Enhancing the AvrX Kernel with Efficient Secure Communication Using Software Thread Integration*

Prasanth Ganesan† and Alexander G. Dean
Center for Embedded Systems Research
Dept. of Electrical and Computer Engineering
North Carolina State University
alex_dean@ncsu.edu

Abstract

This paper presents methods to add efficient cryptographic support to low-performance embedded processors with embedded networks (e.g. sensor networks). Software thread integration (STI) is used to create efficient threads which can perform cryptographic operations during time-slice (TDMA) communication, eliminating most context-switching overhead. The AvrX kernel is enhanced to automatically select the most efficient threads based upon available work, saving processor cycles and power.

The results show that an STI-based implementation enables communication at higher rates while also performing more cryptographic work compared with traditional ISR (interrupt service routine) or busy-wait schemes. Significant performance improvements are found for both the RC4 and RC5 ciphers. First, STI enables cryptographic processing to occur during communication at a bit rate of $f_{cpu}/8$, which is not possible with an ISR approach. Second, cryptographic throughput at lower communication rates increases by up to 200% for both RC4 and RC5.

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. As a result, system designers often move the protocol functions to software to implement custom-fit protocols. In addition, network controller chips are typically expensive in comparison with generic 8- or 16-bit microcontrollers, so software implementations can cut costs. 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 growing more important 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. Conversion from cipher $\Rightarrow$ plain and plain $\Rightarrow$ cipher text are added to message transmission and reception times, so slow implementations of these schemes may delay communication, thus reducing throughput.

The medium access control (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 either with methods which share the processor (by the use of interrupts or some other dynamic control-flow transfer mechanism) or methods which monopolize the processor (e.g. busy-waiting). 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

* This work was supported by NSF CAREER award CCR-0133690
† Now at Symantec: prasanth_ganesan@symantec.com

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 [1, 2, 3, 4, 5] is a back-end 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 real-time threads are met by this method.

As indicated above, cryptographic schemes of security protocols may be bottle-necks in attaining maximum transmission throughput. The implementations of many protocols attempt to offset this, by using hardware accelerators for encryption or decryption. Wired Equivalent Privacy (WEP) of 802.11 [6] 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 [7] attempts to conserve energy and power by adjusting the length of transmission data by using a counter mode. 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 computationally intensive encryption part can be performed earlier while the XOR can be performed at run-time.

This paper reveals the benefits of using software thread integration for the purpose of secure communication. The constraints that are imposed by current software implementations can be eased 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 paper 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 paper is as follows. Section 2 discusses how software thread integration improves upon existing methods for sharing a uniprocessor efficiently. Section 3 shows how to use the integrated threads by modifying the software architecture of an embedded system with a real-time operating system and secure network communications. Section 4 presents the experimental methods and modifications made to AvrX. Section 5 presents results with analysis.

2. Cutting context switches

Traditional methods for sharing a uniprocessor break down for software implementing communication protocol controllers because processing is required very frequently (at least once per bit or once per byte). Control must be passed between threads extremely often, even when compared with typical fast context switch times (e.g. coroutines, interrupt service routines). This limits the idle time which can be shared by other threads. Without a mechanism for recovering this idle time, other work in the system will make no progress when the communication bus is active.

Software thread integration (STI) provides low-cost thread concurrency on general-purpose processors by automatically interleaving multiple (potentially real-time) threads of control into one. STI recovers fine-grain idle time efficiently for use by other threads in the processor.

Communication protocols implemented in software require support from the microcontroller to access the physical medium for transmitting and receiving bits. Depending on the protocol constraints and available hardware in the microcontroller, the software implementation maintains bit or byte level control on transmission.

Bit-banging schemes require software to operate on each bit individually and sequentially. 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 or sampling a specific port bit.

Byte-banging schemes use some hardware to serialize bytes or words onto the bus and deserialize bus bits back into bytes or words. This reduces the software processing overhead and also the frequency of context switches. The software controls low-level communication through control and data registers. A shift register is used to serialize the bytes while the formatting (e.g framing, parity bits) is performed in software or special hardware. The software acts byte-wise by interacting with a data register. Other software work is possible while the transmission/reception completes.

