

## ABSTRACT

GANESAN, PRASANTH Efficiently Adding Secure Communications to Networked Low-End Embedded Systems using Software Thread Integration. (Under the direction of Assistant Professor Alexander Dean).

The wide spread acceptance of distributed embedded networks is motivated by factors such as cost, weight and power consumption. Communication protocols or MAC layer protocols can be implemented in software on microcontrollers to save costs in comparison to dedicated chips. Traditional methods for implementing a protocol's lowest layers (sending and receiving bits and bytes) in software incur significant execution time overhead, which limits system efficiency and peak performance, increasing power consumption.

An additional constraint on communication is security. Wireless networks in particular require secure channels of communication due to the open nature of the RF medium, which makes them vulnerable to attacks. To provide privacy these networks use security protocols. Cryptographic support is a vital ingredient of these protocols. All data transmitted and received is encrypted or decrypted in real time. This further burdens the system processor, forcing lower network bit-rates or higher processor clock speeds and therefore increasing power consumption.

Software Thread Integration (STI) is a software technique which interleaves multiple threads at the machine instruction level. This enables system resources to be used efficiently and eliminates context switch overhead. This technique gives a wonderful opportunity to utilize the free cycles generated in communication protocols by cryptographic algorithms. Real-time work is performed faster, allowing for better throughput rates. Throughput increase is also attained because of more efficient use of the communication channel. Timing constraints of the communication threads are met while cryptographic duties are performed concurrently. This saves many processor cycles as well. In some cases it may also allow hardware to software migration without any effect on the throughput rates.

This thesis proposes a set of methods to add cryptographic support efficiently to networked embedded systems using STI and save processor cycles and power. A system level software architecture is proposed to enable the use of integrated threads efficiently. TDMA threads are integrated with encryption/decryption threads and, using the above technique and proposed architecture, implemented as part of an open source operating

system AVR<sub>X</sub>. The results show that an STI based implementation is more efficient in the case of synchronous transmission compared to traditional ISR or busy-wait schemes. Our analysis of an integrated thread including the RC4 stream cipher at a relative bus speed of  $f_{cpu}/16$  showed a 22% increase in algorithm throughput while data throughput increased by 23%. Further results and benefits are tabulated and evaluated.

**Efficiently Adding Secure Communications to Networked Low-End  
Embedded Systems using Software Thread Integration**

by

**Prasanth Ganesan**

A thesis submitted to the Graduate Faculty of  
North Carolina State University  
in partial satisfaction of the  
requirements for the Degree of  
Master of Science

**Department of Electrical and Computer Engineering**

Raleigh

2003

**Approved By:**

---

Dr. Alexander Dean  
Chair of Advisory Committee

---

Dr. Gregory T. Byrd

---

Dr. Mihail Sichitiu

To my parents . . .

## Biography

Prasanth Ganesan was born in Chennai, India. He graduated with a Bachelor of Engineering degree in Electronics and Communication, from Maulana Azad College of Technology (Regional Engineering College), Bhopal, India in May 1998. He then worked for Infosys Technologies Ltd., Bangalore, India for 3 years as a software Engineer until July, 2001. Following which he undertook a Masters program in Computer Engineering at North Carolina State University.

## Acknowledgements

I thank my adviser Dr.Alexander Dean, without whose insight and guidance this work would not have been possible. I am grateful to my committee members Dr.Gregory Byrd and Dr.Mihail Sichitiu for devoting time and providing valuable inputs.

I also thank all my friends for their ideas and timely assistance, which has proven helpful during the course of this work.

Finally, I would like to thank my parents for their constant support and encouragement, which has been my lifeline many times during the last two years.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Cryptographic Algorithms</b>	<b>5</b>
2.1 Effect of Algorithms . . . . .	5
2.2 Description of the algorithms . . . . .	7
<b>3 Execution Model for Low-Level Communication</b>	<b>8</b>
3.1 Execution Model . . . . .	8
3.1.1 Bit Banging . . . . .	9
3.1.2 Byte Banging . . . . .	9
3.1.3 Other Control Mechanisms . . . . .	10
3.2 Analysis of common implementation schemes . . . . .	12
3.2.1 ISR based implementation . . . . .	12
3.2.2 Busy Wait Implementation . . . . .	16
3.3 Benefits with STI based scheme . . . . .	17
<b>4 Secure Communication</b>	<b>22</b>
4.1 Problem statement . . . . .	22
4.2 Performing Integration . . . . .	23
4.2.1 Case study of RC5 . . . . .	24
4.3 Integration Results and Analysis . . . . .	25
<b>5 Software Architecture</b>	<b>34</b>
5.1 Basic architecture . . . . .	34
5.2 Modifications to software architecture . . . . .	37
5.2.1 Thread Controller . . . . .	38
5.3 OS Support for Integrated Threads . . . . .	40
5.4 Case study of the architecture on AVRX . . . . .	42
5.4.1 AVRX . . . . .	42

5.4.2	Modifications to AVRX . . . . .	44
<b>6</b>	<b>Conclusion and Future Work</b>	<b>47</b>
6.1	Conclusion . . . . .	47
6.2	Future Work . . . . .	47
	<b>Bibliography</b>	<b>50</b>



# List of Figures

1.1	A sample cryptographic scheme . . . . .	3
3.1	SPI Master - Slave Implementation . . . . .	11
3.2	ISR based implementation using SPI bus . . . . .	13
3.3	Timeline for ISR based implementation . . . . .	14
3.4	Busy Wait implementation using SPI bus . . . . .	17
3.5	Timeline for Busy Wait implementation . . . . .	18
3.6	STI code for using the SPI bus . . . . .	19
3.7	Timeline for STI based implementation . . . . .	21
4.1	RC5 before transformation . . . . .	24
4.2	RC5 after transformation . . . . .	26
4.3	Performance increase of the RC4 algorithm for STI . . . . .	27
4.4	Code expansion for the RC4 algorithm . . . . .	27
4.5	Performance increase of the RC5 algorithm for STI . . . . .	28
4.6	Code expansion for the RC5 algorithm . . . . .	28
4.7	Percentage increase in throughput for STI over other schemes . . . . .	29
4.8	Actual throughput vs. clock speeds . . . . .	31
4.9	Clock speed vs. Performance for a constant data rate of 1Mbps . . . . .	33
5.1	Pipe Filter System . . . . .	35
5.2	software architecture . . . . .	36
5.3	Modified architecture . . . . .	37
5.4	Filter Scheme . . . . .	41
5.5	Sample processor workload . . . . .	42
5.6	OS level view of the architecture . . . . .	45

## List of Tables

2.1	Surveyed Cipher Algorithms . . . . .	7
3.1	SPI speed vs. Clock cycles . . . . .	15

# Chapter 1

## Introduction

Increasing numbers of embedded systems rely upon embedded communication networks to improve performance, flexibility and reliability as well as reduce costs, weight, size and installation effort. Sensor networks and other low-end systems often have tight cost constraints and meager power budgets, and both of these factors complicate the use of communication networks. Network controller chips are generally expensive in comparison with a node's microcontroller. As a result, system designers often move the protocol functions to software to cut system costs or to implement custom-fit protocols. Traditional methods for implementing a protocol's lowest layers (sending and receiving bits and bytes) in software incur execution time overhead, which limits system efficiency and peak performance as well as increasing power consumption

Secure communication is gaining focus as embedded networks grow more common. Wireless networks are popular due to their ease of installation, discreet operation, and support for mobility. But these networks can be compromised due to the open nature of the RF medium.

Security requirements and mechanisms vary depending on the nature of the embedded network. The computing power of the embedded system, the purpose for which the embedded network is used (strength of the threat), and the communication mechanism (Communication medium, MAC layer protocols, etc.) determine the kind of security scheme that should be put in place.

All security protocols have cryptographic schemes as a prime component. The

cryptographic algorithms convert plain text to cipher text and vice versa. All protocol signaling may be completed a priori, but conversion from cipher  $\Rightarrow$  plain and plain  $\Rightarrow$  cipher text has to be performed while meeting real-time constraints. The implementation of these schemes may cause delay in actual transmission, thus reducing throughput.

The MAC layer of any wireless protocol dictates exact transmission guidelines. Similarly, low level communication protocol layers need to meet specific timing requirements for synchronizing the transmission and reception of data. Software implementations have to match these precise timing needs. They are currently met by the use of interrupts or nops in busy-wait schemes. Both methods waste cycles to meet timing requirements. The context switch overhead of ISRs become increasingly costly as communication protocol bit rate rises relative to processor speeds. This overhead makes higher data rates impossible. To solve this problem, implementations use the busy-wait scheme, where registers are polled until a change in state takes place. This in turn leads to many cycles where the processor does no work. These techniques also cause inter-byte delays in transmission as context switches or status checks use up important processor cycles before data is placed on the bus. This causes a fall in the effective throughput of the system.

Software Thread Integration [5, 6, 4] is a compiler technique that merges multiple program threads of control into one. The integration uses code transformations to create interleaved code that runs efficiently on general purpose uniprocessors. This technique enables system resources to be used efficiently and eliminates context switch overhead. The timing constraints imposed by any real-time thread are met by this method.

As indicated above, cryptographic schemes of security protocols may be bottlenecks in attaining maximum transmission throughput. Implementation of many protocols attempt to offset this, by using hardware accelerators for encryption or decryption. Wired Equivalent Privacy (WEP) of 802.11 [1] is in most cases implemented in hardware, although most of the MAC layer functions are implemented in software. Even with this hardware implementation, the throughput of 802.11 networks do not attain their maximum when WEP is turned on. The extra component consumes power and adds extra cost to the chip. The security protocol for sensor networks such as SPINS [10] attempts to conserve energy and power by adjusting the length of transmission data by using a counter mode. This scheme is illustrated in figure 1.1. The encryption function is applied to a predetermined text sequence to generate a one time pad. This pad is then XORed with the plaintext. The decryption operation is identical. The advantage of such a scheme is that the computation-

ally intensive encryption part can be performed earlier while the XOR can be performed at run-time.

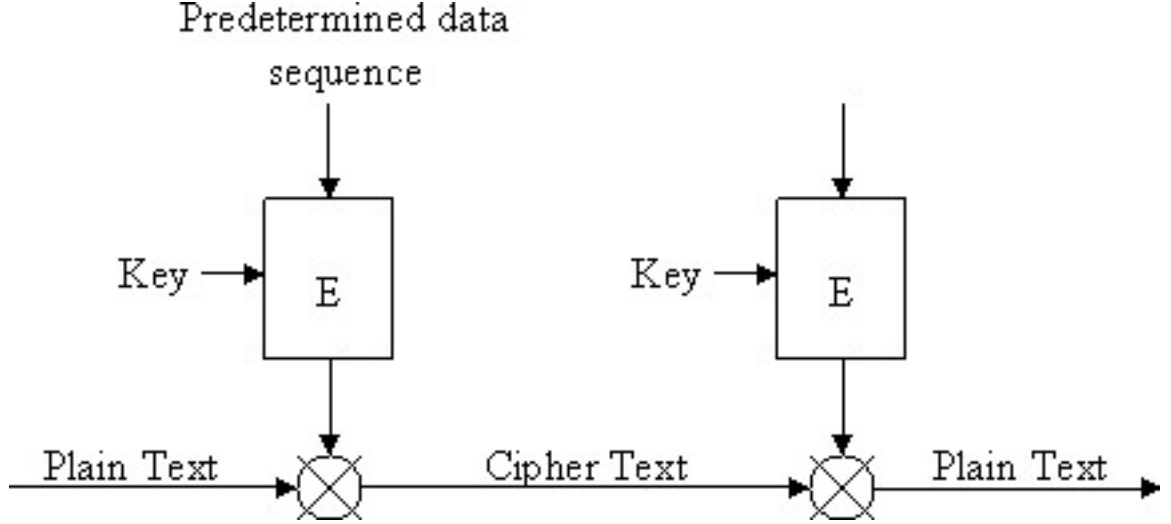


Figure 1.1: A sample cryptographic scheme

STI is an alternative technique to interrupt and busy-wait schemes for the implementation of MAC/Communication protocols. It provides accurate timing and does not waste processor cycles in context switch overheads. It efficiently utilizes the communication channel and allows the system to attain maximum possible throughput. In addition to this it frees up processor cycles to perform work in parallel. By using these freed up cycles, security mechanisms can now use alternate methods than those described above to reduce the cryptographic bottleneck. Communication can be made both secure and fast.

This thesis highlights the benefits of using Software Thread Integration for the purpose of secure communication. The constraints that are imposed by current software implementations can be easily overridden by the use of STI. The much needed concurrency while using encryption schemes with communication is easily achieved, enabling better throughput. Throughput is increased by more efficient use of the communication channel and processor cycles can be saved conserving power in low-level embedded devices and sensor networks. The second contribution of this thesis is to propose a software architecture to use the integrated threads generated by STI more efficiently in a system design. Further, a proposal is made on a generic scheduling scheme, for an operating system to use integrated

threads.

The organization of the thesis is as follows. Chapter 2 focuses on the cryptographic algorithms evaluated in this thesis. The advantage of these algorithms with respect to STI is discussed. Chapter 3 analyzes legacy schemes along with STI and indicates the benefits of STI with respect to throughput and processor cycle conservation. Chapter 4 evaluates the integration of cryptographic schemes with communication threads and captures the benefits. Chapter 5 proposes the software architecture for using integrated threads in a system. Chapter 6 discusses the conclusion and future work.

