m.q0[];
        dec.cipherkey1[] = m.q1[];
        dec.cipherkey2[] = m.q2[];

        enc.sel0 = count.q[0];
        dec.sel0 = count.q[0];
        enc.sel1 = count.q[1];
        dec.sel1 = count.q[1];
        enc.sel2 = count.q[2];
        dec.sel2 = count.q[2];
        enc.sel3 = count.q[3];
        dec.sel3 = count.q[3];

        % OUTPUT %
        out_mux0.data[0][] = enc.q0[];
        out_mux0.data[1][] = dec.q0[];
        out_mux1.data[0][] = enc.q1[];
        out_mux1.data[1][] = dec.q1[];
        out_mux2.data[0][] = enc.q2[];
        out_mux2.data[1][] = dec.q2[];
        DataOUT0[] = out_mux0.result[];
        DataOUT1[] = out_mux1.result[];
        DataOUT2[] = out_mux2.result[];

        COUNTOUT[] = count.q[];
        TERM = count.eq[11];
END;
```

B.1.1 Timing Diagrams, Cipher

B.2 encipher

```
-- encipher.tdf
-- Takes the key and, with the correct input of control signals, performs the
-- encryption of a block of data. mlatch is used to implement the for loop in
-- the code, note the roundconstants are hard coded as input to the multiplexer
-- which feeds them out based on the select input signals.
-- John Ronan, 2000708

INCLUDE "rho.inc";
INCLUDE "theta.inc";
INCLUDE "round.inc";
INCLUDE "lpm_xor.inc";
INCLUDE "mlatch.inc";
INCLUDE "lpm_constant.inc";
INCLUDE "lpm_mux.inc";

CONSTANT REG_LENGTH  = H"20"; -- 32bit Registers
CONSTANT A1_XOR_SIZE = H"2";  -- compare two inputs
CONSTANT A0_XOR_SIZE = H"3";  -- 3 INPUTS
CONSTANT RC_WIDTH    = H"10";
```

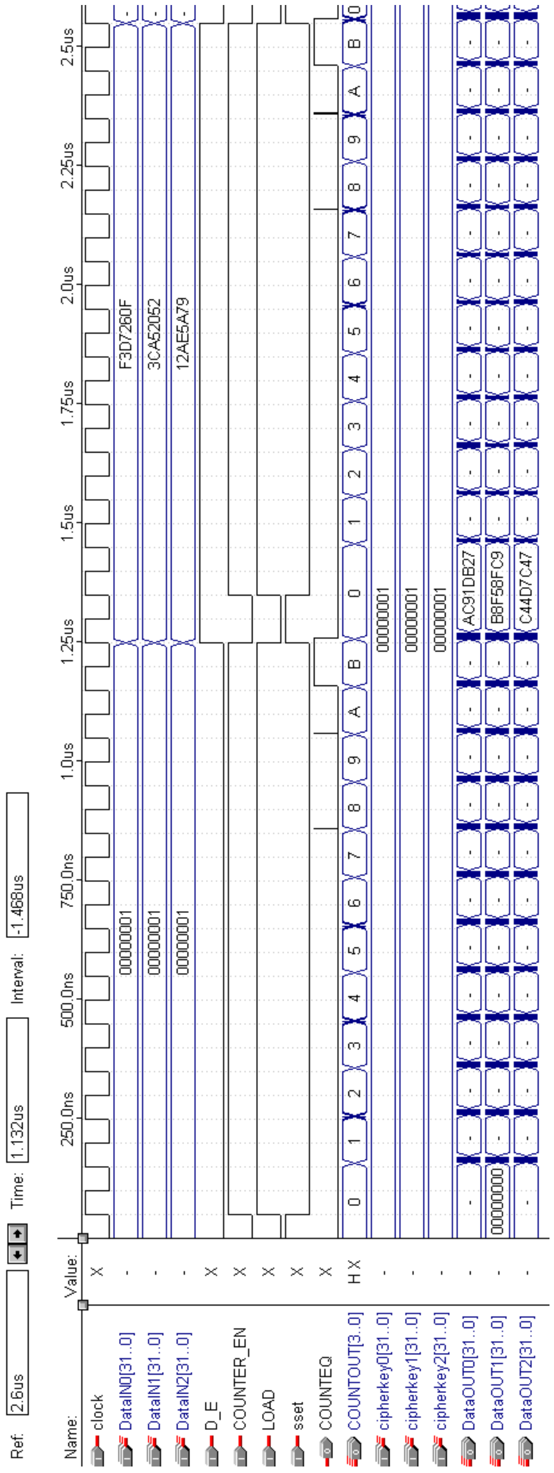

Figure B.1: Cipher, full test timing diagram.

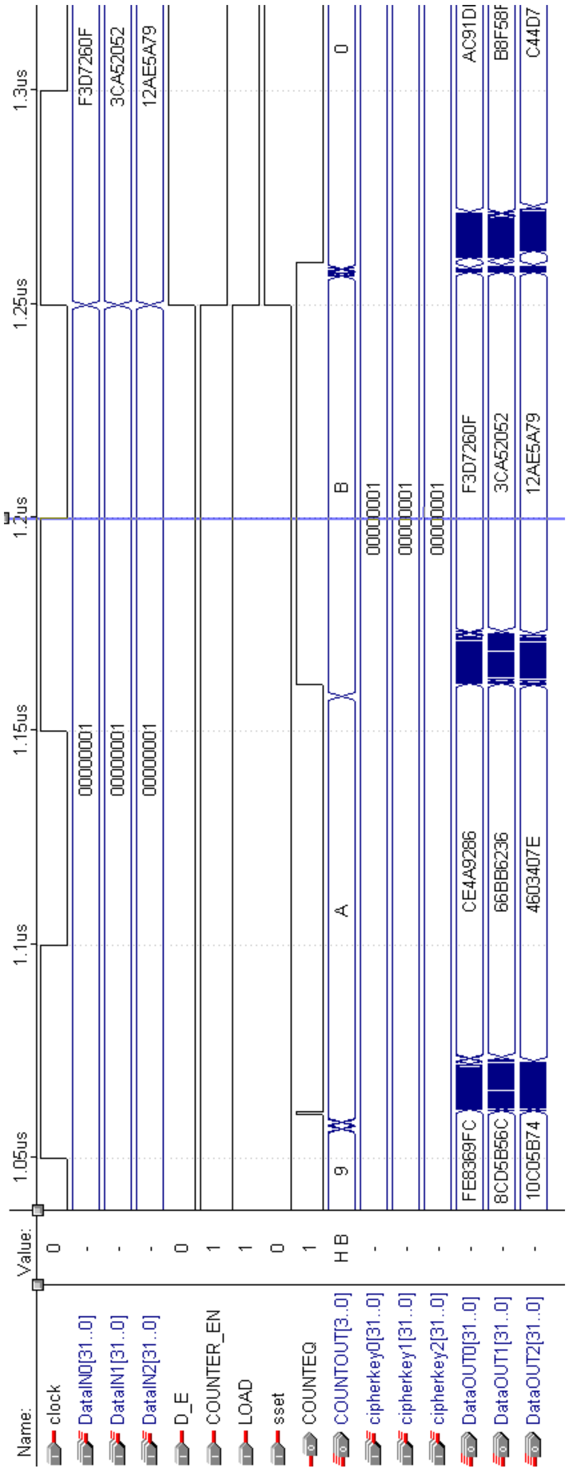

Figure B.2: Cipher, end of encryption cycle, close up.

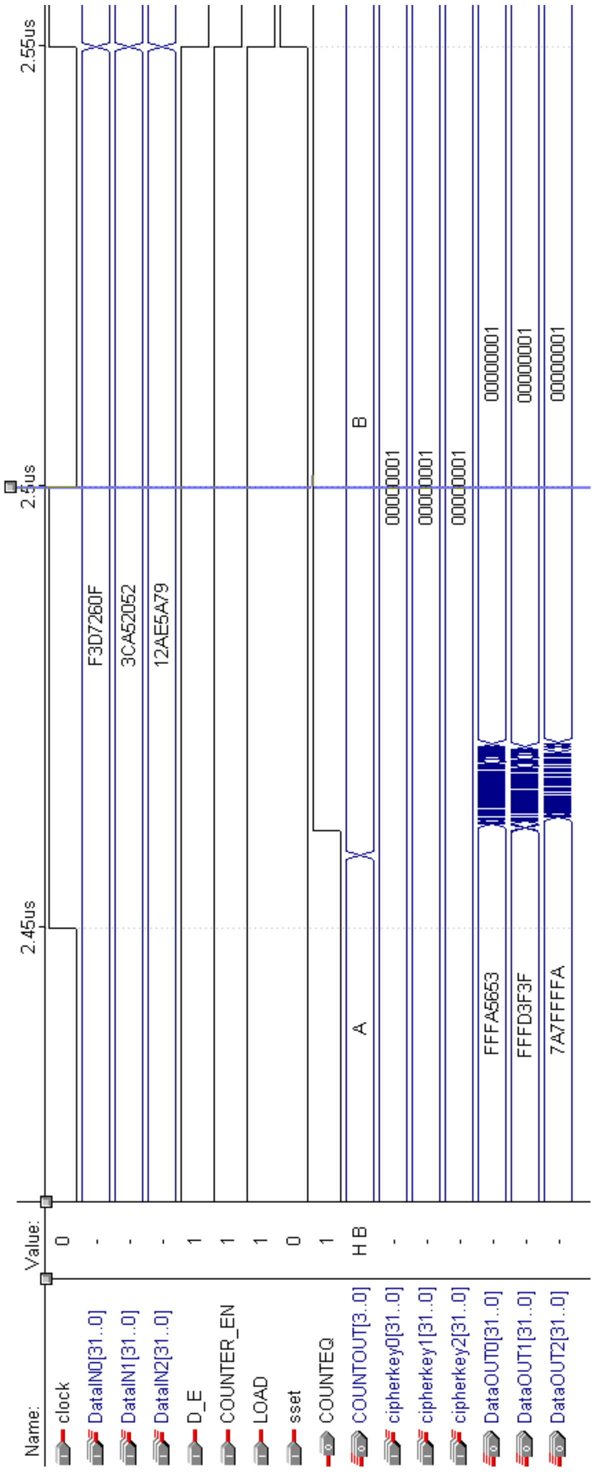

Figure B.3: Cipher, end of decryption cycle, close up.

```
% Test values generated from the Reference code
INPUTS  A0=cfbd782f
        A1=31761927
        A2=234809bf
 OUTPUTS Q0=324a8748
        Q1=023c96d8
        Q2=ecf5f740%

SUBDESIGN encipher
(
        A0[REG_LENGTH - 1..0]        : INPUT;
        A1[REG_LENGTH - 1..0]        : INPUT;
        A2[REG_LENGTH - 1..0]        : INPUT;
        CIPHERKEY0[REG_LENGTH -1..0] : INPUT;  -- encryption key input
        CIPHERKEY1[REG_LENGTH -1..0] : INPUT;
        CIPHERKEY2[REG_LENGTH -1..0] : INPUT;
        LOAD                         : INPUT;  -- control signal to latchs
        CLOCK                        : INPUT;
        SEL0                         : INPUT;  -- control inputs
        SEL1                         : INPUT;
        SEL2                         : INPUT;
        SEL3                         : INPUT;
        Q0[REG_LENGTH - 1..0]        : OUTPUT;
        Q1[REG_LENGTH - 1..0]        : OUTPUT;
        Q2[REG_LENGTH - 1..0]        : OUTPUT;
)

VARIABLE
        r               : round;
        A0_00   : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        A0_01   : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A1_XOR_SIZE);
        A0_02   : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        T               : theta;
        l0              : mlatch WITH(REG_LENGTH = REG_LENGTH);
        l1              : mlatch WITH(REG_LENGTH = REG_LENGTH);
        l2              : mlatch WITH(REG_LENGTH = REG_LENGTH);
        m               : lpm_mux WITH(LPM_WIDTH=RC_WIDTH,LPM_SIZE=H"10",
                                       LPM_WIDTHS=H"4");

BEGIN

        m.sel[0] = SEL0;
        m.sel[1] = SEL1;
        m.sel[2] = SEL2;
        m.sel[3] = SEL3;

        -- Precalculated Round Constants
        m.data[0][] = H"0b0b";
        m.data[1][] = H"1616";
        m.data[2][] = H"2c2c";
        m.data[3][] = H"5858";
        m.data[4][] = H"b0b0";
        m.data[5][] = H"7171";
        m.data[6][] = H"e2e2";
```

```
m.data[7][] = H"d5d5";
m.data[8][] = H"bbbb";
m.data[9][] = H"6767";
m.data[10][] = H"cece";
m.data[11][] = H"8d8d"; -- last round constant
m.data[12][] = H"0000";
m.data[13][] = H"0000";
m.data[14][] = H"0000";
m.data[15][] = H"0000";

-- feedback of round data (for loop)
l0.clock = CLOCK;
l1.clock = CLOCK;
l2.clock = CLOCK;
l0.enable = VCC;
l1.enable = VCC;
l2.enable = VCC;
l0.load = LOAD;
l1.load = LOAD;
l2.load = LOAD;

-- inputs to the latch... data in and feedback.
l0.a0[] = A0[];
l0.r0[] = r.q0[];
l1.a0[] = A1[];
l1.r0[] = r.q1[];
l2.a0[] = A2[];
l2.r0[] = r.q2[];

-- encryption key fed into the round function
r.cipherkey0[] = CIPHERKEY0[];
r.cipherkey1[] = CIPHERKEY1[];
r.cipherkey2[] = CIPHERKEY2[];

-- round constant selected based on input to multiplexer
r.roundconstant[] = m.result[];

-- data from latch to round function
r.a0[] = l0.qout[];
r.a1[] = l1.qout[];
r.a2[] = l2.qout[];

-- round completed
-- a0
-- data into exclusive or
A0_00.DATA[0][] = CIPHERKEY0[];
A0_00.DATA[1][] = l0.qout[];

-- take in the round constant
FOR i IN 0 TO 15 GENERATE
        A0_00.DATA[2][I] = GND;
END GENERATE;
FOR i IN 0 TO 15 GENERATE
        A0_00.DATA[2][I+16] = m.result[i];
END GENERATE;
```

```
        -- a1
        A0_01.DATA[0][] = CIPHERKEY1[];
        A0_01.DATA[1][] = l1.qout[];

        -- a2
        A0_02.DATA[0][] = CIPHERKEY2[];
        A0_02.DATA[1][] = l2.qout[];

        FOR i IN 0 TO 15 GENERATE
                A0_02.DATA[2][I+16] = GND;
        END GENERATE;

        FOR i IN 0 TO 15 GENERATE
                A0_02.DATA[2][I] = m.result[i];
                A0_02.DATA[2][I] = GND;
        END GENERATE;

        -- last operation is theta.
        T.A0[] = A0_00.RESULT[];
        T.A1[] = A0_01.RESULT[];
        T.A2[] = A0_02.RESULT[];

        -- data is now encrypted.
        Q0[] = T.Q0[];
        Q1[] = T.Q1[];
        Q2[] = T.Q2[];
END;
```

B.2.1 Timing diagrams, encipher

B.3 mlatch

```
-- mlatch.tdf
-- Takes two inputs one is data the other is feedback data and selects from one
-- or the other based on the load inputline.
-- John Ronan, 20000708

INCLUDE "dffe.inc";

-- Default value, gets overwritten by outside parameters
PARAMETERS
(
   REG_LENGTH = 4
);

SUBDESIGN mlatch
(
        clock                   : INPUT;
        a0[REG_LENGTH - 1..0]   : INPUT;  -- input data
        r0[REG_LENGTH - 1..0]   : INPUT;  -- feedback input data
        load                    : INPUT;  -- load the latch (used to take in input data)
        enable                  : INPUT;
```

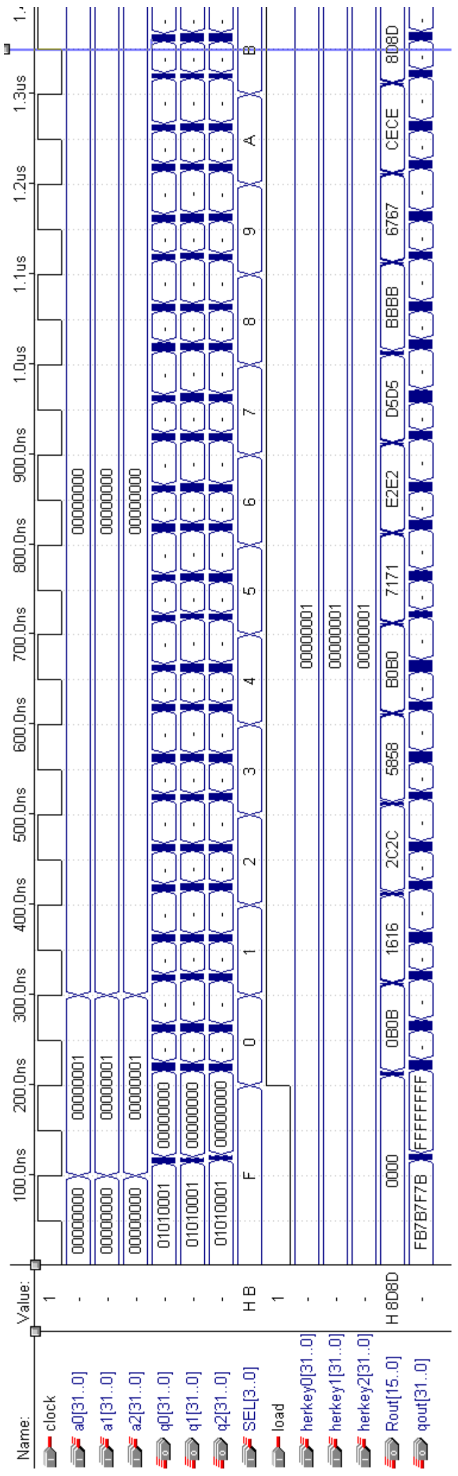

Figure B.4: Encipher, full test timing diagram.

```
        qout[REG_LENGTH - 1..0] : OUTPUT; -- output data
)

VARIABLE
        DT[REG_LENGTH - 1..0]   : dffe;
BEGIN
        -- generates the logic for us, no need to type it all in.
        FOR i IN 1 TO REG_LENGTH GENERATE
                DT[i - 1].clk = CLOCK;
                DT[i - 1].ena = ENABLE;

                -- feedback data
                DT[i - 1].d = RO[i - 1];

                -- input data
                DT[i - 1].clrn = a0[i - 1] or load;
                DT[i - 1].prn = !a0[i - 1] or load;

                -- output
                QOUT[i - 1]     = DT[i - 1].q;
        END GENERATE;
END;
```

B.4 round

```
-- round.tdf
-- Implements one round of the 3 way algorithm in the for loop, includes the
-- rho() operation. This is identical to the code in the round function, just
-- packaged here as it includes rho.
-- John Ronan, 20000708

INCLUDE "lpm_xor.inc";
INCLUDE "rho.inc";

CONSTANT REG_LENGTH  = H"20"; -- 32bit Registers
CONSTANT A1_XOR_SIZE = H"2";  -- compare two inputs
CONSTANT A0_XOR_SIZE = H"3";  -- 3 INPUTS
CONSTANT RC_WIDTH    = H"10";

SUBDESIGN round
(
        A0[REG_LENGTH - 1..0]         : INPUT;
        A1[REG_LENGTH - 1..0]         : INPUT;
        A2[REG_LENGTH - 1..0]         : INPUT;
        CIPHERKEY0[REG_LENGTH -1..0]  : INPUT;
        CIPHERKEY1[REG_LENGTH -1..0]  : INPUT;
        CIPHERKEY2[REG_LENGTH -1..0]  : INPUT;
        ROUNDCONSTANT[RC_WIDTH -1..0] : INPUT;
        Q0[REG_LENGTH - 1..0]         : OUTPUT;
        Q1[REG_LENGTH - 1..0]         : OUTPUT;
        Q2[REG_LENGTH - 1..0]         : OUTPUT;
)
```

```
VARIABLE
        A0_00 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        A1_00 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A1_XOR_SIZE);
        A2_00 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        R0    : rho;

BEGIN
        -- a0
        A0_00.DATA[0][] = CIPHERKEY0[];
        A0_00.DATA[1][] = A0[];

        FOR i IN 0 TO 15 GENERATE
                A0_00.DATA[2][I] = GND;
        END GENERATE;

        FOR i IN 0 TO 15 GENERATE
                A0_00.DATA[2][I+16] = ROUNDCONSTANT[I];
        END GENERATE;

        -- a1
        A1_00.DATA[0][] = CIPHERKEY1[];
        A1_00.DATA[1][] = A1[];

        -- a2
        A2_00.DATA[0][] = CIPHERKEY2[];
        A2_00.DATA[1][] = A2[];

        FOR i IN 0 TO 15 GENERATE
                A2_00.DATA[2][I+16] = GND;
        END GENERATE;

        FOR i IN 0 TO 15 GENERATE
                A2_00.DATA[2][I] = ROUNDCONSTANT[I];
        END GENERATE;

        -- call to rho
        R0.A0[] = A0_00.RESULT[];
        R0.A1[] = A1_00.RESULT[];
        R0.A2[] = A2_00.RESULT[];

        -- OUTPUT
        Q0[] = R0.Q0[];
        Q1[] = R0.Q1[];
        Q2[] = R0.Q2[];
END;
```

B.5 rho

```
-- rho.tdf
 -- This is just a wrapper for theta and pi_gammma_pi to maintain the same structure
-- as the software version. It also makes each component easier to test.
-- John Ronan, 20000708
```

```
INCLUDE "lpm_xor.inc";
INCLUDE "theta.inc";
INCLUDE "pi_gamma_pi.inc";

CONSTANT REG_LENGTH = H"20"; -- 32bit Registers
CONSTANT _XOR_SIZE  = H"2";  -- compare two inputs

% Test values generated from the Reference code
INPUTS  A0=00000001
        A1=00000001
        A2=00000001
OUTPUTS Q0=fb7b7f7b
        Q1=febebfbe
        Q2=ff3f3f3f%

SUBDESIGN rho
(
        A0[REG_LENGTH - 1..0] : INPUT;
        A1[REG_LENGTH - 1..0] : INPUT;
        A2[REG_LENGTH - 1..0] : INPUT;
        Q0[REG_LENGTH - 1..0] : OUTPUT;
        Q1[REG_LENGTH - 1..0] : OUTPUT;
        Q2[REG_LENGTH - 1..0] : OUTPUT;
)

VARIABLE
 T : theta;
 P : pi_gamma_pi;
BEGIN

-- THETA
T.a0[] = a0[];
T.a1[] = a1[];
T.a2[] = a2[];

-- PI_GAMMA_PI
P.a0[] = T.q0[];
p.a1[] = T.Q1[];
P.A2[] = T.Q2[];

-- output
Q0[] = P.Q0[];
Q1[] = P.Q1[];
Q2[] = P.Q2[];

END;
```

B.5.1 Timing diagram, rho

B.6 pi_gamma_pi

```
-- pi_gamma_pi.tdf
-- This is the concatenation of pi_1 , gamma and pi_2 into the one function as
```

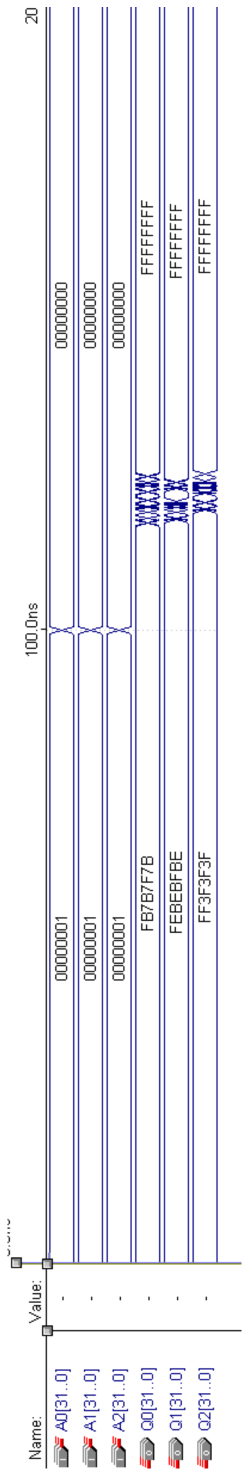

Figure B.5: Rho, full test timing diagram.

