Matthew W. Pagano

Computer Security Architecture

Spring 2008 Semester

Dr. Masson / Dr. Vasconcelos

PSEUDORANDOM NUMBER GENERATORS ON ATMEL AVR AT90USB1287 MICROCONTROLLERS

Summary

The purpose of this report is to provide exhaustive details of my Computer Security Architecture project this spring 2008 semester. Ultimately, the goal has been to prepare a document that an undergraduate student can read to increase his/her comprehension of the Atmel AVR AT90USB1287 microcontroller and its hardware components, the Assembler programming language, random number generation, pseudorandom number generation, and testing methods to determine levels of randomness. This report outlines how to program a random number generator (RNG) on the AT90USB1287 using the on-board thermometer, as well as two pseudorandom number generators (PRNGs) on the AT90USB1287. One of the PRNGs is a Linear Feedback Shift Register (LFSR) and is designated as the "weak" PRNG. The other PRNG is the Advanced Encryption Standard (AES) algorithm and is designated as the "strong" PRNG algorithm. After programming the RNG, weak PRNG, and strong PRNG on the AT90USB1287, this report details the results of randomness tests offered by the National Institute of Standards and Technology (NIST) on both PRNG algorithms. The purpose of this is to demonstrate specifically what it means to have a cryptographically secure PRNG algorithm.

Introduction, Goals, and Objectives

The central motivation of this project is to demonstrate proper implementation of pseudorandom number generators (PRNGs). It will be assumed that the students of this project have some computer science background, but this should be the only prerequisite. During the completion of this project, students will be expected to research the importance of random and pseudorandom number generation within modern computing. Students will be exposed to the difficulties of obtaining sufficient amounts of true random data. This will highlight the realistic need for pseudorandom number generation.

From a high-level perspective, students will analyze two PRNG algorithms. The first PRNG algorithm, the Linear Feedback Shift Register (LFSR), will be simple, but will not generate data that is "random enough." The first PRNG algorithm will not be cryptographically secure in order to demonstrate the significance of this term. The goal is to demonstrate the pitfalls of a PRNG algorithm whose output an attacker can easily differentiate from true random data. This first PRNG algorithm will be referred to as the "weak PRNG algorithm."

The second PRNG algorithm, the Advanced Encryption Standard (AES), will be complex, will produce data that appears to be very random, and will be cryptographically secure. The second PRNG algorithm will be referred to as the "strong PRNG algorithm." Students will contrast these two PRNG algorithms in

order to understand the theoretical and practical differences between weak and strong PRNG algorithms. Students will learn that even supposedly strong PRNG algorithms must be implemented carefully to avoid producing predictable data.

From a low-level perspective, students will be required to code the weak PRNG algorithm in Assembler and execute it on the Atmel AVR AT90USB1287 microcontroller. This will provide students will an excellent opportunity to learn the finer components of Assembler. Coding a secure PRNG algorithm will require a vast knowledge of algorithm analysis, Assembler programming tactics, and debugging on the AT90USB1287. Students should leave the course having sharply increased their skills in all of these areas.

Lastly, students will be required to evaluate their results against the National Institute of Standards and Technology (NIST) suite of randomness tests. The intended result is that pseudorandom data produced by the strong PRNG algorithm will score much higher than that of the weak PRNG algorithm. This should confirm the need for a PRNG algorithm to be thoroughly tested to ensure that it produces strongly random data.

Background and Related Theory

Pseudorandom number generators (PRNGs) are commonly encountered throughout many areas of computer science. The demand for PRNGs results from a need for randomness in numerous applications. Random data is essential for many gaming applications, cryptographic protocols, statistical programs for random sampling, prime numbers generators, sorting algorithms, and so on. Unfortunately, true random data (unpredictable data not produced by deterministic formulas) is often expensive and difficult to obtain. Sources of true randomness that have been used in the past include temperatures of hardware components, feedback from noisy diodes, formulas based on the time of day, keystroke or mouse movement, and nuclear decay. The reality of these forms of randomness is that they often require significant resources to generate relatively low amounts of random data. In short, random data tends not to be cost-effective.

This downside of random data has lead to the widespread use of pseudorandom data. The applications that need randomness often require the random data to be generated quickly and in large amounts. Accordingly, pseudorandom algorithms are designed to take as input small amounts of true random data and then apply various PRNG algorithms to produce high volumes of pseudorandom data. These pseudorandom algorithms are of course deterministic in nature, so the pseudorandom data that is produced is not truly random. However, if the following two conditions are met, the pseudorandom data will be “random enough” for most applications:

1) The true random string provided as input is sufficiently long and comes from a highly unpredictable source. This true random string needs to be regenerated each time the PRNG algorithm is used to prevent predictable patters in successive generations of pseudorandom data.

2) The pseudorandom algorithm has been confirmed to produce pseudorandom data that an attacker bounded by polynomial space and time cannot differentiate from true random data.

When evaluating sources of true random data and pseudorandom algorithms, the developer is advised to use tests offered by the National Institute of Standards and Technology (NIST). The NIST website offers a suite of tests that evaluate and provide quantifiable statistics on the data's level of randomness. NIST has certified that these tests provide an accurate estimation of how random the data is. For example, it is reasonable to expect that true random binary data should have approximately the same number of ones and zeros. One of the NIST randomness tests is therefore to determine if the total number of ones and zeros versus the length of the binary string falls within an acceptable range of randomness.

Solution Implemented and Justification

The preceding sections have provided a detailed overview of the solution implemented and the justifications for the decisions made. What remains is to discuss which PRNG algorithms were selected for the weak and strong algorithms (LFSR and AES, respectively), and justify these selections. The following two lists of points provide these justifications.

<u>Weak PRNG: Linear Feedback Shift Register (LFSR)</u>

- Involves manipulations of bits (often with the EOR operation) within 8-bit or 16-bit registers, so it is ideal for the Atmel microcontroller
- As a result, LFSRs are readily implemented in hardware, which works well in this project; this will also demonstrate to the students what it means to be easily implemented in hardware, which will solidify the point for them when they hear that term with respect to algorithms such as DES
- Output is a stream of ones and zeroes, which can be readily input into the NIST suite of randomness tests
- LFSRs are used in the real world for such applications as GPS, digital broadcasting and communications systems, and some gaming consoles, so the student would not be learning an antiquated algorithm that is solely used for pedagogical purposes
- Relatively simple to understand and code, so it will provide a good starting point for the students to learn and build their confidence (as opposed to the strong PRNG, which will require more time to understand)
- Efficient and fast, so time will not be a significant issue when running this PRNG
- Involves concepts of polynomials to map sequences of transformations and concepts of periodicity, so it will be a good review for the students of the mathematics involved in cryptography

- Involves a caveat of which to be mindful during implementation (ensure that the sequence does not reach an all-zero state, after which point there will be no change); this will be an excellent demonstration to the students of how secure PRNGs can become insecure if not implemented properly

- Outputs of LFSRs are linear, which means that they are susceptible to cryptanalysis; this will perfectly demonstrate to the students the difference between this weak PRNG and a strong PRNG (cryptographically speaking)

- Note: this linear weakness has been corrected in some implementations of LFSR, but the stronger implementations will not be used in order to highlight the difference between a weak and strong PRNG

Strong PRNG: AES

- Considered to be one of the most cryptographically secure (if not the most cryptographically secure) PRNGs available

- Thus, it will do very well on NIST's suite of randomness tests, which will demonstrate the differences to the students between weak and strong PRNGs

- Will demonstrate to the students the need to use openly and freely available cryptographic algorithms that have been thoroughly tested, as opposed to relying on obscurity by using lesser-known algorithms

- Will demonstrate the level of complexity that students can expect from top-tier block ciphers

- Because the students will not be coding AES themselves as they will with LFSR (but will instead get the code from a site like http://point-at-infinity.org/avraes/ as referenced in http://www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/MerkleMicro.pdf), they will learn how to securely implement code from an external source. There will surely be implementation errors associated with this process, so the students will gain experience with dealing with these types of issues.