## Chapter 2

# Cryptographic Algorithms

### 2.1 Effect of Algorithms

The choice of a cryptographic algorithm is very important, not only for its cryptographic properties but also for its compatibility with current methods of software thread integration. A cryptographic algorithm is a mathematical function used for encryption or decryption. This function performs the basic mathematical operations of **ADD, MUL, AND, OR, XOR and rotate/shift** in a pre-determined order on the input plain text or encrypted text. These actions may be coupled with certain memory operations.

STI requires a deterministic run time from its threads. Thus each subset of operations in the threads have to be performed in finite time. Almost all processors perform the above mathematical and data-manipulation operations deterministically. High performance processors with memory hierarchy and sophisticated architectures are not considered, as the target here is low-end microcontrollers. Therefore the temporal determinism needed by STI is not a factor of hardware but software. Many cryptographic algorithms perform a subset of the above operations in loops. The constraint for integration is that these loop counts too have to be deterministic. Cryptographic algorithms depend on input data for loop counts and shift distances. Although many algorithms have fixed loop counts in the implementation, the compiler may introduce loops of its own, depending on how it performs the mathematical operations. For example, the AVR instruction set performs shifts on one bit

at a time. Hence the compiler performs these shifts in a loop depending on the distance to which they need to be performed.

In case these loops or shifts form an integral part of the algorithm and are executed in the middle of the algorithm, they may throw the temporal determinism of later sections off acceptable limits for STI. In these cases the code must be modified at the assembly level to make it amenable to STI, like addition of pads and movement of code between loops. One such transformation performed for the RC5 algorithm is discussed in a later section. The restrictions indicated do not apply to an external loop which runs the algorithm repeatedly for different input data and does not constitute the algorithm itself. Still it constitutes the thread that is integrated. For such loops, STI allows extra loop cycles for the concerned (guest or host) thread to be executed at the end of an integrated thread. Another advantage of cryptographic schemes with respect to STI is that all operations are integer based, simplifying the mathematical function implementations and assuring that only the main processor is involved always.

There are two kinds of cryptosystems: asymmetric and symmetric. Symmetric schemes utilize the same key and algorithms for both encryption and decryption purposes. Asymmetric schemes use different keys for encryption and decryption. The asymmetry provides for a more convenient way of transferring data, but for the security to be strong, the key lengths used are very large (256-1024 bytes). Maintaining and manipulating such large key lengths require very large integer math. Software implementation of these would put a huge strain on resources of current embedded systems. There is no successful published work on software implementations of asymmetric cryptographic schemes for 8-bit architecture. Due to this restraint the analysis in this thesis focuses only on symmetric schemes. However, as the basic tenets of cryptography are the same, the STI principles would work with asymmetric schemes too, provided temporal determinism of the implementation is possible.

The algorithms chosen for the experiment are widely used algorithms, which find their use in a lot of current products. They are indicated in table 2.1. These algorithms form an integral part of many security protocols. RC4 is used in WEP [1] and RC5 has been suggested as a good algorithm for sensor networks [10]. The algorithms have been chosen because of their popularity and applicability to embedded systems. The cryptanalytic strength of the algorithm is not a focus of this experiment or analysis.



Algorithm	Type
RC4	Stream Cipher
RC5	Block Cipher

Table 2.1: Surveyed Cipher Algorithms

## 2.2 Description of the algorithms

### RC4

RC4 [12] is a stream cipher symmetric key algorithm. This algorithm is quite simple and operations involve the addition of 8 bit elements or swapping variables. RC4 uses a variable length key between 1 and 256 bytes to initialize a 256-byte state table. The state table is used for subsequent generation of pseudo-random bytes and then to generate a pseudo-random stream, which is XORed with the plaintext to give the ciphertext. Each element in the state table is swapped at least once. A 128-bit key is used for our experiments.

### RC5

RC5 [12] is a fast symmetric block cipher with a variety of parameters: block size, key size and number of rounds. It primarily consists of three operations: XOR, addition and rotations. These operations are bounded on most embedded processors. We select an RC5 implementation with a 64-bit data block and 64-bit key. The key is used to generate  $(2r + 2)$  32-bit words ( $S[2r+1]$ ) that are used in the encryption and decryption algorithms ( $r$  is the number of rounds). During encryption the plaintext is split into two 32-bit words and a series of XOR, rotation and addition operations are performed on these words in conjunction with the above array  $S$  to generate the ciphertext. The decryption process is similar and involves the above operations in a different order.

## Chapter 3

# Execution Model for Low-Level Communication

Communication protocols implemented in software require support from the microcontroller to access the physical medium for transmitting and receiving bits. Depending on the constraints of the protocol or the micro-controller architecture, the software implementation maintains bit or byte level control on transmission. This chapter examines the facilities provided by a micro-controller for the purpose of communication between devices. The model of how software controls the various peripheral devices is presented. These models are compared and contrasted, and the benefits of STI as an implementation model is discussed.

### 3.1 Execution Model

Software implementation or the execution control of how and when bits are placed on the physical medium are influenced by supported peripheral devices, timing accuracy required and communication protocol complexity. Broadly these control mechanisms can be classified as bit and byte banded schemes.

### 3.1.1 Bit Banging

In this scheme communication is achieved by operation on bits rather than bytes. This is seen in software implementations of many communication protocols where rather than an on-chip interface the software works with general purpose I/O ports. The control and data are sent or received by toggling the values of specific bits on a port.

### 3.1.2 Byte Banging

In this scheme the software controls low-level communication through control and data registers. An 8-bit shift register is used to serialize the bytes while the formatting is performed in software. The software acts byte-wise by interacting with a data register. Parallel work is performed while the transmission/reception completes. In the implementation of byte banged schemes, the software works with (1) interrupts to determine the instance of transmission and reception or (2) busy wait loops that check control signals to identify instances of transmission and reception.

- *Interrupt Service Routines* - To support byte banging, micro-controllers provide a set of registers for control and communication with the hardware. The 8-bit shift register from which data is placed onto the physical medium is called the data register. The bits that indicate the current status with respect to the activity on the medium are part of the status register. The actual signal and control are part of the control register. By setting bits in the control register, an interrupt can be set to occur as soon as a byte is transferred or received. This provides dynamic control to the software to initiate transmission when the actual medium is free. There is however a penalty in the form of context switch overhead. The process of saving registers and switching to an interrupt service routine uses up valuable cycles of processor time.
- *Busy-wait scheme* - In a busy-wait scheme, unlike in the ISR scheme, where the hardware signals the status, the software keeps checking the status of the bus before it transmits. This is enabled by testing bits in the status register, which are set or reset depending on activity between the data register and the bus. The software runs in a loop until there is a status change in the respective bit of the status register. This has low overhead when compared to the ISR scheme indicated above, but no concurrent work can be performed as the control is always with one thread.

Peripheral features of micro-controllers that use byte banging vary depending on the type of transmission (asynchronous or synchronous) or protocol functionality supported. An overview of popular byte-banged schemes is given below. This establishes the base for the scheme to be used in our study and experiments.

- *Serial Peripheral Interface (SPI)* - An SPI register and bus are used to setup synchronous serial communication between two embedded devices, SPI communication operates in a Master - Slave mode, with transmission controlled by the Clock signal generated by the Master. Software control is both interrupt driven as well as implemented as busy-wait loops. SPI allows high speed synchronous transfer. Hence for some higher bit rates only the busy-wait schemes are possible in case the microcontroller operates at a lower frequency. This is to avoid the overhead of switching to Interrupt service routines as explained previously. A detailed discussion of the operation of the SPI bus on the Atmega128 microcontroller, that follows this section, gives a complete picture on the operation of SPI.
- *Dedicated Protocol Controllers* - Many microcontrollers have on-chip support for a number of communication protocols. The protocols themselves are part of hardware with handles for their control provided to the software in the form of data and control registers. These schemes are also generally byte banged with software implementations similar to SPI. One such example would be the I2C bus as part of the chip.

### 3.1.3 Other Control Mechanisms

As previously seen in byte banged schemes, the peripheral device serializes and deserializes the bytes. But some devices, apart from providing the above functionality, also format the bytes. Schemes where extra information is added to the data by the device do not fall under the category of byte-banged schemes. An example of such a scheme is given below:

- *Universal Asynchronous Receiver Transmitter (UART)* - The UART is a common component of most microcontrollers in the market. It follows a byte oriented scheme to handle asynchronous serial communication. It supports parity and uses delineators for each byte transmitted to ensure correctness of data. Software control for transmission and reception of data on the UART is normally interrupt driven. However,

dedicated busy-wait loops to handle the control of receive and transmit functions are not uncommon. A common use of the UART is for programming the microcontroller or establishing communication with a desktop computer.

The focus in this thesis is to analyze the idle times available in the use of a byte banded scheme and how STI enables us to reap these idle times to perform encryption work. SPI is used as the base to evaluate the improvements achieved by an STI scheme. The following section discusses the operation of SPI on an ATmega128.

### SPI on ATmega128

The ATmega128 is the processor used in the evaluation. It is a low power CMOS 8-bit micro-controller based on the AVR enhanced RISC architecture. The micro-controller among other peripheral features supports the Serial Peripheral Interface (SPI).

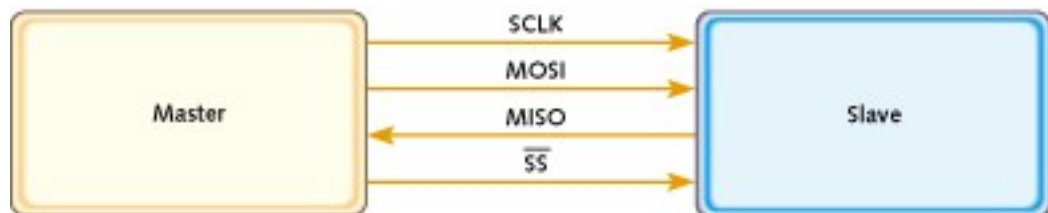


Figure 3.1: SPI Master - Slave Implementation

This allows high-speed synchronous data transfer between the ATmega128 and peripheral devices or between several AVR devices. The CPU utilizing the SPI operates in either the master or slave mode. As shown in figure 3.1 there are four lines that control the operation of SPI.

The SPI Master initiates the communication cycle by pulling low the Slave Select SS pin of the desired Slave. Master and Slave prepare the data to be sent in their respective Shift Registers, and the Master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from Master to Slave on the Master Out Slave In, MOSI, line, and from Slave to Master on the Master In Slave Out, MISO, line. After each data packet, the Master will synchronize the Slave by pulling high the Slave Select, SS, line.

This operation, where two 8-bit shift registers communicate by shifting data onto

and from the bus, is similar in operation to the serializer/deserializer found on some protocol chips. Therefore a study of the SPI scheme provides a generic solution to the operation of many protocol controllers.

In a distributed environment, for example in sensor networks, where there is no physical connection and communication is through the wireless medium, SPI is configured as follows. Separate synchronized clock lines drive the SCLK lines of various nodes. Any node that transmits becomes the master and transmits on the MOSI line. The intended recipient in turn pulls its SS line low and operates as the slave. The transmitter is connected to the MOSI line.

Control is governed by the SPI Control Register. The status register is used to verify the current status and the byte to be transmitted is written to the SPI data register while the received byte is read from it. An ISR can be setup to execute each time a byte is transmitted or received.

## **3.2 Analysis of common implementation schemes**

Software implementations of bit/byte bang communications have to meet exact timing constraints. Two primary methods of implementation that allow this are a Interrupt based scheme and a busy-wait scheme. Details of both these methods and how they perform are discussed below. The model that is used to compute the transmission rates and idle times is based on the AVR architecture. The free cycles computed could vary with other architectures.

### **3.2.1 ISR based implementation**

An interrupt service routine executes at each instance that a bit or byte needs to be transferred. In this specific scenario, the encryption thread runs while the ISR performs the complete job of transmission. The ISR writes data into the SPI register, sets the SPIE flag and returns. The same ISR would execute for both the transmission and reception threads. Therefore a check will have to be performed to determine what specific action must be taken inside the ISR.

The code shown in figure 3.2 in actual implementation would give the timeline

as shown in figure 3.3. This shows the activity on the micro-controller as well as the peripheral device (SPI bus). The main thread initializes the SPI data transfer and enables the interrupt. Then it continues to perform other tasks which in our case can be considered to be the encryption/decryption function. The ISR runs at regular intervals to interact with the SPI data register for transmission or reception. Once data is written onto the data register, the shift register puts bits on the bus, depending on the pre-selected bus speed. Regular context switches occur between the cryptographic thread and the ISR.