```
-- they are never used seperately it makes sense to put them together
-- John Ronan, 20000708

INCLUDE "lpm_xor.inc";

CONSTANT REG_LENGTH = H"20"; -- 32bit Registers
CONSTANT _XOR_SIZE  = H"2";  -- compare two inputs

% Test values generated from the Reference code
INPUTS  A0=677b6449
        A1=586c045e
        A2=1853287f

OUTPUTS Q0=9a48e30d
        Q1=a5ca75a0
        Q2=c1e1df6b%

SUBDESIGN pi_gamma_pi
(
        A0[REG_LENGTH - 1..0] : INPUT;
        A1[REG_LENGTH - 1..0] : INPUT;
        A2[REG_LENGTH - 1..0] : INPUT;
        Q0[REG_LENGTH - 1..0] : OUTPUT;
        Q1[REG_LENGTH - 1..0] : OUTPUT;
        Q2[REG_LENGTH - 1..0] : OUTPUT;
)
VARIABLE
 A0TMP  : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = _XOR_SIZE);
 A2TMP  : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = _XOR_SIZE);
 A0TMP1 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = _XOR_SIZE);
 A2TMP1 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = _XOR_SIZE);
 Q1EXOR : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = _XOR_SIZE);
 A2LS01[REG_LENGTH - 1..0]  : NODE;
 A2RS31[REG_LENGTH - 1..0]  : NODE;
 A0LS22[REG_LENGTH - 1..0]  : NODE;
 A0RS10[REG_LENGTH - 1..0]  : NODE;
 A0ALS01[REG_LENGTH - 1..0] : NODE;
 A0ARS31[REG_LENGTH - 1..0] : NODE;
 A2ALS22[REG_LENGTH - 1..0] : NODE;
 A2ARS10[REG_LENGTH - 1..0] : NODE;
BEGIN
-- pi_1
        -- left shift by 1
        A2LS01[0] = GND;
        FOR i IN 0 TO 30 GENERATE
                A2LS01[I+1] = A2[I];
        END GENERATE;

        -- right shift 31
        A2RS31[0] = A2[31];
        FOR I IN 1 TO 31 GENERATE
                A2RS31[I] = GND;
        END GENERATE;

        -- left shift 22
        A0LS22[21..0] = GND;
        FOR i IN 0 TO 9 GENERATE
                A0LS22[I+22] = A0[I];
        END GENERATE;


        -- right shift 10
        A0RS10[31..22] = GND;
        FOR I IN 0 TO 21 GENERATE
                A0RS10[I] = A0[I+10];
        END GENERATE;

-- gamma, the non-linear step and feed into input for pi_2
        -- a0, left side of xor
        A0TMP.DATA[0][] = (A0LS22[] OR A0RS10[]);
        A0TMP.DATA[1][] = (A1[] OR (NOT (A2LS01[] OR A2RS31[])));

        A0ALS01[0] = GND;
                FOR I IN 0 TO 30 GENERATE
                A0ALS01[I+1] = A0TMP.RESULT[I];
        END GENERATE;

        -- a0, right side of xor
        A0TMP1.DATA[0][] = A0LS22[] OR A0RS10[];
        A0TMP1.DATA[1][] = A1[] OR (NOT (A2LS01[] OR A2RS31[])) ;

        A0ARS31[0] = A0TMP1.RESULT[31];
                FOR I IN 1 TO 31 GENERATE
                A0ARS31[I] = GND;
                END GENERATE;

        -- a2, left side of xor
        A2TMP.DATA[0][] = A2LS01[] OR A2RS31[];
        A2TMP.DATA[1][] = (A0LS22[] OR A0RS10[]) OR (NOT A1[]);

        A2ALS22[21..0] = GND;
        FOR i IN 0 TO 9 GENERATE
        A2ALS22[I+22] = A2TMP.RESULT[I];
        END GENERATE;

        -- a2, right side of xor
        A2TMP1.DATA[0][] = (A2LS01[] OR A2RS31[]);
        A2TMP1.DATA[1][] = (A0LS22[] OR A0RS10[]) OR (NOT A1[]);
        A2ARS10[31..22] = GND;
                FOR I IN 0 TO 21 GENERATE
                A2ARS10[I] = A2TMP1.RESULT[I+10];
        END GENERATE;

        -- feed output of middle 32 bits directly to output as they're not affected by pi_2
        Q1EXOR.DATA[0][] = A1[];
        Q1EXOR.DATA[1][] = ((A2LS01[] OR A2RS31[]) OR (NOT (A0LS22[] OR A0RS10[])));
        Q1[] = Q1EXOR.RESULT[];

-- pi_2, and feed to output
        Q0[] = A0ALS01[] OR A0ARS31[];
        Q2[] = A2ALS22[] OR A2ARS10[];

END;
```

B.6.1 Timing diagram, pi_gamma_pi

B.7 theta

```
-- theta.tdf
-- optimised theta, reduces the number of xor operations required and thus saves
-- space on silicon
-- John Ronan, 20000708
INCLUDE "lpm_xor.inc";

CONSTANT REG_LENGTH     = H"20"; -- 32bit Registers
CONSTANT TEMP_XOR_SIZE  = H"3";  -- 3 inputs
CONSTANT TEMP2_XOR_SIZE = H"2";  -- 2 inputs
CONSTANT OUT_XOR_SIZE   = H"7";  -- 7 inputs

% Test values generated from the Reference code
INPUTS  A0=d2f05b5e
        A1=d6144138
        A2=cab920cd
 OUTPUTS Q0=ffeabf3f
        Q1=30771926
        Q2=90a50fd5 %

SUBDESIGN theta
(
 A0[REG_LENGTH - 1..0] : INPUT;
 A1[REG_LENGTH - 1..0] : INPUT;
 A2[REG_LENGTH - 1..0] : INPUT;
 Q0[REG_LENGTH - 1..0] : OUTPUT;
 Q1[REG_LENGTH - 1..0] : OUTPUT;
 Q2[REG_LENGTH - 1..0] : OUTPUT;
)

VARIABLE
 TEMPLS16[REG_LENGTH - 1..0]   : NODE;
 TEMPRS16[REG_LENGTH - 1..0]   : NODE;
 TEMP3LS08[REG_LENGTH - 1..0] : NODE;
 TEMP3RS08[REG_LENGTH - 1..0] : NODE;
 TEMP4LS08[REG_LENGTH - 1..0] : NODE;
 TEMP4RS08[REG_LENGTH - 1..0] : NODE;
 TEMP5LS08[REG_LENGTH - 1..0] : NODE;
 TEMP5RS08[REG_LENGTH - 1..0] : NODE;
 A0LS08[REG_LENGTH - 1..0]     : NODE;
 A0LS24[REG_LENGTH - 1..0]     : NODE;
 A0RS24[REG_LENGTH - 1..0]     : NODE;
 A1LS08[REG_LENGTH - 1..0]     : NODE;
 A1LS24[REG_LENGTH - 1..0]     : NODE;
 A1RS24[REG_LENGTH - 1..0]     : NODE;
 A2LS08[REG_LENGTH - 1..0]     : NODE;
```

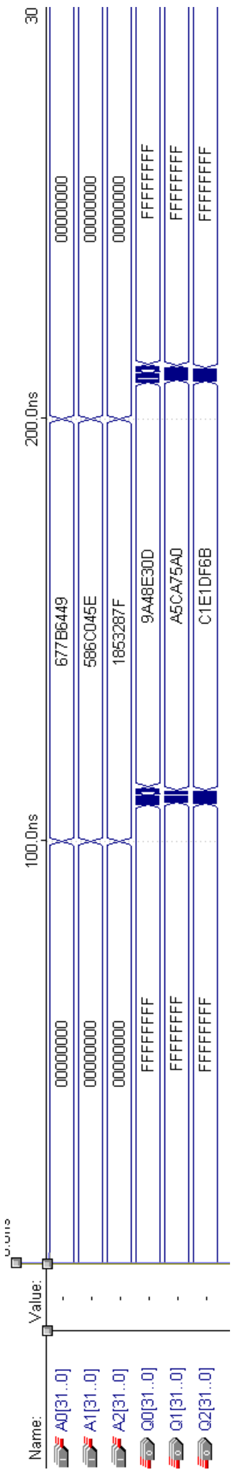

Figure B.6: Pi_gamma_pi, full test timing diagram.

```
 A2LS24[REG_LENGTH - 1..0]    : NODE;
 A2RS24[REG_LENGTH - 1..0]    : NODE;
 -- 3 input exclusive or
 TEMP : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = TEMP_XOR_SIZE);

 -- 2 input exclusive or
 TEMP2 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = TEMP2_XOR_SIZE);
 TEMP3 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = TEMP2_XOR_SIZE);
 TEMP4 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = TEMP2_XOR_SIZE);
 TEMP5 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = TEMP2_XOR_SIZE);

 -- 7 input exclusive or
 B0 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = OUT_XOR_SIZE);
 B1 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = OUT_XOR_SIZE);
 B2 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = OUT_XOR_SIZE);

BEGIN
-- no shift
        TEMP.data[0][] = A0[];
        TEMP.data[1][] = A1[];
        TEMP.data[2][] = A2[];

-- left and right shift 16
        TEMPLS16[15..0] = GND;
        TEMPRS16[REG_LENGTH - 1.. 16] = GND;

        FOR i IN 0 TO 15 GENERATE
                        TEMPLS16[i+16] = TEMP.result[i];
                        TEMPRS16[i] = TEMP.result[i+16];
        END GENERATE;

        -- store the results of the above in temporary space
        TEMP2.data[0][] = TEMPLS16[];
        TEMP2.data[1][] = TEMPRS16[];

-- left and right shift 8
        A0LS08[7..0] = GND;
        A1LS08[7..0] = GND;
        A2LS08[7..0] = GND;
        TEMP3LS08[7..0] = GND;
        TEMP3RS08[REG_LENGTH - 1..24] = GND;
        TEMP4LS08[7..0] = GND;
        TEMP4RS08[REG_LENGTH - 1..24] = GND;
        TEMP5LS08[7..0] = GND;
        TEMP5RS08[REG_LENGTH - 1..24] = GND;

        FOR i in 0 to 23 GENERATE
                A0LS08[i+8] = A0[i];
                A1LS08[i+8] = A1[i];
                A2LS08[i+8] = A2[i];
                TEMP3LS08[i+8] = A0[i];
                TEMP3RS08[i] = A0[i+8];
                TEMP4LS08[i+8] = A1[i];
                TEMP4RS08[i] = A1[i+8];
                TEMP5LS08[i+8] = A2[i];
```

```
                        TEMP5RS08[i] = A2[i+8];
                END GENERATE;

                -- store
                TEMP3.data[0][] = TEMP3LS08[];
                TEMP3.data[1][] = TEMP3RS08[];
                TEMP4.data[0][] = TEMP4LS08[];
                TEMP4.data[1][] = TEMP4RS08[];
                TEMP5.data[0][] = TEMP5LS08[];
                TEMP5.data[1][] = TEMP5RS08[];

        -- left and right shift 24
                A0LS24[23..0] = GND;
                A0RS24[REG_LENGTH - 1..8] = GND;
                A1LS24[23..0] = GND;
                A1RS24[REG_LENGTH - 1..8] = GND;
                A2LS24[23..0] = GND;
                A2RS24[REG_LENGTH - 1..8] = GND;

                FOR i IN 0 TO 7 GENERATE
                                A0LS24[i+24] = A0[i];
                                A0RS24[i] = A0[i+24];
                                A1LS24[i+24] = A1[i];
                                A1RS24[i] = A1[i+24];
                                A2LS24[i+24] = A2[i];
                                A2RS24[i] = A2[i+24];
                END GENERATE;

        -- feed into three 7 input xors.
                B0.data[0][] = A0[];
                B0.data[1][] = TEMP2.result[];
                B0.data[2][] = TEMP5.result[];
                B0.data[3][] = A1RS24[];
                B0.data[4][] = A0LS24[];
                B0.data[5][] = A2RS24[];
                B0.data[6][] = A0LS08[];

                B1.data[0][] = A1[];
                B1.data[1][] = TEMP2.result[];
                B1.data[2][] = TEMP3.result[];
                B1.data[3][] = A2RS24[];
                B1.data[4][] = A1LS24[];
                B1.data[5][] = A0RS24[];
                B1.data[6][] = A1LS08[];

                B2.data[0][] = A2[];
                B2.data[1][] = TEMP2.result[];
                B2.data[2][] = TEMP4.result[];
                B2.data[3][] = A0RS24[];
                B2.data[4][] = A2LS24[];
                B2.data[5][] = A1RS24[];
                B2.data[6][] = A2LS08[];

        -- output
                Q0[] = B0.result[];
        Q1[] = B1.result[];
        Q2[] = B2.result[];
END;
```

B.7.1 Timing diagram, theta

B.8 decipher

```
-- encipher.tdf
-- Takes the key, and with the correct input of control signals, performs the
-- encryption of a block of data. mlatch is used to implement the for loop similiar
-- to the software version, note the roundconstants are, similiar to the encrypt
-- module, hard coded as inputs to the a multiplexer, but in this case they are a
--  different set of constants.
-- John Ronan, 20000708

INCLUDE "rho.inc";
INCLUDE "theta.inc";
INCLUDE "mu.inc";
INCLUDE "round.inc";
INCLUDE "lpm_xor.inc";
INCLUDE "mlatch.inc";
INCLUDE "lpm_constant.inc";
INCLUDE "lpm_mux.inc";

CONSTANT REG_LENGTH  = H"20"; -- 32bit Registers
CONSTANT A1_XOR_SIZE = H"2";  -- compare two inputs
CONSTANT A0_XOR_SIZE = H"3";  -- 3 INPUTS
CONSTANT RC_WIDTH    = H"10";

% Test values generated from the Reference code
INPUTS  A0=cfbd782f
        A1=31761927
        A2=234809bf
OUTPUTS Q0=00000001
        Q1=00000001
        Q2=00000001 %

SUBDESIGN decipher
(
        A0[REG_LENGTH - 1..0]        : INPUT;  -- intput data
        A1[REG_LENGTH - 1..0]        : INPUT;
        A2[REG_LENGTH - 1..0]        : INPUT;
        CIPHERKEY0[REG_LENGTH -1..0] : INPUT;  -- decryption key
        CIPHERKEY1[REG_LENGTH -1..0] : INPUT;
        CIPHERKEY2[REG_LENGTH -1..0] : INPUT;
        LOAD                         : INPUT;  -- latch control
        CLOCK                        : INPUT;
        SEL0                         : INPUT;  -- multliplexer control
        SEL1                         : INPUT;
        SEL2                         : INPUT;
        SEL3                         : INPUT;
        Qout[REG_LENGTH - 1..0]      : OUTPUT; -- Diagnostic output
```

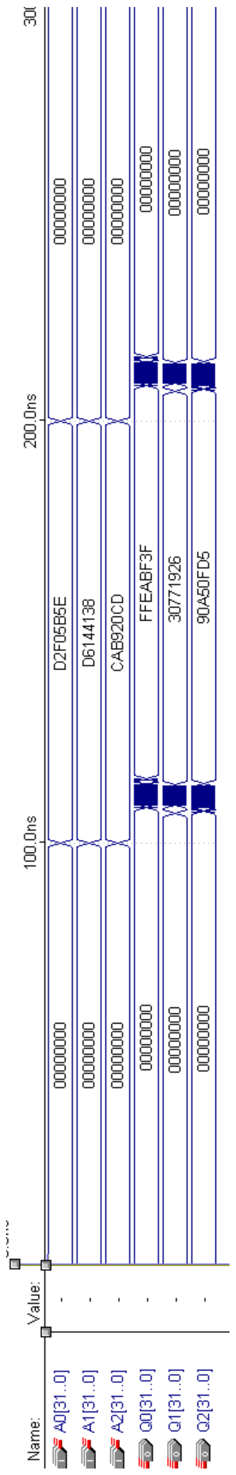

Figure B.7: Theta, full test timing diagram.

```
        Rout[RC_WIDTH - 1..0]        : OUTPUT; -- Diagnostic output
        Q0[REG_LENGTH - 1..0]        : OUTPUT; -- output data
        Q1[REG_LENGTH - 1..0]        : OUTPUT;
        Q2[REG_LENGTH - 1..0]        : OUTPUT;
)

VARIABLE
        r     : round;
        A0_00 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        A0_01 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A1_XOR_SIZE);
        A0_02 : lpm_xor WITH(LPM_WIDTH = REG_LENGTH, LPM_SIZE = A0_XOR_SIZE);
        T0    : theta;
        MU0   : mu;
        MU1   : mu;
        l0    : mlatch WITH(REG_LENGTH = REG_LENGTH);
        l1    : mlatch WITH(REG_LENGTH = REG_LENGTH);
        l2    : mlatch WITH(REG_LENGTH = REG_LENGTH);
        m     : lpm_mux WITH(LPM_WIDTH=RC_WIDTH, LPM_SIZE=H"10", LPM_WIDTHS=H"4");

BEGIN
        m.sel[0] = SEL0;
        m.sel[1] = SEL1;
        m.sel[2] = SEL2;
        m.sel[3] = SEL3;

        -- precalculated round constants
        m.data[0][] = H"b1b1";
        m.data[1][] = H"7373";
        m.data[2][] = H"e6e6";
        m.data[3][] = H"dddd";
        m.data[4][] = H"abab";
        m.data[5][] = H"4747";
        m.data[6][] = H"8e8e";
        m.data[7][] = H"0d0d";
        m.data[8][] = H"1a1a";
        m.data[9][] = H"3434";
        m.data[10][] = H"6868";
        m.data[11][] = H"d0d0"; -- last round constant
        m.data[12][] = H"0000";
        m.data[13][] = H"0000";
        m.data[14][] = H"0000";
        m.data[15][] = H"0000";

        -- mu, bit reversal function
        MU0.A0[] = A0[];
        MU0.A1[] = A1[];
        MU0.A2[] = A2[];

        -- until we meet mu again, this is all the same as the encryption function
        -- feedback of round data (for loop)
        l0.clock = CLOCK;
        l1.clock = CLOCK;
        l2.clock = CLOCK;
        l0.enable = VCC;
        l1.enable = VCC;
```

```
l2.enable = VCC;
l0.load = LOAD;
l1.load = LOAD;
l2.load = LOAD;

-- inputs to the latch.. data in and feedback
l0.a0[] = MU0.Q0[];
l0.r0[] = r.q0[];
l1.a0[] = MU0.Q1[];
l1.r0[] = r.q1[];
l2.a0[] = MU0.Q2[];
l2.r0[] = r.q2[];

-- diagnostic output
Qout[] = r.q0[];

-- decryption key fed into the round function
r.cipherkey0[] = CIPHERKEY0[];
r.cipherkey1[] = CIPHERKEY1[];
r.cipherkey2[] = CIPHERKEY2[];

-- round constant selected based on input to multiplexer
r.roundconstant[] = m.result[];

-- data from latch to round function
r.a0[] = l0.qout[];
r.a1[] = l1.qout[];
r.a2[] = l2.qout[];

-- round completed
-- a0
-- data into exclusive or
AO_00.DATA[0][] = CIPHERKEY0[];
AO_00.DATA[1][] = l0.qout[];

FOR i IN 0 TO 15 GENERATE
        AO_00.DATA[2][I] = GND;
END GENERATE;

FOR i IN 0 TO 15 GENERATE
        AO_00.DATA[2][I+16] = m.result[i];
END GENERATE;

-- a1
AO_01.DATA[0][] = CIPHERKEY1[];
AO_01.DATA[1][] = l1.qout[];

-- a2
AO_02.DATA[0][] = CIPHERKEY2[];
AO_02.DATA[1][] = l2.qout[];

FOR i IN 0 TO 15 GENERATE
        AO_02.DATA[2][I+16] = GND;
END GENERATE;
```

```
        FOR i IN 0 TO 15 GENERATE
                A0_02.DATA[2][I] = m.result[i];
                A0_02.DATA[2][I] = GND;
        END GENERATE;