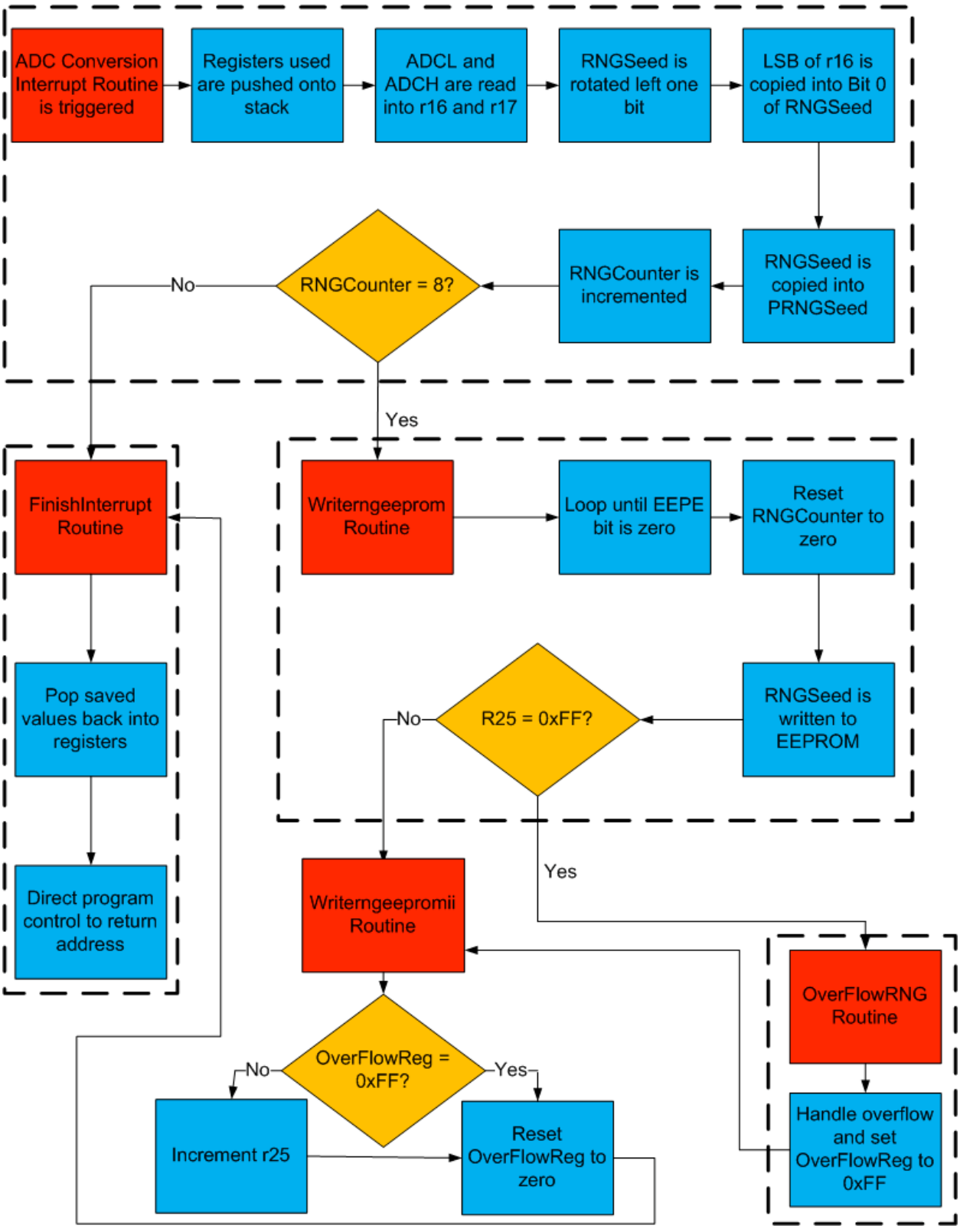
- As a short side assignment, students should be expected to write a short report on the history of AES; this will familiarize them with the modern process of developing cryptographically secure PRNGs, as the story of AES does this exceedingly well

Detailed Description of the Implementation with Flowcharts and Code Snippets

The implementation of this project can be divided into four sections: 1) RNG; 2) Weak PRNG (LFSR); 3) Strong PRNG (AES); 4) suite of the randomness tests offered by NIST and subsequent results.

RNG

A flowchart of the RNG is shown below:



The basis of the RNG is to read data from the thermometer on board the AT90USB1287 microcontroller. The on-board thermometer is a resistor that displays various levels of resistance that are directly

affected by the temperature of the microcontroller. The resistance of this resistor increases as the temperature decreases, which is referred to as a Negative Temperature Coefficient (NTC). This resistor is consequently labeled as a temperature-sensitive resistor, or more commonly as a thermistor. Because the values read from the thermistor fluctuate in proportion to the temperature of the microcontroller, the thermistor can be effectively used as a thermometer for the microcontroller. The assumption behind the RNG is that the temperature of the microcontroller, or at the very least the least significant bits of the temperature, will be consistently fluctuating and sufficiently unpredictable to serve as a source of randomness. Should this premise be correct, RNG data can be obtained by reading values from the thermistor.

Data can be read from the thermistor using the on-board 8-channel Analog Multiplexer. The 8-channel Analog Multiplexer is connected to 8 voltage inputs from one of the microcontroller's ports. These voltage inputs are labeled ADC0, ADC1, ADC2, ADC3, ADC4, ADC5, ADC6, and ADC7. The thermistor is the ADC0 voltage input. ADC0 sends analog voltage data to the 8-channel Analog Multiplexer, which then sends this analog data to the on-board Analog-to-Digital Converter (ADC). The ADC converts the analog data from the Multiplexer into 10-bit values. This system of the ADC and 8-channel Analog Multiplexer is referred to as the ADMUX. The digital data from the ADMUX is the sent to the ADC Data Register. Because the ADC generates 10-bit values, the ADC Data Register is comprised of two 8-bit registers, labeled ADCH (ADC High) and ADCL (ADC Low). ADCL contains the least significant 8 bits of the ADC output; ADCH contains the 2 most significant bits of the ADC output in bits 0 and 1 of the ADCH register. The programmer can then read data from this system through the ADC Data Registers (ADCH and ADCL). In the specific case of the thermistor, temperature data can be read by configuring the ADMUX to read data from ADC0 (the voltage input of the thermistor) and then reading in data from the ADC Data Registers (ADCH and ADCL).

The remainder of this section will be dedicated to a thorough explanation of the RNG code. Code snippets will be shown first and will be followed by an evaluation of what these snippets signify.

```
.org 0x0000
rjmp main

.org 0x003A
call ADC_ISR
```

The org directive in Assembler is an abbreviation for "origin" and indicates to the compiler to which address within the code segment to direct the flow of execution for a specific routine. In this case, the first line shows that the program will start at address 0x0000 within the code segment. This is typically done because the processor begins execution at address 0000. Address 0x0000 is therefore referred to as the "reset vector". The next line is a relative jump to the label of the "main" routine to begin the program.

The last two lines of the code snippet above relate to the ADC interrupt routine. The purpose of an interrupt is to stop program execution in order to perform a specified routine. After the interrupt routine has completed, it is required that program control be transferred back to the location where

program execution was originally stopped. This requirement necessitates the use of the stack in order to store the address to which program execution must be returned after the interrupt routine has completed (named the return address).

The ".org 0x003A" informs the compiler of the address of the ADC interrupt vector. As discussed above, because the processor begins execution at address 0000, address 0x0000 is referred to as the "reset vector". The vectors that follow the reset vector within program space are addresses 0x0001, 0x0002, 0x0003, etc. These vectors are labeled "interrupt vectors". If an interrupt is triggered, program control is transferred to the position in program space given by interrupt vector that corresponds to the type of interrupt triggered. The specific interrupt vectors that correspond to each type of interrupt are hardware-dependent. For the AT90USB1287, the programmer can use Atmel's usb1287def.inc file to find this information. The usb1287def.inc file contains the following line that is pertinent to the ADC interrupt vector:

```
.equ    ADCCaddr        = 0x003a        ; ADC Conversion Complete
```

This line indicates that the address of the ADC Conversion interrupt vector is 0x003A. This is why the .org directive shown above specifies 0x003A as the address to which to direct program control in the event of an interrupt. The line that follows uses a call statement to give the compiler the name of the routine to execute if this interrupt is triggered. As a result, if the ADC Conversion interrupt is triggered, the ADC_ISR routine will be executed and then program control will be redirected to the return address. More information on the ADC_ISR routine will be given below.

It is important to note that the command immediately following the org directive to the reset vector (.org 0x0000) has to be a relative jump (rjmp main) in order to jump over the interrupt vectors that are listed below (in this case, the ADC Conversion interrupt).