---

```

SIGNAL(SIG_SPI)
/* signal handler for spi transmission/reception complete interrupt */
{
    if(in transmission mode)
        if(all data not transmitted)
        {
            write data into the SPDR register for transmission
        }
    else /*in reception mode*/
        if(all data not received)
        {
            read data from the SPDR register
        }
}

```

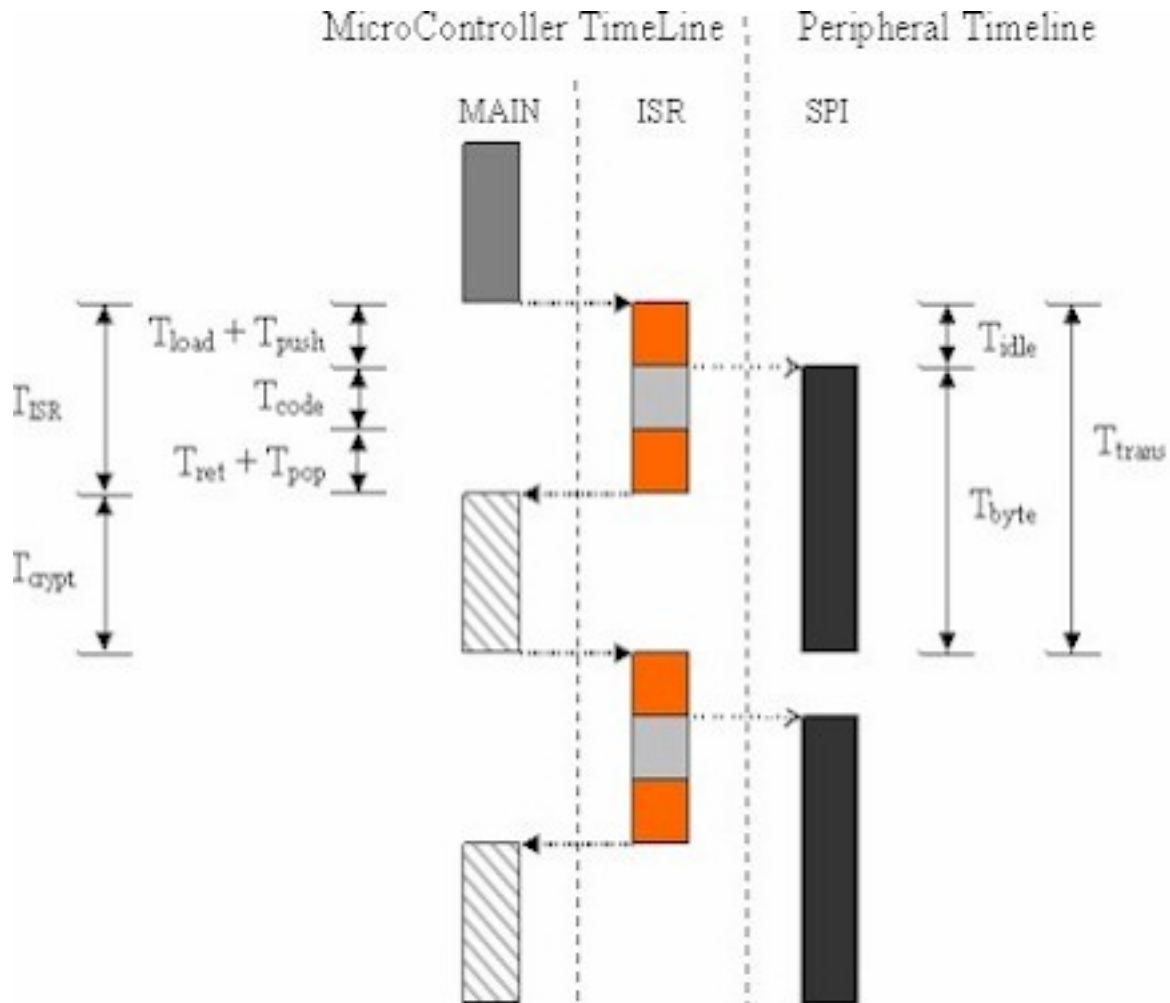
Note: Initialization would be performed in the main code.

Figure 3.2: ISR based implementation using SPI bus

---

The SPI bus on the Atmega128 operates at different data rates. These rates are a fraction of the CPU clock. If  $f_{cpu}$  is the speed of the CPU then the SPI bus operates at a frequency of  $f_{cpu}/2^n$  where  $n$  varies from 1 to 7. This operating frequency is set in the SPI Control Register.

For continuous transmission the throughput is limited by the speed of the SPI bus. The time taken for the transmission of a byte is given by  $T_{byte}$  while the actual transmission time that affects the throughput is  $T_{trans}$ . These values are multiples of the CPU clock period. The work performed while this transmission is taking place is shown by



$T_{byte}$	Time taken for data to be transferred in/out of the SPDR register
$T_{ISR}$	Total time taken for an ISR to execute
$T_{crypt}$	Time spent performing work in the main routine
$T_{load}$	Time taken for the ISR to load
$T_{push}$	Time taken for the registers to be pushed onto the stack
$T_{pop}$	Time taken for the registers to be popped off the stack
$T_{ret}$	Time taken for the ISR to return to the main context
$T_{trans}$	Actual Time taken for the transmission of a byte
$T_{idle}$	Inter byte delay on the bus when no data is transmitted

Figure 3.3: Timeline for ISR based implementation



SPI speed	$T_{trans}$ in clock cycles
$f_{cpu}/2$	16
$f_{cpu}/4$	32
$f_{cpu}/8$	64
$f_{cpu}/16$	128
$f_{cpu}/32$	256
$f_{cpu}/64$	512
$f_{cpu}/128$	1024

Table 3.1: SPI speed vs. Clock cycles

the equation

$$T_{trans} = T_{ISR} + T_{crypt} \quad (3.1)$$

During the transmission of each byte of data the work performed by the CPU is given by  $T_{ISR}$  and  $T_{crypt}$ . Here  $T_{ISR}$  is given by equation 3.2 and  $T_{crypt}$  is the useful work performed by the processor when the SPI bus is busy. In the current scenario this work is considered to be encryption/decryption work.

$$T_{ISR} = T_{load} + T_{push} + T_{code} + T_{pop} + T_{ret} \quad (3.2)$$

The various parameters of equation 3.2 are explained in the diagram 3.3. The value of  $T_{ISR}$  is a constant for a given processor frequency .

To increase the throughput of data transmission or reduce the value of  $T_{trans}$ , equation 3.1 indicates that the value of  $T_{crypt}$  has to reduce. Therefore for an ISR based implementation, the limit on the maximum throughput would be when  $T_{crypt}$  is set to zero giving

$$T_{trans} = T_{ISR} \quad (3.3)$$

Consider table 3.1 for the number of clock cycles taken for various SPI bus speeds. As indicated before, for an ISR based scheme to work at each of these data speeds the limiting factor would be  $T_{ISR}$ . By measuring the times of the various components of  $T_{ISR}$  for the Atmega128 we see that  $T_{ISR} = 130$  cycles. Hence from the table an ISR based implementation can not work when the SPI speed is  $f_{cpu}/16$  and below.

Another important factor is the value of  $T_{idle}$ . This value determines the actual throughput of the bus. There is a time lag between when the Interrupt flag is set to when the next byte is placed in the SPI data register. This lag is mainly due to the overhead of a

context switch. The byte transmission throughput is  $f_{cpu}/2^n * 8$  and  $T_{byte}$  is the transmission time. However, the actual transmission time  $T_{trans}$  is given by the equation below:

$$T_{trans} = T_{byte} + T_{idle} \quad (3.4)$$

Thus depending on overall context switch overhead there can be a substantial fall in throughput. As  $T_{idle}$  is constant for any clock frequency  $f_{cpu}$ , the detrimental effect on the throughput increases as the bus is driven at a higher speed for a given clock. This problem forces most implementations to be busy-wait loops written in C or assembly.

### 3.2.2 Busy Wait Implementation

In a busy-wait implementation the main thread places the data to be transferred in the SPI data register. Once the data is transmitted, the SPIF flag is set in the status register. This flag is tested in a busy wait loop and new data is placed on the SPI data register once this flag is set.

Implementation of the code in figure 3.4 would give the timeline shown in figure 3.5 with respect to activity on the micro-controller and the SPI bus. No other thread is allowed to run while the data transmission is going on.

As bytes are transmitted continuously, the transmission time of a byte ( $T_{trans}$ ) that influences that actual throughput is given by

$$T_{trans} = T_{code} + T_{busy-wait} \quad (3.5)$$

The value  $T_{trans}$  is discrete, ie: it varies by a factor of  $2^n$  while  $T_{code}$  is a constant. Thus a very high throughput can be achieved by adjusting the  $T_{busy-wait}$  to a very small value so that  $T_{trans}$  is adjusted linearly driven by the variable  $2^n$ .

But if data is not being transmitted with the highest throughput, then cycles are wasted in the  $T_{busy-wait}$  period. During this period the processor does not perform any useful work.

Secondly as seen in figure 3.5,  $T_{trans}$  is again limited by the idle time  $T_{idle}$ .  $T_{idle}$  comes into the equation as the process for testing the bit in the status register and looping control runs into tens of cycles. There is still a slight time lag between when the SPI data register becomes empty and the next byte is written for transmission. This value is small and does not affect the throughput significantly when transmitting with lower data rates.

However, it can cause an impact on higher data rates by causing a fall in maximum available throughput.

---

```

SPI_MasterTransmit(...)
{
    while(data buffer not empty)
    {
        Start transmission; /* SPDR = data */
        Wait for transmission to complete;
        /* while(!(SPSR & (1<<SPIF))) ; */
    }
}

SPI_SlaveReceive(...)
{
    while(all data not received)
    {
        Wait for reception complete;
        /* while(!(SPSR & (1<<SPIF))) ; */
        Read data register; /* data = SPDR */
    }
}

```

Note: Initialization performed outside this code section.

Figure 3.4: Busy Wait implementation using SPI bus

---

### 3.3 Benefits with STI based scheme

The STI technique removes the need for using interrupts or checking bits on the status registers as indicated in sections 3.2.1 or 3.2.2. The SPI code is reduced to interacting with the data register (figure 3.6). The timing constraints are all pre-determined statically while performing thread integration.

The time for transmission on the SPI bus is fixed.  $T_{trans}$  is constant for a selected SPI clock rate and does not depend on the instructions being executed. Unlike the previous two implementations,  $T_{trans}$  and  $T_{byte}$  as shown in figure 3.7 are the same. There is no idle

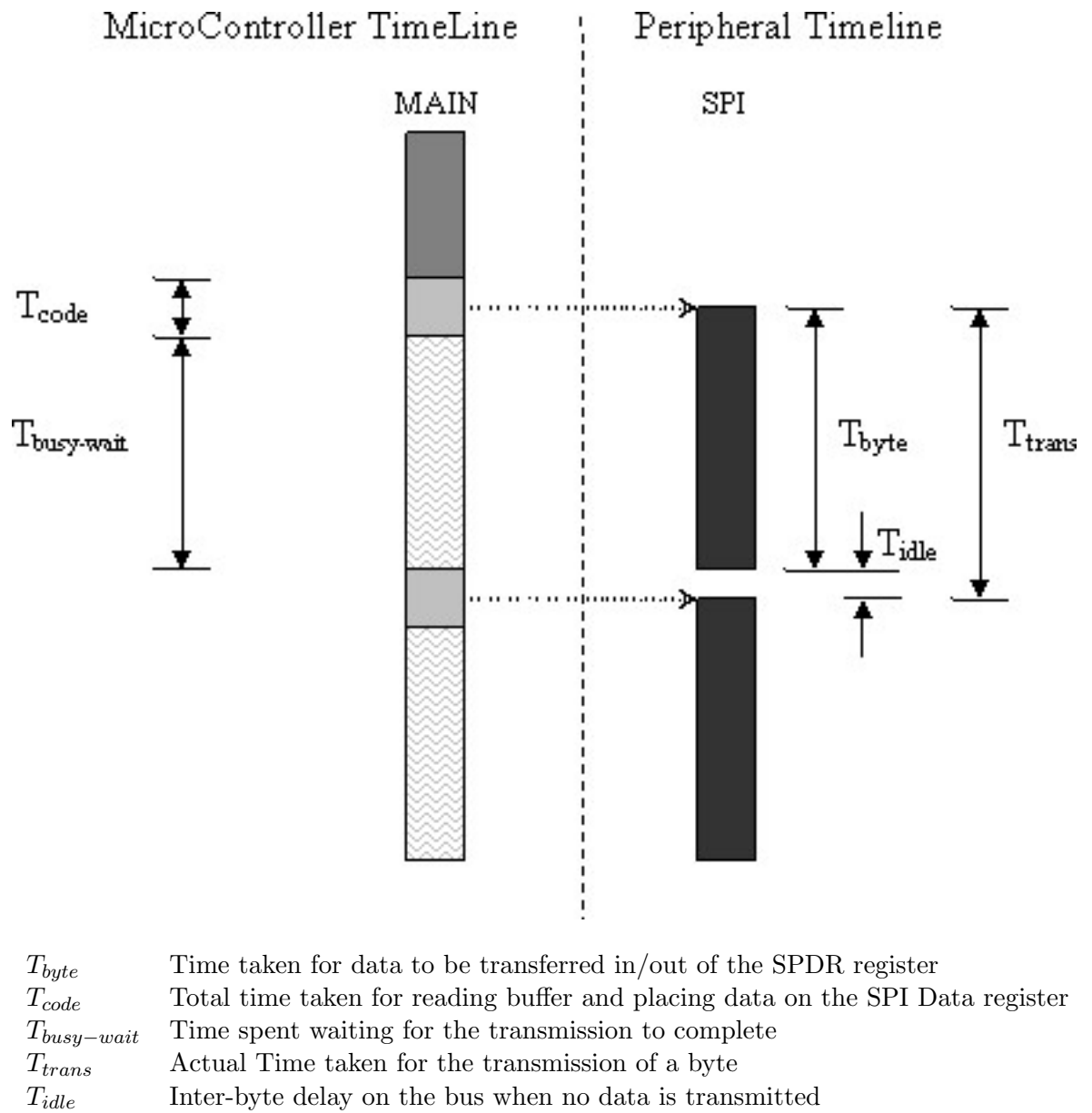


Figure 3.5: Timeline for Busy Wait implementation

time between consecutive transmissions of bytes. STI allows data to be moved to/from the data register immediately, once it becomes empty/full. There is no interbyte delay wasted in checking status registers or context switch overheads. In effect, without any change to the settings of the control register or microcontroller clock speed, there is an improvement in the effective throughput of the system.