        -- theta
        T0.A0[] = A0_00.RESULT[];
        T0.A1[] = A0_01.RESULT[];
        T0.A2[] = A0_02.RESULT[];

        -- and bit reversal
        MU1.A0[] = T0.Q0[];
        MU1.A1[] = T0.Q1[];
        MU1.A2[] = T0.Q2[];

        -- diagnostic output
        Rout[] = m.result[];
        -- data is now decrypted
        Q0[] = MU1.Q0[];
        Q1[] = MU1.Q1[];
        Q2[] = MU1.Q2[];
END;
```

B.8.1 Timing diagram, decipher

B.9 mu

```
-- Mu function
 -- Mu reverses the bits in a longword, used in the decryption
-- John Ronan, 20000709

CONSTANT REG_LENGTH = H"20"; -- 32bit Registers

SUBDESIGN mu
(
        A0[REG_LENGTH - 1..0] : INPUT;
        A1[REG_LENGTH - 1..0] : INPUT;
        A2[REG_LENGTH - 1..0] : INPUT;
        Q0[REG_LENGTH - 1..0] : OUTPUT;
        Q1[REG_LENGTH - 1..0] : OUTPUT;
        Q2[REG_LENGTH - 1..0] : OUTPUT;
)
BEGIN

FOR i IN 0 TO 31 GENERATE
        Q0[i] = A2[31-i];
        Q1[i] = A1[31-i];
        Q2[i] = A0[31-i];
END GENERATE;

END;
```

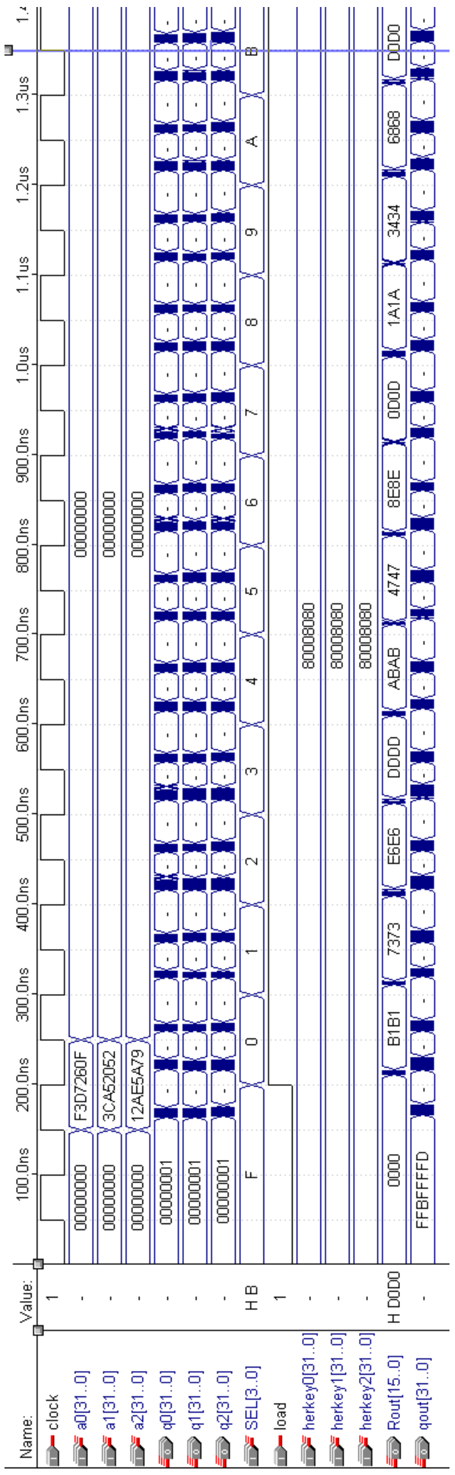

Figure B.8: Decipher, full test timing diagram.

B.10 inputlogic

```
-- inputlogic.tdf
-- 32 bit 3 output demus.
-- John Ronan, 20000722

INCLUDE "lpm_dff.inc";
INCLUDE "lpm_ff.inc";

CONSTANT REG_LENGTH = H"20"; --32bit Registers

SUBDESIGN inputlogic
(
        CLOCK                                   : INPUT;
        DataIN[REG_LENGTH - 1..0]               : INPUT;
        ENABLE                                  : INPUT;
        RESET                                   : INPUT;
        DataOUT0[REG_LENGTH - 1..0]             : OUTPUT;
        DataOUT1[REG_LENGTH - 1..0]             : OUTPUT;
        DataOUT2[REG_LENGTH - 1..0]             : OUTPUT;
)

VARIABLE
        inputsel                :lpm_dff WITH(LPM_WIDTH=3, LPM_AVALUE=H"00000");
        inputlatch0             :lpm_dff WITH(LPM_WIDTH=REG_LENGTH);
        inputlatch1             :lpm_dff WITH(LPM_WIDTH=REG_LENGTH);
        inputlatch2             :lpm_dff WITH(LPM_WIDTH=REG_LENGTH);

BEGIN
-- Demux Controller
        inputsel.clock   = not CLOCK;

-- 3 D types.
        inputsel.enable = ENABLE AND (NOT inputsel.q[2]);
        inputsel.data[0] = (not inputsel.q[2]);
        inputsel.data[1] = inputsel.q[0];
        inputsel.data[2] = inputsel.q[1];
        inputsel.aset = RESET;

-- input latch - 3 32 bit latches
        inputlatch0.data[] = DataIN[];
        inputlatch0.clock = inputsel.q[0];
        inputlatch1.data[] = DataIN[];
        inputlatch1.clock = inputsel.q[1];
        inputlatch2.data[] = DataIN[];
        inputlatch2.clock = inputsel.q[2];

        DataOUT0[] = inputlatch0.q[];
        DataOUT1[] = inputlatch1.q[];
        DataOUT2[] = inputlatch2.q[];
END;
```

B.11 key-authenthication-key

```
-- kak.tdf
-- One input, two outputs, The choice of output is either the internal
-- Key Authenthication Key or the Session key coming in from a Latch.
-- John Ronan, 20000722

INCLUDE "lpm_dff.inc";
INCLUDE "lpm_mux.inc";

CONSTANT REG_LENGTH = H"20"; %32bit Registers%

SUBDESIGN kak
(
        KAKSEL                                          : INPUT;
        KEY0IN[REG_LENGTH - 1..0]                       : INPUT;
        KEY1IN[REG_LENGTH - 1..0]                       : INPUT;
        KEY2IN[REG_LENGTH - 1..0]                       : INPUT;
        CIPHERKEY0[REG_LENGTH - 1..0]                   : OUTPUT;
        CIPHERKEY1[REG_LENGTH - 1..0]                   : OUTPUT;
        CIPHERKEY2[REG_LENGTH - 1..0]                   : OUTPUT;
)

VARIABLE
        kak0 : lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");
        kak1 : lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");
        kak2 : lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"1", LPM_SIZE=H"2");

BEGIN
        kak0.sel[0] = KAKSEL;
        kak1.sel[0] = KAKSEL;
        kak2.sel[0] = KAKSEL;

-- Hard coded KAK key
        kak0.data[0][] = H"00000001";
        kak1.data[0][] = H"00000001";
        kak2.data[0][] = H"00000001";

-- Session Key coming in from Latch
        kak0.data[1][] = KEY0IN[];
        kak1.data[1][] = KEY1IN[];
        kak2.data[1][] = KEY2IN[];

        CIPHERKEY0[] = kak0.result[];
        CIPHERKEY1[] = kak1.result[];
        CIPHERKEY2[] = kak2.result[];
END;
```

B.12 feedbacklogic

```
-- fblogic.tdf
-- Used in several places, Basically a 96 bit latch.
-- John Ronan, 20000722
```

```
INCLUDE "lpm_dff.inc";
CONSTANT REG_LENGTH = H"20"; -- 32bit Registers

SUBDESIGN fblogic
(
        CLOCK                        : INPUT;
        ENABLE                       : INPUT;
        LATCHENABLE                  : INPUT;
        DataIN0[REG_LENGTH - 1..0]   : INPUT;
        DataIN1[REG_LENGTH - 1..0]   : INPUT;
        DataIN2[REG_LENGTH - 1..0]   : INPUT;
        ACLR                         : INPUT;
        LatchOUT0[REG_LENGTH - 1..0] : OUTPUT;
        LatchOUT1[REG_LENGTH - 1..0] : OUTPUT;
        LatchOUT2[REG_LENGTH - 1..0] : OUTPUT;
)

VARIABLE
        cipherkeylatch0 : lpm_dff WITH(LPM_WIDTH=REG_LENGTH, LPM_AVALUE="0000000");
        cipherkeylatch1 : lpm_dff WITH(LPM_WIDTH=REG_LENGTH, LPM_AVALUE="0000000");
        cipherkeylatch2 : lpm_dff WITH(LPM_WIDTH=REG_LENGTH, LPM_AVALUE="0000000");

BEGIN
        cipherkeylatch0.aset    = ACLR;
        cipherkeylatch1.aset    = ACLR;
        cipherkeylatch2.aset    = ACLR;

        cipherkeylatch0.clock = CLOCK;
        cipherkeylatch1.clock = CLOCK;
        cipherkeylatch2.clock = CLOCK;

        cipherkeylatch0.enable = LATCHENABLE;
        cipherkeylatch1.enable = LATCHENABLE;
        cipherkeylatch2.enable = LATCHENABLE;

        cipherkeylatch0.data[] = DataIN0[];
        cipherkeylatch1.data[] = DataIN1[];
        cipherkeylatch2.data[] = DataIN2[];

        LatchOUT0[] = cipherkeylatch0.q[];
        LatchOUT1[] = cipherkeylatch1.q[];
        LatchOUT2[] = cipherkeylatch2.q[];
END;
```

B.13 outputlogic

```
-- outputlogic.tdf
-- Multiplexer
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_counter.inc";
INCLUDE "tim_cnt.inc";
```

```
CONSTANT REG_LENGTH = H"20"; %32bit Registers%

SUBDESIGN outputlogic
(
        CLOCK                       : INPUT;
        ENABLE                      : INPUT;
        SSET                        : INPUT;
        DataIN0[REG_LENGTH - 1..0]  : INPUT;
        DataIN1[REG_LENGTH - 1..0]  : INPUT;
        DataIN2[REG_LENGTH - 1..0]  : INPUT;
        DataOUT[REG_LENGTH - 1..0]  : OUTPUT;
)

VARIABLE
        out :lpm_mux WITH(LPM_WIDTH=REG_LENGTH, LPM_WIDTHS=H"2", LPM_SIZE=H"4");
        -- I had to define my own style counter, as there was a conflict in the
        -- compilation stage with some element in the PCI core.
        -- Tim Symons of altera helped here hence it is defined
        -- as tims counter.
        count                   :tim_cnt;
BEGIN

count.sset = SSET;
count.clock = CLOCK;
count.cnt_en = ENABLE;

out.sel[0] = count.q[0];
out.sel[1] = count.q[1];

out.data[0][] = DataIN0[];
out.data[1][] = DataIN1[];
out.data[2][] = DataIN2[];
out.data[3][] = VCC;
DataOUT[] = out.result[];
END;
```

B.13.1 Timing diagram, outputlogic

B.14 cbcreset

```
-- cbcreset.tdf
-- If and of the lines Key_Data, E_D or CBCRESET are toggled, then the
-- Feedback register for CBC mode is reset to 0 by a pulse on the
-- RESET output. Note the CBCRESET Input line is automatically pulsed by
-- the IOR interface so we just pass it through.
-- John Ronan, 20000722

SUBDESIGN cbcreset
(
CLOCK     : INPUT;
E_D       : INPUT;
Key_Data  : INPUT;
CBCRESET  : INPUT;
```

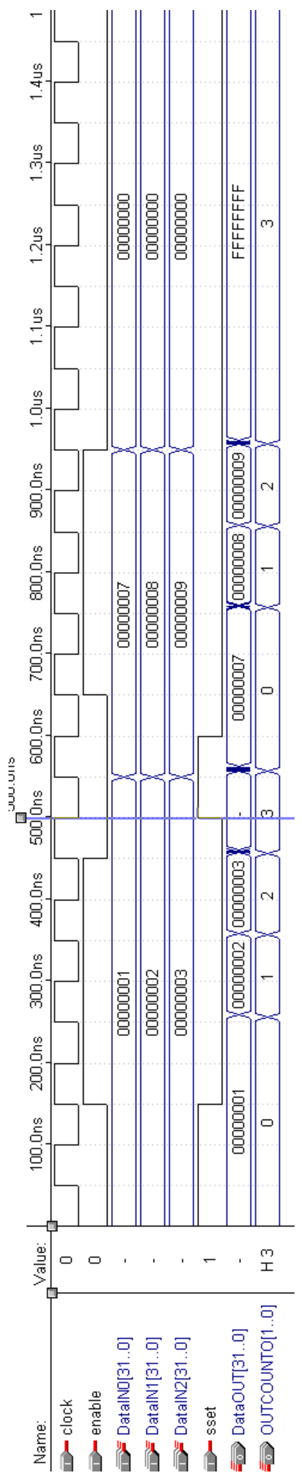

Figure B.9: Outputlogic, full test timing diagram, close up.

```
RESET     : OUTPUT;
)
VARIABLE
    e_ddelay0           :dffe;
    e_ddelay1           :dffe;
    keydelay0           :dffe;
    keydelay1           :dffe;
BEGIN
e_ddelay0.clk= CLOCK;
e_ddelay1.clk = CLOCK;
e_ddelay0.d = E_D;
e_ddelay1.d = e_ddelay0.q;

keydelay0.clk = CLOCK;
keydelay1.clk = CLOCK;
keydelay0.d = Key_Data;
keydelay1.d = keydelay0.q;

RESET = (E_D xor e_ddelay1.q) or (Key_Data xor keydelay1.q) or (CBCRESET);
END;
```

B.15 core

```
-- core.tdf
-- Encapsulates all functions except control, to allow decryption and storage
-- of session keys. ECB/CBC modes of operation, and translation from
-- 32 to 96 bits on input and output.
-- John Ronan, 20000721

INCLUDE "inputlogic.inc";
INCLUDE "cipher.inc";
INCLUDE "kak.inc";
INCLUDE "fblogic.inc";
INCLUDE "outputlogic.inc";
INCLUDE "lpm_mux.inc";
INCLUDE "lpm_xor.inc";
INCLUDE "cbcreset.inc";

CONSTANT REG_LENGTH = H"20"; %32bit Registers%
CONSTANT COUNT_LENGTH = H"4";
CONSTANT COUNTER_TERMINATE = H"C";

SUBDESIGN core
(
        DataIN[REG_LENGTH - 1..0]  : INPUT;
        RESET                      : INPUT; -- Block Reset
        CLOCK                      : INPUT;
        Key_Data                   : INPUT; -- Working with a Key or Data
        E_D                        : INPUT; -- Encipher or Decipher Mode
        ECB_CBC                    : INPUT; -- ECB or CBC mode
        ILENABLE                   : INPUT; -- Enable Input for Input Latch
        ILASET                     : INPUT; -- Reset Input for Input Latch
        OLSSET                     : INPUT; -- Reset Input for Output Latch
        OLENABLE                  : INPUT; -- Enable Input for Output Latch
        CENABLE                   : INPUT; -- Enable Input for Cipher Block
        CBCFBENABLE               : INPUT; -- Enable Input for CBC Latch
        CBCRESETIN                : INPUT; -- External (from IOR) Reset Input for CBC Latch
        DataOUT[REG_LENGTH - 1..0] : OUTPUT;

)

VARIABLE
        c          :cipher;      -- cipher block
        k          :kak;         -- key authenthication key block
        il         :inputlogic;
        kfb        :fblogic;     -- session key feedback logic
        fb         :fblogic;     -- Data Latch after cipher block
        cbcfb      :fblogic;     -- CBC Mode feedback logic
        ol         :outputlogic;
        cbc_reset  :cbcreset;    -- cbcreset block
        ctermdelay :dffe;        -- single D latch to delay signal one clock pulse

BEGIN

-- CBC Mode Reset
cbc_reset.CLOCK = CLOCK;
cbc_reset.Key_Data = Key_Data;
cbc_reset.E_D = E_D;
cbc_reset.CBCRESET = CBCRESETIN;

-- If the key gets changed or if we switch from encryption to decryption, we
-- definitely want to reset all the feedback registers to 0;
--CBCRESETO = (E_D xor e_ddelay1.q) or (Key_Data xor keydelay1.q) or CBCRESET;
CBCRESETO = cbc_reset.RESET; -- Reset Pulse

ctermdelay.clk = CLOCK;
ctermdelay.d = c.TERM;

-- Key authenthication Key mux. KAK hardcoded into data[0] below. CipherKey loaded from the feedback latch
k.kaksel = Key_Data;
k.KEY0IN[] = kfb.LatchOUT0[];
k.KEY1IN[] = kfb.LatchOUT1[];
k.KEY2IN[] = kfb.LatchOUT2[];

-- Input Logic, converts 32 bits to 96
il.clock = CLOCK;
il.DataIN[] = DataIN[];
il.ENABLE = ILENABLE; -- enable the input latch
il.reset = ILASET;    -- reset the input latch

-- cipher
c.clock = CLOCK;
c.counter_en = CENABLE; -- enable the counter (loop)
c.sset = (not CENABLE) or cbc_reset.reset or (not RESET); -- Reset block
c.load = CENABLE; -- load the data into the cipher block
c.E_D = E_D;      -- enciper or decipher

-- input from kak block, gives kak or data key
```

```
c.cipherkey0[] = k.CIPHERKEY0[];
c.cipherkey1[] = k.CIPHERKEY1[];
c.cipherkey2[] = k.CIPHERKEY2[];

-- Session Key fedback latch
kfb.CLOCK       = CLOCK;
kfb.ENABLE = not Key_data; -- Only enable when setting new session key
kfb.ACLR        = not RESET;   -- Reset Latch
kfb.LATCHENABLE = ctermdelay.q and (not Key_Data); -- Only enable when setting
-- new session key and when the data out of the cipher block is valid

fb.CLOCK = CLOCK;
fb.ENABLE = Key_Data; -- Enable only when processing data, thus we don't leak
                      -- the new key back out to the outside world.
fb.ACLR = (not RESET) or cbc_reset.RESET; -- Reset on system reset or from control line
fb.LATCHENABLE = ctermdelay.q; -- data out of the cipher block becomes valid

cbcfb.CLOCK = CLOCK;
cbcfb.ENABLE = Key_Data and ECB_CBC; -- Enable only when processing data and
                                     -- when in CBC mode.
cbcfb.ACLR = (not RESET) or cbc_reset.RESET;  -- Reset on system reset or
                                              -- from control line
cbcfb.LATCHENABLE = CBCFBENABLE; -- enable while in CBC mode.

-- Cipher Key is fed into the KAK block from cipher block output
kfb.DataIN0[] = c.DataOUT0[];
kfb.DataIN1[] = c.DataOUT1[];
kfb.DataIN2[] = c.DataOUT2[];

-- only need to feed data to fb latch while operating on data
fb.DataIN0[] = c.DataOUT0[] and Key_Data;
fb.DataIN1[] = c.DataOUT1[] and Key_Data;
fb.DataIN2[] = c.DataOUT2[] and Key_Data;

-- ECB Mode = fb.LatchOUT and not ECB_CBC
-- CBC Mode Enciphering = fb.LatchOUT and (not E_D) and ECB_CBC
-- CBC MODE Deciphering = il.DataOUT  and E_D      and ECB_CBC
cbcfb.datain0[] = (fb.LatchOUT0[] and (not ECB_CBC)) xor ((((fb.LatchOUT0[] and (not E_D))
xor (il.DataOUT0[] and E_D)) and ECB_CBC) and Key_Data);
cbcfb.datain1[] = (fb.LatchOUT1[] and (not ECB_CBC)) xor ((((fb.LatchOUT1[] and (not E_D))
xor (il.DataOUT1[] and E_D)) and ECB_CBC) and Key_Data);
cbcfb.datain2[] = (fb.LatchOUT2[] and (not ECB_CBC)) xor ((((fb.LatchOUT2[] and (not E_D))
xor (il.DataOUT2[] and E_D)) and ECB_CBC) and Key_Data);

--  in ECB Mode il.DataOUT is xored with 0
--  in CBC Encryption its xored with the output from the feedback loop
c.DataIN0[] = il.DataOUT0[] xor ((cbcfb.LatchOUT0[] and (not E_D) and ECB_CBC) and Key_Data);
c.Datain1[] = il.DataOUT1[] xor ((cbcfb.LatchOUT1[] and (not E_D) and ECB_CBC) and Key_Data);
c.Datain2[] = il.DataOUT2[] xor ((cbcfb.LatchOUT2[] and (not E_D) and ECB_CBC) and Key_Data);

-- in ECB Mode ol.DataIN = fb.dataout
-- in CBC Encryption ol.DataIN = fb.dataout xor cbcfb.LatchOUT
ol.DataIN0[] = (fb.LatchOUT0[] xor ((cbcfb.LatchOUT0[] and E_D and ECB_CBC)) and Key_Data);
ol.DataIN1[] = (fb.LatchOUT1[] xor ((cbcfb.LatchOUT1[] and E_D and ECB_CBC)) and Key_Data);
ol.Datain2[] = (fb.LatchOUT2[] xor ((cbcfb.LatchOUT2[] and E_D and ECB_CBC)) and Key_Data);
```

```
-- output latch
ol.ENABLE = OLENABLE and Key_Data; -- Only enable when dealing with data
ol.SSET = OLSSET and Key_Data;     -- Ditto
ol.clock = CLOCK;

-- 32 bits of Data out.
DataOUT[] = ol.DataOUT[] and Key_Data; -- Nothing leaves unless we're processing data

END;
```

B.16 controlncore

```
-- controlncore.tdf
-- Control circuitry for the device module, state machine stalls if either
-- Input Fifo is Empty (But will output current values in core) or output
-- Fifo is Full. Hence these two lines control the whole process.
-- Generates an interrupt 18 clock cycles after the Input Fifo is empty
-- This is ample time for all the data to have been processed
-- John Ronan, 20000407

INCLUDE "lpm_counter.inc";
INCLUDE "lpm_compare.inc";
INCLUDE "lpm_dff.inc";