Most microcontrollers support serial communication with a UART (Universal Asynchronous Receiver/Transmitter). This peripheral is byte-oriented (framing each byte with start and stop bits, as well as an optional parity bit), so it cannot support arbitrary communica-

tion protocols with non-byte message. A more useful peripheral is the *Serial Peripheral Interface (SPI)*, which serializes and deserializes bytes but adds no framing or parity information. SPI status flags which indicate completion of byte transmission or reception; these flags can trigger interrupts.In this paper we use the SPI peripheral for communication because simplifies STI, as it reduces real-time I/O event rates by a factor of eight.

2.1. Traditional implementations

Two common methods for scheduling the processintg needed for communication are an interrupt based scheme and a busy-wait scheme. Details of both these methods and how they perform are discussed below.

2.1.1. ISR-based implementation The timeline shown in figure 1 shows the activity on the microcontroller as well as the peripheral device (SPI bus) for an interrupt-based approach. The main thread initializes the SPI data transfer and enables the interrupt. Then it continues to perform other tasks (e.g. encryption or decryption). The ISR runs periodically to interact with the SPI data register for transmission or reception.

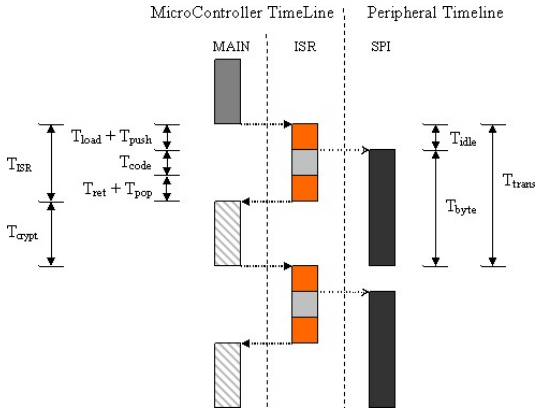

Figure 1. Timeline for ISR approach

We define the times for several activities associated with SPI communication:

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

Depending on overall context switch overhead and the latency between SPI transmission completion and SPI reloading, there can be a substantial drop in throughput. The detrimental effect on the throughput increases as the bus is driven at a higher speed for a given clock. This problem artificially limits processor and network performance.

2.1.2. 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. The timeline shown in figure 2 shows activity on the microcontroller and the SPI bus. No other thread can run while the data transmission is occuring. Additional timing terms must be considered:

T_{code} Total time taken for reading buffer and placing data on the SPI Data register

$T_{busy-wait}$ Time spent waiting for the transmission to complete

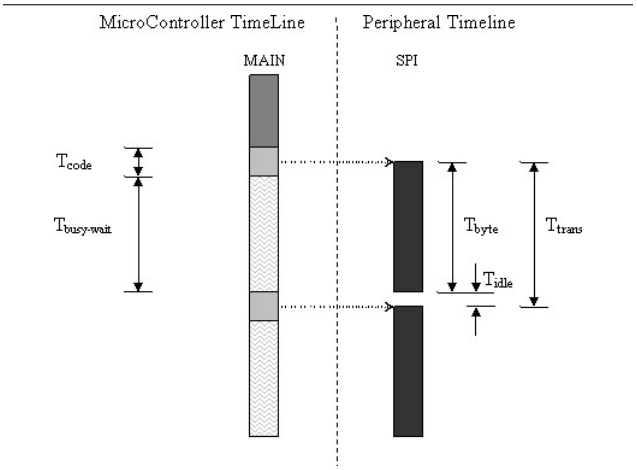

Figure 2. Timeline for busy-wait approach

2.2. Software thread integration

Software thread integration is a back-end compiler technique that provides fine-grain concurrency on generic processors by eliminating many context switches [2, 3, 4, 5,

8, 9]. By eliminating the need for special architectural features it allows generic, low-cost processors to replace more expensive specialized devices. STI reduces the clock speed needed to implement given functionality on a generic processor, saving money, power, and energy and simplifying design efforts.

STI increases instruction level parallelism and increasing concurrency. The latter is useful for implementing real-time applications which require frequent context switches.When a thread with internal, fine-grain real-time requirements (release times and deadlines on specific I/O instructions) are scheduled for execution on a sufficiently fast CPU, gaps will appear in the schedule of primary instructions, as illustrated by the white gaps in the black bar. These gaps are pieces of idle time which can be reclaimed to perform useful secondary work. STI recovers fine-grain idle time efficiently and automatically.