---

```

SPI_MasterTransmit(...)
{
    if(all data not transmitted)
    {
        Start transmission; /* SPDR = data */
    }
}

SPI_SlaveReceive(...)
{
    if(all data not received)
    {
        Read data register; /* data = SPDR */
    }
}

```

Note: Initialization would be performed outside this code section.

Figure 3.6: STI code for using the SPI bus

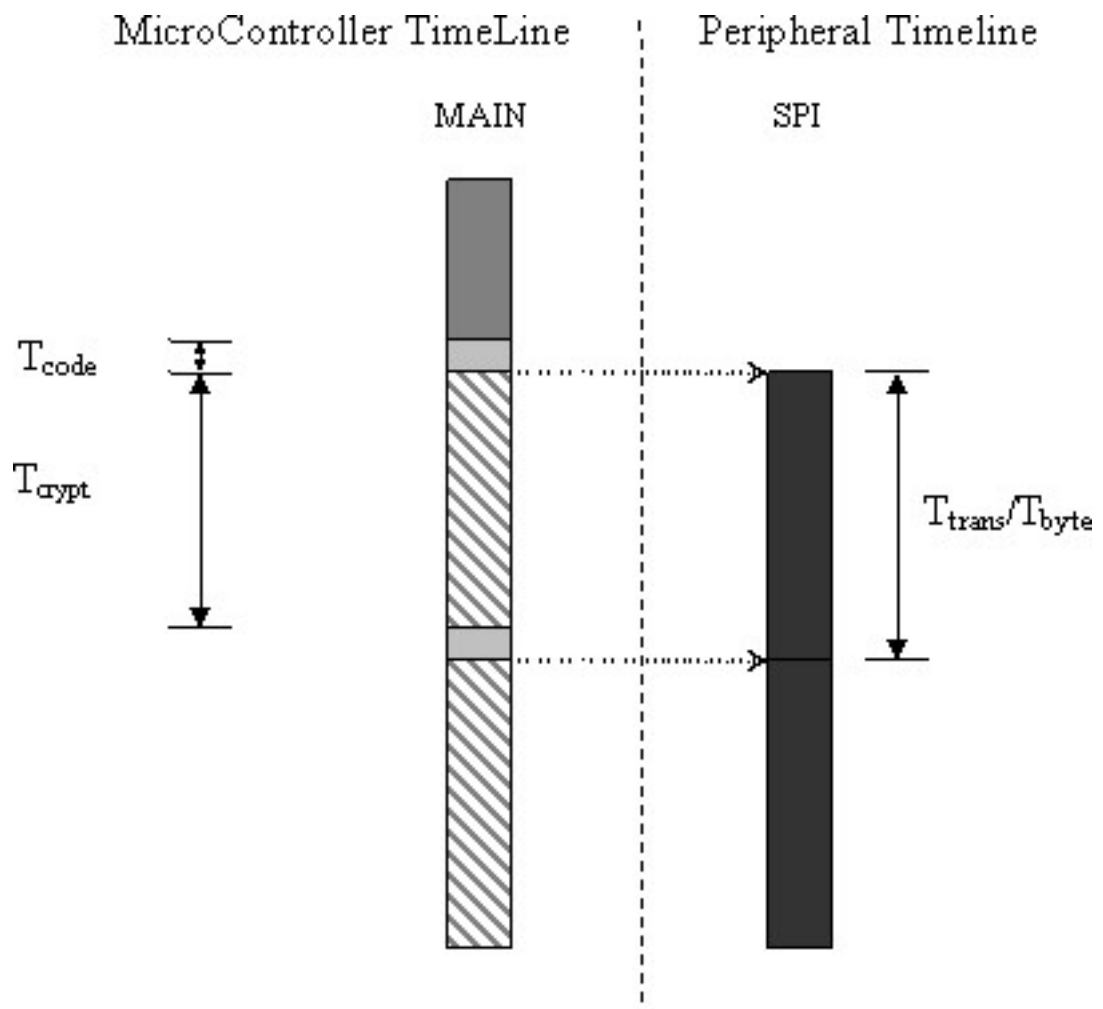
---

The context switch instructions or the busy-wait cycles are also freed up. These extra cycles can be used efficiently to perform useful work. For example, of the 128 cycles needed for transmission with a bus speed of  $f_{clk}/16$ , almost 120 cycles can be reaped by STI to perform cryptographic work. This reduces security scheme overheads as they are now performed at the expense of reaped clock cycles. A major benefit depending on the amount of cycles freed is that it allows the processor to sleep longer, increasing battery life. Secondly, the relaxed security bottle-neck enables better data rates. Faster data rates have the biggest benefit in sensor networks as it makes efficient use of the transmitter and saves power.

The above discussion can be summed up into two benefits:

- longer sleep times for the microcontroller.
- faster data throughput rates at the same clock speed.

In addition to the above there is also a region where interrupt service routines fail. The context switch overhead makes it impossible for an ISR to put data onto the SPI bus greater than a specific speed. This region of higher throughput can only be addressed by busy-wait and STI schemes. STI allows better performance in this region too, due to better synchronization with no interbyte idle times and extra work performed. A quantitative analysis of the benefits is presented in the next chapter.



- $T_{trans}$  Time taken for data to be transferred in/out of the SPDR register  
 $T_{code}$  Total time taken for reading buffer and placing data on the SPI Data register  
 $T_{crypt}$  Time spent performing cryptographic work

Figure 3.7: Timeline for STI based implementation

## Chapter 4

# Secure Communication

The previous chapter analyzed the improvements of STI over traditional methods of implementing communication protocols. Having established the case for STI this chapter focuses on the use of cryptographic schemes as an application that benefits from STI when used in conjunction with communication protocols.

### 4.1 Problem statement

As part of the study, a simplified version of a TDMA scheme is used and no specific security protocols are considered. The focus is to observe transmission and reception threads in conjunction with encryption and decryption methods. This also enables a simple implementation of the software architecture proposed in the next section. The cryptographic and communication threads are integrated and relative performance and benefits tabulated.

The RC4 algorithm has an initialization and cipher routine. Only the cipher part is considered as it is the only work to be done in real-time. The cipher part is integrated with a transmission thread. The transmission thread continually checks the buffer for data and writes it onto the SPDR register. This write into the SPDR register is constrained temporally. STI allows this write at exact instances by its code transformations.

The RC5 algorithm has an initialize, encrypt and decrypt routine. Only the encrypt and decrypt part are considered for integration as they are executed in real-time while



the initialize part would be executed only once at the start of a session. The transmission thread is integrated as indicated for the RC4 thread.

The  $f_{cpu}/16$  is studied as a break even point between the STI and ISR based schemes. Below this bus speed, ISRs become ineffective, as the context switch overhead is much higher than the period within which data has to be placed onto the SPI bus. An STI based scheme still provides high data throughput and good cryptographic performance at this bus speed. As the data rates rise or clock speeds increase and when the data rates get very slow or clock speeds fall, the benefits of STI are limited. This is studied in detail in section 4.3.

## 4.2 Performing Integration

Integration planning and integration are performed as indicated in [3]. Timing analysis of the guest thread and host thread is performed. Depending on the timing requirements of the guest code, transformations and padding are performed while inserting the guest code in the host code. Register reallocation is performed before integration, enabling both threads to run in the integrated version without any interference.

STI however requires that host code must have predictable timing constraints. Basic timing jitters, like unequal duration of two branches of a predicate, are padded to result in exact cycle count for all possible paths of the code.

Cryptographic operations as discussed in section 2 depend on multiple mathematical operations performed repeatedly. The nature of the algorithm may require that the count be dependent on the input data or plain text. This causes an indeterminacy in some loop cycle counts. The nature of most symmetric algorithms is such that the overall algorithm cycle count can be made a constant by padding code paths. In some cases, however, internal loop counts vary, but the sum of all loop counts is determinate. This problem can be addressed in STI by splitting loops to small determinate sections and/or adding guest triggering which causes extra checks to be performed in all loop counts. Solving the internal indeterminacy by these methods causes inefficient use of code space and reduces the amount of useful host work performed in guest idle time.

However the indeterminacy can be solved in most algorithms by performing certain transformations to code segments. These reduce the code overhead and allow efficient code

merging. A case study of such a transformation is shown in the next subsection.

### 4.2.1 Case study of RC5

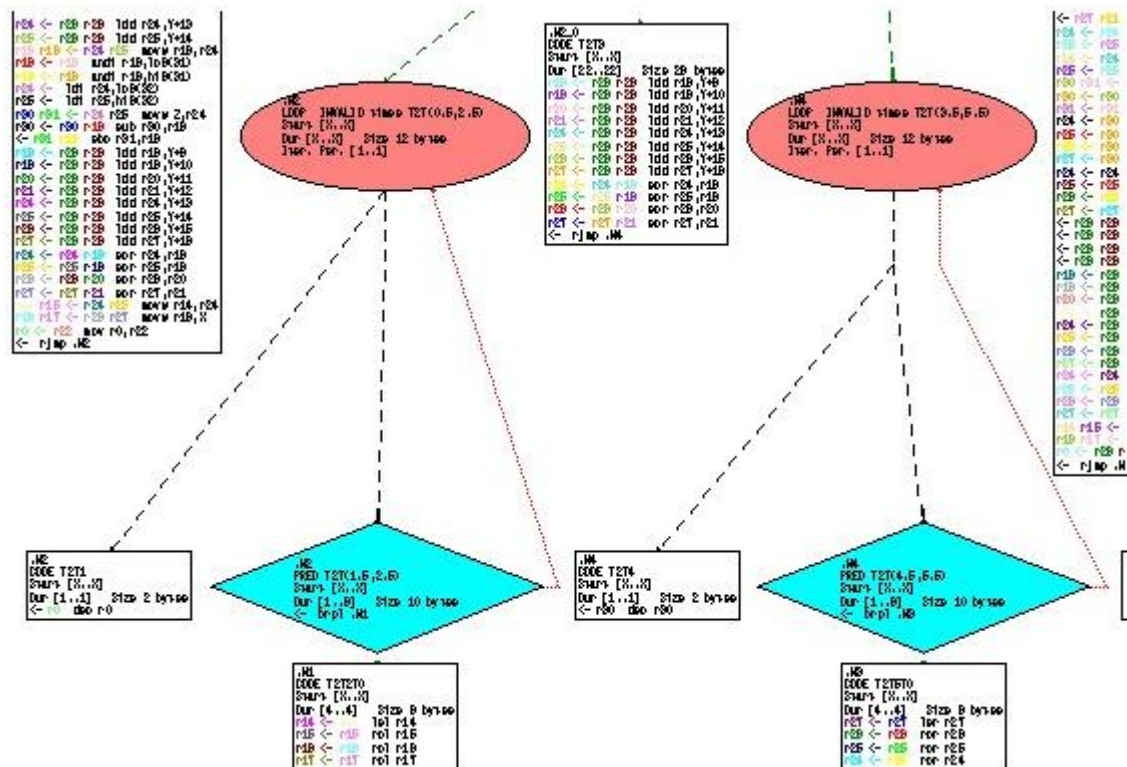


Figure 4.1: RC5 before transformation

The RC5 algorithm has 3 basic operations: XOR, addition and rotations. The rotations are constant time operations on most processors. But the count of these rotations depend on both the key and data. In the case of the AVR architecture, the rotations have to be performed only by the use of single-bit left-shift and right-shift operations. To complete the full rotation, the shift operations have to be performed in a loop. As the rotation count is non-linear, the loop counts are not constant and timing prediction is not possible.

However the rotations are all performed on 32 bit words. Hence, the rotation algorithm can be made to have constant time with the following (figure 4.1):

- shift the 32-bit word to the left for a count  $c$  with the count dependent on the data and the key.
- shift the 32-bit word to the right for a count  $32 - c$ .
- XOR the two values to get a rotate left by count  $c$

This makes all the rotate operations into 32 bit operations. This still leaves an issue that can make thread integration complicated. There are two loops with counts  $c$  and  $32 - c$ . Since  $c$  is a variable, for guest code with short inter-node timing difference, integration would again be complex, leading to bigger code size and more guest triggering mechanisms which shave off host time.

This can be offset by making the overall loop count to be a constant of 32 rotations. This is can be achieved by doing the following (figure 4.2):

- place a predicate inside the loop that tests for the count  $c$  and performs either the left or the right shift operations.
- reallocate registers and initialize them before entering the loop such that registers on either side of the predicate are not common.
- pad one of the edges so that both the edges execute for the same cycle count.