These lines of code begin the ADC_ISR routine. The ADC_ISR routine will be called whenever there is an interrupt triggered by a reading from the thermistor:

```
push r16
in r16, SREG
push r16
push r17
```

Recall that an interrupt stops program execution, performs a specific routine, and then transfers program execution to the original location where it was stopped. It is very important to ensure that the state of the program that is not related to the interrupt is not modified by the interrupt routine. Most notably, the interrupt should not permanently modify any registers (including the status register) that the interrupt uses. As a result, it is common practice to push the values of any registers used by the interrupt onto the stack before performing any other actions. These values remain on the stack until the interrupt is completed, after which these values are popped off the stack into the corresponding registers. When program control is transferred back to the return address, the registers (including the status register) have the same values as they did immediately before the interrupt was triggered.

Because the stack is a Last-In-First-Out (LIFO) operation, values must be popped off in the reverse order as they were pushed on. This is why the last lines of the ADC_ISR routine are the following:

```
pop r17
pop r16
out SREG, r16
pop r16
reti
```

Note that reti is the command used to indicate that an interrupt has finished and that program control should be transferred back to the return address.

```
lds r16, ADCL
lds r17, ADCH
```

This indicates that the value of ADCL (least significant eight bits) is being read into the r16 register and the value of ADCH (most significant two bits) is being read into the r17 register.

```
lsl RNGSeed
```

RNGSeed is the register that holds the eight bits of randomness that have been gathered so far by the RNG. Every time the ADC Conversion interrupt is called, the bit of randomness that was least recently added to RNGSeed (i.e., the oldest) is removed, and new bit of randomness is extracted from the thermistor and added to RNGSeed. This is analogous to a First-In-First-Out (FIFO) method. Bits of randomness are added to RNGSeed from right to left using the lsl command, as shown in the code snippet above. In other words, if the bits of a register are labeled from left to right at Bit 7, Bit 6, Bit 5, Bit 4, Bit 3, Bit 2, Bit 1, and Bit 0, a new bit of randomness is added to RNGSeed by shifting all of the bits over one bit to the left. Bit 6 is moved to Bit 7, Bit 5 is moved to Bit 6, Bit 4 is moved to Bit 5, Bit 3 is moved to Bit 4, Bit 2 is moved to Bit 3, Bit 1 is moved to Bit 2, Bit 0 is moved to Bit 1, and a zero is placed in Bit 0. This effectively pops off Bit 7, slides everything over to the left one bit, and places a zero in Bit 0, which is what the lsl command does. The new bit of randomness extracted from the ADC Conversion interrupt can then be added to Bit 0, as will be described below.

```
andi r16, 0b00000001
andi RNGSeed, 0b11111110
or RNGSeed, r16
```

The above code snippet is the algorithm used to add the newest bit of randomness to Bit 0 of RNGSeed. Recall that r16 contains the 8 least significant bits of the 10-digit digitized value of the latest thermistor reading. Empirical data from this study showed that the readings of the thermistor tended to be largely homongenous. This was quite detrimental to this study because of the need for highly unpredictable data to be used as the RNG. In order to remedy the homogeneity observed from the thermistor, only the least significant bit (LSB) of the readings from the thermistor was used in the RNG. The assumption was that the LSB would be the bit that would be fluctuating most frequently and unpredictably. As a result, the goal of this algorithm is to copy Bit 0 of r16 (the LSB of the data read from the thermistor) into Bit 0 of RNGSeed (now that the lsl command has made RNGSeed ready to accept the newest bit of randomness into its Bit 0).

The algorithm shown above makes use of the following four facts: 1) Performing an AND operation with any binary digit and zero always yields zero; 2) Performing an AND operation with any binary digit and one always yields the original binary digit; 3) Performing an OR operation with any binary digit and zero always yields the original binary digit; 4) Performing an OR operation with any binary digit and one always yields one. The first step of the algorithm shown above, "andi r16, 0b00000001", zeroes Bits 1-7 of r16 and allows Bit 0 of r16 to keep its original value. The second step, "andi RNGSeed, 0b11111110", allows Bits 1-7 of RNGSeed to keep their original values and zeroes Bit 0. Because Bits 1-7 of r16 are zero and because Bits 1-7 of RNGSeed are their original values, performing an OR operation on RNGSeed and r16 and storing the result in RNGSeed ensures that Bits 1-7 of RNGSeed retain their original values. Because Bit 0 of r16 is its original value and Bit 0 of RNGSeed is zero, performing an OR operation on RNGSeed and r16 and storing the result in RNGSeed ensures that Bit 0 of RNGSeed takes on the value of Bit 0 of r16. This was indeed the goal of this algorithm. Note that in order to ensure that the result of the OR operation is copied to RNGSeed, RNGSeed had to be the first operand in the OR statement, as shown in the last line of this algorithm.

mov PRNGSeed, RNGSeed

Because PRNGSeed is the first operand of the mov command, this command copies the contents of the RNGSeed register into the PRNGSeed register. This command is present within the ADC Conversion interrupt so that PRNGSeed always contains the latest version of RNGSeed. As will be discussed later, when the PRNG functionality of this program is called, the PRNGSeed register is used as the seed to the PRNG algorithm. Due to this, it is vital that PRNGSeed be updated immediately after RNGSeed is updated so that the PRNG algorithm can make use of the latest randomly generated numbers.

inc RNGCounter

As will be described later, in order to generate large volumes of random data for NIST testing, bytes of data generated from the ADC Converstion interrupt are written to EEPROM memory within the interrupt. EEPROM is a component on the AT90USB1287 microcontroller that functions as a type of memory. EEPROM is used in this code to record the random data that is generated by the thermistor to be evaluated later by the NIST tests. Recall, however, that each run of the interrupt only generates one bit of randomness, not one byte. If the RNGSeed register were copied into EEPROM every time the interrupt was run, patterns of bits would emerge. This can be observed from the following example. Suppose that at any given time, RNGSeed contained the following byte: 10010101. Suppose that the next run of the interrupt generated a 0 for the random digit. RNGSeed would then become 00101010. If each run of the interrupt contained code to write the contents of RNGSeed into EEPROM, EEPROM would then contain the following string of data: 1001010100101010. Observe that Bits 1-7 (0010101, counting from right to left) and Bits 8-14 (0010101) are identical.

The solution implemented in this code is to only write to EEPROM for every eight runs of the interrupt. This ensures that all bits of RNGSeed will have been replaced by new random bits from the thermistor before writing to EEPROM. Because nothing is written to EEPROM before all bits have been replaced with newly generated values from the thermistor, no patterns should form. This solution is implemented by placing a counter register in the interrupt, named RNGCounter. RNGCounter is initialized to zero in the main routine and is incremented every time the interrupt is run. When RNGCounter reaches eight, Bits 0-7 will have been rewritten with newly generated values from the thermistor, and RNGSeed is then written to EEPROM. Later in the code, RNGCounter is reset to zero to begin the next byte generation of RNGSeed.

```
cpse RNGCounter, EightRegister
rjmp finishinterrupt

writerngeeprom:
```

This segment of code is designed to test whether RNGCounter has reached eight. If so, RNGSeed should be written to EEPROM and RNGCounter should be reset to zero, as described above. The register EightRegister is declared to be 0x08 in the main routine and is not changed at any other point in the code. RNGCounter is compared to EightRegister to determine if RNGCounter equals eight. If so, the cpse command evaluates as true, skips the next statement (rjmp finishinterrupt), and begins the writerngeeprom routine. If RNGCounter is not equal to eight, the next statement is executed (rjmp finishinterrupt). Program control is thus transferred to the finishinterrupt routine, in which the ADC Converstion interrupt is ended without writing anything to EEPROM.

```
writerngeeprom:

sbic EECR,EEPE
rjmp writerngeeprom
```