STI uses a control dependence graph (CDG, a subset of the program dependence graph [10]) to represent each function in a program. In this hierarchical graph, control dependence regions such as conditionals and loops are represented as non-leaf nodes, and assembly language instructions are stored in leaf nodes. Conditional nesting is represented vertically while execution order is horizontal. The CDG is well-suited for holding a program for STI because this structure simplifies analysis and transformation through its hierarchy.

STI involves moving primary code into the correct position within the secondary code for execution at the correct time. A tight target time range may fall completely within a secondary node, forcing movement down into that node or its subgraph. Before code motion the secondary and primary threads are statically analyzed for timing behavior, with best and worst cases predicted. During integration, programmer-supplied timing directives guide integration. Conditionals are padded to equalize their duration regardless of the path followed. Code is placed within a loop through splitting and peeling or guarded execution. Portions of primary and secondary loops which overlap are handled through loop fusion. Loop unrolling matches the idle time in the primary function loop iteration with the work in the secondary function loop iteration. Remaining loop iterations are executed by clean-up loops.

Modern high-performance CPUs and memory hierarchies have features such as branch prediction, out-of-order execution, prefetching and caching which greatly reduce the temporal determinism which STI requires when used for real-time applications. However, this is a non-issue. STI targets the large number of applications which neither need nor can afford these CPUs and memory systems. For perspective, in 2001 75% of the 8 billion microprocessors sold were four- and eight-bit units [11]. These microcontrollers run applications which are not computationally intensive (typically needing no more than 30 MHz clock rates), and do not need more parallelism or faster clock rates. They lack sophisticated microarchitectures and memory systems, and typically cannot afford them. However, reducing the execution cycles needed for an application is still important as that enables the use of simpler, less expensive processors which run at lower clock speeds, reducing power consumption and simplifying hardware development.

We have developed our optimizing post-pass compiler Thrint in C++ over the past five years. Thrint targets the AVR architecture, an 8-bit load/store architecture from Atmel. Thrint parses AVR assembly code, builds control flow and dependence graphs, measures idle time and timing jitter, evaluates register data flow, attempts to predict loop iteration counts, plans integration, pads timing variations in conditionals, moves and replicates code regions, unrolls, splits and peels loops, verifies timing correctness of integrated code and finally regenerates a file with flat assembly code.

2.2.1. 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 2.1.1 or 2.1.2. The timeline for the STI based scheme is shown in figure 3. The SPI code is reduced to interacting with the data register; the ISR overhead is removed, improving performance. The timing constraints are all pre-determined statically while performing thread integration. Furthermore, SPI activity can be scheduled back-to-back as there is no longer any interrupt latency to delay the transmission of the next character.

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. The context switch instructions or the busy-wait cycles are 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. Second, 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.

To summarize, STI enables longer sleep times for the microcontroller and faster data throughput rates at the same clock speed. In addition, there is 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.

2.2.2. Costs of STI implementation There are several costs associated with the integrated code. First, all conditionals are padded to the worst-case duration, potentially slowing down the code. Second, code size increases due to replicated into conditionals and loop transformations(splitting, peeling, fusion and copying). In general, the total impact of code expansion is minor, considering the savings in clock speed or increased throughput. Third, the code is tailored specifically for a fixed clock speed and the timing of an instruction set architecture implementation. Changes in either of these requires re-integration.

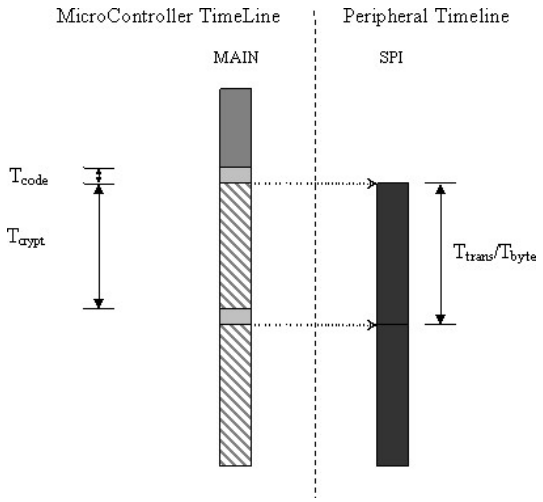

Figure 3. Timeline for STI approach

3. Software architecture

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. Furthermore, the work of one task may be postponed to occur interleaved (integrated) with another. With concurrency considerations being affected, there is the need to modify the software architecture to meet this demand.