This ensures a constant loop count, and guest code can now be introduced into the loop efficiently using the current STI methods.

### 4.3 Integration Results and Analysis

Figure 4.3 shows the increase in performance of the RC4 algorithm with an STI scheme in comparison to an ISR based scheme. As there is no parallel work in a busy-wait scheme, any extra work by an STI scheme is a benefit. But as seen in the graph, the ISR scheme for lower data rates allows concurrent work. The throughput of the RC4 algorithm is in effect 22% higher than an ISR scheme at bus speeds of both  $f_{cpu}/16$  and  $f_{cpu}/32$ . For lower data rates however, both ISRs and STI schemes have enough cycles for the RC4 algorithm to be able to complete 100% of the work. There are extra free cycles generated by an STI based scheme which can be used for handling other MAC/Communication Protocol functionalities.

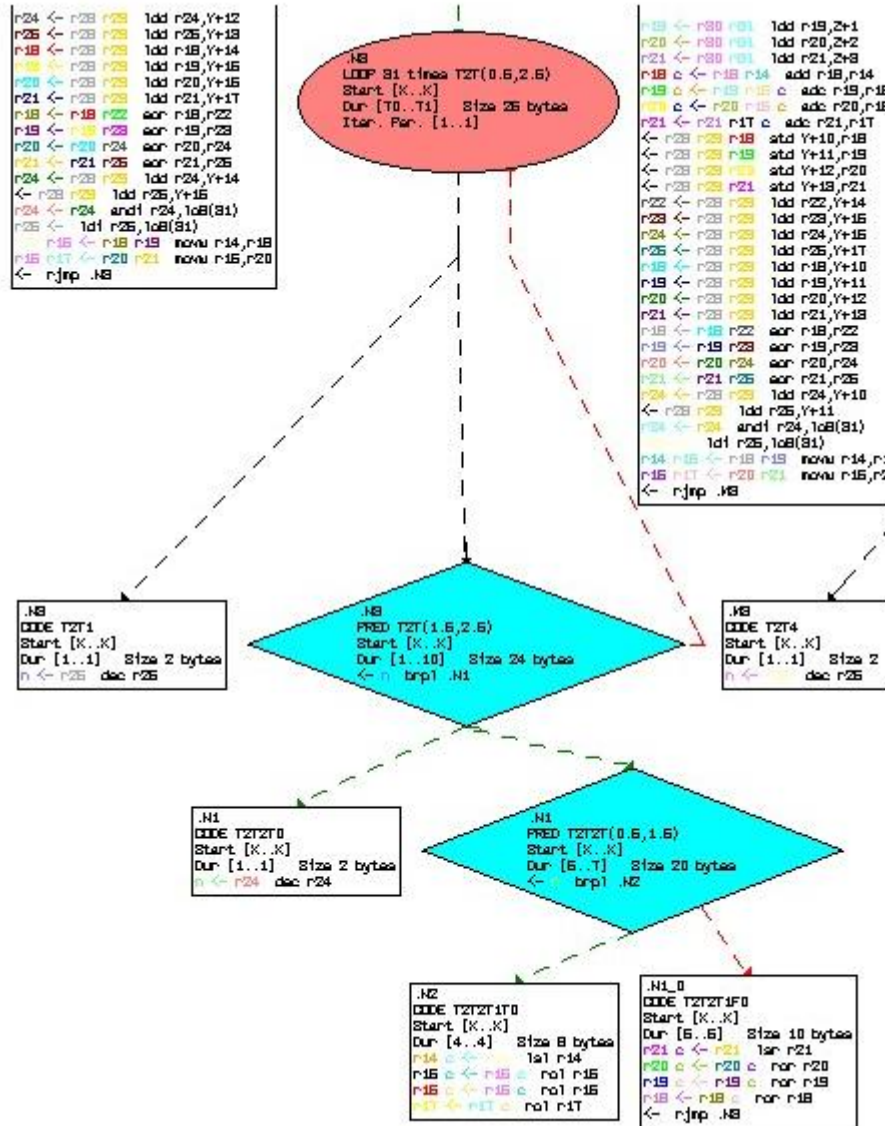


Figure 4.2: RC5 after transformation

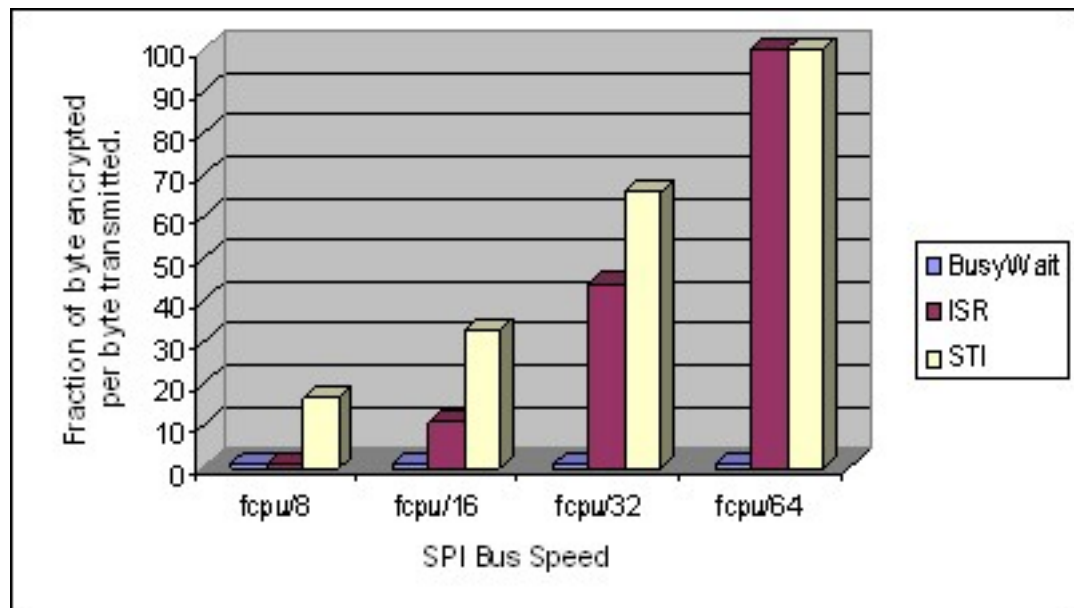


Figure 4.3: Performance increase of the RC4 algorithm for STI

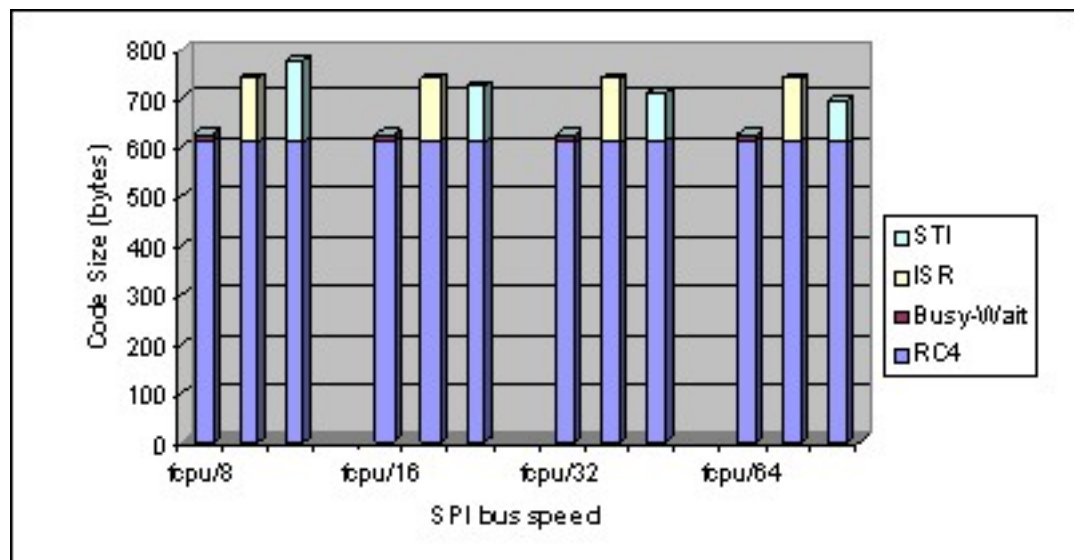


Figure 4.4: Code expansion for the RC4 algorithm

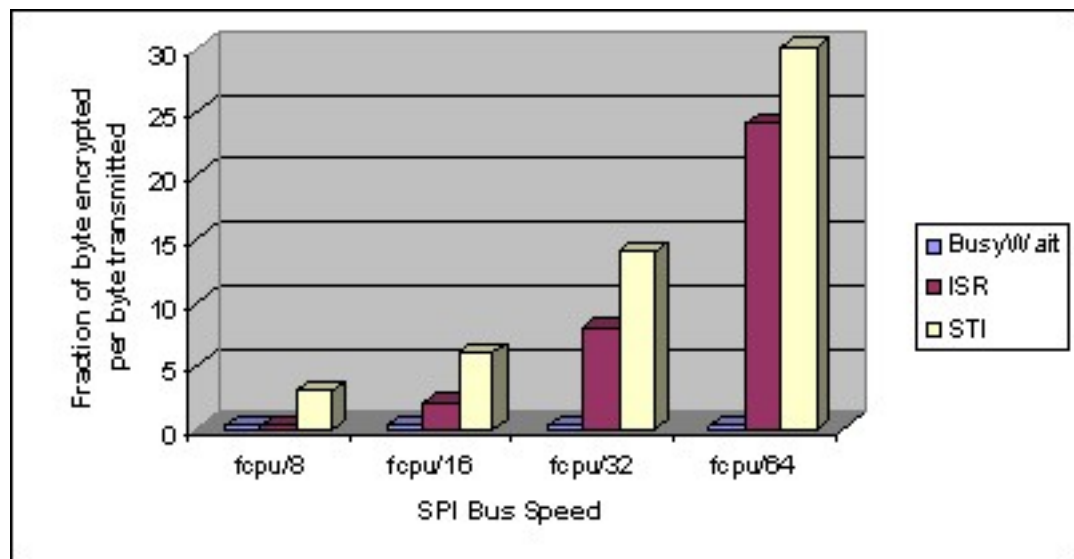


Figure 4.5: Performance increase of the RC5 algorithm for STI

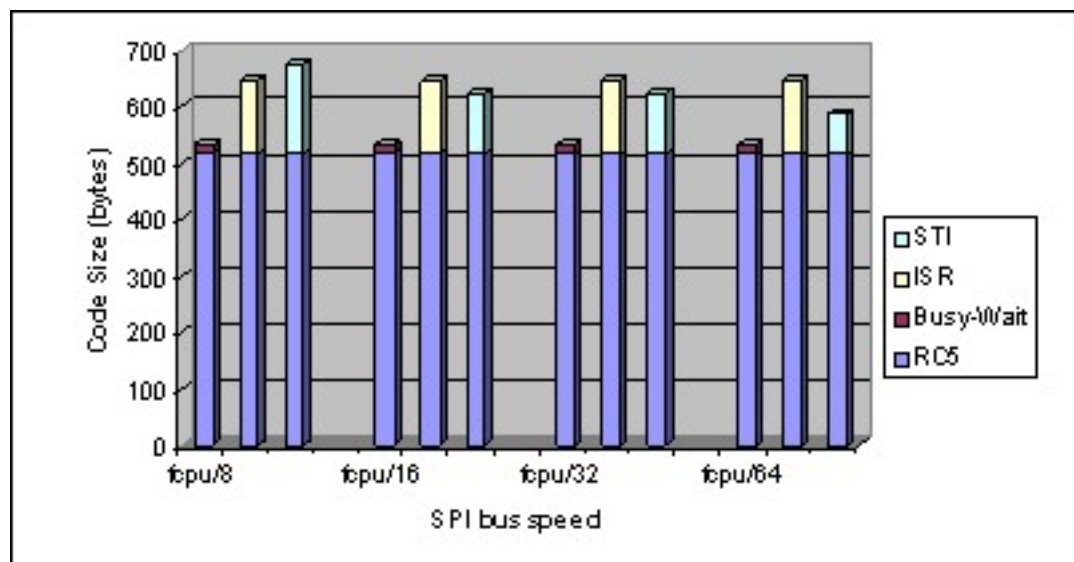


Figure 4.6: Code expansion for the RC5 algorithm

Figure 4.4 shows the comparative increase in code sizes when using STI, busy-wait and STI based schemes for the RC4 algorithm. The overhead in a busy-wait scheme is due to the check performed on the status register in a loop. The ISR code size overhead is due to context switch work of saving, re-initialization and restoration of registers. The STI code size increase is due to padding and multiple copies of the transmission code placed to meet the exact timing constraints. Even here as the data rates fall the overhead falls for STI, making it a better scheme than ISRs, as more concurrent work is performed.

The graph follows a similar path for the RC5 algorithm as well (figures 4.5 & 4.6). However the relative benefits are quite low compared to RC4. This is because of the relative algorithm complexity between RC4 and RC5 schemes. While RC4 takes 350 cycles to encrypt a byte, RC5 requires almost 1600 cycles to encrypt a byte. Secondly RC5 has a lot of internal loops, so guest triggering mechanisms cause a small overhead in both cycle count and code size.