This segment of code is a loop that verifies that EEPROM is not currently being written to before writing RNGSeed to EEPROM. EEPROM offers a register named EEPROM Control Register (EECR) that holds the status of EEPROM. Bit 1 of the EECR is the EEPE EEPROM Programming Enable (EEPE) bit. When the programmer is ready to write to EEPROM, the EEPE bit must be set to one. After setting the EEPE bit to one, the CPU is halted for two cycles before the next instruction is executed. It is very important that the program not attempt to write to EEPROM while another write to EEPROM is already taking place. Due to this, the code shown above contains a loop that continues to loop until the EEPE bit is zero. The first line of the loop (sbic EECR, EEPE) skips the next instruction if the bit in question (EEPE) is clear, meaning that it is set to zero. The translation is that if EEPE is set to zero, there is no other EEPROM write taking place, so the rjmp writerngeeprom instruction can be skipped. This prevents the loop from looping again and continues with the next instruction in the writerngeeprom routine. If EEPE is not set to zero, the rjmp writerngeeprom instruction is not skipped and is thus exectuted, which begins the loop again. This is referred to as a polling looop. As a side note, the programmer must be sure that an interrupt not take place during an EEPROM write. However, because this EEPROM write is occurring inside of an interrupt, this is not a concern for this code.

```
ldi RNGCounter, 0x00
```

If program execution has entered the writerngeeprom routine, it is because RNGCounter became equal to eight. Now RNGCounter must be reset to zero so that the tally on the next byte of RNGSeed can begin.

```
out EEARH, r26
out EEARL, r25
```

In the AT90USB1287, EEPROM is 4096 bytes in size (approximately 4KB). The byte addresses of EEPROM therefore range from 0 to 4095. Because the maximum number that one byte can represent is 255 (if all bits are 1, as in 11111111), the full range of EEPROM addresses requires two bytes. Accordingly, the

AT90USB1287 has two registers to represent EEPROM addresses, named EEARH (high byte) and EEARL (and low byte). EEARL represents EEPROM addresses 0 – 255 and EEARH represents EEPROM addresses 256 – 4096. EEARH only needs the use of its Bits 0-3 because 4096 can be represented with 12 bits (8 from EEARL and Bits 0-3 of EEARH).

r25 and r26 hold the addresses of EEARL and EEARH, respectively, that correspond to the byte of EEPROM to which the next byte of data will be written. In order to maximize the amount of data that can be stored in EEPROM, this program begins writing to EEPROM at address 0. R25 and r26 are accordingly set to zero in the main routine. R25 and r26 are incremented accordingly in the interrupt to ensure that bytes of data are written to EEPROM sequentially. This ensures that every byte of EEPROM is written to in numerical order. The steps shown above (out EEARH, r26 *and* out EEARL, r25) load the proper values into EEARH and EEARL so that data is written to the correct byte in EEPROM.

out EEDR, RNGSeed

EEPROM contains a data register named EEPROM Data Register, or EEDR. EEDR contains the byte of data to be written to EEPROM. The statement above loads the contents of RNGSeed into EEDR so that RNGSeed may be written to EEPROM.

sbi EECR,EEMPE
sbi EECR,EEPE

Bit 2 of EEPROM is the EEPROM Master Programming Enable (EEMPE) bit. As discussed above, when the programmer is ready to write to EEPROM, the EEPE bit must be set to one. However, no writing to EEPROM will occur unless the EEMPE bit is also set to one. The EEMPE bit is reset to zero in four cycles after the EEMPE bit is set to one. Consequently, after the EEMPE bit is set, writing to EEPROM will occur only if the EEPE bit is set to one within four clock cycles. If not, EEMPE will time out and be set to zero. This is why the EEPE bit is set to one (using the sbi command) immediately after EEMPE is set to one (using the sbi command). After performing both of these commands, the byte in the EEDR register (which contains the contents of the RNGSeed register) will be written to the next byte in EEPROM. This is how random data generated by the thermistor is written to EEPROM.

cp r25, FullRegister
breq overflowrng

This segment of code is designed to address the overflow issue in EEPROM. As discussed above, the full range of byte addresses in EEPROM requires the use of two registers, EEARH and EEARL. In this program, the registers r25 and r26 hold the values of the next byte in EEPROM to which the current value of RNGSeed is written. R25 and r26 correspond directly to EEARL and EEARH, respectively. Because of the need for two registers, a simple increment will not suffice. There has to be some method of organization between r25 and r26 to ensure that all byte addresses in EEPROM are used. Consider the following example. Suppose r25 is currently set to 11111111 (0xFF) and r26 is currently set to 00000000 (0x00). This corresponds to byte address 255 (recall that byte addresses in EEPROM range from 0 to 4095, for a total of 4096 bytes). The next byte address should be 256, which would correspond to r25 being set to 00000000 (0x00) and r26 being set to 00000001 (0x01). However, simply incrementing r25 will not convert 11111111 (0xFF) to 00000000 (0x00) [for r25] *and* 00000000 (0x00) to 00000001 (0x01) [for r26]. An overflow mechanism must therefore be implemented for situations such as this. The overflow mechanism is only necessary when r25 equal 11111111 (0xFF) because that is

when the increment needs to overflow into r26. The lines of code shown above address this situation. The FullRegister register is set to 0xFF in the main routine and is not changed at any location in the code. The cp command compares the value of r25 to the static value of FullRegister (which is always 0xFF in this code). If the two registers are equal, the breq command branches and program control is sent to the overflowrng routine. Overflowrng handles the overflow and returns program control to the next line of code after the "breq overflowrng" line (writerngeepromii:). Note that in order to return control to the line of code following "breq overflowrng", that line of code has to be a routine so that the rjmp command has a routine to which to perform a relative jump. Rjmp cannot perform a relative jump to a line of code that is not the label of a routine. If the r25 and FullRegister registers are not equal, the "breq overflowrng" line of code is skipped altogether and the line of code after "breq overflowrng" (writerngeepromii:)is executed.

In order to effectively demonstrate the logic of the code, the overflowrng routine will now be examined.

```
overflowrng:

        inc r26

        ldi r25, 0x00

        ldi OverFlowReg, 0xFF

        rjmp writerngeepromii
```

The proper method of handling the overflow is to increment r26 and reset r25 to zero, as shown in the overflow example given above. The first two lines of code in overflowrng handle these tasks. The next line of code is "ldi OverFlowReg, 0xFF". The OverFlowReg register is a status flag that indicates whether an overflow has just been performed. If OverFlowReg is zero, no overflow has just taken place; if OverFlowReg is 0xFF, an overflow has just taken place. Because an overflow has just taken place in the overflowrng routine, the OverFlowReg register is set to 0xFF in the "ldi OverFlowReg, 0xFF" command. Lastly, as described above, program control is transferred to the writerngeepromii routine using the "rjmp writerngeepromii" command.

```
writerngeepromii:

        cpse OverFlowReg, FullRegister
        inc r25

        ldi OverFlowReg, 0x00
```

The "cpse OverFlowReg, FullRegister" command determines whether an overflow has just been performed. It does so by testing whether OverFlowReg is equal to FullRegister, the latter of which is always set to 0xFF. If so, an overflow has just been performed (as described above), and so the cpse statement skips the next command (inc r25). "Inc r25" must be skipped in the event of an overflow because r25 is reset to zero in the overflowrng routine. If r25 were then incremented again, a byte in EEPROM would be missed entirely. This is obviously not the proper method to handle an overflow, so the "inc r25" command is skipped if OverFlowReg equals FullRegister (i.e. in the event of an overflow). If there has been no overflow, "cpse OverFlowReg, FullRegister" evaluates to false and thus "inc r25" is

executed. This is done so that the next byte of EEPROM will be written to during the next interrupt. Lastly, now that the overflow has been properly handled (if one has taken place), the OverFlowReg status flag is reset to zero, which indicates no overflow for the next time an interrupt occurs.

finishinterrupt:

```
pop r17
pop r16
out SREG, r16
pop r16
reti
```

As described above, the finishinterrupt routine finishes the interrupt and returns control to the return address where program control was stopped to perform the interrupt. All registers used in the interrupt (i.e. SREG, r16, and r17) are reset to the values they held before the interrupt by popping these saved values off the stack. The "reti" command returns program control to the return address, which is saved on the stack.