CONSTANT WIDTH = 5;                  -- No of bits in in count
CONSTANT INT_COUNTER_TERMINATE = 18; -- No of clock cycles after
-- Input Fifo Empties that all the data has been processed and output

CONSTANT CBC_COUNTER_TERMINATE = 24;

SUBDESIGN controlncore
(
        clk                     : INPUT;
        reset                   : INPUT;
        IFifoEMPTY              : INPUT; -- Control Line from Input Fifo
        OFifoFULL               : INPUT; -- Control Line from Output Fifo
        INT_RESET               : INPUT; -- Reset Interrupt
        INT_REQUEST             : OUTPUT;
        CENABLE                 : OUTPUT; -- Cipher block Enable line
        CBCFBENABLE             : OUTPUT; -- CBC Latch Enable line
        ILASET                  : OUTPUT; -- Input Latch Reset Line
        ILENABLE                : OUTPUT; -- Input Latch Enable line
        OLSSET                  : OUTPUT; -- Output Latch Reset Line
        OLENABLE                : OUTPUT; -- Output Latch Enable Line
        IFifoRREQ               : OUTPUT; -- Reqest data from the Input Fifo
        OFifoWREQ               : OUTPUT; -- Request data to be sent to the Output Fifo
                                          -- OfifoFull precludes this
)
VARIABLE
   ss: MACHINE OF BITS (CENABLE, CBCFBENABLE,ILASET, ILENABLE,OLSSET, OLENABLE, FiFORREQ)
       WITH STATES (s0   =    B"0000000",
                    s1   =    B"0010000",
                    s2   =    B"0001001",
                    s3   =    B"0001001",
                    s4   =    B"0001001",
                    s5   =    B"0000000",
                    s6   =    B"1000000",
                    s7   =    B"1000000",
                    s8   =    B"1000000",
                    s9   =    B"1000000",
                    s10  =    B"1000000",
                    s11  =    B"1000000",
                    s12  =    B"1000000",
                    s13  =    B"1000000",
                    s14  =    B"1000000",
                    s15  =    B"1000000",
                    s16  =    B"1000000",
                    s17  =    B"1000000",
                    s18  =    B"0000100",
                    s19  =    B"0000010",
                    s20  =    B"0000010",
                    s21  =    B"0000010",
                    s22  =    B"0100000");

        interruptcount : lpm_counter WITH(LPM_WIDTH=WIDTH, LPM_MODULUS=INT_COUNTER_TERMINATE,
                                          LPM_SVALUE=H"00000");
        intcomparator  : lpm_compare WITH(LPM_WIDTH=WIDTH);
        d1             : lpm_dff WITH(LPM_WIDTH=1);
        ififorreqdelay :dffe;
BEGIN

   ss.clk   = not clk;
   ss.reset = not reset;
TABLE
   --  current  current     next
   --  state    input       state

      ss,      (IFifoEmpty) or (OFifoFULL)      =>   ss;

      s0,      1      =>   s0;
      s0,      0      =>   s1;
      s1,      1      =>   s2;
      s1,      0      =>   s2;
      s2,      1      =>   s3;
      s2,      0      =>   s3;
      s3,      1      =>   s4;
      s3,      0      =>   s4;
      s4,      1      =>   s5;
      s4,      0      =>   s5;
      s5,      1      =>   s6;
      s5,      0      =>   s6;
      s6,      1      =>   s7;
      s6,      0      =>   s7;
      s7,      1      =>   s8;
      s7,      0      =>   s8;
      s8,      1      =>   s9;
      s8,      0      =>   s9;
```

```
        s9,      1      =>   s10;
        s9,      0     =>   s10;
        s10,     1      =>   s11;
        s10,     0      =>   s11;
        s11,      1      =>   s12;
        s11,      0      =>   s12;
        s12,      1      =>   s13;
        s12,      0      =>   s13;
        s13,      1      =>   s14;
        s13,      0      =>   s14;
        s14,      1      =>   s15;
        s14,      0      =>   s15;
        s15,      1      =>   s16;
        s15,      0      =>   s16;
        s16,      1      =>   s17;
        s16,      0      =>   s17;
        s17,      1      =>   s18;
        s17,      0      =>   s18;
        s18,      1      =>   s19;
        s18,      0      =>   s19;
        s19,      1      =>   s20;
        s19,      0      =>   s20;
        s20,      1      =>   s21;
        s20,      0      =>   s21;
        s21,      1      =>   s22;
        s21,      0      =>   s22;
        s22,      1      =>   s0;
        s22,      0      =>   s0;
    END TABLE;

-- Traps the Rising edge of IfifoEMPTY and 18 pulses later
-- it raises an Interrupt to the PCI core.  The reset then
-- sets the counter back to 0 and doesnt start counting
-- again until the next rising edge
        d1.data[0] = VCC;
        d1.aclr = INT_RESET;
        d1.clock = IfifoEMPTY;
        d1.enable = VCC;

-- Retiming Delay
        ififorreqdelay.clk = clk;
        ififorreqdelay.d = FiFORREQ;

-- Delay until all data is out of the core
        interruptcount.clock = clk;
        interruptcount.cnt_en = d1.q[0] and (not intcomparator.aeb);
        interruptcount.sset = INT_RESET;
        interruptcount.aclr = not reset;

-- Terminate the count
        intcomparator.dataa[] = interruptcount.q[];
        intcomparator.datab[] = INT_COUNTER_TERMINATE - 1;

-- Interrupt when the Input Fifo is empty AND the Core is empty.
        INT_REQUEST = intcomparator.aeb;
        IFifoRREQ   = ififorreqdelay.q;
        OfifoWREQ   = OLENABLE; -- just happens to be the same signal :)
END;
```

B.16.1 Timing diagram, Controlncore

B.17 crypto

```
-- crypto.tdf
-- Encapsulates the cryptographic "core" and the control circuitry.
-- Also added last bit of Interrupt circuitry on rising Edge of
-- Output Fifo Full.

INCLUDE "core.inc";
INCLUDE "controlncore.inc";

SUBDESIGN crypto
(
        DataIN[32 - 1..0]  : INPUT;
        -- Control Lines in from IOR
        RESET              : INPUT;
        CBCRESET           : INPUT;
        OFifoFull          : INPUT;
        IFifoEmpty         : INPUT;
        CLOCK              : INPUT;
        Key_Data           : INPUT;
        E_D                : INPUT;
        ECB_CBC            : INPUT;
        INT_RESET          : INPUT;
        INT_ENABLE         : INPUT;
        -- Control line out to IOR
        INT_REQUEST        : OUTPUT;
        -- Data out
        DataOUT[32 - 1..0] : OUTPUT;
        -- Request Lines to the Input and Output Fifos
        OFifoWREQ          : OUTPUT;
        IFifoRREQ          : OUTPUT;
)

VARIABLE
    c       : core;
    ctrl    : controlncore;
    delay0  : dffe;
BEGIN

ctrl.clk = CLOCK;
ctrl.RESET = RESET and (not CBCRESET);
-- Input lines to the Control Circuitry
ctrl.IFifoEMPTY = IFifoEMPTY;
ctrl.OFifoFULL  = OFifoFULL;
ctrl.INT_RESET  = INT_RESET;

c.Key_Data = Key_Data;
```

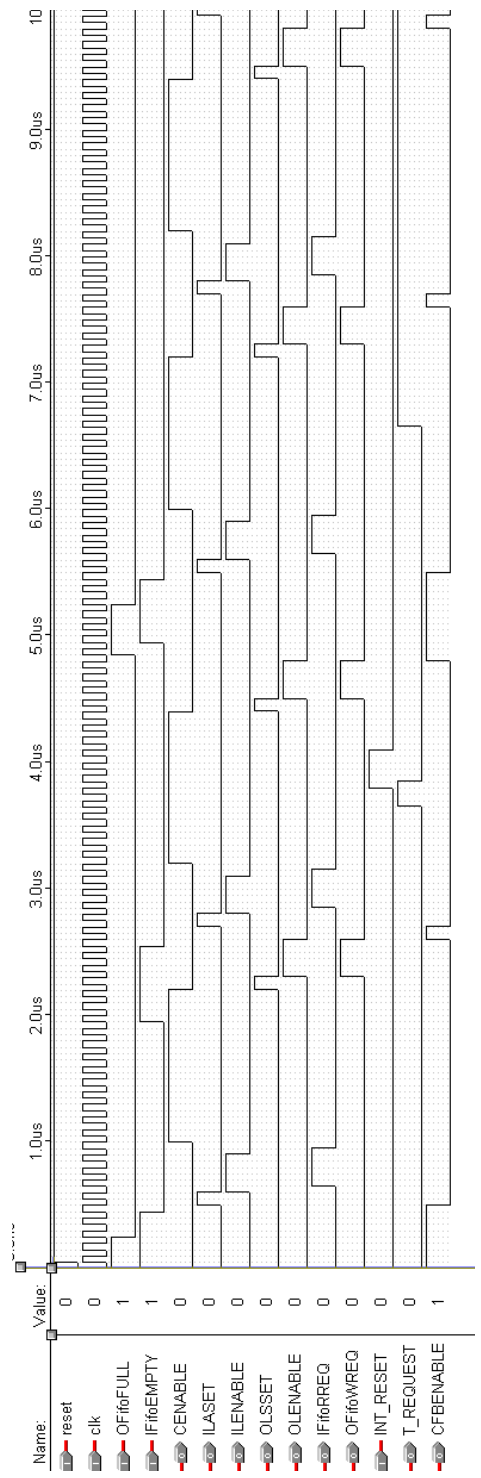

Figure B.10: Control circuitry, full test timing diagram.

```
c.E_D      = E_D;
c.RESET    = RESET;
c.CLOCK    = CLOCK;

-- Interrupt on rising edge of OfifoFull
delay0.clk  = OFifoFULL;
delay0.d    = VCC;
delay0.clrn = not INT_RESET;

-- Connect all the control and data lines to the core
c.CBCFBENABLE = ctrl.CBCFBENABLE and ECB_CBC;
c.DATAIN[]    = DataIN[];
c.ILENABLE    = ctrl.ILENABLE;
c.ILASET      = ctrl.ILASET;
c.OLENABLE    = ctrl.OLENABLE;
c.OLSSET      = ctrl.OLSSET;
c.CENABLE     = ctrl.CENABLE;
c.CBCRESETIN  = CBCRESET;
c.ECB_CBC     = ECB_CBC;

-- Only when in Data mode do we write anything out.
OFifoWREQ = ctrl.OFifoWREQ and Key_Data;
IFifoRREQ = ctrl.IFifoRREQ;

-- Interrupt Request line
INT_REQUEST = (ctrl.INT_REQUEST or delay0.q) and INT_ENABLE;
DataOUT[]   = c.DataOUT[];

END;
```

B.17.1 Timing Diagrams, Crypto

B.18 display_ctrl.vhd

```
--------------------------------------------------------------
--
-- display_ctrl.vhd
--
-- PCI core universal examples
-- v 4.4
--
-- This module controls PLDA prototyping boards leds/displays
-- Leds blink and FIFO size is displayed
--
-- (c) PLD Applications 1999-2000
--
-- PL : 31-jan-2000
-- Modified by John Ronan, May 2000
--
--------------------------------------------------------------
```

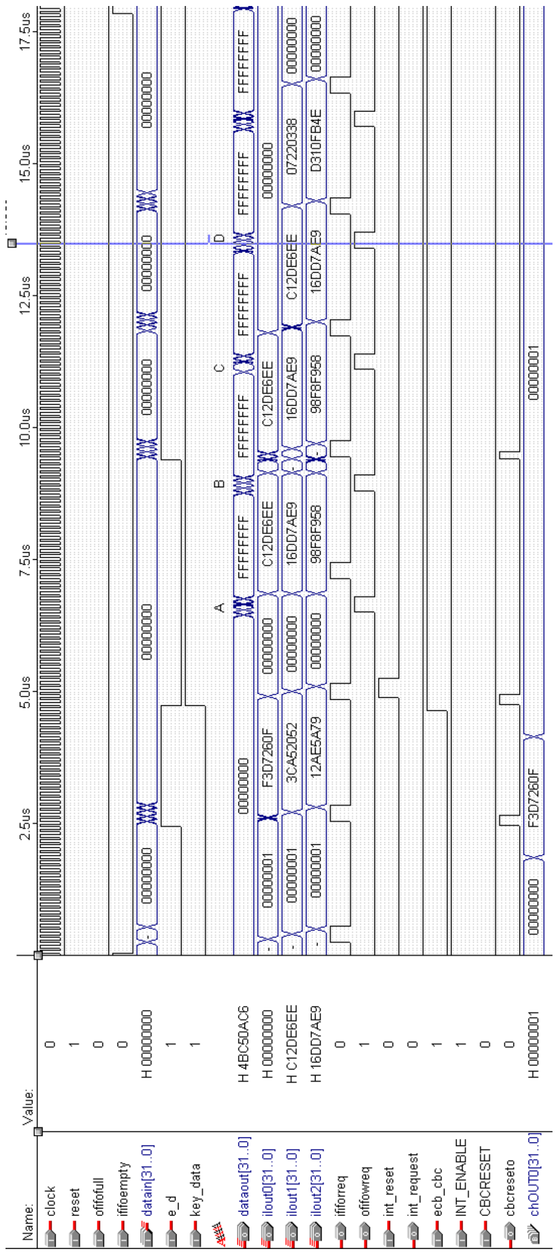

Figure B.11: Crypto, full test timing diagram.

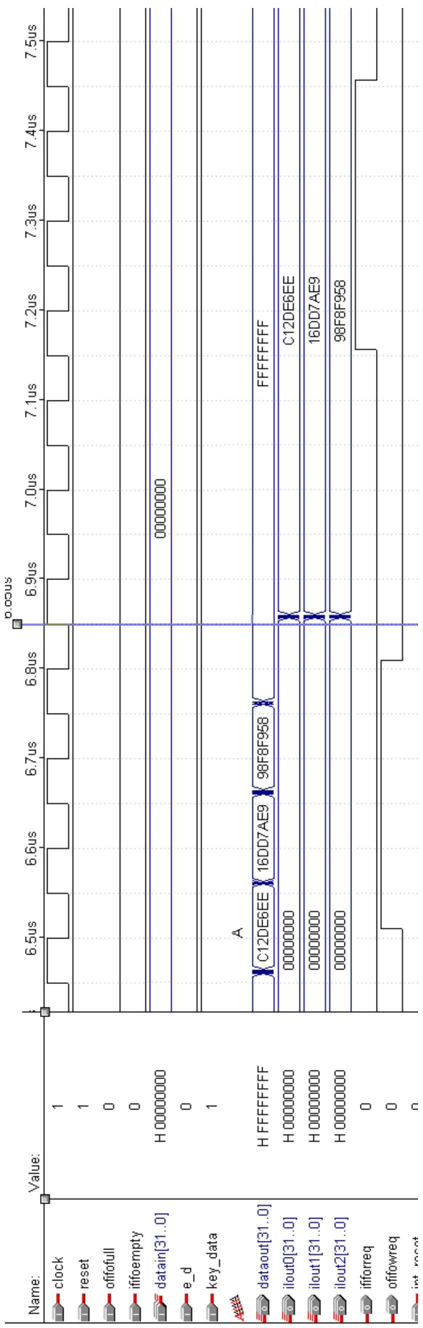

Figure B.12: Crypto, end of first encryption cycle, close up.

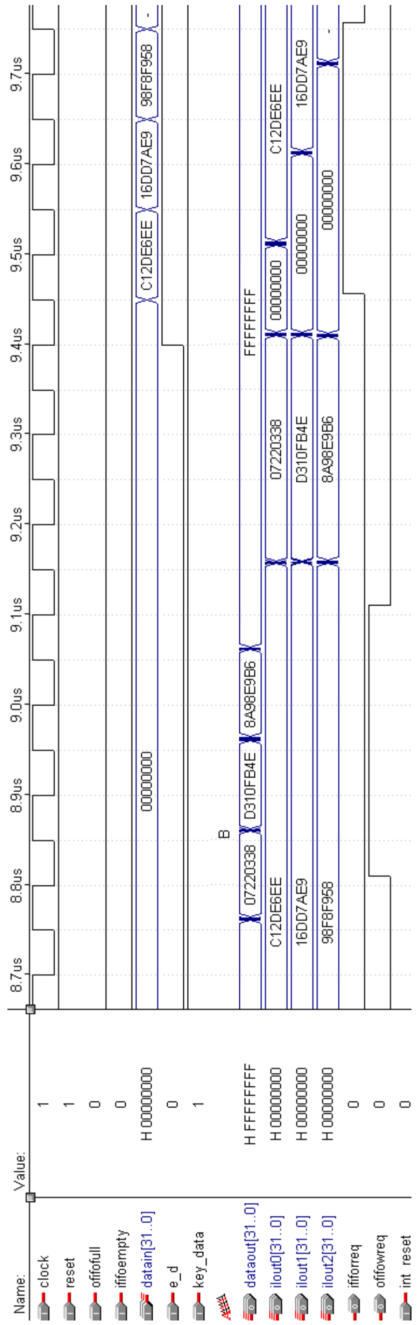

Figure B.13: Crypto, end of second encryption cycle, close up.

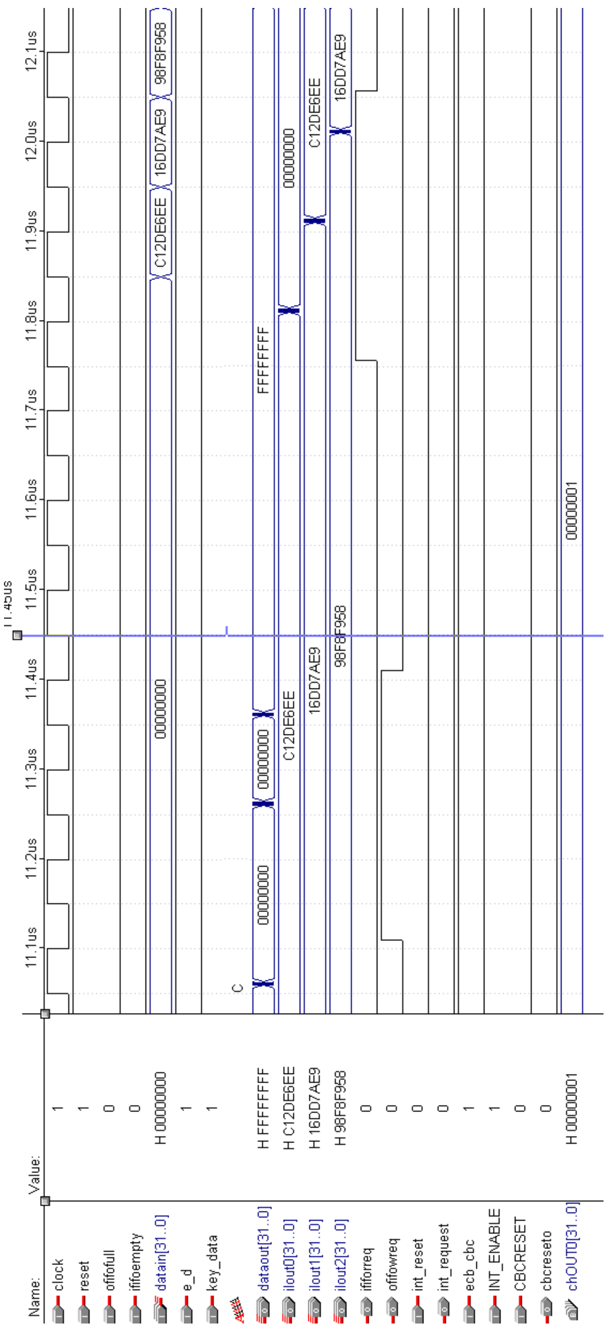

Figure B.14: Crypto, end of first decryption cycle, close up.

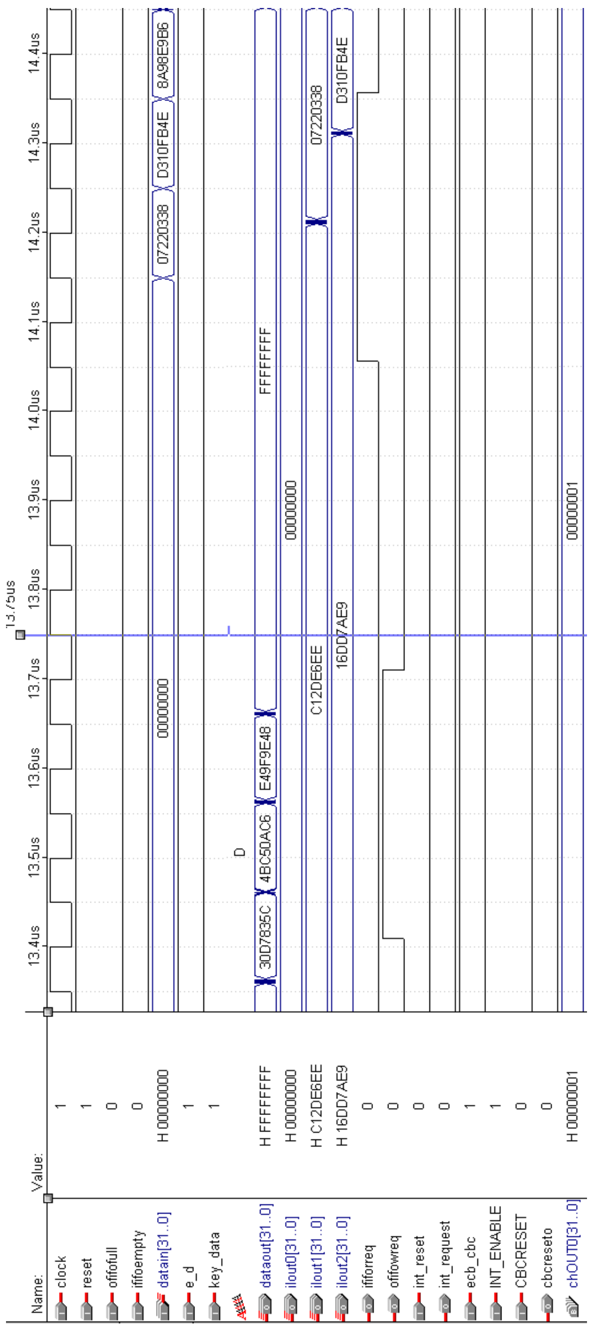

Figure B.15: Crypto, end of second decryption cycle, close up.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity display_ctrl is
        port
                (
                clk             : in    std_logic;
                rst             : in    std_logic;
                ior_read        : in    std_logic;
                ior_write       : in    std_logic;
                c_data_out      : in    std_logic_vector (31 downto 0);
                in_fifo_full    : in    std_logic;
                in_fifo_size    : in    std_logic_vector (9 downto 0);
                out_fifo_size   : in    std_logic_vector (9 downto 0);
                out_fifo_full   : in    std_logic;
                c_conf_in       : out   std_logic_vector (31 downto 0);

                -- Displays
                led_d2                  : out std_logic;        -- Gen10k leds
                led_d3                  : out std_logic;
                disp_a                  : out std_logic_vector (7 downto 0);   -- Pci-prod displays
                disp_b                  : out std_logic_vector (7 downto 0);

                -- crypto These are the control lines out from the ior register
                key_data                : out   std_logic;
                e_d                     : out   std_logic;
                ecb_cbc                 : out   std_logic;
                crypto_reset            : out   std_logic;
                cbcreset                : out   std_logic;
                crypto_int_reset        : out   std_logic; -- unused
                crypto_int_enable       : out   std_logic;
                crypto_int_request      : in    std_logic;

                -- Interrupt control
                int_active              : in  std_logic_vector (3 downto 0);
                int_request             : out std_logic_vector (3 downto 0);
                int_reset               : out std_logic_vector (3 downto 0)
                );
end display_ctrl;

----------------------------------------------------------------

architecture structural of display_ctrl is
        -- LCD display decoder
        component hex_decoder
        port(
                bit_code      : in std_logic_vector(3 downto 0);
                decimal_point : in std_logic;
                hex_digit     : out std_logic_vector(7 downto 0));
        end component;

        signal count_r : std_logic_vector (1 downto 0);
    signal i_key_data, i_e_d, i_ecb_cbc, i_crypto_int_enable, i_crypto_reset, i_cbc_reset : std_logic;
```

```
begin
        ------------------------------------------------------------------------
        -- Manage IOR read/write
        -- bit 0                : interrupt status/interrupt reset
        -- bit 1                : interrupt enable
        -- bit 2                : ecb or cbc mode
        -- bit 3                : encipher or decipher mode
        -- bit 4                : key or data
        -- bit 5        : cbcreset, 1 resets
        -- bit 6       : crypto_reset, 0 resets
        -- bit 7               : spare/unused
        -- bits 18..8   : out_fifo_size
        -- bit  19      : out_fifo_full
        -- bits 30..20  : in_fifo size
        -- bit 31       : in_fifo_full
        ior : process (clk,rst)
        begin
                if rst='0' then
                        count_r             <= (others=>'0');
                        i_crypto_reset      <= '0';
                        i_crypto_int_enable <= '0';
                        i_cbc_reset         <= '1';
                        i_key_data          <= '0';
                        i_ecb_cbc           <= '0';
                        i_e_d               <= '1';
                elsif rising_edge (clk) then
                        count_r <= unsigned (count_r) + '1'; -- increment counter
                        if ior_write='1' then
                                count_r             <= (others=>'0');                   -- reset counter
                                i_crypto_reset      <= c_data_out(6);
                                i_cbc_reset         <= c_data_out(5);
                                i_key_data          <= c_data_out(4);
                                i_e_d               <= c_data_out(3);
                                i_ecb_cbc           <= c_data_out(2);
                                i_crypto_int_enable <= c_data_out(1);
                        elsif count_r ="11" then
                                 -- If we're not being written to and counter hits this value then reset
                                i_crypto_reset <= '1';
                                i_cbc_reset    <= '0';
                        end if;
                end if;
        end process;

        c_conf_in <=in_fifo_full & in_fifo_size & out_fifo_full & out_fifo_size & "0" & i_crypto_reset
                   & i_cbc_reset & i_key_data & i_e_d & i_ecb_cbc & i_crypto_int_enable & int_active(0)
                        when ior_read = '1' else (others=>'0');

        key_data          <= i_key_data;
        e_d               <= i_e_d;
        ecb_cbc           <= i_ecb_cbc;
        crypto_int_enable <= i_crypto_int_enable;
        cbcreset          <= i_cbc_reset;
        crypto_reset      <= i_crypto_reset;
```

```
-- Generate interrupts when backend requests it.
        int_request(0)          <= crypto_int_request and i_crypto_int_enable;
        crypto_int_reset        <= '0'; -- unused
        int_reset(0)            <= '1' when ior_write='1' and c_data_out(0)='1' else '0';
        int_request(3 downto 1) <= "000";       -- PCI core implements only 1 interrupt line
        int_reset(3 downto 1)   <= "000";

-- Diagnostic use of onboard Leds
        lcdb : hex_decoder port map (
                bit_code      => in_fifo_size(9 downto 6), -- Encode top bits of input fifo size
                decimal_point => i_key_data,               -- Decimal point shows value of Key_Data line
                hex_digit     => disp_b                    -- Send out the value to the Led Display
                );
        lcda : hex_decoder port map (
                bit_code      => out_fifo_size(9 downto 6), -- Encode top bits of output fifo size
                decimal_point => i_crypto_int_enable,       -- Decimal point shows value of Interrupt enable line
                hex_digit     => disp_a                     -- Send out the value to the Led Display
                );
end structural;
```