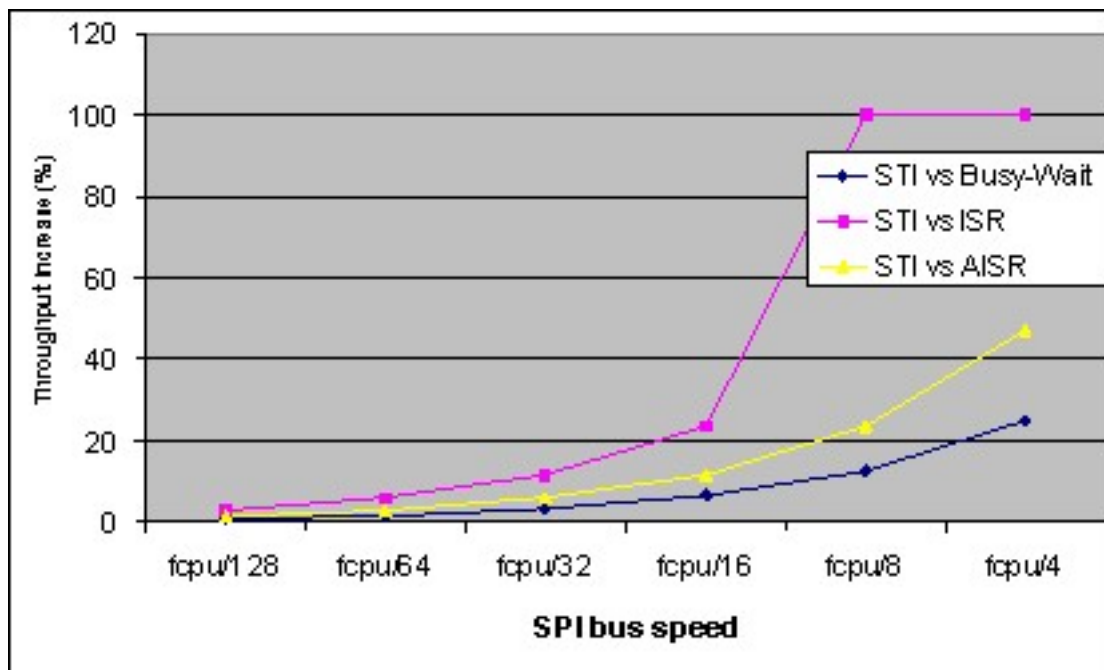


Figure 4.7: Percentage increase in throughput for STI over other schemes

Figure 4.7 maps the increase in throughput using an STI based scheme. Neglecting the parallel work performed, only the percentage increase in throughput of the STI scheme

is considered here. As the bus speed increases for a given system clock, the percentage by which the throughput rises is significant. The idle times between successive transmission of bytes in an ISR based scheme or busy-wait scheme cause a fall in the effective throughput of the data rates. The maximum available data rates are not achieved due to the overhead of the context switch or due to time wasted in reading status registers. Busy-wait schemes give comparable results with respect to throughput. However for higher data rates STI schemes perform better. In the graph the three lines indicate the percentage improvement of an STI based scheme over three other methods. The busy-wait and ISR methods are as indicated before. The AISR line refers to a hypothetical ISR scheme on the AVR microcontroller where there is an alternate register set available for the ISR. Even though registers are not saved during a context switch, the following overhead still occurs:

- Finish current instruction: up to 4 cycles (CALL, RET, RETI)
- Push program counter onto stack in Data SRAM (pointed to by SPH and SPL): 4 cycles
- Push status register onto stack: 2 cycles
- Follow interrupt vector (jump instruction): 3 cycles

This results in a delay of 9 to 13 cycles before ISR starts for the Alternate register set ISR (AISR) scheme. The graph proves that even with such a scheme, an STI based scheme is more efficient. The inconvenience at higher data rates for STI is that there are very few cycles to perform useful work. Therefore as seen in figure 4.3 the improvement in performance for RC4 at high data rates is between 8 and 15%.

The results indicate that while parallel work is an added benefit, we attain better data rates with STI. The same data without any change to clock speed or bus speed can be transmitted faster. This effective increase in data rates, allows the transmitter in case of wireless devices to be on for a much shorter period. Maximum power in sensor networks is dissipated in transmission/reception and shorter transmission time effectively increases battery life.

Figure 4.8 shows that for a system which transmits at 1Mbps, maximum throughput is attained only by STI schemes. This graph shows that by using STI we can effectively run the processor at a slower clock to achieve the same transmission speed as an ISR scheme. An ISR scheme fails to transmit at 1Mbps when the micro-controller clock is run at 4 &



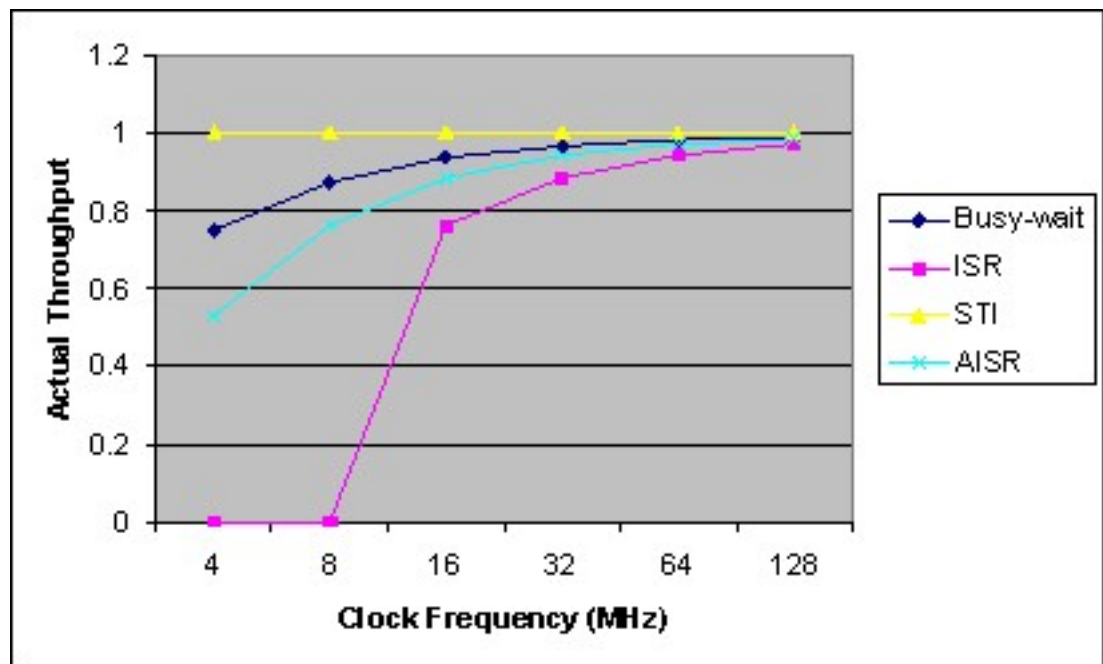


Figure 4.8: Actual throughput vs. clock speeds

8 MHz. But an STI based scheme still achieves the maximum throughput at the slower speeds.

Figure 4.9 maps the effect of clock speed on performance of parallel work. The comparison can be viewed as the percentage of cryptographic work performed to encrypt/decrypt 1 byte while transmitting 1 byte. The clock speed is increased keeping the bus speed or data rate a constant. The graph shows that with increasing clock speeds the benefit provided by STI is less significant. With higher clock speeds, there are more cycles that can be utilized for concurrent work. Hence the gain over an ISR based scheme is reduced. But at slower speeds, STI can boost system performance significantly. The performance gain for a cryptographic scheme largely depends on the scheme itself. RC4 shows quick gains, but the gains for RC5 are more pronounced only at higher data rates. However, as the clock speeds increase, the difference in performance of STI and ISR also narrows. Due to the relative complexity of RC5 over RC4, there are extra overheads in an integrated version of RC5 as indicated earlier. Therefore RC5 performs similar to an Alternate register set ISR scheme.

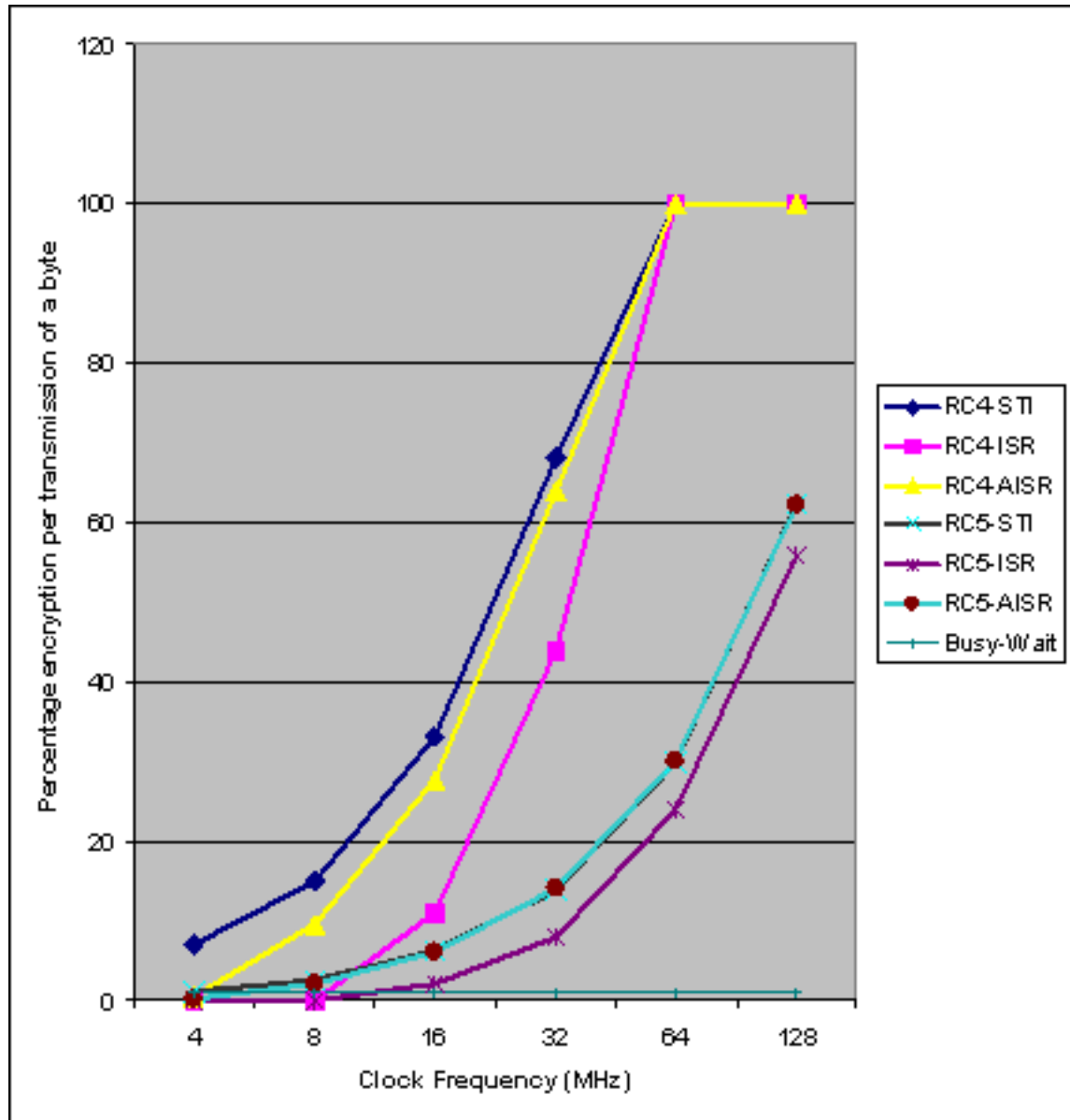


Figure 4.9: Clock speed vs. Performance for a constant data rate of 1Mbps

## Chapter 5

# Software Architecture

Threads in any execution model follow an order determined statically at compile time or dynamically by a scheduler. The finer details of the model are driven by the software architecture adapted for the system. The decisions made for scheduling depend on various factors like system state, task periods, task deadlines, component interactions and so on.

With the adoption of the integrated thread model, new scheduling variables enter the fray. An integrated thread runs depending on which tasks require work to be performed simultaneously. Further there is the consideration that the work of one task be postponed to work in parallel to another. This also leads to the aspect of scalability of current popular architectures to work well in this scenario. With concurrency considerations being affected, there is the need to tinker with the software architecture to meet this demand.

### 5.1 Basic architecture

The software architecture is presented as a conjunction of pipe-filter and layered styles. A pipe filter style [8, 14, 13, 9] has focus on the data flow in the system. There are a number of computational components where output from one component becomes the input for the next. In the purest form the components share no data or state, ie: the components are completely separated. Figure 5.1 illustrates a pipe filter style. Many implementations of communication protocols follow this style where processing is divided

into components (filters) and communication between the components is through message passing or intermediate buffers (pipes).