Main

Below is the first portion of the main routine:

```
main:
    wdr                 ; Reset watchdog timer

    ldi r16, 0xF0       ; Enable R16's bits 7-4 (1111 0000)
    out DDRD, r16       ; Set PortD[7-4] as output (leds)

    ldi r16, 0xCF       ; Disable R16's bits 5-4 (1100 1111)
    out DDRE, r16       ; Set PortE[5-4] as input (Joystick: Right & Down)

    ldi r16, 0x1F       ; Disable R16's bits 7-5 (0001 1111)
    out DDRB, r16       ; Set PortB[7-5] as input (Joystick: Select,Up & Left)

    ; Reset joystic's ports
    ldi r16, 0x30       ; Enable R16's bits 5-4 (0011 0000)
    out PortE, r16
    ldi r16, 0xE0       ; Enable R16's bits 7-5 (1110 0000)
    out PortB, r16
```

This portion of main was copied verbatim from the Lights program that was given and explained to students in the first week of the Computer Security Architecture course. It is therefore assumed that any reader of this report will have had experience with this segment of code. No further discourse on the subject is needed as a result.

```
ldi r16, low(RAMEND)
out SPL, r16
ldi r16, high(RAMEND)
```

```
out SPH, r16
```

The purpose of this code segment is to initialize the stack for use. The use of the stack is necessary in this program because this program implements interrupts, as discussed in the RNG section. When using the AT90USB1287 microcontroller, the on-board static RAM (SRAM) memory is used to function as the stack. The stack must first be initialized before use by properly configuring the stack pointer (SP). The SP is a two-byte pointer that can be accessed in the same manner as a port. The two bytes of SP are Stack Pointer High (SPH) and Stack Pointer Low (SPL). SPH holds the most significant address byte (MSB); SPL holds the least significant address byte (LSB). This is similar to the EEARH and EEARL bytes of EEPROM.

The code segment above initializes the stack by setting the SP to the highest address of SRAM, labeled RAMEND. As is traditionally true with stack architecture, the stack of the AT90USB1287 grows downward, starting from higher addresses and moving downward toward lower addresses. This is why the SP is initialized to the highest address of SRAM and not the lowest address. RAMEND represents the highest address of SRAM and is hardware-specific. The aforementioned usb1287def.inc file for the AT90USB1287 hardware has the following line to give the two-byte address of RAMEND:

```
.equ    RAMEND          = 0x20ff
```

0x20FF is therefore the highest address of SRAM in the AT90USB1287. The MSB and LSB of this address are 0x20 and 0xFF, respectively. Due to this, high(RAMEND) evaluates to 0x20 and low(RAMEND) evaluates to 0xFF. These addresses are loaded into SPH and SPL, respectively, using the code segment shown above. Once SPH and SPL are initialized, the programmer can make use of the stack.

```
ldi r16, 0b01000000
sts ADMUX, r16
```

Recall from the RNG section the ADMUX component of the AT90USB1287. This segment of code sets the configuration of ADMUX that define how ADMUX will be used throughout the program. The usb1287def.inc file shows that the bits of ADMUX represent the following configurations:

```
; ***** AD_CONVERTER *****************
; ADMUX - The ADC multiplexer Selection Register
.equ    MUX0   = 0      ; Analog Channel and Gain Selection Bits
.equ    MUX1   = 1      ; Analog Channel and Gain Selection Bits
.equ    MUX2   = 2      ; Analog Channel and Gain Selection Bits
.equ    MUX3   = 3      ; Analog Channel and Gain Selection Bits
.equ    MUX4   = 4      ; Analog Channel and Gain Selection Bits
.equ    ADLAR  = 5      ; Left Adjust Result
.equ    REFS0  = 6      ; Reference Selection Bit 0
.equ    REFS1  = 7      ; Reference Selection Bit 1
```

The numbers 0 -7 shown above correspond to the eight bits of the ADMUX register. Setting any of these bits to 1 enables the corresponding feature; setting any of these bits to 0 disables the corresponding feature. The code segment shown above indicates that the “Reference Selection Bit 0” (Bit 6) was enabled and that the remaining features (Bit 1-5 and Bit 7) were disabled. Research from various sites such as those given in the “References” section of this report suggested to disable all of the features of

ADMUX. Unfortunately, this did not produce the intended results. Trial and error ultimately provided the solution, which was to enable the Reference Selection Bit 0.

```
ldi r16, 0b11101111
sts ADCSRA, r16
```

The ADCSRA register is a register associated with the ADC compononent of the AT90USB1287, just as is the ADMUX register. As with the ADMUX register (discussed above), the bits of the ADCSRA register all represent various configurations that can be enabled or disabled. The usb1287def.inc file gives the following text regarding the ADCSRA register's bits:

```
; ADCSRA - The ADC Control and Status register
.equ    ADPS0  = 0      ; ADC  Prescaler Select Bits
.equ    ADPS1  = 1      ; ADC  Prescaler Select Bits
.equ    ADPS2  = 2      ; ADC  Prescaler Select Bits
.equ    ADIE   = 3      ; ADC Interrupt Enable
.equ    ADIF   = 4      ; ADC Interrupt Flag
.equ    ADATE  = 5      ; ADC Auto Trigger Enable
.equ    ADSC   = 6      ; ADC Start Conversion
.equ    ADEN   = 7      ; ADC Enable
```

Just as with the ADMUX register, each of these features can be enabled by setting the bit to one, and disabled by setting the bit to zero. The code segment shown above gives the configuration used in this project. Below is an explanation of each:

- Bits 0-2 "determine the division factor between the XTAL frequency and the input clock to the ADC", as stated at http://www.analoglab.com/adc.html. What this quote means is beyond the scope of this project and this report. It is sufficient to say that empirical testing and data showed that the optimum configuration of these three bits for the goals of this project was 111.
- Bit 3 is the ADC Interrupt Enable. This bit must be enabled for the ADC Conversion interrupt to trigger, so obviously this must be set to one.
- Bit 4 is the ADC Interrupt Flag. The hardware of the AT90USB1287 modifies this bit based on whether an ADC converstion interrupt has been completed. As such, the programmer need not concern himself/herself with this bit. This bit should be disabled (zero) at first because no interrupt has yet taken place.
- Bit 5 is the ADC Auto Trigger Enable. Setting this bit to one enables Free Running mode, in which the ADC continuously updates the hardware with new values. Because the programmer needs to generate large amounts of random data, this bit needs to be enabled. The alternative (setting the bit to zero) represents Single Conversion mode, in which only one interrupt routine is triggered.
- Bit 6 is the ADC Start Conversion. This bit should be set when the programmer is ready to begin the first ADC converstion interrupt, so it should be set to one.
- Bit 7 is the ADC Enable. Setting this bit enables the ADC in general, so this bit should be set to one.

```
ldi RNGSeed, 0x00
ldi PrevSeed, 0x00
```

```
ldi TempSeed, 0x00
ldi PRNGSeed, 0x00
ldi ZeroCounter, 0x00
ldi RNGCounter, 0x00
ldi FullRegister, 0xFF
ldi r26, 0x00
ldi r25, 0x00
ldi OverFlowReg, 0x00
ldi EightRegister, 0x08
```

This list is simply the initializations of all registers used in this program.

```
Sei
```

Sei is the command needed to enable the Interrupt bit on the Status Register (SREG). If the Interrupt bit on SREG is clear, no interrupts can take place. If the Interrupt bit on SREG is set, interrupts can begin taking place. Obviously, this bit needs to be set.