Appendix C

Device Driver

C.1 3way.h

```
// $Header: //j0n/module/3way.h#6 $
// John Ronan July 2000

#include "common.h"
#ifndef _3way_h
#define _3way_h

void theta (WORD32 * a);        // linear step
void mu (WORD32 * a);           // bit flipper
void pi_1 (WORD32 * a);         // bit shifter
void gamma (WORD32 * a);        // the nonlinear step
void pi_2 (WORD32 * a);         // bit shifter
void rho (WORD32 * a);          // round

#endif // _3way_h
```

C.2 3way.c

```
// $Header: //j0n/module/3way.c#6 $
// John Ronan June 2000

#include "common.h"
#include "3way.h"

void
theta (WORD32 * a)
{

  WORD32 b[3], temp, temp2, temp3, temp4, temp5;

  temp = a[0] ^ a[1] ^ a[2];
  temp2 = (temp << 16) ^ (temp >> 16);
  temp3 = (a[0] << 8) ^ (a[0] >> 8);
  temp4 = (a[1] << 8) ^ (a[1] >> 8);
  temp5 = (a[2] << 8) ^ (a[2] >> 8);

  b[0] =
    a[0] ^ temp2 ^ temp5 ^ (a[1] >> 24) ^ (a[0] << 24) ^ (a[2] >> 24) ^ (a[0]
                                                                         <<
                                                                         8);
  b[1] =
    a[1] ^ temp2 ^ temp3 ^ (a[2] >> 24) ^ (a[1] << 24) ^ (a[0] >> 24) ^ (a[1]
                                                                         <<
                                                                         8);
  b[2] =
    a[2] ^ temp2 ^ temp4 ^ (a[0] >> 24) ^ (a[2] << 24) ^ (a[1] >> 24) ^ (a[2]
                                                                         <<
                                                                         8);
  a[0] = b[0];
  a[1] = b[1];
  a[2] = b[2];
}

void
mu (WORD32 * a)
{
  int i;
  WORD32 b[3];

  b[0] = b[1] = b[2] = 0;
  for (i = 0; i < 32; i++)
    {
      b[0] <<= 1;
      b[1] <<= 1;
      b[2] <<= 1;
      if (a[0] & 1)
          b[2] |= 1;
      if (a[1] & 1)
          b[1] |= 1;
      if (a[2] & 1)
          b[0] |= 1;
      a[0] >>= 1;
      a[1] >>= 1;
      a[2] >>= 1;
    }
  a[0] = b[0];
  a[1] = b[1];
  a[2] = b[2];
}

// Optimised version
void
pi_gamma_pi (WORD32 * a)
{

  WORD32 b0, b2;
  b2 = (a[2] << 1) | (a[2] >> 31);
  b0 = (a[0] << 22) | (a[0] >> 10);
  a[0] = (b0 ^ (a[1] | (~b2))) << 1 | ((b0 ^ (a[1] | (~b2))) >> 31);
  a[2] = ((b2 ^ (b0 | (~a[1]))) << 22) | ((b2 ^ (b0 | (~a[1]))) >> 10);
  a[1] = a[1] ^ (b2 | (~b0));

}

void
rho (WORD32 * a)
{
  theta (a);
  pi_gamma_pi (a);
}
```

C.3 common.h

```
// $Header: //j0n/module/common.h#8 $
// John Ronan, June 2000

#ifndef _common_h
#define _common_h
#include <linux/types.h>
#include <asm/spinlock.h>

#define WORD32 u32 // Note this is only defined for kernel space.

#define STRT_E 0x0b0b // round constant of first encryption round
#define STRT_D 0xb1b1 // round constant of first decryption round
#define NMBR 11       // number of rounds is 11
#define TWAY_BLOCKSIZE 3
#define TWAY_BYTE_BLOCKSIZE 12
#define KEYFILE "/etc/kakfile"
#define RANDOMDEV "/dev/random"

#undef PINFO
#define PINFO(fmt, args...) printk( KERN_DEBUG "crypto: " fmt, ## args)

#undef PDEBUG             // undef it, just in case
  #ifdef CRYPTO_DEBUG
    #ifdef __KERNEL__
// This one if debugging is on, and kernel space
      #define PDEBUG(fmt, args...) printk( KERN_DEBUG "crypto: " fmt, ## args)
    #else
// This one for user space
      #define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
    #endif
  #else
    #define PDEBUG(fmt, args...) // not debugging: nothing
  #endif

typedef struct software_key
```

```
{
  // crypto stuff
  WORD32 masterKey[TWAY_BLOCKSIZE];
  WORD32 encipherKey[TWAY_BLOCKSIZE];
  WORD32 decipherKey[TWAY_BLOCKSIZE];
  WORD32 encipherRoundConstant[NMBR+1];
  WORD32 decipherRoundConstant[NMBR+1];
  WORD32 initialisationVector[TWAY_BLOCKSIZE];

}softwarekey;
#endif // _common_h
```

C.4 software.h

```
// $Header: //j0n/module/software.h#9 $
// John Ronan, June 2000

#ifndef _software_h
#define _software_h
#include "common.h"

void swInit (struct software_key *key);
ssize_t swWrite (struct file *file, const char *const buffer, size_t length,
                 loff_t * offset);
ssize_t swRead (struct file *file, char *const buffer, size_t length,
                loff_t * offset);
void swRoundInit (struct software_key *const key);
void swEncipher (const struct software_key *const key, WORD32 * const a);
void swDecipher (const struct software_key *const key, WORD32 * const a);
void swKakDecipher (const struct software_key * const key, WORD32 * const a);
int swIsDeviceReadyForData (const struct Crypto_Dev *const cDev);
int swIsOutputBufferEmpty (const struct Crypto_Dev *const cDev);
#endif // _software_h
```

C.5 Software.c

```
// $Header: //j0n/module/software.c#10 $
// John Ronan, June 2000

#include <linux/vmalloc.h>
#include <asm/uaccess.h>        // Copy_from/to_user
#include "3way.h"
#include "driver.h"             // cDev
#include "software.h"
#include "hardware.h"           // PLDA_BUFFER_SIZE

rwlock_t sw_lock = RW_LOCK_UNLOCKED;

void
swInit (softwarekey * SoftwareKey)
{
  PDEBUG ("Inside SoftwareInit\n");
  // load the KeyAuthenthicationKey from file
  SoftwareKey->masterKey[0] = 0x00000001;
  SoftwareKey->masterKey[1] = 0x00000001;
  SoftwareKey->masterKey[2] = 0x00000001;

  // Default encryption key.
  SoftwareKey->encipherKey[0] = SoftwareKey->decipherKey[0] = 0x00000001;
  SoftwareKey->encipherKey[1] = SoftwareKey->decipherKey[1] = 0x00000001;
  SoftwareKey->encipherKey[2] = SoftwareKey->decipherKey[2] = 0x00000001;

  // Default initialisation vector.
  SoftwareKey->initialisationVector[0] = 0;
  SoftwareKey->initialisationVector[1] = 0;
  SoftwareKey->initialisationVector[2] = 0;

  // Has to be called every time the encipherKey is changed
  swRoundInit (SoftwareKey);
}

// Crypto functions
// call decrypt using the masterkey as the key and this as the data.
void
swRoundInit (softwarekey * const SoftwareKey)
{
  int i = 0;
  WORD32 encipherStart = STRT_E;
  WORD32 decipherStart = STRT_D;

  // Initialise round constants
  PDEBUG ("Roundinit (%p)\n", SoftwareKey);
  for (i = 0; i <= NMBR; i++)
    {
      SoftwareKey->encipherRoundConstant[i] = encipherStart;
      encipherStart <<= 1;
      if (encipherStart & 0x10000)
          encipherStart ^= 0x11011;
    }

  for (i = 0; i <= NMBR; i++)
    {
      SoftwareKey->decipherRoundConstant[i] = decipherStart;
      decipherStart <<= 1;
      if (decipherStart & 0x10000)
          decipherStart ^= 0x11011;
    }
  // Initialise decryption key
  theta (SoftwareKey->decipherKey);
  mu (SoftwareKey->decipherKey);
  // Key's initialised.. checked against reference code...
}

void
swEncipher (const softwarekey * const SoftwareKey, WORD32 * const a)
{
  // Hardware implementation should encompass all of this function.
  int i = 0;
  for (i = 0; i < NMBR; i++)
    {
      // XOR with vector that depends on encryption Key and round number.
      a[0] ^=
        SoftwareKey->
        encipherKey[0] ^ (SoftwareKey->encipherRoundConstant[i] << 16);
      a[1] ^= SoftwareKey->encipherKey[1];
      a[2] ^=
        SoftwareKey->encipherKey[2] ^ SoftwareKey->encipherRoundConstant[i];
      // Encryption Round
      rho (a);
    }

  // Extra application of Vector and theta()
  a[0] ^=
    SoftwareKey->
    encipherKey[0] ^ (SoftwareKey->encipherRoundConstant[NMBR] << 16);
  a[1] ^= SoftwareKey->encipherKey[1];
  a[2] ^=
    SoftwareKey->encipherKey[2] ^ SoftwareKey->encipherRoundConstant[NMBR];
  theta (a);
  // block of data is now encrypted.
}

void
swDecipher (const softwarekey * const SoftwareKey, WORD32 * const a)
{
  int i = 0;
  // Reverse bits in each Longword.
  mu (a);
  for (i = 0; i < NMBR; i++)
    {
      // XOR with vector that depends on decryption Key and round number.
      a[0] ^=
        SoftwareKey->
        decipherKey[0] ^ (SoftwareKey->decipherRoundConstant[i] << 16);
      a[1] ^= SoftwareKey->decipherKey[1];
      a[2] ^=
        SoftwareKey->decipherKey[2] ^ SoftwareKey->decipherRoundConstant[i];

      // Encryption Round same as encryption
      rho (a);
    }

  // Extra application of Vector and theta()
  a[0] ^=
    SoftwareKey->
    decipherKey[0] ^ (SoftwareKey->decipherRoundConstant[NMBR] << 16);
  a[1] ^= SoftwareKey->decipherKey[1];
  a[2] ^=
    SoftwareKey->decipherKey[2] ^ SoftwareKey->decipherRoundConstant[NMBR];
  theta (a);
  // Reverse bits to get plaintext
  mu (a);
}

void
swKakDecipher (const softwarekey * const SoftwareKey, WORD32 * const a)
{
  WORD32 tempKey[3];
  int i = 0;

  for (i = 0; i < 3; i++)
    tempKey[i] = SoftwareKey->masterKey[i];

  theta (tempKey);
  mu (tempKey);
  // Reverse bits in each Longword.
  mu (a);
  for (i = 0; i < NMBR; i++)
    {
      // XOR with vector that depends on decryption Key and round number.
      a[0] ^= tempKey[0] ^ (SoftwareKey->decipherRoundConstant[i] << 16);
      a[1] ^= SoftwareKey->decipherKey[1];
      a[2] ^= tempKey[2] ^ SoftwareKey->decipherRoundConstant[i];

      // Encryption Round same as encryption
      rho (a);
    }

  // Extra application of Vector and theta()
  a[0] ^= tempKey[0] ^ (SoftwareKey->decipherRoundConstant[NMBR] << 16);
  a[1] ^= SoftwareKey->decipherKey[1];
  a[2] ^= tempKey[2] ^ SoftwareKey->decipherRoundConstant[NMBR];
  theta (a);

  // Reverse bits to get plaintext
  mu (a);
}

ssize_t
swWrite (struct file *file, const char *buffer, size_t length,
         loff_t * offset)
{
  CryptoDev *cDev = file->private_data;

  WORD32 *tempPtr = 0;
  WORD32 blockCount = 0;
  WORD32 l_buffer[TWAY_BLOCKSIZE];
  WORD32 l_tempBuffer[TWAY_BLOCKSIZE];
  int i = 0;

  PDEBUG ("swWrite\n");

  write_lock (&sw_lock);
  PDEBUG ("Buffer Size %d, length %d\n", PLDA_BUFFER_SIZE, length);
  if (length > PLDA_BUFFER_SIZE)
```

```
    length = PLDA_BUFFER_SIZE;

  if (copy_from_user (cDev->oFifoBuf, buffer, length))
    {
      PINFO ("User Space Buffer is not valid %s(%d)", __FILE__, __LINE__);
      return -EFAULT;
    }

  blockCount = length;
  tempPtr = (WORD32 *) cDev->oFifoBuf;
  PDEBUG ("before blockcount %d, tempPtr %p\n", blockCount, tempPtr);

  // So we're encrypting
  if (cDev->encDec == 1)
    {
      for (i = 0; i < 3; i++)
        {
          // Seed cbc mode
          l_buffer[i] = cDev->softwareKeys.initialisationVector[i];
        }

      // Begin cbc mode encipherment.
      while (blockCount >= 1)
        {
          for (i = 0; i < TWAY_BLOCKSIZE; i++)
            {
              // xor with previous ciphertext
              tempPtr[i] = l_buffer[i] ^ tempPtr[i];
            }

          swEncipher (&cDev->softwareKeys, tempPtr);

          // Store generated ciphertext for next round
          for (i = 0; i < TWAY_BLOCKSIZE; i++)
            {
              l_buffer[i] = tempPtr[i];
            }
          PDEBUG ("blockcount %d, tempPtr %p max (%p)\n", blockCount, tempPtr,
                  (cDev->oFifoBuf + PAGE_SIZE));
          // Increment pointer.
          tempPtr += TWAY_BLOCKSIZE;
          blockCount -= TWAY_BYTE_BLOCKSIZE;
        }
      length = length - blockCount;
    }

  // Decryption on write
  else
    {
      if (cDev->encDec == 2)
        {
          blockCount = length;
          tempPtr = (WORD32 *) cDev->oFifoBuf;

          for (i = 0; i < TWAY_BLOCKSIZE; i++)
```

```
            {
              // Seed cbc mode
              l_buffer[i] = cDev->softwareKeys.initialisationVector[i];
            }
          // Begin cbc mode decipherment.
          while (blockCount >= TWAY_BLOCKSIZE)
            {
              for (i = 0; i < TWAY_BLOCKSIZE; i++)
                {
                  l_tempBuffer[i] = tempPtr[i];
                }
              swDecipher (&cDev->softwareKeys, tempPtr);
              // XOR old ciphertext block with new plaintext block to remove CBC
              for (i = 0; i < TWAY_BLOCKSIZE; i++)
                  tempPtr[i] = tempPtr[i] ^ l_buffer[i];
              // Store original ciphertext block for next decipher operation
              for (i = 0; i < TWAY_BLOCKSIZE; i++)
                  l_buffer[i] = l_tempBuffer[i];
              PDEBUG ("blockcount %d, tempPtr %p\n", blockCount, tempPtr);
              // Increment pointer.
              tempPtr += TWAY_BLOCKSIZE;
              blockCount -= TWAY_BYTE_BLOCKSIZE;
            }
          length = length - blockCount;
        }
    }
  PDEBUG ("outside if blockcount %d, tempPtr %p\n", blockCount, tempPtr);
  // End software
  cDev->outputBufferLength = length;
  write_unlock (&sw_lock);
  wake_up_interruptible (&cDev->readq);
  PDEBUG ("software write returning\n");
  return length;
}

ssize_t
swRead (struct file * file, char *const buffer, size_t length,
        loff_t * offset)
{
  CryptoDev *cDev = file->private_data;

  write_lock (&sw_lock);
  PDEBUG ("Software read(%p)\n", file);

  // If the requested length is greater than what we have stored, then we
  // return what we have and the number of bytes returned... this is the correct semantics
  // for the read function.
  if (length > cDev->outputBufferLength)
    length = cDev->outputBufferLength;

  // If this call fails it means that some of the data could not be copied due
  // to an invalid buffer.
  if (copy_to_user (buffer, cDev->oFifoBuf, length))
    {
      PINFO ("User Space Buffer is invalid %s(%d)", __FILE__, __LINE__);
```

```
        return -EFAULT;
      }
    // Ok data has been read from it now...
    //cDev->isData = 0;
    cDev->outputBufferLength = 0;

    // Allow any writers to continue
    write_unlock (&sw_lock);
    wake_up_interruptible (&cDev->writeq);
    return length;
}

int
swIsDeviceReadyForData (const struct Crypto_Dev *const cDev)
{
  return swIsOutputBufferEmpty (cDev);
}

int
swIsOutputBufferEmpty (const struct Crypto_Dev *const cDev)
{
  if (cDev->outputBufferLength == 0)
    return 1;
  else
    return 0;
}
```