3.1. Basic architecture

The software architecture is a conjunction of pipe-filter and layered styles. A pipe filter style [12, 13, 14, 15] focuses 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. 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). Most communication 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 protocol data units (PDUs). A software architecture can have multiple views [15, 16, 17]. Thus the layered architecture may have a different view when seen with the aim of design [18]. 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 4 provides an overview of the general architecture.

3.2. Modifications to software architecture

The medium access control/communication protocol (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 including the encryption scheme. The MAC/CP layer are composed of an input pipe which holds the PDU from the higher layers that it receives for transmission. The output pipe is 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 are also 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 are the filters. The transmission and reception threads are responsible for bit or byte banging the data in the cipher text pipes.

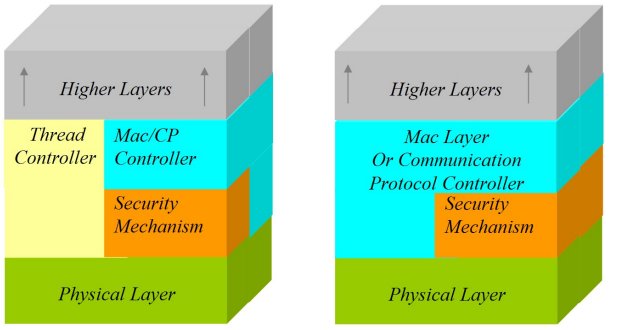

Figure 4. Original and modified architectures

Depending on the indication from the Mac/CP controller and the data present in the pipe, a decision block determines which specific filters are active. 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 is called the pre-filter and post-filter block depending on when this control function is invoked.

Figure 4 shows where the thread control functionality resides 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, but 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.

3.3. Thread controller

The security and MAC/CP layers contain multiple filters and pipes. Eight filters connect four pipes. The four pipes transfer data to and from the application (higher layer) and network (physical layer). The filters fall into two categories: discrete (encryption, decryption, transmission and reception) and integrated (encryption with transmission, encryption with reception, decryption with transmission, and decryption with 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.

3.3.1. Pre filter control block This block decides which thread (filter) has to run based on various criteria: the 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, and how long the work can be postponed or advanced. Specific applications may have other implementation-specific decisions which must be considered.

3.3.2. 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 if the data is needed urgently, run the encryption or decryption thread that clears the pipes; determine if the higher layers filters can use unencrypted data, and determine if work can be postponed so that an integrated thread can run the next time.

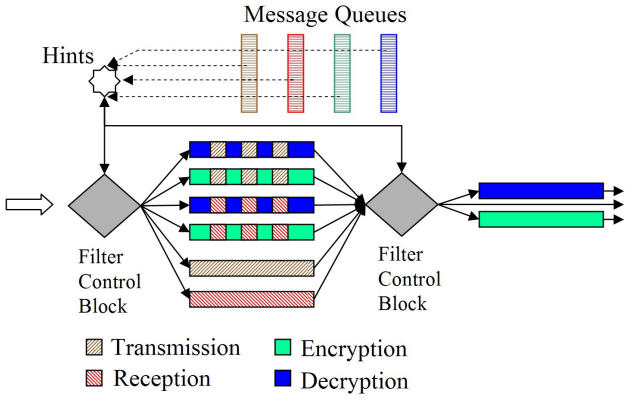

Figure 5. Filter scheme

Figure 5 shows a block diagram of the thread controller system interacting with the pipes and filters. Another view as a processor timeline can be seen in Figure 6. 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.

4. Experiments

We implement a simple time slice (TDMA) network communication 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. Encrypt and decrypt functions operate on both encrypt/decrypt and transmit/receive buffers. Integrated threads 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.