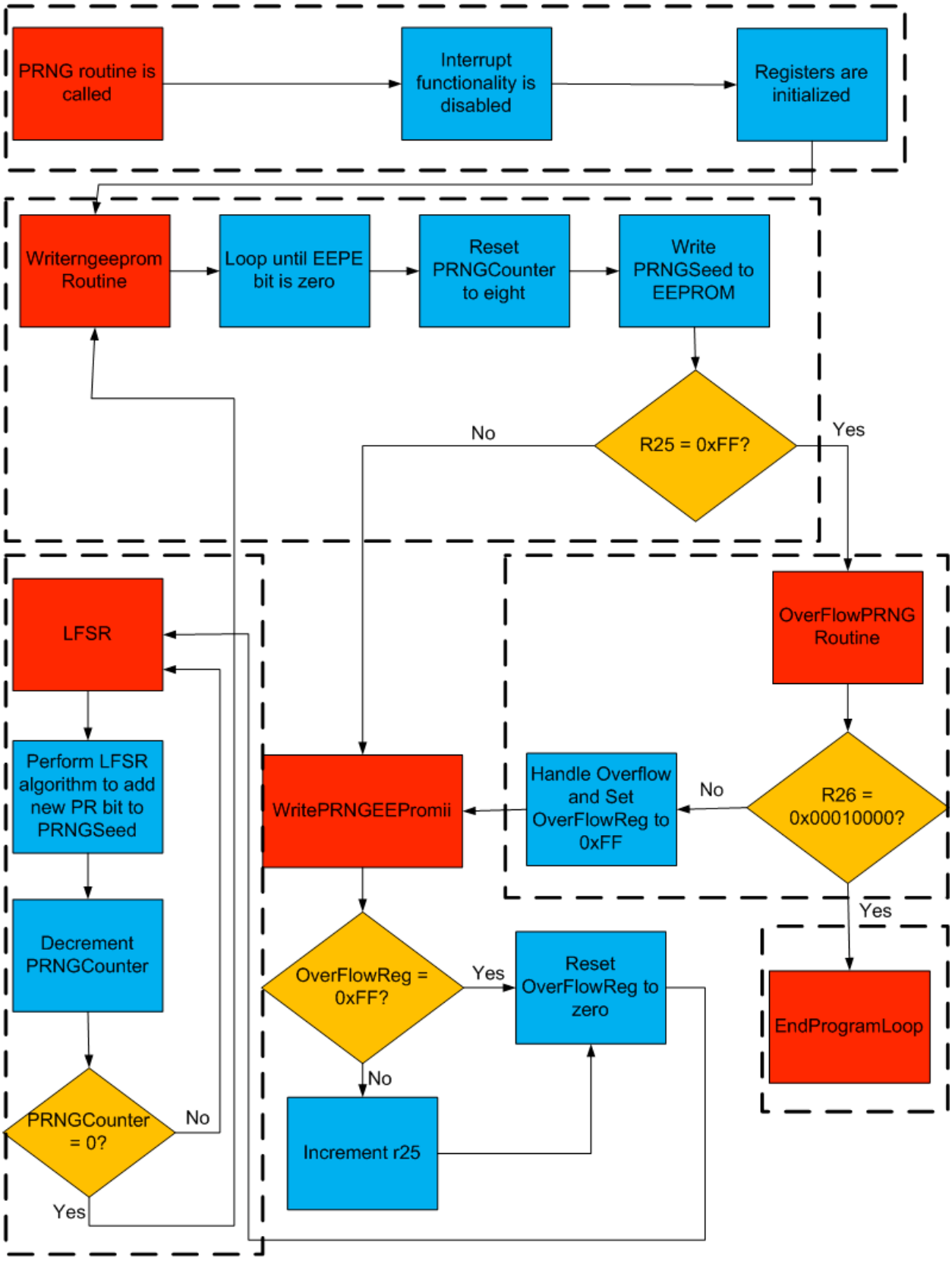
```
loop:

        wdr

        in r17, PinB
        in r18, PinE

        sbrs r17, 6
        rjmp prng

        sbrs r18, 4
        rjmp prng

        sbrs r17, 7
        rjmp prng

        sbrs r18, 5
        rjmp prng

        sbrs r17, 5
        rjmp prng

        rjmp loop
```

This loop is analogous to the “LedLoop” of the Lights program that students analyzed in the first week of the Computer Security Architecture class. The purpose of this loop is twofold. First, because the last line of this loop is “rjmp loop”, it ensures that the program runs continuously without stopping. This is vital because the RNG needs to gather large amounts of data. If the program stops, no more data can be gathered. Adding the “loop: . . . rjmp loop” statements to this program ensures that the program will run indefinitely. Second, this loop gives the user the option to move the joystick in any direction to

jump to the prng routine. As will be discussed below, when the prng routine is accessed, all RNG and interrupt functionality ceases, and pseudorandom data is generated.

Weak PRNG (LFSR)

A flowchart of the weak PRNG (LFSR) algorithm is shown below:


PRNG routine is called
Interrupt functionality is disabled
Registers are initialized
Writerngeeprom Routine
Loop until EEPE bit is zero
Reset PRNGCounter to eight
Write PRNGSeed to EEPROM
R25 = 0xFF?
No
Yes
LFSR
Perform LFSR algorithm to add new PR bit to PRNGSeed
Decrement PRNGCounter
PRNGCounter = 0?
No
Yes
WritePRNGEEPromii
OverFlowReg = 0xFF?
Yes
No
Reset OverFlowReg to zero
Increment r25
OverFlowPRNG Routine
Handle Overflow and Set OverFlowReg to 0xFF
No
R26 = 0x00010000?
Yes
EndProgramLoop

prng:

```
cli

sts ADCSRA, ZeroCounter
```

These are the first two lines of the prng routine. The first priority of the prng routine is to completely disable all interrupts. When the prng routine is called, it is assumed that the user has finished gathering random data using the ADC Conversion interrupt routine, and would like to use that random data as input into the PRNG. No further random data need be collected; the interrupt functionality should be disabled as a result. Disabling the interrupt functionality is also necessary to avoid disrupting or interfering with the prng routine. Accordingly, the cli command clears the Interrupt bit on the Status Register (SREG). This is the opposite effect of the sei command. When the Interrupt bit on SREG is cleared, no further interrupts can take place. Although it should not be necessary, this program also clears all of the ADC enable bits of the ADCSRA register as a redundant means of canceling the ADC Conversion interrupt.

```
ldi r25, 0x00
ldi r26, 0x00
ldi OverFlowReg, 0x00
```

These are standard initializations to prepare for the prng routine.

```
writeprngeeprom:

    sbic EECR,EEPE
    rjmp writeprngeeprom

    wdr
```

This is the beginning of the writeprngeeprom routine. The first two lines of this routine are the polling loop to ensure that the EEPE bit of EECR is clear (see the RNG section for more details). The next command, wdr, simpl clears the watchdog.

```
ldi PRNGCounter, 0x08
```

Recall from the RNG section that RNGSeed was not written to EEPROM until all of the eight bits in the RNGSeed byte were rewritten with fresh random data. This was done to prevent unnecessary patterns from forming. The same methodology is used for the prng routine. PRNGCounter is the counter that stores the number of bits of PRNGSeed that have been replaced with fresh pseudorandom data. PRNGCounter is set to eight and is decremented every time a new bit is added to PRNGSeed. When PRNGCounter reaches zero, PRNGSeed is written to EEPROM.

```
mov PRNGByteForEEProm, PRNGSeed

out EEARH, r26
out EEARL, r25

out EEDR, PRNGByteForEEProm
```

```
sbi EECR,EEMPE
sbi EECR,EEPE
```

As discussed above, when PRNGCounter reaches zero, PRNGSeed is written to EEPROM. The code segment shown above is the algorithm used to do this. It is mostly identical to the algorithm used in the RNG section to write RNGSeed to EEPROM. In this case, PRNGSeed is first copied to another register (PRNGByteforEEProm), and that register (PRNGByteforEEProm) is then written to EEPROM.

```
        cp r25, FullRegister
        breq overflowprng

writeprngeepromii:

        cpse OverFlowReg, FullRegister
        inc r25

        ldi OverFlowReg, 0x00
```

Because the prng routine writes to all bytes of EEPROM (address bytes 0 – 4096), just as does the RNG routine, it uses the same methodology and algorithms as does the RNG routine to write to EEPROM. The code segment shown above is identical to the corresponding code segment in the RNG routine because the algorithm is exactly the same. For a more detailed discussion on what this code segment does, the reader is encouraged to review the RNG section.

```
lfsr:

        mov PrevSeed, PRNGSeed

        mov TempSeed, PRNGSeed

        lsl TempSeed
        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed
```

The code segment shown above is the beginning of the lfsr routine. The lfsr routine is where the Linear Feedback Shift Register (LFSR) algorithm is actually performed. In order to understand the lfsr routine, it is necessary to review the general LFSR algorithm.

The Fibonacci LFSR algorithm works as follows:

1) A register containing a predetermined number of bits is filled with ones and zeroes. This is the seed of the LFSR. In this project, because the AT90USB1287 offers 8-bit registers, the LFSR seed was determined to have 8 bits.
2) The "taps" are chosen. The taps are the bits within the register that will be used to determine the next bit generated by the LFSR. The list of taps in the LFSR is referred to as the tap sequence.
3) It is to the user's advantage to choose taps that produce a maximum LFSR. A maximum LFSR will cycle through all possible states of ones and zeroes that the register can assume (except for the state in which all bits are zero, which will never change, no matter how many iterations) before repeating a state. Increasing the number of unique states that the LFSR produces before repeating a state is advantageous so that patterns do not form in the pseudorandom data. The user is thus advised to choose taps that produce a maximal LFSR. In this project, the tap sequence is the 8^{th} bit, the 6^{th} bit, the 5^{th} bit, and the 4^{th} bit, which is labeled [8, 6, 5, 4].
4) The values of the taps are EORed together to form the next bit to be added to the register. Because the [8, 6, 5, 4] tap sequence was chosen for this project, the 8^{th}, 6^{th}, 5^{th}, and 4^{th} bits are EORed together to produce the next bit to be added to the register.
5) The bits of the register are shifted either one bit to the left or one bit to the right. The bit generated in step four is copied into the vacant bit that remained from the shift of this step.
6) This procedure is repeated to continuously generate pseduorandom data.