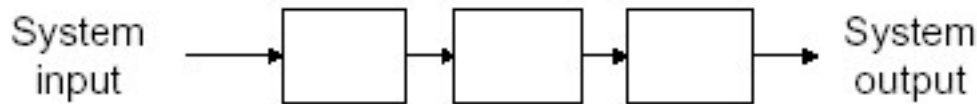


Figure 5.1: Pipe Filter System

Most protocols are implemented in a layered fashion, in adherence to the OSI model. Different layers perform different functions and communication between the layers is in the form of PDUs. A software architecture can have multiple views [9, 15, 2]. Thus the layered architecture may have a different view when seen with the aim of design [7]. This could very well be the pipe filter style.

The software architecture model for implementing the low level communication functionality is layered along with other functionality. It can be seen as forming the lower layers protocol stack. Figure 5.2 provides an overview of the general architecture.

The MAC/CP controller takes the data/packet from the higher layers and encapsulates it with its headers, and byte or bit bangs the data to be transmitted by the physical layer. Additional control data may also be sent.

The security block depicts the security mechanism in place including the protocol and encryption scheme. The cipher text production could follow a plain encryption policy where data from the higher layer is worked on by an algorithm; or follow a security scheme, for example: a seeding sequence is used to generate random text which is encrypted and XORed with the data to be transmitted. This layer comes into play in case the Mac/CP layer sees a requirement for the transmitted data to be cipher text.

It is here the MAC/CP layer and security mechanism can be seen as a pipe filter architecture in implementation. In an earlier description of the pipe filter architecture, it was mentioned that the scheme was composed of components which have a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams

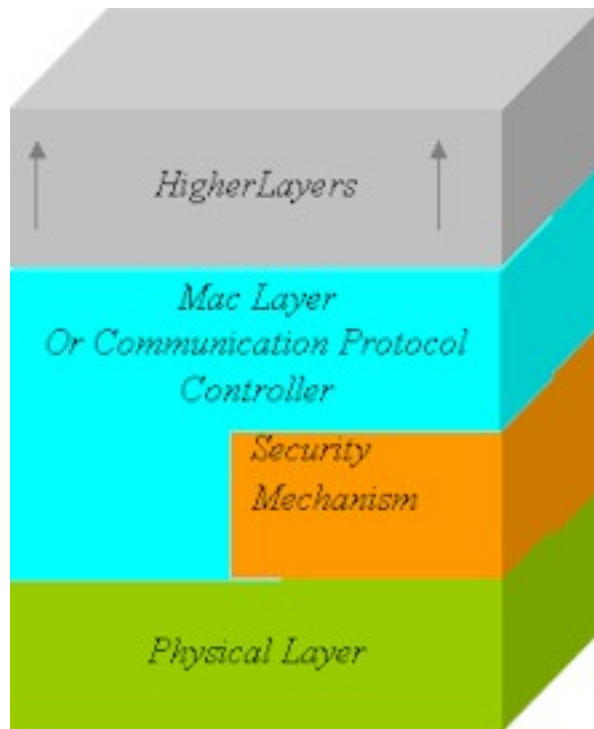


Figure 5.2: software architecture

and computing incrementally so that the output begins before input is consumed. Hence components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed "pipes".

The MAC/CP layer can be composed of an input pipe which is the PDU from the higher layers that it receives for transmission. The output pipe could be the control information for transmission or plain text to the encryption block. Similarly the security mechanism has 2 input pipes: The plain text for encryption and the cipher text for decryption. There would also be 2 output pipes from the security mechanism: The plain text from decryption and the cipher text from encryption. The MAC/CP protocol controls and the security functions would be the filters. The transmission and reception threads would be responsible for bit or byte banging the data in the cipher text pipes.

## 5.2 Modifications to software architecture

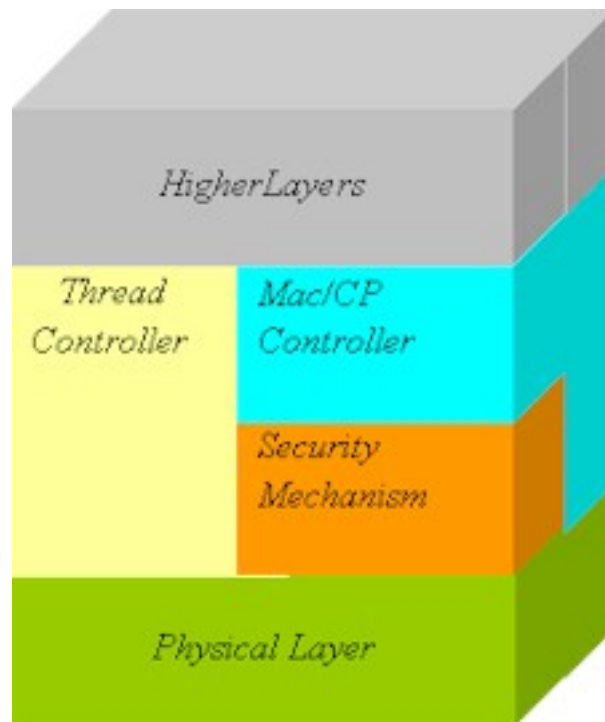


Figure 5.3: Modified architecture

The new architecture is a modification of the pipe filter mechanism. In pure form the pipe filter mechanism does not consider state information of the components. But with integrated threads this becomes a necessity. Integrated threads need to be run when there is concurrent work to be performed. There is a need to know which two threads have work waiting, so that the appropriate integrated version can be run. The pure pipe filter method does not account for this. Also if work has to be postponed, this decision has to be taken dynamically. Hence apart from a filter running to consume a pipe, a decision needs to be taken as to which instance of a filter will run. In other words the policy should account for the scenario where the filter runs independently or runs in conjunction with another filter in the form of a integrated thread.

The state information becomes vital in this instance. Depending on the indication from the Mac/CP controller and the data present in the pipe, a decision block determines

which specific filters are going to be in action. Although the MAC/CP controller determines when the transmission or reception has to begin, the actual thread to be run is determined by a thread controller filter. This has been called as the pre-filter and post-filter block depending on when this control function is invoked. The next section discusses the considerations for this block.

Figure 5.3 shows where the thread control functionality would reside in a layered architecture. This is not a requirement of any layer, but with integrated threads running, there needs to be information sharing between layers. Data transmission is not totally independent in this system. It needs to maintain status information regarding the interaction of the MAC/CP layer with the higher layers. This enables the right thread to be picked for execution.

### 5.2.1 Thread Controller

To elaborate the concept of this architecture, a simple approach is taken to the functionalities provided by the security and MAC/layers. The overall picture would contain 8 filters that connect 4 pipes.

#### Pipes:

- Data output from the higher layer.
- Data input to the higher layer.
- Data received from the physical layer.
- Data output to the physical layer.

#### Filters:

- Integrated thread: Encryption and Transmission
- Integrated thread: Decryption and Transmission
- Integrated thread: Encryption and Reception
- Integrated thread: Decryption and Reception



- Encryption
- Decryption
- Transmission
- Reception

The thread controller functionality is split into two filter control blocks. As there are specific timing constraints on when the transmission and reception threads have to run. These blocks run before and after the execution of a hard real-time thread which in this case is the communication thread. These blocks identify when and which integrated thread has to run and maintain state information for making this decision. A look into the decision making hints in these blocks is indicated below. The filter control blocks are named pre and post control blocks to indicate the time that they execute and make the decision.

#### **Pre Filter Control Block:**

This makes a decision as to which thread (filter) has to run based on some of the following criteria:

- Amount of data to be encrypted or decrypted
- Whether the Mac/CP controller has determined if it is going to be transmission or reception
- Priority assigned to the specific queue. That is which queue has more critical data to be processed. For example: Data in the encryption queue may be greater, but the transmission slot may be far away, hence it would be more useful to address the decryption block.
- How long the work be postponed or advanced.

The actual system may have many constraints and the actual decision block would depend on the exact system adopted. The above points are only a hint and there may in fact be many other implementation specific decisions which may also have to be considered.

### Post Filter Control Block:

Depending on the thread that has run, the hints or state information on running integrated threads may have to be updated. Certain variables that help the next run of the filter control block may have to be set. Also this block may have to make a decision as to service certain pipes that cannot wait to be handled until the next transmission/reception phase. Some hints that may be set or needed are indicated below.

- Run the encryption or decryption thread that clears the pipes, if the data is needed urgently.
- Determine if the higher layers filters can use unencrypted data and provide indication.
- Determine if work can be postponed so that an integrated thread can run the next time.

Figure 5.4 shows a block diagram of the thread controller system interacting with the pipes and filters. Another view as a processor workload can be seen in Figure 5.5. The thread controller scheme can be visualized as shown in the diagram. Whenever there is work for both the MAC/CP and security to be performed, a trigger enables the pre filter control block to execute. The control flow shows that the pre filter control block looks at hints from the message queues. Based on the hints a decision for a specific integrated filter is run. After the integrated filter runs, the post filter control block makes a decision for an encryption or decryption thread to run. The control then switches back to the other system tasks. To be scalable a system architecture would also require that these filters can be registered before the system is initialized. These filters share state information with each other along with the message queues and this execution model would be independent of the other threads in the system.

## 5.3 OS Support for Integrated Threads

The above architecture would aid the use of integrated threads. But the functionality should ideally reside inside the Operating System, to abstract away the implementation details from the user. This simplifies the design of other components of the system. However the user would specify which threads are integrated, as this could vary with application.

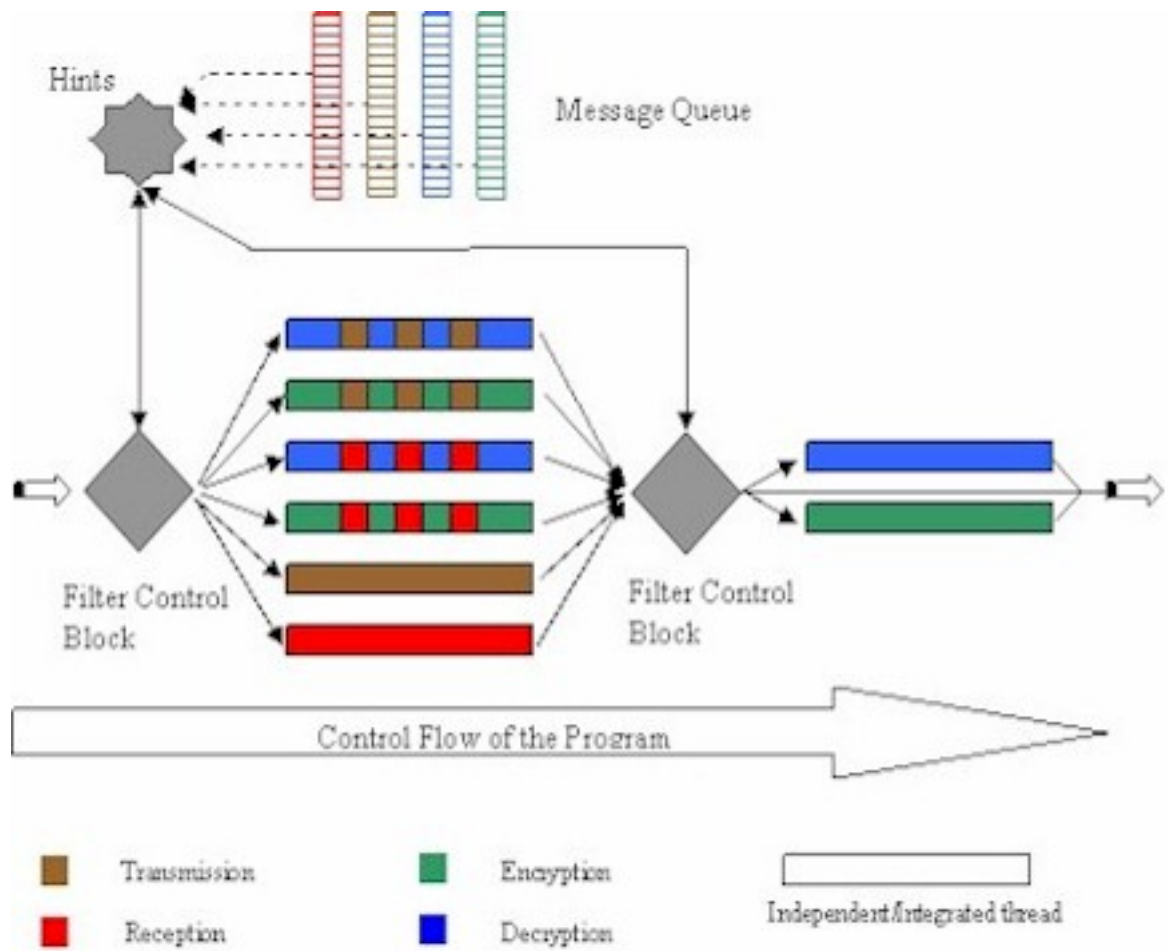


Figure 5.4: Filter Scheme



Figure 5.5: Sample processor workload

Hence the decision of scheduling the integrated thread must be built into the operating system, but the work to be performed by the thread must be provided to the user. However in a system where the MAC/CP functionality is fixed, the whole implementation can be incorporated into the Operating System.

Thus our actual filters are present as part of application code while the filter control blocks are pushed into the kernel.

The adoption of this architecture allowing the operating system to have control on running the integrated threads is indicated in the next section.