Figure 6. Sample processor timeline

4.1. Cryptographic algorithms

The algorithms chosen for the experiment are widely used algorithms, which are used in many security protocols and products. RC4 is used in WEP [6] and RC5 has been suggested as a good algorithm for sensor networks [7]. The algorithms have been chosen because of their popularity and applicability to embedded systems. The cryptanalytic strength of the algorithms is not a focus of this experiment or analysis.

4.1.1. RC4 RC4 [19] 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.

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 checks the buffer for data and writes it into the SPDR register. This write into the SPDR register is constrained temporally. STI makes this write occur at exact instants by its code transformations.

4.1.2. RC5 RC5 [19] 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.

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.

4.2. AVR processor

The target architecture is AVR from Atmel, and features 8 bit native word size, 32 general-purpose registers, and limited support for 16 bit operations. The Atmega 128 processor [20] is inexpensive (about $3 in volume) and provides 128 kilobytes of Flash program memory, 4 kilobytes of onboard data SRAM, no caches, up to 64 kilobytes of off-chip SRAM and numerous peripherals. The CPU core features a two-stage pipeline; most instructions take one cycle, but some take more (branches, multiplies, calls, returns, loads and stores). The C compiler used is GCC 3.2 [21].

AVR microcontrollers feature a SPI peripheral on-chip. The SPI bus can operate at different data rates, which are a fraction of the CPU clock. With a CPU clock speed of f_{cpu}, the SPI bus operates at a frequency of $f_{cpu}/2^n$ where n can be set to integers from 1 to 7, resulting in SPI speeds of $f_{cpu}/2$ to $f_{cpu}/128$.

The SPI speed of $f_{cpu}/16$ is studied as a break even point between the STI and ISR-based schemes. Above 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 fall.

4.3. AvrX kernel

AvrX is a fully pre-emptive, priority driven scheduler [22]. AvrX provides APIs for control of 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.

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 when a semaphore is blocked or released, the message queue is accessed, a timer expires, or a task voluntarily relinquishes control or sleeps. Task switching control exists primarily in two functions _Prolog and _Epilog. _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.

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.

4.4. Modifications to AvrX

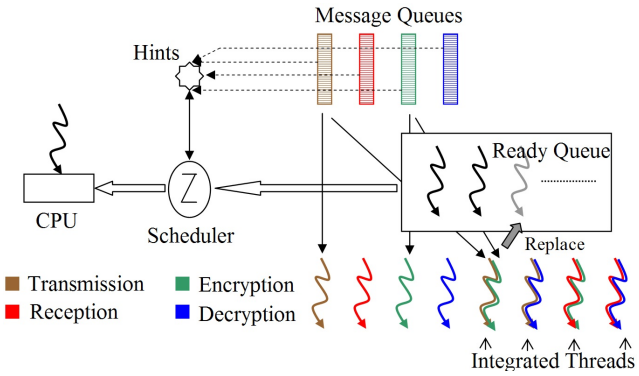

Figure 7. OS level view of the architecture

Figure 7 indicates how the scheduler coordinates information to run the integrated threads. The scheduler, indicated by the circle in the diagram, uses hints based on information in the message queue to determine whether to replace a thread in the ready queue with an integrated thread. It then decides which thread from the ready queue to run.

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: encrypt/decrypt) are part of the message queues. Hence the user functions have to perform their task (eg:encrypt/decrypt) and save the resultant array in the global buffers.

5. Results and analysis

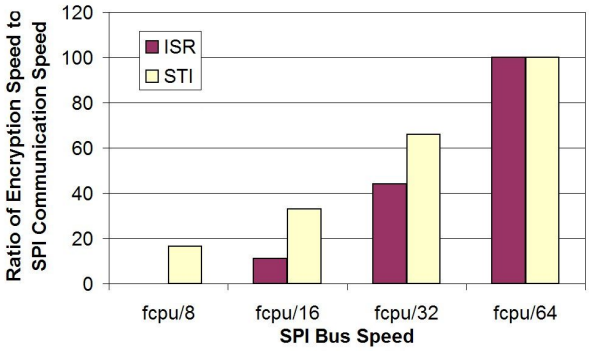

Figure 8. Speed ratios for RC4 + SPI communication implementations

This section examines the performance of the integrated threads. We evaluate the ratio between the time for encrypting (or decrypting) a byte and the speed of communicating a byte on the SPI bus. A ratio of less than 100% indicates the cryptography cannot keep up with the communication, so some work must be done before transmission