To return to the code of this project:

```
lfsr:

        mov PrevSeed, PRNGSeed

        mov TempSeed, PRNGSeed
```

The contents of PRNGSeed are copied to the PrevSeed and TempSeed registers so that these two registers can perform the LFSR algorithm.

```
lsl TempSeed
lsl TempSeed

eor PrevSeed, TempSeed
```

Because the tap sequence is [8, 6, 5, 4], the first step is to EOR the 8^{th} and 6^{th} bits. TempSeed is shifted to the left twice to isolate Bit 6 of TempSeed in the leftmost bit position. Now PrevSeed holds the 8^{th} bit of PRNGSeed in its leftmost bit position, and TempSeed holds the 6^{th} bit of PRNGSeed in its leftmost position. The EOR operation between PrevSeed and TempSeed will EOR these two leftmost bits and store the result in the leftmost bit of PrevSeed. PrevSeed now holds the result of the EOR operation of the 8^{th} and 6^{th} bits of PRNGSeed, as desired.

```
lsl TempSeed

eor PrevSeed, TempSeed
```

TempSeed is now shifted left so that the 5th bit of PRNGSeed is now in the leftmost bit position of TempSeed. This bit is EORed with PrevSeed and stored in PrevSeed. This is equivalent to taking the result of the EOR of the 8th and 6th bits of PRNGSeed (the step just completed) and EORing it with the 5th bit of PRNGSeed. The result is stored in PrevSeed because PrevSeed is the first operand in the EOR command.

```
lsl TempSeed

eor PrevSeed, TempSeed
```

Lastly, TempSeed is shifted left so that the 4th bit of PRNGSeed is now in the leftmost bit position of TempSeed. This bit is EORed with PrevSeed and stored in PrevSeed. This is equivalent to taking the result of the EOR of the 8th, 6th, and 5th bits of PRNGSeed (the step just completed) and EORing it with the 4th bit of PRNGSeed. This completes the tap sequence, as the 8th, 6th, 5th, and 4th bits of PRNGSeed have now been EORed. The result of this operation is stored in the leftmost bit of PrevSeed.

```
lsr PRNGSeed

andi PrevSeed, 0b10000000
andi PRNGSeed, 0b01111111
or PRNGSeed, PrevSeed
```

Essentially, the purpose of this algorithm is as follows. PRNGSeed is shifted to the right once to make room for the newest bit generated in the steps above. This means that Bit 7 (leftmost bit) is now vacant and ready to accept the newest generated bit. The three lines of code that follow copy the leftmost bit of PrevSeed (newest generated bit, as described above) into the leftmost bit of PRNGSeed. This algorithm is identical to that used in the RNG routine. The reader is encouraged to review that section in the RNG explanation for more detail.

```
dec PRNGCounter
```

Now that a new bit of pseudorandom data has been added to PRNGSeed, the PRNGCounter can be decremented.

```
cpse ZeroCounter, PRNGCounter
rjmp lfsr

rjmp writeprngeeprom
```

The first statement of the code segment shown above tests to see if ZeroCounter is equal to PRNGCounter. ZeroCounter is a register that statically stores 0x00. ZeroCounter was initialized in the main routine to hold 0x00 and its value is never changed throughout the program. As a result, comparing ZeroCounter to PRNGCounter is actually a test to determine if PRNGCounter equals 0x00. If PRNGCounter does equal zero, it means that the lfsr routine has been performed eight times. This means that each of the eight bits in the PRNGSeed byte has been replaced by freshly generated pseduorandom data. If this is the case, the “cpse ZeroCounter, PRNGCounter” statement will evaluate to true, which will cause program control to skip the next statement (rjmp lfsr). Program control will

therefore be transferred to the “rjmp writeprngeeprom” statement. The writeprngeeprom routine will then be executed; as discussed above, this will cause PRNGSeed to be written to EEPROM. On the other hand, if PRNGCounter is not equal to 0x00, it means that lfsr has not been yet been performed eight times. In this case, the lfsr needs to be run again. The “cpse ZeroCounter, PRNGCounter” statement will evaluate to false, the “rjmp lfsr” statement will not be skipped and will be executed this time, and program control will be transferred to the lfsr routine to be run again.

```
overflowprng:

        ldi RNGSeed, 0b00010000

        cp r26, RNGSeed
        breq endprogramloop

        inc r26

        ldi r25, 0x00

        ldi OverFlowReg, 0xFF

        rjmp writeprngeepromii

endprogramloop:

        rjmp endprogramloop
```

The last portion of code is the overflow routine used for the prng (overflowprng). This routine is almost identical to the overflow routine used for the rng (overflowrng), with one exception. With the RNG, the program allows EEPROM to be overwritten indefinitely. As long as the user does not terminate the program, EEPROM will continue to write over itself. This is not the case with the PRNG in order to provide the programmer with built-in debugging and testing. When the PRNG routine is called, the program first writes out the seed used for the PRNG to EEPROM at byte address zero. This is a good test to determine if the PRNG is working correctly because the first byte of EEPROM should always be the seed of the PRNG. If the program allows the PRNG to freely overwrite EEPROM as does the RNG, this seed at byte address zero will be overwritten. The code segment shown above includes the following code to prevent this from happening:

```
        ldi RNGSeed, 0b00010000

        cp r26, RNGSeed
        breq endprogramloop

.

.

.
```

```
endprogramloop:

        rjmp endprogramloop
```

As previously described, EEPROM ranges from byte addresses 0 – 4095. In order to prevent EEPROM from being overwritten, the program needs to stop when the EEARH and EEARL registers evaluate to 4096. Recall that this program controls EEARH and EEARL with registers r26 and r25, respectively. Consequently, when r26 and r25 evaluate to 4096, the program should terminate. R26 and r25 evaluate to 4096 when r26 is equal to 0b00010000 and r25 is equal to 0b00000000 because 2 ^ 12 = 4096 (the bit that is set to one in r26 is the 12^{th} bit of r26 and r25).. It is sufficient, though, to simply test to determine if r26 is equal to 0b00010000. This is what the first two lines of the sub-code snippet shown above does. If r26 does not equal 0b00010000, the program control follows the same overflow algorithm as used in the RNG. If r26 does equal 0b00010000, program control is transferred to a routine named endprogramloop. The purpose of endprogramloop is to run indefinitely, but not perform any actions, until the user terminates the program.

Results of NIST Randomness Tests on Weak and Strong PRNGs

All of the tests executed on the weak and strong PRNGs were developed by the NIST Random Number Generation Technical Working Group (NIST RNG-TWG). These tests were implemented because this group's test suite if often cited as the industry standard and state-of-the-art in random number generation. The tests listed below were chosen for this project.

- Frequency test
- Runs test
- Cumulative Sums (Forward) test
- Cumulative Sums (Reverse) test
- Linear Complexity test
- Overlapping Templates test

For each of these tests, a test statistic named the p-value is generated. NIST considers data to be sufficiently random is the p-value is greater than 0.01.

The Frequency (Monobit) Test was chosen because, as stated in the NIST RNG-TWG software documentation, "It is recommended that the Frequency test be run first, since this supplies the most basic evidence for the existence of non-randomness in a sequence, specifically, non-uniformity." Due to the fact that the Frequency (Monobit) Test was chosen, the use of the Frequency (Block) Test was excluded because these two tests examined the same type of non-randomness, namely having too many ones or zeroes.

The Runs and the Longest-Run-of-Ones (Block) Tests both examine the oscillation of ones and zeroes within a string to determine randomness. The Runs Test measures the total number of runs within the target string, whereas the Longest-Run-of-Ones (Block) Test measures the longest run of ones within blocks of the target string. The Runs Test was chosen because the sample size is relatively low (4096 bytes).