C.6 driver.h

```
// driver.h - ioctl definitions and device dependant structure
// $Header: //j0n/module/driver.h#10 $
// John Ronan, June 2000

#ifndef __driver_h
#define __driver_h

#include "common.h"
#include <linux/ioctl.h>
#include <linux/tqueue.h>
// The major device number. We can't rely on dynamic
// registration any more, because ioctls need to know
// it.
#define MAJOR_NUM 100

#define CRYPTO_DEC_USE_COUNT _IO(MAJOR_NUM, 10)
#define CRYPTO_ENC_DEC _IOW(MAJOR_NUM, 0, int)
#define CRYPTO_ECB_CBC _IOW(MAJOR_NUM, 1, int)
// Set the message of the device driver
#define CRYPTO_SET_IV  _IOW(MAJOR_NUM, 2, char *)
#define CRYPTO_SET_KEY _IOW(MAJOR_NUM, 3, char *)

#define CRYPTO_RESET   _IO(MAJOR_NUM, 4)
```

```
// Get the IV from the device driver
#define CRYPTO_GET_IV _IOR(MAJOR_NUM, 3, char *)

// The name of the device file
#define DEVICE_FILE_NAME "/dev/crypto"
#define KAK_FILE_NAME "/etc/kakfile"
#define RANDOM_DEVICE "/dev/random"

typedef struct Crypto_Dev
{
  u16 deviceOpen;
  uid_t deviceOwner;
  u8 encDec;                    // 0 for pass through 1 for encrypt 2 for decrypt
  u32 inputBufferLength;        // this may be redundant
  u32 outputBufferLength;
  u8 dataPending;               // data left in the output fifo.
  u8 encUsedIV;                 // Have we used the IV in the encryption stream
  u8 decUsedIV;                 // Ditto for decryption stream.
  void *iFifoBuf;
  void *oFifoBuf;
  struct pci_dev *dev;
  struct wait_queue *readq, *writeq, *openq;
  struct software_key softwareKeys;
  struct tq_struct crypto_queue;
}
CryptoDev;


#endif // driver.h
```

C.7 driver.c

```
// $Header: //j0n/module/driver.c#13 $
// Create an input/output character device
// Copyright (C) 1999-2000 John Ronan

#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif

// Deal with CONFIG_MODVERSIONS
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#ifndef NOSYM
#define EXPORT_SYMTAB           // Yes, I export a symbol table, has to be defined before Module.h is included
#endif
```

```
// The necessary header files
// Standard in kernel modules
#include <linux/kernel.h>       // We're doing kernel work
#include <linux/module.h>       // Specifically, a module
#include <asm/io.h>

// This will be required when I get the key stuff into the module
extern void get_random_bytes (void *, int);

// For character devices
// The character device definitions are here
#include <linux/fs.h>

// Necessary because we use proc fs
#include <linux/proc_fs.h>

// A wrapper which does next to nothing at
// at present, but may help for compatibility
// with future versions of Linux
#include <linux/wrapper.h>
#include <linux/malloc.h>
// 3way stuff
#include "common.h"
#include "driver.h"
#include "software.h"
#include "hardware.h"

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>        // for get_user and put_user
#endif

#define SUCCESS 0
// Device Declarations *******************************

// The name for our device, as it will appear in
// /proc/devices
#define DEVICE_NAME "crypto"

// default mode=0 try and init hardware board.
int mode = 0;
MODULE_PARM (mode, "i");

CryptoDev cryptoDevice =
  { 0, 0, 0, 0, 0, 0, 0, 0, NULL, NULL, NULL, NULL, NULL, NULL };

static int usingHardware = 0;   // Are we using hardware or software

// Get 4K..(2 ^ 12). pg 218 Linux Device Drivers.. only portable way to do it.
// Fifo's on the Board will thus be 4K each. Calculated at compile time,
// thus no runtime overhead. used in AllocateDmaBuffers & FreeDmaBuffers
static int order =
  (PLDA_SOFT_BUF_SIZE - PAGE_SHIFT > 0) ? PLDA_SOFT_BUF_SIZE - PAGE_SHIFT : 0;

int
isDeviceReadyForData (const struct Crypto_Dev *const cDev)
```

```
{
  if (usingHardware == 1)
    return pldIsDeviceReadyForData (cDev);
  if (usingHardware == 0)
    return swIsDeviceReadyForData (cDev);
  PINFO ("Serious error %s(%d)\n", __FILE__, __LINE__);
  return 0;
}

int
isDeviceOutputEmpty (const struct Crypto_Dev *const cDev)
{
  if (usingHardware == 1)
    return pldIsOutputBufferEmpty (cDev);
  if (usingHardware == 0)
    return swIsOutputBufferEmpty (cDev);
  PINFO ("Serious error %s(%d)\n", __FILE__, __LINE__);
  return 0;
}

int
AllocateDmaBuffers (void)
{
#ifdef CRYPTO_DEBUG
  char *c1;
  char *c2;
  int i;
#endif

  cryptoDevice.oFifoBuf = (void *) __get_free_pages (GFP_DMA, order);
  cryptoDevice.iFifoBuf = (void *) __get_free_pages (GFP_DMA, order);

  if (!cryptoDevice.oFifoBuf || !cryptoDevice.iFifoBuf)
    {
      PINFO ("Error allocating DMA pages %s(%d)\n", __FILE__, __LINE__);
      return 1;
    }
  PINFO ("Allocated 2 Dma Buffers of %d Bytes\n", 1 << PLDA_SOFT_BUF_SIZE);
  PDEBUG ("Outputbuffer (%p) InputBuffer (%p)\n", cryptoDevice.oFifoBuf,
          cryptoDevice.iFifoBuf);
  // Colors the pages for debugging purposes

#ifdef CRYPTO_DEBUG
  c1 = (char *) cryptoDevice.oFifoBuf;
  c2 = (char *) cryptoDevice.iFifoBuf;

  for (i = 0; i < PAGE_SIZE; i++)
    {
      *(c1 + i) = 0xFF;
      *(c2 + i) = 0xFF;
    }
  PDEBUG ("Outputbuffer End (%p) InputBuffer End (%p)\n", c1 + i, c2 + i);
#endif
  return SUCCESS;
}
```

```
void
FreeDmaBuffers (void)
{
  if (cryptoDevice.iFifoBuf)
      free_pages ((unsigned long) cryptoDevice.iFifoBuf, order);
  if (cryptoDevice.oFifoBuf)
      free_pages ((unsigned long) cryptoDevice.oFifoBuf, order);
}

ssize_t
crypto_proc_output (struct file *file,  // The file read
                    char *buf,  // The buffer to put data to (in the user segment)
                    size_t len, // The length of the buffer
                    loff_t * offset)    // Offset in the file - ignore
{
  static int finished = 0;
  //int i;
  int pageSize = PAGE_SIZE;
  char message[PAGE_SIZE + 1];

  u32 ior = 0, ofifolength = 0, ififolength = 0;

  // We return 0 to indicate end of file, that we have
  // no more information. Otherwise, processes will
  // continue to read from us in an endless loop.
  if (finished)
    {
      finished = 0;
      return 0;
    }
  len = 0;

  if (usingHardware == 1)
    {
      pci_read_config_dword (cryptoDevice.dev, IOR_REG, &ior);
      ofifolength = ((ior >> PLDA_OFIFO_POSN) & PLDA_FIFO_MASK);
      ififolength = ((ior >> PLDA_IFIFO_POSN) & PLDA_FIFO_MASK);
      len += sprintf (message,
                      "\nPld ofifolen:\t\t%d\noutputBufferLength:\t%d\nPld ififolen:
                       \t\t%d\ninputBufferLength:\t%d\nencDec:\t\t\t%d\nDeviceOpen
                       \t\t%d\nusingHardware:\t\t%d\nPage Size is:\t\t %d\n",
                      ofifolength, cryptoDevice.outputBufferLength,
                      ififolength, cryptoDevice.inputBufferLength,
                      cryptoDevice.encDec, cryptoDevice.deviceOpen,
                      usingHardware, pageSize);
    }

  if (usingHardware == 0)
    {
      len += sprintf (message,
                      "\noutputBufferLength:\t%d\ninputBufferLength:\t%d\nencDec:
                       \t\t\t%d\nDeviceOpen\t\t%d\nusingHardware:\t\t%d\nPage Size is:\t\t %d\n",
                      cryptoDevice.outputBufferLength,
                      cryptoDevice.inputBufferLength,
                      cryptoDevice.encDec, cryptoDevice.deviceOpen,
                      usingHardware, pageSize);
    }
  if (len > pageSize)
    len = pageSize;

  if (copy_to_user (buf, message, len))
    return -EFAULT;

  finished = 1;
  return len;
}

int
crypto_proc_permission (struct inode *inode, int op)
{
  // We allow everybody to read from our module
  if (op == 4)
    return 0;

  // If it's anything else, access is denied
  return -EACCES;
}

// The file is opened - we don't really care about
// that, but it does mean we need to increment the
// module's reference count.
int
crypto_proc_open (struct inode *inode, struct file *file)
{
  PDEBUG ("proc_open\n");
  MOD_INC_USE_COUNT;
  return 0;
}

// The file is closed - again, interesting only because
// of the reference count.
int
crypto_proc_close (struct inode *inode, struct file *file)
{
  PDEBUG ("proc_close\n");
  MOD_DEC_USE_COUNT;
  return SUCCESS;

}

struct file_operations File_Ops_4_Our_Proc_File = {
  NULL,                         // lseek
  crypto_proc_output,           // "read" from the file
  NULL,                         // "write" to the file
  NULL,                         // readdir
  NULL,                         // select
  NULL,                         // ioctl
  NULL,                         // mmap
  crypto_proc_open,             // Somebody opened the file
  NULL,                         // flush, added here in version 2.2
  crypto_proc_close,            // Somebody closed the file
};

struct inode_operations Inode_Ops_4_Our_Proc_File = {
  &File_Ops_4_Our_Proc_File,
  NULL,                         // create
  NULL,                         // lookup
  NULL,                         // link
  NULL,                         // unlink
  NULL,                         // symlink
  NULL,                         // mkdir
  NULL,                         // rmdir
  NULL,                         // mknod
  NULL,                         // rename
  NULL,                         // readlink
  NULL,                         // follow_link
  NULL,                         // readpage
  NULL,                         // writepage
  NULL,                         // bmap
  NULL,                         // truncate
  crypto_proc_permission        // check for permissions
};


// Directory entry
struct proc_dir_entry Our_Proc_File = {
  0,                            // Inode number - ignore, it will be filled by
  // proc_register[_dynamic]
  6,                            // Length of the file name
  "crypto",                     // The file name
  S_IFREG | S_IRUGO | S_IWUSR,
  1,
  0, 0,                         // The uid and gid for the file -
  // we give it to root
  96,                           // The size of the file reported by ls.
  &Inode_Ops_4_Our_Proc_File,
  // A pointer to the inode structure for
  // the file, if we need it. In our case we
  // do, because we need a write function.
  NULL
    // The read function for the file. Irrelevant,
    // because we put it in the inode structure above
};

// This function is called whenever a process attempts
// to open the device file
int
crypto_open (struct inode *inode, struct file *file)
{
  CryptoDev *cDev = &cryptoDevice;
  int num = MINOR (inode->i_rdev);
  int retval = SUCCESS;

  if (!cDev->iFifoBuf || !cDev->oFifoBuf)
```

```
    {
      PINFO ("Buffers not allocated %s(%d)\n", __FILE__, __LINE__);
      return 0;
    }

  // Only one physical device... this is a bit of overkill maybe
  if (num > 0)
    return -ENODEV;
  PDEBUG ("device_open(%p)(%p)\n", inode, file);

  // We don't want to talk to two processes at the
  // same time
  while (cDev->deviceOpen && (cDev->deviceOwner != current->uid) &&     // Allow user
         (cDev->deviceOwner != current->euid) &&        // Allow whover did su
         !suser ())             // Allow root
    {
      if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;
      interruptible_sleep_on (&cDev->openq);    // Put this process to sleep.
      if (signal_pending (current))     // a signal arrived
        return -ERESTARTSYS;    // Tell Filesystem Layer to handle it
      // else loop
    }

  if (cDev->deviceOpen == 0)
    {
      cDev->deviceOwner = current->uid; // Grab it
      if (usingHardware == 1)
        {
          if (pldRequestInterrupt (cDev))
            retval = 1;
        }
    }
  cDev->deviceOpen++;
  file->private_data = cDev;

  // Card is found... Still haven't done anything... must set key first..
  MOD_INC_USE_COUNT;
  return retval;
}

int
crypto_release (struct inode *inode, struct file *file)
{
  CryptoDev *cDev = file->private_data;
  PDEBUG ("device_release(%p,%p)\n", inode, file);

  // We're now ready for our next caller
  cDev->deviceOpen--;

  if (cDev->deviceOpen == 0)
    {
      if (usingHardware == 1)
        {
          // Disable onboard interrupts and free interrupt line
        pldFreeInterrupt (cDev);
      }
    wake_up_interruptible (&cDev->openq);     // awake other waiting uid's
  }
  // If the module gets 'stuck' running the 'ioctl' program will decreast this value and allow it to be unloaded
  MOD_DEC_USE_COUNT;
  return 0;
}

// This function is called whenever a process which
// has already opened the device file attempts to
// seek through it.  It doesn't make sense for my device
long long
crypto_lseek (struct file *file, long long offset, int whence)
{
  PDEBUG ("lseek(%p)\n", file);
  return -ESPIPE;               // Illegal Seek
}

// This function is called whenever a process which
// has already opened the device file attempts to
// read from it.
ssize_t
crypto_read (struct file * file, char *buffer,  // The buffer to fill with the data
             size_t length,     // The length of the buffer
             loff_t * offset)   // offset to the file
{
  CryptoDev *cDev = file->private_data;
  int retval = 0;

  // Paranoia
  if (!cDev->oFifoBuf)
    {
      return -EFAULT;
    }
  User has to read an even number of block lengths from the device
  if (length < TWAY_BLOCKSIZE - 1 || length % TWAY_BYTE_BLOCKSIZE)
    {
      PDEBUG ("Block size Error %s(%d) %d\n", __FILE__, __LINE__, length);
      return -EIO;            // I/O error possibly the best signal to return.
    }

  // Ok there's nothing in the Output Fifo and theres nothing in the buffer so we can do
  // absolutely nothing... put the process to sleep.
  while (isDeviceOutputEmpty (cDev))
    {
      // If the device was opened in nonblocking mode, we tell 'em to try again
      if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;
      interruptible_sleep_on (&cDev->readq);
      if (signal_pending (current))     // a signal arrived
        return -ERESTARTSYS;    // tell the filesystem layer to handle it
    }

  // Ok now we're getting down to it.  we have data that we can read either
  // In the output Fifo or our buffer.
  if (usingHardware == 1)
    {
      retval = pldRead (file, buffer, length, offset);
    }
  if (usingHardware == 0)
    {
      retval = swRead (file, buffer, length, offset);
    }
  // This is ok for now, but this has to be made interrupt aware.
  // hence this will probably move to an interrupt handler.
  wake_up_interruptible (&cDev->writeq);
  return retval;
}


// This function is called when somebody tries to
// write into our device file.
static ssize_t
crypto_write (struct file *file,
              const char *buffer, size_t length, loff_t * offset)
{

  CryptoDev *cDev = file->private_data;
  unsigned long retval;

  // In case a NULL buffer gets passed down.. paranoia
  if (!cDev->iFifoBuf)
    {
      return -EFAULT;
    }

  // From an email with Alessandro Rubini.  If the length is greater than the FiFo Length, then
  // just write what we can and put the caller to sleep. return the no of bytes written.

  if (length < TWAY_BLOCKSIZE - 1 || length % TWAY_BYTE_BLOCKSIZE)
    {
      PDEBUG ("Data needs to be even multiples of 12 bytes %d\n",
              (int) length);
      return -EIO;              // I/O error possibly the best signal to return.
    }

  // if the input buffer is not empty then put the caller to sleep.
  while (!isDeviceReadyForData (cDev))
    {
      // If the device was opened in nonblocking mode, we tell 'em to try again
      if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;

      interruptible_sleep_on (&cDev->writeq);

      if (signal_pending (current))     // a signal arrived
        return -ERESTARTSYS;    // tell the filesystem layer to handle it
      // otherwise loop
    }
  if (usingHardware == 1)
    {
      retval = pldWrite (file, buffer, length, offset);
    }
  else
    {
      retval = swWrite (file, buffer, length, offset);
    }
  return retval;

}

int
crypto_ioctl (struct inode *inode, struct file *file, unsigned int ioctl_num,   // The number of the ioctl
              unsigned long ioctl_param)        // The parameter to it
{
  WORD32 *temp = (WORD32 *) ioctl_param;
  CryptoDev *cDev = file->private_data;

  // Switch according to the ioctl called
  switch (ioctl_num)
    {
    case CRYPTO_DEC_USE_COUNT:
      PDEBUG ("Decrement Use Count\n");
      MOD_DEC_USE_COUNT;
      break;
    case CRYPTO_RESET:
      PDEBUG ("Reset\n");
      if (usingHardware == 1)
        pldDeviceReset (cDev->dev);
      break;
    case CRYPTO_ENC_DEC:
      cDev->encDec = ioctl_param;       // 0 pass through, 1 encipher, 2 decipher
      PDEBUG ("Encdec %d", cDev->encDec);
      if (usingHardware == 1)
        {
          pldSetEncipherMode (cDev);
        }
      break;
    case CRYPTO_ECB_CBC:
      if (usingHardware == 1)
        {
          pldSetStreamMode (cDev, ioctl_param);
        }
      break;
    case CRYPTO_SET_IV:
      PDEBUG ("Set IV\n");
      // Receive a pointer to a message (in user space)
      // and set that to be the device's message.
      // Get the parameter given to ioctl by the process
#ifdef DEBUG
      cDev->softwareKeys.initialisationVector[0] = temp[0];
      cDev->softwareKeys.initialisationVector[1] = temp[1];
      cDev->softwareKeys.initialisationVector[2] = temp[2];
```

```
#endif
      // see /usr/src/linux/drivers/char/random.c
      get_random_bytes ((void *) cDev->softwareKeys.initialisationVector, 12);
      // new IV's
      cDev->encUsedIV = cDev->decUsedIV = 0;
      break;
    case CRYPTO_SET_KEY:
      PDEBUG ("Set Key\n");
      cDev->encDec = 2;         // Decrypting.
      // Disables IV being xored with Key
      cDev->encUsedIV = cDev->decUsedIV = 1;
      if (usingHardware == 1)
        {
          // This is very dangerous... ioctl_param is a 64 bit value
          pldPrepareForWritingKey (cDev->dev);
          crypto_write (file, (char *) temp, 12, 0);
        }
      if (usingHardware == 0)
        {
          // Decipher using the KAK
          swKakDecipher (&cDev->softwareKeys, temp);
          //swDecipher(&cDev->softwareKeys, temp);
          // temp is now our new key :)
          PDEBUG ("Keys going to be\n %08lx %08lx %08lx\n", temp[0], temp[1],
                  temp[2]);
          cDev->softwareKeys.encipherKey[0] =
            cDev->softwareKeys.decipherKey[0] = temp[0];
          cDev->softwareKeys.encipherKey[1] =
            cDev->softwareKeys.decipherKey[1] = temp[1];
          cDev->softwareKeys.encipherKey[2] =
            cDev->softwareKeys.decipherKey[2] = temp[2];
          // prepare new decryption key
          swRoundInit (&cDev->softwareKeys);
          // we're done
        }
      cDev->encUsedIV = cDev->decUsedIV = 0;
      cDev->encDec = 0;
      PDEBUG ("Key Changed ioctl\n");
      break;
    case CRYPTO_GET_IV:
      {
        // Return the current IV
        // process - the parameter we got is a pointer,
        // fill it.
        char Message[97];
        u32 *tempPtr = (u32 *) cDev->softwareKeys.encipherKey;
        int i = 0;

        for (i = 0; i < 2; i++)
          tempPtr[i] = cDev->softwareKeys.encipherKey[i];

        Message[96] = '\0';
        if (copy_to_user ((char *) ioctl_param, Message, 97))
          {
            PINFO ("User Space Buffer is invalid %s(%d)", __FILE__, __LINE__);
            return -EFAULT;
          }
      }
      break;
    default:
      // It makes sense to return -EINVAL here but conformance to Posix
      // requires -ENOTTY (Writing Linux Device Drivers pp 99-100)
      return -ENOTTY;
    }
  return SUCCESS;
}

// Module Declarations **************************
struct file_operations crypto_Fops = {
  crypto_lseek,                // seek
  crypto_read, crypto_write, NULL,      // readdir
  NULL,                        // select
  crypto_ioctl,                // ioctl
  NULL,                        // mmap
  crypto_open,
  NULL,                        // flush
  crypto_release               // a.k.a. close
};

// Initialize the module - Register the character device
int
init_module ()
{
  int ret_val = 0;

  if (AllocateDmaBuffers ())
    {
      PINFO ("Failed to allocate memory %s(%d)\n", __FILE__, __LINE__);
      ret_val++;               // Failure
    }

  // OK now we either have found a board or we're using Software emulation.
  // We can override the driver and tell it to use software emulation.
  /// this is done by "insmod module.o mode=1"
  if (mode == 0 && ret_val == 0)
    {
      // Can only fail with PCI errors.
      if (!pldInit (&cryptoDevice))
        usingHardware = 1;
      else
        PINFO ("Crypto Board not found, using software emulation\n");
    }
  else
      swInit (&cryptoDevice.softwareKeys);
  // Register the character device (atleast try)

  ret_val =
    ret_val + module_register_chrdev (MAJOR_NUM, DEVICE_NAME, &crypto_Fops);
  // Success if proc_register[_dynamic] is a success,
  // failure otherwise
  ret_val = ret_val + proc_register (&proc_root, &Our_Proc_File);

  // Negative values signify an error
  if (ret_val < 0)
    {
      PINFO ("%s failed with %d\n",
             "Sorry, registering the character device ", ret_val);
      return 1;
    }

  PINFO ("The major device number is %d.\n", MAJOR_NUM);
  PINFO ("To talk to the driver,\n");
  PINFO ("'mknod %s c %d 0' will create the device\n", DEVICE_FILE_NAME,
         MAJOR_NUM);

#ifdef NOSYM
  EXPORT_NO_SYMBOLS;
#endif
  return 0;
}

// Cleanup - unregister the appropriate file from /proc
void
cleanup_module ()
{
  int ret;

  proc_unregister (&proc_root, Our_Proc_File.low_ino);
  // Unregister the device
  ret = module_unregister_chrdev (MAJOR_NUM, DEVICE_NAME);

  FreeDmaBuffers ();
  // If there's an error, report it
  if (ret < 0)
    PINFO ("Error in module_unregister_chrdev: %d\n", ret);
}
```