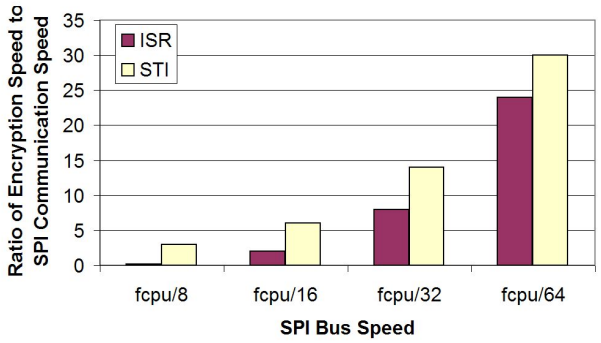

Figure 9. Speed ratios for RC5 + SPI communication implementations

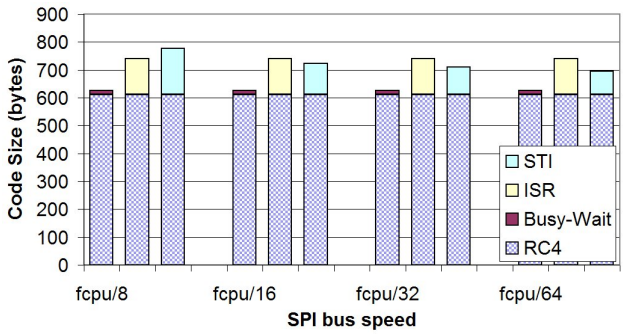

Figure 10. Code memory expansion for RC4 + SPI communication implementions

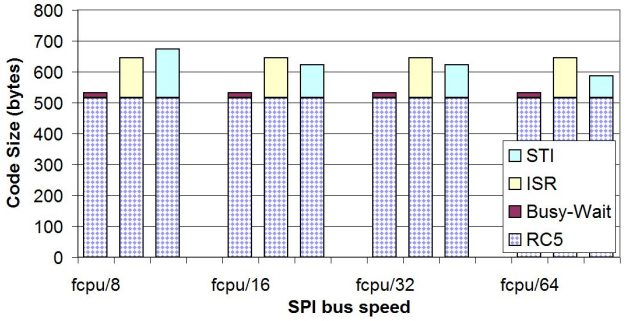

Figure 11. Code memory expansion for RC5 + SPI communication implementations

or after reception. Figure 8 shows the increase in performance of the RC4 algorithm with an STI scheme in comparison with the other schemes. As there is no interleaved work in a busy-wait scheme, any extra work enabled by STI is a benefit. The throughput of the RC4 algorithm with STI is 50% and 200% higher than an ISR scheme at bus speeds of $f_{cpu}/32$ and $f_{cpu}/16$, respectively. For lower data rates ($f_{cpu}/64$ and below), both the ISR and STI schemes have enough cycles for the RC4 algorithm to keep up with the SPI communication. At this point the extra free cycles freed by an STI based scheme can be used for handling other MAC/Communication Protocol Controller functionalities. At higher data rates ($f_{cpu}/8$), the STI implementation is able to encrypt at 16% of the communication rate, while the ISR approach is unable to do any such work.

RC5 requires almost 1600 cycles to encrypt a byte, while RC4 takes only 350 cycles. As a result, none of the approaches allow encryption or decryption to keep up with communication at the speeds examined. However, as shown in figure 9, STI still improves cryptographic performance over using ISRs, with improvements of 25% at $f_{cpu}/64$, 75% at $f_{cpu}/32$ and 200% at $f_{cpu}/16$. At $f_{cpu}/8$ the ISR-based method will not work, as the overhead of interrupt response fully loads the processor, while STI-based encryption is able to run at 3% of the communication speed.

Figures 10 and 11 show 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.