## 5.4 Case study of the architecture on AVRX

### 5.4.1 AVRX

AVRX is a fully pre-emptive, priority driven scheduler. The tasks run on one of sixteen priority levels. AVRX provides APIs for control on tasks, semaphores, message queues and timer management. The kernel, written in assembly is available as a library of functions. The modifications to support the above architecture are made partly in the library and partly in the system implementation using the APIs.

The next few subsections indicate the working of the operating system and the structure of various OS level functionalities is discussed.

## Task Scheduling

Tasks with the same priority execute on a cooperative basis using round robin scheduling. There is no time slice for a task. Task switching can occur in only one of the following scenarios:

- Block/Release of a semaphore
- Access to message queue ( send a message or wait for a message )
- Expiration of a timer.
- Task voluntarily relinquishes control or sleeps.

Task switching control exists primarily in two functions `_Prolog` and `_Epilog`. All tasks expect the one that is currently running are queued on a ready queue. In any of the above scenarios a call to `_Prolog` is made. `_Prolog` saves the process stack and updates task control information. The control now switches from the running task to the kernel. Based on which of the above scenarios made the call for scheduling, the respective OS level actions are performed. A final call to `_Epilog` picks the first task off the ready queue and restores its stack. The control now switches back to the application thread.

The semaphore, message queue and timer control operate in a sandwich mode between a `_Prolog` and `_Epilog` call. These control functions add or delete tasks in the run and wait queue which is later picked by the `_Epilog` call.

To summarize, the job of scheduling like identifying new tasks, blocking a task on a queue or releasing a blocked task are all part of the functionality of the itemized list above. Finally, the dispatching as indicated above is handled by the `_Epilog` call.

## Tasks

Although not absolutely necessary, a task typically is a routine with an entry, some initialization and then an endless loop. The endless loop typically involves blocking, or waiting, on a semaphore. That might be explicit when a call to block on a semaphore is made (`AvrXWaitSemaphore`), or it might be implicit in the case of when a call is made to the kernel when waiting on timers or message queues (`AvrXWaitTimer` or `AvrXWaitMessage`). These last two items actually block on a semaphore embedded in the timer or message data structure.

## Semaphore

Semaphores in AVRX are simply pointers to a task queue. Each time a task blocks on a semaphore, it dequeues itself from the run queue and places itself in the semaphore queue. Both mutexs and waiting semaphores are possible in this scheme.

## Message Queue

The message queue unlike in most operating systems is not bound to a process or a set of processes. Any process can utilize the message queue to send a message. It is up to the programmer that the correct task gathers the message. However control is maintained by the message control block which is a part of each message. This enables specific processes to interact. The message control block contains a semaphore, to which a process sending a message queues itself after sending the message. This process is released once the message has been received by the intended recipient. A pointer in the message control block points towards the actual message.

## Timer

Timers are handled in a special manner in AVRX. The timer interrupt handler makes kernel calls using the `_Prolog` and `_Epilog` calls and operates in the kernel mode. The timer queue again follows a special scheme. It implements a sorted list of adjusted timeouts such that the interrupt handler is only decrementing the first element at all times. When the first element times out, the code then signals the semaphore inside the `TimerQueue` element. If a task is waiting on that semaphore, then it is placed onto the `RunQueue`. At this point the time event for which the task was waiting for has occurred and it is ready to run.

### 5.4.2 Modifications to AVRX

#### Problem Statement

We have a simplified version of a TDMA scheme. There are transmission and reception threads that interact with transmit and receive buffers. These threads operate by

byte banging. TDMA control functions are not considered in the current scenario. Encrypt and decrypt functions operate on both encrypt/decrypt and transmit/receive buffers. Integrated threads which were analyzed previously form the filters which are selected depending on which of the buffers has data for consumption. The transmit and receive threads have exact timing requirements for their execution instance.

### Task Scheduling

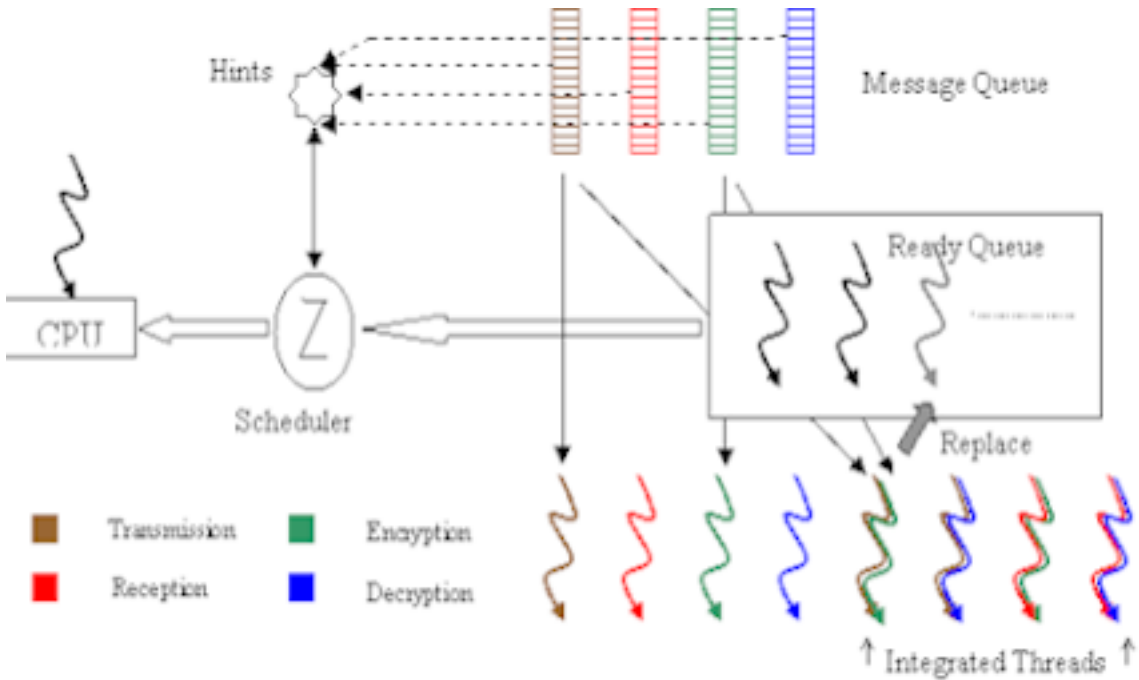


Figure 5.6: OS level view of the architecture

Figure 5.6 indicates how the scheduler works to run the integrated threads. The scheduler completely does the work of the filter control blocks. The scheduling function handled by the message queue is modified here use the hints from the message buffers to run our integrated threads.

The implementation allows the user to register the functions, in this case the

integrated and encryption/decryption threads. Registration consists of filling up an array of function pointers that now point to functions that act as filters. These registered user functions send messages on the same message queue. However the message control block has been enhanced to indicate the identity of the function that sent the message. The buffers are distinguished in this fashion. The previous implementation of the message queue blocked the calling function on a semaphore and passed the message to the function waiting for the specific message. The scheduler now intervenes and makes a decision on handling this message based on whether a transmission or reception is about to take place. Based on the decision, that, this message which needs to be encrypted or decrypted can be handled in conjunction with the transmission/reception thread, an update is sent to the application to run the integrated thread instead of the original transmission/reception thread. This is an index to the array of function pointers. This function/integrated thread runs based on the specified timing constraint of the transmit/receive threads.

The application now has the complete control on what work needs to be integrated with the transmission and reception threads. It accordingly registers functions using the array of function pointers provided to it by the OS.

The application has a high priority task that now receives each message that was posted, but the actual function that receives the message has now been determined by the OS. The application just calls the function pointer provided to it by the OS and the appropriate job is performed.

Here it must be noted that the message buffers that are used by the transmission and reception threads are global while the message buffers used by the other threads (eg: encry/decry) are part of the message queues. Hence the user functions have to perform their task (eg: encry/decry) and save the resultant array in the global buffers.

If jobs other than that of transmission and reception have to be performed, then the thread decision logic that is part of the OS has to be changed accordingly.



## Chapter 6

# Conclusion and Future Work

### 6.1 Conclusion

This thesis proposes a set of methodologies to secure communication in low level embedded devices using software thread integration. STI is used to replace traditional methods of communication protocol implementations like interrupts and busy-wait schemes. Efficient use of the communication channel is achieved enabling maximum possible throughput of transmission. STI frees up processor cycles which enables work to be performed in parallel. These free cycles can be reaped to perform cryptographic work and improve throughput of security schemes. Transformations necessary to enhance cryptographic support is also indicated. A software architecture is proposed that provides a structure for using integrated threads in a system. OS support for integrated threads is also discussed.

### 6.2 Future Work

1. Currently the focus has been on symmetric security schemes, particularly on a few stream and block ciphers. Hashing functions and asymmetric schemes have not been handled. These are CPU intensive compared to most symmetric functions. A hash needs to be attached only at the end of a large packet. Hence STI may enable us to start the hash process while the transmission takes place. There would be some

tradeoffs like a minimum message size and transmission speed. The applicability of hash or asymmetric schemes to low level embedded networks will also have to be considered. Another aspect of many hash/asymmetric functions is that their execution cycles are data dependent. Therefore STI concepts may have to be further developed before handling Hash/asymmetric schemes.

2. This experiment focused only on Atmega processors. Here, some operations like shifts have to be executed in loops. This causes execution cycles to be lost. Conversely, the Atmega128 processor has a 2 cycle multiply instruction, which helps speed-up math operations. Other processors may have different features or RISC instructions that can be exploited by STI to allow more parallel execution. Other mechanisms will have to be investigated, such as asynchronous transmission without the use of a SPI bus and how STI can be beneficial in such a scenario.
3. Modifications to STI concepts to allow extra loop cycles of the host thread to be executed before the guest is triggered would permit immediate encryption and transmission. Current STI methods start the execution of host and guest threads simultaneously in the integrated thread. When work by one of the threads finishes the other thread execution continues until it completes. This drops a constraint in the current encryption/decryption scheme wherein transmission/reception of data can occur only if the encryption/decryption occurred in the previous instance.
4. OS level support for STI can be evaluated further. A dynamic thread scheduling scheme can be derived which identifies cluster threads with identical deadlines and schedules an integrated thread. This would dynamically improve power savings in the system, depending on the frequency of these threads. This analogy is similar in some aspects to the Dynamic Voltage/Frequency scaling schemes [11], where the scheduler changes frequency of processor to allow threads to run for longer periods but still meet their deadline. Here rather than execution time increasing, the execution instant would be moved within the deadline.

A brief idea of how this algorithm would work is as follows.

- Determine if an integratable thread has been scheduled.
- Depending on factors such as system utilization by tasks, number of threads at the same priority level, etc determine a period  $\alpha$  that is a factor of the deadline

of the task.  $\alpha$  would give the time that the integratable thread can be postponed, without fear of missing the threads deadline.

- If within this period  $\alpha$  another task gets scheduled that is integratable with this thread, then the scheduler runs the integrated thread instead. A decision will have to be taken such that this integratable thread is schedulable within the deadline of the first thread.

5. Modifications to security protocols or MAC layer protocols may be identified to reap benefits of STI. Properties that allow work to be postponed or preponed may determine a scheme that is unique to a protocol.
6. Due to temporal determinism required by STI, some cryptographic algorithms have to be modified at the assembly level. This scenario can be seen when internal loops are data dependent, making timing analysis difficult. However many code structures are inter-related in a cryptographic algorithm. That is, a set of basic blocks may have a relation that can be exploited to bring timing determinism back to the code. This is currently done manually. These relations can be identified and STI can be enhanced to allow these code transformations to be performed by the compiler.

# Bibliography

- [1] IEEE Std 802.11. *Wireless LAN medium access control (MAC) and physical layer (PHY) specification*. 1997.
- [2] Paul C. Clements and Linda N. Nothrop. Software architecture: an executive overview. In Alan W. Brown, editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, pages 55–68. IEEE Computer Society Press, 1996.
- [3] A. Dean. Compiling for concurrency: Planning and performing software thread integration. In *The 23rd IEEE International Real-Time Systems Symposium*, December 2002.
- [4] Alexander G. Dean and Richard R. Grzybowski. A high- temperature embedded network interface using software thread integration. In *Second Workshop on Compiler and Architectural Support for Embedded Systems*, Washington, DC, October 1999.
- [5] Alexander G. Dean and John Paul Shen. Hardware to software migration with real-time thread integration. In *Proceedings of the 24th EUROMICRO Conference*, pages 243–252, Vasteras, Sweden, August 1998.
- [6] Alexander G. Dean and John Paul Shen. Techniques for software thread integration in real-time embedded systems. In *Proceedings of the 19th Symposium on Real-Time Systems*, pages 322–333, Madrid, Spain, December 1998.
- [7] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

- [8] P. Clements L. Bass and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
- [9] Rikard Land. A brief survey of software architecture. Technical report, Malardalen Real-Time Research Center, Malardalen University, Vasteras, Sweden, 2002.
- [10] A. Perrig, R. Szewczyk, V. Wen, D. Cullar, and J. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of MOBICOM*, 2001.
- [11] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, pages 251–259, Rome, Italy, July 2001.
- [12] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.
- [13] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems, April 1996.
- [14] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [15] A. Wall. Software architecture for real-time systems. Technical report, Malardalen Real-Time Research Center, Malardalen University, Vasteras, Sweden, May 2000.