C.8 hardware.h

```
// All PLDA specific items in here.
// $Header: //j0n/module/hardware.h#10 $
// John Ronan, June 2000

#ifndef __hardware_h
#define __hardware_h
#include <linux/pci.h>
#include "driver.h"             // needed for Crypto_Device struct

// PLDA BOARD
#define PLDA_MAX_DEV        0x01
#define PLDA_VENDOR_ID_PLD  0x1556
#define PLDA_CRYPTO_ID      0x6a71      // jr in hex.
```

```
#define PLDA_FIFO_SIZE      1023       //biggest multiple of 3 32-bit words that will fit in 1024 words
// Used to test large buffer sizes
//#define PLDA_FIFO_SIZE      8184      //biggest multiple of 3 32-bit words that will fit in 1024 words
#define PLDA_BUFFER_SIZE    (PLDA_FIFO_SIZE * 0x20)/8   // size is in bytes
#define PLDA_SOFT_BUF_SIZE  0xc // 2 ^ 12 4K, Used to figure out 'order'
                                // parameter for dma_pages
// Used to test large buffer sizes
//#define PLDA_SOFT_BUF_SIZE  0xf       // 2 ^ 12 4K, Used to figure out 'order'
                                        // parameter for dma_pages

// Registers
#define IOR_REG         0x40
#define TAR0            0x48
#define DCR0            0x4c
#define TAR1            0x50
#define DCR1            0x54
#define TAR2            0x58
#define DCR2            0x5c
#define TAR3            0x60
#define DCR3            0x64

//PCI Commands
#define PLDA_IACK           0x0
#define PLDA_SPAECIAL       0x1
#define PLDA_IO_READ        0x2
#define PLDA_IO_WRITE       0x3
#define PLDA_MEM_READ       0x6
#define PLDA_MEM_WRITE      0x7
#define PLDA_CONFIG_READ    0xA
#define PLDA_CONFIG_WRITE   0xB
#define PLDA_MEM_READ_MULT  0xC
#define PLDA_DUAL_ADDR_CYC  0XD
#define PLDA_MEM_READ_LINE  0xE
#define PLDA_MEM_WRI        0xF

// DMA
#define PLDA_DMA_REQUEST    0x9 // PCI Core USer Guide Page 39
#define PLDA_DMA_FINISHED   0x2
#define PLDA_DMA_INPROGRESS 0x1

//
#define PLDA_INTERRUPT_RESET     0x1    // interrupt status/reset   (bit 0)
#define PLDA_INTERRUPT_STATUS    0x1
#define PLDA_INTERRUPT_ENABLE    0x2    // interrupt enable         (bit 1)
#define PLDA_INTERRUPT_DISABLE   0xFFFFFFFD     // interrupt disable
#define PLDA_ECB_MODE            0xFFFFFFFB     // Zero Bit for ECB Mode    (bit 2)
#define PLDA_CBC_MODE            0x4    // Set Bit for CBC Mode
#define PLDA_ENCIPHER            0xFFFFFFF7     // Zero Bit for Encipher    (bit 3)
#define PLDA_DECIPHER            0x8    // Set bit for Decipher
#define PLDA_KEY                 0xFFFFFFEF     // Zero Bit for Key         (bit 4)
#define PLDA_DATA                0x10   // Set bit for Data 0x10

#define PLDA_CBC_RESET_HI        0x20   // Set bit to zero to reset register
#define PLDA_CBC_RESET_LOW       0xFFFFFFDF     // Set bit to zero to reset register
#define PLDA_RESET               0xFFFFFFBF     // Reset the backend..
```

```
#define PLDA_FIFO_MASK            0x3FF
#define PLDA_IFIFO_POSN           20
#define PLDA_OFIFO_POSN           8
// bits 7 is unused
// bits 18-8 are the output fifo size
// bit 19 is out_fifo_full
// bits 30-20 are the input fifo size
// bit 31 is input fifo full

int pldInit (struct Crypto_Dev *const cDev);
ssize_t pldWrite (struct file *file, const char *const buffer, size_t length,
                  loff_t * offset);
ssize_t pldRead (struct file *file, char *const buffer, size_t length,
                 loff_t * offset);
void pldDmaFromHardware (struct Crypto_Dev *const cDev);
void pldDmaToHardware (struct Crypto_Dev *const cDev);

int pldRequestInterrupt (struct Crypto_Dev *const cDev);
void pldFreeInterrupt (struct Crypto_Dev *const cDev);
void pldInterrupt (int irq, void *dev_id, struct pt_regs *regs);
void pldBhInterrupt (void *data);

int pldIsDeviceReadyForData (const struct Crypto_Dev *const cDev);
int pldIsInputBufferEmpty (const struct Crypto_Dev *const cDev);
int pldIsInputFifoEmpty (const struct Crypto_Dev *const cDev);
int pldIsOutputFifoEmpty (struct Crypto_Dev *const cDev);
int pldIsOutputBufferEmpty (const struct Crypto_Dev *const cDev);

void pldIsDma0Busy (struct pci_dev *const pciDev, int clearFlag);
void pldIsDma1Busy (struct pci_dev *const pciDev, int clearFlag);

int pldGetOutputFifoLength (struct pci_dev *const pciDev);
int pldGetInputFifoLength (struct pci_dev *const pciDev);

void pldCbcModeReset (struct pci_dev *const pciDev);
void pldDeviceReset (struct pci_dev *const pciDev);

void pldSetEncipherMode (const struct Crypto_Dev *const cDev);
void pldSetStreamMode (const struct Crypto_Dev *const cDev, const int mode);
void pldPrepareForWritingKey (struct pci_dev *const pciDev);

#endif // __hardware_h
```

C.9 hardware.c

```
// $Header: //j0n/module/hardware.c#11 $
// John Ronan, June 2000

#include <linux/mm.h>
#include <linux/interrupt.h>
#include <asm/io.h>             // needed for virt_to_bus()
#include <asm/uaccess.h>        // copy_from/to_user
```

```
#include "hardware.h"
#include "common.h"
// Note all pci bus access stuff is  little endian so bring in the converstion fucntions.

rwlock_t cDev_lock = RW_LOCK_UNLOCKED;

// Allocates ram for the buffers
// Sets up the structure for the bottom half handler.
// Takes in a pointer to the Device Specific Data Stricture
// and returns 0 if succesful.
int
pldInit (CryptoDev * const cDev)
{
  int retVal = 0;
  if (!pci_present ())
    {
      PINFO ("No PCI Bios Present\n");
      retVal = 1;
    }
  else
    {
      cDev->dev =
        pci_find_device (PLDA_VENDOR_ID_PLD, PLDA_CRYPTO_ID, cDev->dev);
      if (cDev->dev)
        {
          int i = 0;
          PINFO ("Found Crypto Device Bus %i, Function %02i\n",
                 cDev->dev->bus->number, cDev->dev->devfn);
          PINFO ("VendorID %x, DeviceID %x, IRQ %d\n", cDev->dev->vendor,
                 cDev->dev->device, cDev->dev->irq);
          pldDeviceReset (cDev->dev);
          // Fill the task structure, used for the bottom half handler
          cDev->crypto_queue.routine = pldBhInterrupt;
          cDev->crypto_queue.data = cDev;

          for (i = 0; i < 3; i++)
            cDev->softwareKeys.initialisationVector[i] = 0;

          // Success
          retVal = 0;
        }
      else
        {
          // Failure
          retVal = 1;
        }
    }
  return retVal;
}

void
pldDeviceReset (struct pci_dev *const pciDev)
{
  u32 ior;
  PDEBUG ("Resetting Back end\n");
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  ior = ior & PLDA_RESET;
  pci_write_config_dword (pciDev, IOR_REG, ior);
}

ssize_t
pldRead (struct file *file, char *const buffer,
         size_t length, loff_t * offset)
{
  CryptoDev *cDev = file->private_data;
  unsigned long flags;

  PDEBUG ("Hardware Read\n");
  write_lock_irqsave (&cDev_lock, flags);
  length = cDev->outputBufferLength;
  // This just checs that there is data for us to read.
  if (cDev->outputBufferLength != 0)
    {
      // if there is no data left in the output fifo
      // then we XOR the last 12 bytes of the buffer with the
      // IV
      if (cDev->decUsedIV == 0 && cDev->encDec == 2)
        {

          int i = 0;
          WORD32 *tempBuf = cDev->oFifoBuf;
          PDEBUG ("Removing IV\n");
          for (i = 0; i < 3; i++)
            {
              tempBuf[i] =
                tempBuf[i] ^ cDev->softwareKeys.initialisationVector[i];
            }

          cDev->decUsedIV = 1;
        }
      if (copy_to_user (buffer, cDev->oFifoBuf, cDev->outputBufferLength))
        {
          write_unlock_irqrestore (&cDev_lock, flags);
          return -EFAULT;
        }

      cDev->outputBufferLength = 0;
      if (cDev->dataPending)
        {
          cDev->dataPending = 0;
          pldDmaFromHardware (cDev);
        }

    }
  write_unlock_irqrestore (&cDev_lock, flags);
  wake_up_interruptible (&cDev->writeq);
  return length;
}

// prototype looks like write(). If there is no data in the kernel space buffer copy to kernel space
// if there is no data in the input buffer copy to the PLDA device.
ssize_t
pldWrite (struct file * file, const char *const buffer,
          size_t length, loff_t * offset)
{
  CryptoDev *cDev = file->private_data;
  unsigned long flags;

  PDEBUG ("Hardware Write\n");
  if (cDev->inputBufferLength == 0 && pldIsInputFifoEmpty (cDev))
    {

      write_lock_irqsave (&cDev->lock, flags);
      if (length > PLDA_BUFFER_SIZE)
        length = PLDA_BUFFER_SIZE;
      if (copy_from_user (cDev->iFifoBuf, buffer, length))
        return -EFAULT;
      // If we've just begun an encryption then we send the IV to the input fifo
      PDEBUG ("Before if, encUsedIV = %d, %d\n", cDev->encUsedIV, length);
      if (cDev->encUsedIV == 0 && cDev->encDec == 1)
        {
          int i = 0;
          WORD32 *tempBuf = cDev->iFifoBuf;
          PDEBUG ("Adding IV\n");
          cDev->encUsedIV = 1;
          for (i = 0; i < 3; i++)
            {
              tempBuf[i] =
                tempBuf[i] ^ cDev->softwareKeys.initialisationVector[i];
            }
        }

      cDev->inputBufferLength = length;
      pldDmaToHardware (cDev);
      cDev->inputBufferLength = 0;      // clear to write again
      write_unlock_irqrestore (&cDev->lock, flags);
    }
  PDEBUG ("end_hardware_write\n");
  // Return how much we actually wrote..
  return length;
}

// Transfers data to PLDA device.
// No return value. Should I change this?
void
pldDmaToHardware (struct Crypto_Dev *const cDev)
{
  u32 addr;
  u32 val;
  pldIsDma0Busy (cDev->dev, 0);
  // If we get here, there's room in the Fifo
  if (cDev->inputBufferLength == 0)
    {
      PDEBUG ("Cannot copy Input data to copy\n");
      return;
    }

  if (cDev->inputBufferLength >
      (PLDA_BUFFER_SIZE - pldGetInputFifoLength (cDev->dev)))
    {
      PDEBUG ("Not enough Room %d \n", pldGetInputFifoLength (cDev->dev));
      return;
    }

  // Get the physical address of memory for our dma transfer
  addr = virt_to_bus (cDev->iFifoBuf);
  // construct the dma control word from page 39 of PLDA PCI Core users Guide.
  val =
    ((cDev->inputBufferLength /
      4) << 8) | (PLDA_MEM_READ << 4) | PLDA_DMA_REQUEST;

  if (pci_write_config_dword (cDev->dev, TAR0, addr))
    {
      PINFO ("Error writing to TAR0 %s(%d)\n", __FILE__, __LINE__);
      return;
    }

  // check all return values
  if (pci_write_config_dword (cDev->dev, DCR0, val))
    {
      PINFO ("Error writing to DCR0 %s(%d)", __FILE__, __LINE__);
      return;
    }
    pldIsDma0Busy (cDev->dev, 1);
  return;
}



// If there is data in the Fifo take it out. All of it.
void
pldDmaFromHardware (struct Crypto_Dev *const cDev)
{
  u32 addr = 0, val = 0, size = 0;
  PDEBUG ("dma_from_hardware\n");
  // If there is data in the kernel space buffer or the Fifo is empty we return
  if (cDev->outputBufferLength != 0)
    {
      PDEBUG ("buffer full, Size is %d\n", cDev->outputBufferLength);
      cDev->dataPending = 1;
      return;
    }

  pldIsDma1Busy (cDev->dev, 0);
  if ((size = pldGetOutputFifoLength (cDev->dev)) == 0)
    {
      PDEBUG ("No data on board");
      return;
    }
  // PLDA PCI CORE Users guide page 39
  addr = virt_to_bus (cDev->oFifoBuf);
  val = (size << 8) | (PLDA_MEM_WRITE << 4) | PLDA_DMA_REQUEST;
  // write to Tar register
  // Checkma for ongoing transaction
  if (pci_write_config_dword (cDev->dev, TAR1, addr))
    {
      PINFO ("Error writing to TAR1 %s(%d)\n", __FILE__, __LINE__);
      return;
    }
  if (pci_write_config_dword (cDev->dev, DCR1, val))
    {
      PINFO ("Error writing to DCR0 %s(%d)\n", __FILE__, __LINE__);
      return;
    }

  pldIsDma1Busy (cDev->dev, 1);
  cDev->outputBufferLength = size * 4;
  // No need to wake reader.. done afterwards.
  return;
}

int
pldRequestInterrupt (struct Crypto_Dev *const cDev)
{
  u32 ior;
  // if return value is greater than 0 then we've an error.
  //PDEBUG("1st devid %p\n",cDev);
  if (request_irq
      (cDev->dev->irq, pldInterrupt,
       SA_INTERRUPT | SA_SHIRQ | SA_SAMPLE_RANDOM, "crypto", cDev))
    {
      PINFO ("Couldn't grab interrupt %s(%d)", __FILE__, __LINE__);
       return 1;
    }
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  ior = ior | PLDA_INTERRUPT_ENABLE;
  pci_write_config_dword (cDev->dev, IOR_REG, ior);

  return 0;
}

void
pldFreeInterrupt (struct Crypto_Dev *const cDev)
{
  u32 ior;
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  ior = ior | (PLDA_INTERRUPT_RESET & PLDA_INTERRUPT_DISABLE);
  pci_write_config_dword (cDev->dev, IOR_REG, ior);
  free_irq (cDev->dev->irq, cDev);
  PDEBUG ("IRQ, Should now be free %d\n", cDev->dev->irq);
}

// We can count interrupts with this... delete later.
static atomic_t hardwareint;
```

```
void
pldInterrupt (int irq, void *dev_id, struct pt_regs *regs)
{
  u32 ior;
  CryptoDev *cDev = dev_id;
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  // Make sure the interrupt belings to us.
  if (!(ior & PLDA_INTERRUPT_STATUS))
    return;
  //Disable and reset the interrupt
  ior = ior & (PLDA_INTERRUPT_DISABLE | PLDA_INTERRUPT_RESET);
  pci_write_config_dword (cDev->dev, IOR_REG, ior);
  // schedule the bottom half handler to run and do the work for us.
  queue_task (&cDev->crypto_queue, &tq_immediate);
  mark_bh (IMMEDIATE_BH);
  atomic_inc (&hardwareint);
  return;
}

void
pldBhInterrupt (void *data)
{
  CryptoDev *cDev = data;
  u32 ior;
  unsigned long flags;
  write_lock_irqsave (&cDev_lock, flags);
  pldDmaFromHardware (cDev);
  pldDmaToHardware (cDev);
  write_unlock_irqrestore (&cDev_lock, flags);
  wake_up_interruptible (&cDev->writeq);
  wake_up_interruptible (&cDev->readq);
  // Re-enable interrupts
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  ior = ior | PLDA_INTERRUPT_ENABLE;
  pci_write_config_dword (cDev->dev, IOR_REG, ior);
  return;
}

// is the kernel input buffer empty?
int
isInputBufferEmpty (const struct Crypto_Dev *const cDev)
{
  unsigned long flags;
  read_lock_irqsave (&cDev_lock, flags);
  if (cDev->inputBufferLength == 0)
    {
      read_unlock_irqrestore (&cDev_lock, flags);
      return 1;
    }
  else
    {
      read_unlock_irqrestore (&cDev_lock, flags);
      return 0;
    }
}
```

```
// is the kernel input buffer empty?
int
pldIsOutputBufferEmpty (const struct Crypto_Dev *const cDev)
{
  unsigned long flags;
  read_lock_irqsave (&cDev_lock, flags);
  if (cDev->outputBufferLength == 0)
    {
      read_unlock_irqrestore (&cDev_lock, flags);
      return 1;
    }
  else
    {
      read_unlock_irqrestore (&cDev_lock, flags);
      return 0;
    }
}

int
pldIsDeviceReadyForData (const struct Crypto_Dev *const cDev)
{
  return pldIsInputFifoEmpty (cDev);
}

// is the kernel input buffer empty?
int
pldIsInputFifoEmpty (const struct Crypto_Dev *const cDev)
{
  u32 ior;
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  if ((ior >> PLDA_IFIFO_POSN & PLDA_FIFO_MASK) == 0)
    return 1;
  else
    return 0;
}

// return 1 if buffer is empty 0 otherwise
int
pldIsOutputFifoEmpty (struct Crypto_Dev *const cDev)
{
  u32 ior;
  pci_read_config_dword (cDev->dev, IOR_REG, &ior);
  // If the buffer is empty then return 1
  if ((ior >> PLDA_OFIFO_POSN & PLDA_FIFO_MASK) == 0)
      return 1;
  else
      return 0;
}

// waits for the dma channel to become free.
void
pldIsDma0Busy (struct pci_dev *const pciDev, int clearFlag)
{
  u32 dma_status = 0;
  if (clearFlag == 0)
    {
      do{
          pci_read_config_dword (pciDev, DCR0, &dma_status);
          if (dma_status & PLDA_DMA_INPROGRESS)
            PINFO ("Dma channel 0 is busy\n");
        }
      while ((dma_status & PLDA_DMA_INPROGRESS));
    }

  if (clearFlag == 1)
    {
      do{
          pci_read_config_dword (pciDev, DCR0, &dma_status);
        }
      while (!(dma_status & PLDA_DMA_FINISHED));
      pci_write_config_dword (pciDev, DCR0, (dma_status & PLDA_DMA_FINISHED));
    }
}

// waits for the DMA channel to become free.
void
pldIsDma1Busy (struct pci_dev *const pciDev, int clearFlag)
{

  u32 dma_status = 0;
  if (clearFlag == 0)
    {
      do{
          pci_read_config_dword (pciDev, DCR1, &dma_status);
          if (dma_status & PLDA_DMA_INPROGRESS)
            PINFO ("Dma channel 1 is busy\n");
        }
      while ((dma_status & PLDA_DMA_INPROGRESS));
    }

  if (clearFlag == 1)
    {
      pci_read_config_dword (pciDev, DCR1, &dma_status);
      do{
          pci_read_config_dword (pciDev, DCR1, &dma_status);
        }
      while (!(dma_status & PLDA_DMA_FINISHED));
      pci_write_config_dword (pciDev, DCR1, (dma_status & PLDA_DMA_FINISHED));
    }
  return;
}

// Returns the length of the output Fifo
int
pldGetOutputFifoLength (struct pci_dev *const pciDev)
{
  u32 ior;
  int length;
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  length = (ior >> PLDA_OFIFO_POSN & PLDA_FIFO_MASK);
  return length;
}

// Returns the length of the input Fifo
int
pldGetInputFifoLength (struct pci_dev *const pciDev)
{
  u32 ior;
  int length;
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  length = (ior >> PLDA_IFIFO_POSN & PLDA_FIFO_MASK);
  return length * 4;            // bytes
}

void
pldCbcModeReset (struct pci_dev *const pciDev)
{
  u32 ior;
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  ior = ior | PLDA_CBC_RESET_HI;
  pci_write_config_dword (pciDev, IOR_REG, ior);
  return;
}

void
pldSetEncipherMode (const struct Crypto_Dev *const cDev)
{
  u32 ior = 0;
  if (cDev->encDec == 1)        // Encrypting
    {
      pci_read_config_dword (cDev->dev, IOR_REG, &ior);
      if (!(ior & PLDA_DECIPHER))
        {
          PDEBUG ("Previously encrypting\n");
          //If we wer previously enciphering
          // Toggle the PLDA_CBC_RESET line
          pldCbcModeReset (cDev->dev);
        }
      // Check if the last operation was also an encrypt
      // If so then we toggle the cbc_reset flag
      // Set the board to encipher
      ior = (ior & PLDA_ENCIPHER) | PLDA_DATA;
      pci_write_config_dword (cDev->dev, IOR_REG, ior);
    }

  if (cDev->encDec == 2)        // Decrypting
    {
      pci_read_config_dword (cDev->dev, IOR_REG, &ior);
      // Check if the last operation was also a decrypt
      // if so then we toggle the cbc reset
      if (ior & PLDA_DECIPHER)
        {
          pldCbcModeReset (cDev->dev);
        }
      ior = ior | (PLDA_DECIPHER | PLDA_DATA);
      pci_write_config_dword (cDev->dev, IOR_REG, ior);
    }
}

void
pldSetStreamMode (const struct Crypto_Dev *const cDev, const int mode)
{
  u32 ior = 0;
  if (mode == 0)                // ECB mode
    {
      pci_read_config_dword (cDev->dev, IOR_REG, &ior);
      ior = ior & PLDA_ECB_MODE;
      PDEBUG ("Ioctl - ECB_MODE\n");
      pci_write_config_dword (cDev->dev, IOR_REG, ior);
    }
  if (mode == 1)                // CBC mode
    {
      pci_read_config_dword (cDev->dev, IOR_REG, &ior);
      ior = ior | PLDA_CBC_MODE;
      pci_write_config_dword (cDev->dev, IOR_REG, ior);
      PDEBUG ("Ioctl - CBC_MODE\n");
    }
}

void
pldPrepareForWritingKey (struct pci_dev *const pciDev)
{
  u32 ior = 0;
  pldDeviceReset (pciDev);
  // Key, ECB Mode, deciphering.
  pci_read_config_dword (pciDev, IOR_REG, &ior);
  ior = (ior & PLDA_KEY & PLDA_ECB_MODE) | PLDA_DECIPHER;
  pci_write_config_dword (pciDev, IOR_REG, ior);
}
```