We now consider the impact of STI on variations of these experiments and other applications. First, the benefit of STI comes from eliminating interrupt overhead. If the encryption and decryption times are large compared with the communication time, there will be a significant amount of this overhead. The larger the relative overhead, the greater the impact of STI will be. Modifications to the cryptographic parameters (e.g. rounds, key length) or algorithm will change STI's impact according to the relative overhead. Second, STI pads away timing uncertainty in conditionals, increasing average execution time toward the worst-case. Loops of unknown duration in the cryptographic code will require the use of less efficient integration techniques, reducing the performance enhancement and increasing code size further. Finally, as the communication thread is hard real-time, the processor used must have fully predictable timing. This requirement is not much of an issue for low-end processors such as the AVR, but becomes a problem when instruction processing throughput requirements lead

to caches, deep pipelines, speculative and out-of-order execution and branch prediction.

6. Conclusions

This paper proposes a set of methods 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. STI frees up processor cycles which enables work to be performed concurrently. These free cycles can be reaped to perform cryptographic work and improve throughput of security schemes. A software architecture is proposed that provides a structure for using integrated threads in a system. OS support for integrated threads is also discussed.

References

[1] A. G. Dean and J. P. Shen, "Hardware to software migration with real-time thread integration," in *Proceedings of the 24th EUROMICRO Conference*, Vasteras, Sweden, August 1998, pp. 243–252.

[2] ——, "Techniques for software thread integration in real-time embedded systems," in *Proceedings of the 19th Symposium on Real-Time Systems*, Madrid, Spain, December 1998, pp. 322–333.

[3] A. G. Dean and R. 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.

[4] B. Welch, S. Kanaujia, A. Seetharam, D. Thirumalai, and A. G. Dean, "Extending sti for demanding hard-real-time systems," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, November 2003, pp. 41–50.

[5] N. J. Kumar, S. Shivshankar, and A. G. Dean, "Asynchronous software thread integration for efficient software implementations of embedded communication protocol controllers," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, June 2004.

[6] I. S. 802.11, *Wireless LAN medium access control (MAC) and physical layer (PHY) specification*, 1997.

[7] A. Perrig, R. Szewczyk, V. Wen, D. Cullar, and J. Tygar, "SPINS: Security protocols for sensor networks," in *Proceedings of MOBICOM*, 2001. [Online]. Available: citeseer.nj.nec.com/perrig02spins.html

[8] A. G. Dean, "Compiling for concurrency: Planning and performing software thread integration," in *Proceedings of the Sixth Workshop on Interaction between Compilers and Computer Architectures*, Cambridge, MA, Feb 2002.

[9] A. G. Dean and J. P. Shen, "System-level issues for software thread integration: Guest triggering and host selection," in *Proceedings of the 20th Symposium on Real-Time Systems*, Scottsdale, Arizona, December 1999, pp. 234–245.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, July 1987.

[11] E. Nisley, "Rising tides," *Dr. Dobb's Journal*, vol. 346, Mar 2003.

[12] P. L. Bass and R. Kazman, *Software Architecture in Practice*. Addison Wesley, 1998.

[13] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] M. Shaw and P. Clements, "A field guide to boxology: Preliminary classification of architectural styles for software systems," April 1996. [Online]. Available: citeseer.nj.nec.com/shaw96field.html

[15] R. Land, "A brief survey of software architecture," Malardalen Real-Time Research Center, Malardalen University, Vasteras, Sweden, Tech. Rep., 2002. [Online]. Available: citeseer.nj.nec.com/land02brief.html

[16] A. Wall, "Software architecture for real-time systems," Malardalen Real-Time Research Center, Malardalen University, Vasteras, Sweden, Tech. Rep., May 2000. [Online]. Available: citeseer.nj.nec.com/wall00software.html

[17] P. C. Clements and L. N. Nothrop, "Software architecture: an executive overview," in *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, A. W. Brown, Ed. IEEE Computer Society Press, 1996, pp. 55–68. [Online]. Available: citeseer.nj.nec.com/clements96software.html

[18] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds. Singapore: World Scientific Publishing Company, 1993, pp. 1–39.

[19] B. Schneier, *Applied Cryptography*. John Wiley & Sons, 1996.

[20] *Atmega 128: 8-Bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*, Atmel Corporation. [Online]. Available: http://www.atmel.com/dyn/resources/prod documents/doc2467.pdf

[21] *avr-gcc 3.2*. [Online]. Available: http://www.avrfreaks.net/-AVRGCC/index.php

[22] L. Barello, *AvrX Real Time Kernel*. [Online]. Available: http://www.barello.net/avrx