C.10 mode.c

```
// $Header: //j0n/module/mode.c#5 $
// John Ronan, June 2000

#include <fcntl.h>              /* open */
#include <unistd.h>             /* exit */
#include <sys/ioctl.h>          /* ioctl */
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <errno.h>
#include <sys/stat.h>
#include "driver.h"
```

```
#define BUF 12
void
ioctl_encDec (int file_desc, int encDec)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_ENC_DEC, encDec);
  if (ret_val < 0)
    {
      printf ("ioctl_encDec failed:%d\n", ret_val);
      exit (-1);
    }
}
void
ioctl_ecbCbc (int file_desc, int ecbcbc)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_ECB_CBC, ecbcbc);
  if (ret_val < 0)
    {
      printf ("ioctl_ecbcbc failed:%d\n", ret_val);
      exit (-1);
    }
}

void
ioctl_set_IV (int file_desc, WORD32 * initialisationVector)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_SET_IV, initialisationVector);
  if (ret_val < 0)
    {
      printf ("ioctl_Set_IV:%d\n", ret_val);
      exit (-1);
    }
}

void
ioctl_set_key (int file_desc, WORD32 * initialisationVector)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_SET_KEY, initialisationVector);
  if (ret_val < 0)
    {
      printf ("ioctl_Set_Key:%d\n", ret_val);
      exit (-1);
    }
}

void
ioctl_reset (int file_desc)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_RESET);
  if (ret_val < 0)
```

```
    {
      printf ("reset failed %d", ret_val);
      exit (1);
    }
}

void
ioctl_decUseCount (int file_desc)
{
  int ret_val;
  ret_val = ioctl (file_desc, CRYPTO_DEC_USE_COUNT);
  if (ret_val < 0)
    {
      printf ("ioctl_hardreset failed:%d\n", ret_val);
      exit (-1);
    }
}
void
usage (char **argv)
{
  printf
    ("mode <0|1|2|3|5>\n\t0 = Reset\n\t1 = Encipher\n\t2 = Decipher
     \n\t3 = ECB-Mode\n\t4 = CBC-Mode\n\t5 = Set Key\n\t6 = Set IV");
#ifdef CRYPTO_DEBUG
  printf ("\n\t9 = DEC_USE_COUNT");
#endif
  printf ("\n\tOR\npld_read <name> <length>\n");
  printf ("pld_write <name>\n");
}

int
main (int argc, char **argv)
{

  int file_desc;
  int mode = 0;
  int infp = 0;
  int outfp = 0;
  int size = 0;
  unsigned char *buf = 0;
  unsigned char *tempBuf = 0;
  int len;
  int result;
  WORD32 iv[12];
  WORD32 k1[12];

  k1[0] = 0xf3d7260F;
  k1[1] = 0x3ca52052;
  k1[2] = 0x12ae5a79;

  iv[0] = 0x00000001;
  iv[1] = 0x00000001;
  iv[2] = 0x00000001;

// Open the device
```

```
    file_desc = open (DEVICE_FILE_NAME, O_RDWR);

    if (file_desc < 0)
      {
        printf ("Can't open device file: %s\n", DEVICE_FILE_NAME);
        exit (-1);
      }

    if (strncmp (argv[0], "mode", 5) == 0 && argc == 1)
      {
        printf ("argv is %s\n", argv[0]);
        usage (argv);

      }

    if (strncmp (argv[0], "mode", 5) == 0 && argc == 2)
      {

        mode = *argv[1] - 48;
        switch (mode)
          {
          case 0:
            printf ("Reset\n");
            ioctl_reset (file_desc);
            break;
          case 1:
            printf ("Encrypt\n");
            ioctl_encDec (file_desc, 1);
            break;
          case 2:
            printf ("Decrypt\n");
            ioctl_encDec (file_desc, 2);
            break;
          case 3:
            printf ("ECB\n");
            ioctl_ecbCbc (file_desc, 0);
            break;
          case 4:
            printf ("CBC\n");
            ioctl_ecbCbc (file_desc, 1);
            break;
          case 5:
            printf ("Set Key\n");
            ioctl_set_key (file_desc, k1);
            break;
          case 6:
            printf ("Set Key\n");
            ioctl_set_IV (file_desc, iv);
            break;

  #ifdef CRYPTO_DEBUG
          case 9:
            printf ("Decrementing MOD_USE_COUNT");
            ioctl_decUseCount (file_desc);
            break;
```

```
#endif
        default:
          printf ("Default\n");
        }
    }

  if (strncmp (argv[0], "pld_read", 9) == 0 && argc == 3)
    {
      printf ("pld_read: ");
      // Open device for reading.
      if (
          (outfp =
           open (argv[1], O_CREAT | O_EXCL | O_WRONLY,
                 S_IRUSR | S_IWUSR)) < 0)
        {
          printf ("Cannot open %s\n", argv[1]);
          exit (1);
        }

      printf ("size %d ", atoi (argv[2]));
      len = atoi (argv[2]);
      buf = malloc (sizeof (char *) * len);

      if (!buf)
        {
          printf ("Couldn't alloc memory\n");
          exit (1);
        }

      tempBuf = buf;
      // read the data from the device
      while (len > 0)
        {
          result = read (file_desc, tempBuf, len);
          if (result < 0)
            {
              if (errno == EINTR)
                {
                  printf ("EINTR\n");
                }
              if (errno != EINTR)
                {
                  printf ("System Callinterrupted reading file (%s)(%d)",
                          strerror (errno), __LINE__);
                  exit (1);
                }
            }
          else
            {
              tempBuf += result;
              len -= result;
            }
        }

      //write it out to drive
      len = atoi (argv[2]);
      tempBuf = buf;
      // write the data to the file
      while (len > 0)
        {
          result = write (outfp, tempBuf, len);
          if (result < 0)
            {
              if (errno == EINTR)
                {
                  printf ("EINTR\n");
                }
              if (errno != EINTR)
                {
                  printf ("System Callinterrupted reading file (%s)(%d)",
                          strerror (errno), __LINE__);

                  exit (1);
                }
            }
          else
            {
              tempBuf += result;
              len -= result;
            }
        }
      printf ("%s closing\n", argv[0]);
      close (infp);
      close (outfp);
      free (buf);
    }

  if (strncmp (argv[0], "pld_write", 10) == 0 && argc == 2)
    {
      printf ("pld_write: ");
      // Open device for reading.
      if ((infp = open (argv[1], O_RDONLY)) < 0)
        {
          printf ("Cannot open %s", argv[1]);
          exit (1);
        }

      len = lseek (infp, 0, SEEK_END);
      size = len;
      lseek (infp, 0, SEEK_SET);
      printf ("size %d ", len);
      buf = malloc (sizeof (char *) * len);

      if (!buf)
        {
          printf ("Couldn't alloc memory\n");
          exit (1);
        }

      tempBuf = buf;
    // read the data from the file
    while (len > 0)
      {
        result = read (infp, tempBuf, len);
        if (result < 0)
          {
            if (errno == EINTR)
              {
                printf ("EINTR\n");
              }
            if (errno != EINTR)
              {
                printf ("System Callinterrupted reading file (%s)(%d)",
                        strerror (errno), __LINE__);
                exit (1);
              }
          }
        else
          {
            tempBuf += result;
            len -= result;
          }
      }

    //write it out to drive
    len = size;
    tempBuf = buf;
    // write the data to the device
    while (len > 0)
      {
        result = write (file_desc, tempBuf, len);
        if (result < 0)
          {
            if (errno == EINTR)
              {
                printf ("EINTR\n");
              }
            if (errno != EINTR)
              {
                printf ("System Callinterrupted reading file (%s)\n",
                        strerror (errno));
                exit (1);
              }
          }
        else
          {
            tempBuf += result;
            len -= result;
          }
      }
    printf ("%s closing\n", argv[0]);
    close (infp);
    close (outfp);
    free (buf);
  }
  return 0;
}
```

C.11 Makefile

```
# Makefile for a basic kernel module
# $Header: //j0n/module/Makefile#6 $
# John Ronan June 2000

CC=gcc
LINK= gcc -g
MODCLFAGS =
LDFLAGS =

all: module.o mode
ifeq ($HOSTTYPE,i386)
  MODCFLAGS = -D__KERNEL__ -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -pipe \
              -DMODULE  -DLINUX -DNOSYM #-DCRYPTO_DEBUG
  LDFLAGS = elf_i386
else
  MODCFLAGS = -D__KERNEL__ -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -pipe \
              -mno-fp-regs -ffixed-8 -mcpu=ev5 -Wa,-mev6 -DMODULE -DNOSYM #-DCRYPTO_DEBUG
  LDFLAGS = elf64alpha
endif

module.o:       driver.o 3way.o software.o hardware.o
                ld -m $(LDFLAGS) -r -o module.o driver.o 3way.o software.o hardware.o

driver.o:       driver.c driver.h hardware.h software.h common.h /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c driver.c

hardware.o:     hardware.c hardware.h driver.h common.h /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c hardware.c

software.o:     software.c software.h 3way.h software.h driver.h /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c software.c

3way.o:         3way.c 3way.h common.h /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c 3way.c

mode:           mode.c driver.h
                $(CC) $(MODCFLAGS) mode.c -o mode
clean:
        @-rm -f *.o core module.o mode >& /dev/null
```

C.12 testenc.sh

```
#!/bin/sh
# These 4 commands
mode 5 # set key
mode 6 # set IV
mode 4 # cbc mode
mode 1 # encipher
pld_write $1 &> /dev/null&
pld_read $1enc $1
```

C.13 testdec.sh

```
#!/bin/sh
mode 5 # set key
mode 6 # set IV
mode 4 # cbc mode
mode 2 # decipher
pld_write $1enc &> /dev/null &
pld_read $1dec $1
# $1 is the file name and length
```

Bibliography

[3co] 3com. http://www.3com.com/products/nics/3cr990.html.

[Ada94] C.M Adams. Simple and Effective Key Scheduling for Symmetric Ciphers. In *Workshop on Selected Areas in Cryptography — Workshop Record*, pages 129–133, Kingston, Ontario, 5-6 May 1994.

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[Appa] PLD Applications. PCI Core Getting Started Guide. http://www.plda.com/download.htm.

[Appb] PLD Applications. PCI10K-PROD B User's Guide. http://www.plda.com/download.htm.

[AT93] C.M Adams and S.E. Tavares. Designing S-Boxes for Ciphers Resistant to Differential Cryptanalysis. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, pages 181–190, Rome, Italy, 15-16 Feb 1993.

[Bac86] Maurice J. Bach. *The Design of The UNIX Operating System.* Prentice Hall, INC., Englewood Cliffs, New Jersey 07632, 1986.

[BDR$^+$96] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Weiner. Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security. http://www.counterpane.com/keylength.html, January 1996.

[Ben96] Randolf Bentson. *Inside Linux: A Look at Operating Systems Development.* Specialized Systems Consultants, 1996.

[BFRV92] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, May 1992.

[BKPS93] L. Brown, M. Kwan, J. Pieprzyk, and J. Seberry. Improving Resistance to Differential Cryptanalysis and the Redesign of LOKI. In *Advances in Cryptology ASIACRYPT '91 Proceedings*, pages 36–50. Springer-Verlag, 1993.

[BR00] Stephen Brown and Johnathan Rose. Architecture of FPGAs and CPLDs: A Tutorial. http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.html, February 2000.

[CER97] CERT/CC. Cert advisory ca-97.21 (sgi buffer overflow vulnerabilities). http://www.cert.org/advisories/CA-97.21.sgi_buffer_overflow.html, July 1997.

[CER99a] CERT/CC. Cert advisory ca-97.05 (mime conversion buffer overflow in sendmail version 8.8.3 and 8.8.4). http://www.cert.org/advisories/CA.97.05.sendmail.html, January 1999.

[CER99b] CERT/CC. Cert advisory ca-99-07 (iis buffer overflow). http://www.cert.org/advisories/CA-97-07-IIS-Buffer-Overflow.html, June 1999.

[CER99c] CERT/CC. Cert advisory ca-99-13 (multiple vulnerabilities in wu-ftpd). http://www.cert.org/advisories/CA-97-13-wuftpd.html, October 1999.

[CF98] Crispin Cowan and Castor Fu. Death, Taxes and Imperfect Software: Surviving the Inevitable. In *Proceedings of the ACM New Security Paradigms Workshop '98*, September 1998.

[chp] IBM Cryptographics cards home page. http://www.ibm.com/security/cryptocards/.

[CHW00] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, April 2000.

[Ci] Chrysalis-its. http://www.chrysalis-its.com/.

[Coh97] Fredrick B. Cohen. *Protection and Security on the Information Super-Highway.* Dunno Press, 1997.

[Cora] Altera Corporation. Introduction to altera. http://www.altera.com/document/an/an006.pdf.

[Corb] Altera Corporation. PCI Bus Applications In Altera Devices. http://www.altera.com/document/an/an041.pdf.

[Cor95] Altera Corporation. *MAX+PLUS II AHDL.* Altera Corporation, 2610 Orchard Parkway, San Jose, CA 985134-2020, November 1995.

[Cur] Matt Curtin. Avoiding bogus encryption products: Snake Oil FAQ. http://www.interhack.net/people/cmcurtin/snake-oil-faq.html.

[Dae95] J. Daeman. *Cipher and Hash Function Design.* PhD thesis, Katholieke Universiteit Leuven, Mar 1995.

[DeH00] André DeHon. The Density Advantage of Configurable Computing. *IEEE Computer*, 33(4):41–49, April 2000.

[DGV94a] J. Daemen, R. Govaerts, and J. Vandewalle. A New Approach to Block Cipher Design. In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 18–32. Springer-Verlag, 1994.

[DGV94b] J. Daemen, R. Govarts, and J. Vandewalle. Weak Keys for IDEA. In *Advances in Cryptology CRYPTO '93 Proceedings*, pages 224–230. Springer-Verlag, 1994.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in crytography. *IEEE Transactions on Information Theory*, IT-11:644–654, November 1976.

[dil99] dildog@lopht.com. any local user can gain administrator privileges and/or take full control over the system. L0pht Security Advisory, February 1999. http://www.l0pht.com/advisories.html.

[FW99] Kevin Fenzi and Dave Wreski. Linux security howto, April 1999.

[Gee99] Dan Geer. privacy in the real world. *;login*, October 1999.

[GO91] G. Garon and R. Outerbridge. DES Watch: An Examination of the Suffiency of the Data Encryption Standard for Financial Institutions

Information Security in the 1990's. *Cryptolagia*, 15(3):177–193, July 1991.

[Gol99] Emmanuel Goldstein. Inside Cover. *2600*, 16(1), 1999.

[Goo] Richard Gooch. Linux Kernel API Changes. http://www.atnf.csiro.ar/~rgooch/linux/docs/.

[GSB$^+$00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[Gut00] Peter Gutmann. The Design of a Cryptographic Security Architecture. In *Proceedings of the 9th Usenix Security Symposium*, August 2000.

[H$^+$88] E. Hamdy et al. Dieletric-based antifuse for logic and memory ICs. *IEEE International Electron Devices Meeting Technical Digest*, pages 786–789, 1988.

[HA94] F. Hendessi and M. R. Araf. A succesful attack against the DES. *Third canadian workshop on Information Theory and Applications*, pages 78–90, May 1994.

[Hei98] Eric Heimburg. Monitoring System Events by Subclassing the Shell. *Windows Developers Journal*, 9(2):35, February 1998.

[Ito00] Naomaru Itoi. Secure Coprocessor Integration wiht Kerberos V5. In *Proceedings of the 9th Usenix Security Symposium*, August 2000.

[Kah96] David Kahn. *The CodeBreakers*. Scribner, 1230 Avenue of the Americas, New York, NY10020, 1996.

[Kal93] B.S. Kalaski. A Survey of encryption Standards. *IEEE Micro*, 13(6):74–81, December 1993.

[KR98] Brian Kernighan and Denis Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1998.

[KSW] John Kelsey, Bruce Schneier, and David Wagner. Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. http://www.cs.berkeley.edu/~daw/papers.

[Lai92] X. Lai. On the Design and Security of Block Block Ciphers. In *ETH Series in Information Provessing*, volume 1. Konstanz:Hartung-Gorre-Verlag, 1992.

[Lew91] Donald Lewine. *POSIX Programmers Guide.* O'Reilly & Associates, 1991.

[LSM$^+$98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, October 1998.

[Men98] Phunda Menta. Linux and Random Source Bleaching. *Phrack*, 1998. http://www.phrack.com/, Issue 54.

[ml99] mudge@lopht.com and lumpy. Users can de-obfuscate and retreive the hidden shell code. L0pht Security Advisory, October 1999. http://www.l0pht.com/advisories.html.

[mud99] mudge@lopht.com. Users of the tool password apraiser are unwittingly publishing nt user passwords to the internet. L0pht Security Advisory, October 1999. http://www.l0pht.com/advisories.html.

[OD95] J. Oldfield and R. Dorf. *Field-Programmable Gate Arrays.* John Wiley & Sons, New York, 1995.

[oST93] National Institute of Standards and Technology. FIPS PUB 46-2, Data Encryption Standard, 30 December 1993.

[PJ98] Meilir Page-Jones. *Practical Guide to Structured Systems Design.* Prentice Hall, INC., Englewood Cliffs, New Jersey 07632, May 1998.

[Pom] Ori Pomerantz. The Linux Kernel Module Programming Guide. http://sunsite.unc.edu/mdw/LDP/lkmpg/mpg.html.

[Ran99] Marcus J. Ranum. Selling Security: Fear leads to ... the Dark Side. *;login*, November 1999.

[RC97] Mark Russinovich and Bryce Cogswell. Windows NT System-Call Hooking. *Dr. Dobbs Journal*, page 42, January 1997.

[Rit99] Terry Ritter. Cryptography: Is staying with the Herd Really Best? *IEEE Computer*, 32(8):94–95, August 1999.

[Rit00] Terry Ritter. The truth about cryptography. *IEEE Computer*, 33(2):7–8, February 2000.

[Ros00] Greg Rose. T3 Tutorial: Cryptographic Algorithms Revealed. In *9th Usenix Seurity Symposium*, August 2000. QUALCOMM Australia.

[Rub98] Allesandro Rubini. *Linux Device Drivers.* O'Reilly & Associates, 1998.

[Sch94a] B. Schneier. Description of a New Varable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, 1994.

[Sch94b] B. Schneier. The Blowfish Encryption Algorithm. *Dr. Dobb's Journal*, 19(4):38–40, Apr 1994.

[Sch96] Bruce Schneier. *Applied Cryptography Second Edition:protocols, algorithms, and source code in C.* John Wiley & Sons, 1996.

[Sch99a] Bruce Schneier. CRYPTO-GRAM. http://www.counterpane.com, September 1999.

[Sch99b] Bruce Schneier. Security Pitfalls in Cryptography. http://www.counterpane.com/pitfalls.html, September 1999.

[Sch99c] Bruce Schneier. Why Cryptography Is Harder Than It Looks. http://www.counterpane.com/whycrypto.html, September 1999.

[Sch00] Bruce Schneier. CRYPTO-GRAM. http://www.counterpane.com, January 2000.

[Sha49] C. Shannon. Communications Theory of Secrecy Systems. *Bell Systems Technical Journal*, pages 656–715, 1949.

[sil99a] sili@lopht.com. Attackers can remotely add default route entries. L0pht Security Advisory, August 1999. http://www.l0pht.com/advisories.html.

[Sil99b] Robert D. Silverman. Exposing the Mythical MIPS Year. *IEEE Computer*, 32(8):22–26, August 1999.

[Sin99] Simon Singh. *The Code Book.* Fourth Estate Limited, 6 Salem Road, London W2 4BU, 1999.

[SM98] Richard M. Stallman and Roland McGrath. *GNU Make, Version 3.77.* Free Software Foundation, 1998.

[Smi97] Richard E. Smith. *Internet Cryptography.* Addison Wesley Longman, Inc., Massachusetts, 1997.

[Spy] Spyrus. http://www.spyrus.com/.

[SR00] Andrew M. Stean and Elanor G. Rieffel. Beyond bits: The future of quantum information processing. *IEEE Computer*, 33(1):38–45, January 2000.

[SS98] Nicko van Someren and Adi Shamir. Playing hide and seek with stored keys, 22 September 1998. Presented at Financial Cryptography 1999.

[Sta98] William Stallings. *Cryptography and network security: principles and plactice.* Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, second edition, 1998.

[Ste92] Richard Stevens. *Advanced Programming in the Unix Environment.* Addison Wesley Publishing Co., 1992.

[SW99] S. W. Smith and S. H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks, (Special Issue on Computer Network Security)*, 31:831–860, April 1999.

[Tom95] Tom Shanley & Don Anderson. *PCI System Architecture.* Annabooks, fourth edition, September 1995.

[Tri94] S. Trimberger, editor. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, 1994.

[W$^+$95] Matt Welsh et al. Linux Kernel Hackers Guide. http://metalab.unc.edu/mdw/LDP/khg/HyperNews/get/khg.html, c1995.

[Wei93] M. J. Weiner. Efficient DES Key Search. presented at the rump session of CRYPTO' 93, August 1993.

[Wei94] M. J. Weiner. Efficient DES Key Search. Technical Report TR-244, School of Computer Science, Carleton University, May 1994.

[wel99] weld@lopht.com. Web users can view sensitive information in microsoft iis 4.0 web server. L0pht Security Advisory, May 1999. http://www.l0pht.com/advisories.html.

[Wie93] Lauren Ruth Wiener. *Digital Woes: Why We Should Not Depend On Software*. Addison-Wesley, 1993.