Both the Overlapping Template Mapping Test and the Non-overlapping Template Test determine if a pre-specified target substring occurs too frequently for the entire string to be considered random. If the

target substring is found beginning at a certain bit within the entire string, the Non-overlapping Test continues checking by proceeding to the bit after the **last** bit of the target substring match. The Overlapping Test instead proceeds to only the bit after the **first** bit of the target substring match. The Overlapping Test was chosen to ensure that all possible matches were considered. It was accepted for the purposes of this study the NIST RNG-TWG's suggested default template length of 9 bits.

The Cumulative Sums Test evaluates the partial summations of the terms in the target string, provided that the zeros are converted to (-1). The absolute value of the difference between these cumulative partial summations and zero is compared to what would be expected from a truly random string of the same size. This test was chosen because its examination of sequential dependence is relatively different from the nature of the other tests. There was no use of the Random Excursions Test or the Random Excursions Variant Test because these tests were very similar in nature to the Cumulative Sums Test. The use of any two of these tests would have generated redundant results.

Weak PRNG: LFSR

The original intention of this project was to use the thermistor to produce random data that would serve as a seed for the two PRNGs (i.e. LFSR and AES). Unfortunately, the thermistor did not generate data was sufficiently random for the purpose of evaluating the effectiveness of PRNG algorithms. If a PRNG accepts as input a seed that is not sufficiently random, the resulting pseudorandom data will be highly predictable. This will certainly cause the PRNG algorithm to fail the NIST tests.

The main goal of this study was to provide a thorough evaluation of two PRNG algorithms. In order to ensure that the evaluation is an accurate representation of the best results to be expected from each algorithm, the random data generated by the thermistor was not used. Instead, a seed created by the programmer was hard-coded into the code for each PRNG. The PRNG then used these hard-coded seeds as input to the PRNG algorithm. These seeds were designed to be as random as possible.

The hard-coded random seed for the weak PRNG (LFSR) was 10010110. The code given in the Appendix was modified to use this string as the random seed and to fill EEPROM will 4096 bytes of pseudorandom data. This data was then run against the NIST tests. The modified code to test the LFSR algorithm with the hard-coded seed was as follows:

```
.include "usb1287def.inc"

.device AT90USB1287

.DEF RNGSeed = R19
.DEF PrevSeed = R20
.DEF TempSeed = R21
.DEF PRNGSeed = R22
.DEF PRNGCounter = R23
.DEF PRNGByteForEEProm = R24
.DEF ZeroCounter = R27
.DEF RNGCounter = R28
.DEF FullRegister = R29
.DEF OverFlowReg = R30
.DEF EightRegister = R31
```

```
.org 0x0000
rjmp main

main:

        wdr

        ldi r16, low(RAMEND)
        out SPL, r16
   ldi r16, high(RAMEND)
   out SPH, r16

        ldi RNGSeed, 0x00
        ldi PrevSeed, 0x00
        ldi TempSeed, 0x00
        ldi PRNGSeed, 0x00
        ldi ZeroCounter, 0x00
        ldi RNGCounter, 0x00
        ldi FullRegister, 0xFF
        ldi r26, 0x00
        ldi r25, 0x00
        ldi OverFlowReg, 0x00
        ldi EightRegister, 0x08

prng:

        ldi r25, 0x00
        ldi r26, 0x00
        ldi OverFlowReg, 0x00

        ldi PRNGSeed, 0b10010110

writeprngeeprom:

        sbic EECR,EEPE
        rjmp writeprngeeprom

        wdr

        ldi PRNGCounter, 0x08

        mov PRNGByteForEEProm, PRNGSeed

        out EEARH, r26
        out EEARL, r25

        out EEDR, PRNGByteForEEProm
```

```
        sbi EECR,EEMPE
        sbi EECR,EEPE

        cp r25, FullRegister
        breq overflowprng

writeprngeepromii:

        cpse OverFlowReg, FullRegister
        inc r25

        ldi OverFlowReg, 0x00

lfsr:

        mov PrevSeed, PRNGSeed

        mov TempSeed, PRNGSeed

        lsl TempSeed
        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed

        lsr PRNGSeed

   andi PrevSeed, 0b10000000
        andi PRNGSeed, 0b01111111
        or PRNGSeed, PrevSeed

        dec PRNGCounter

        cpse ZeroCounter, PRNGCounter
        rjmp lfsr

        rjmp writeprngeeprom

overflowprng:
```

```
        ldi RNGSeed, 0b00010000

        cp r26, RNGSeed
        breq endprogramloop

        inc r26

        ldi r25, 0x00

        ldi OverFlowReg, 0xFF

        rjmp writeprngeepromii

endprogramloop:

        rjmp endprogramloop:
```

It should be noted that the JTAG device for the AT90USB1287 was used to read data from EEPROM. The use of this device was invaluable to this project because there was no other way to read EEPROM as easily. The one drawback of the JTAG device was that it appeared to add metadata to EEPROM when saving this data to a file. In order to remove this metadata, the following grep command was used on all EEPROM data files generated by the JTAG device:

```
grep -o ".\{34\}$" test.hex | ./scrb 0x0A | ./htobin > test.out
```

This grep statement also converted the EEPROM hex file into a binary file, which is what worked best with NIST's test.

Below are the results of NIST's tests on the weak PRNG (LFSR). Recall that in order for the tests to be successful, p must be greater than 0.01:

- Frequency test produced a p-value of 0.661749 (SUCCESSFUL)
- Runs test produced a p-value of 0.041918 (SUCCESSFUL)
- Cumulative Sums (Forward) test produced a p-value of 0.378538 (SUCCESSFUL)
- Cumulative Sums (Reverse) test produced a p-value of 0.160236 (SUCCESSFUL)
- Linear Complexity test produced a p-value of 0.609283 (SUCCESSFUL)
- Overlapping Templates test produced a p-value of 0.397277 (SUCCESSFUL)

Strong PRNG: AES

The Advanced Encryption Standard (AES) is widely considered to be one of the strongest (if not the strongest) encryption algorithm in use today. It is therefore appropriate that AES be used as the strong PRNG algorithm. However, the AES algorithm is quite complex. It would be far beyond the scope of any undergraduate project for the student to code the AES algorithm in Assembler. Instead, students of this project are required to download Assembler coding of the AES Fantastic algorithm from http://point-at-infinity.org/avraes/ in order to use AES on the AT90USB1287. As a side note, this should teach to the students the value of only using open-source and freely tested cryptographic algorithms for their

encryption needs. It will also teach students how to learn and incorporate into their projects an algorithm written by another programmer. This is a skill that they will often need to use in the future.

In the Appendix, the reader will find the AES Fantastic code given at http://point-at-infinity.org/avraes/ with modifications used to write the AES pseudorandom data to EEPROM. All modifications are clearly marked in the comments directly adjacent to the code. All of the techniques used in the modified AES Fantastic code were used in either the RNG or weak PRNG code given in the project (or both). As a result, no further explanation of the modified AES Fantastic code will be given in this section to avoid redundancy. The reader is encouraged to read the code explanations in the RNG or weak PRNG sections to obtain more details.

The AES Fantastic code at http://point-at-infinity.org/avraes/ hard-codes a 16-byte plaintext string into registers r0 – r15 of the AT90USB1287. The modified AES Fantastic code writes this plaintext string into the first 16 bytes of EEPROM. The AES Fantastic code then performs AES encryption on the 16 bytes of plaintext and produces the corresponding 16 bytes of ciphertext. The modified AES Fantastic code then writes this ciphertext to EERPOM. In order to completely fill EEPROM with pseudorandom data created with AES, the 16 bytes of ciphertext are then re-encrypted with AES to produce an additional 16 bytes of ciphertext. The additional 16 bytes of ciphertext are then written to EEPROM. This process of taking as input the 16 bytes of ciphertext produced by the last step, re-encrypting them with AES, and writing the resultant new 16 bytes of ciphertext to EEPROM is repeated until EEPROM is full (4096 bytes in total).

Unfortunately, the modified AES Fantastic code did not appear to work correctly. There were several checks set up to test whether the modified AES fantastic code was successful, such as: 1) Was all of EEPROM being filled? 2) Was all of EEPROM being filled only once? 3) Were the first 16 bytes of EEPROM the original 16-byte plaintext coded into the AES fantastic code? 4) Did any patterns emerge in the resulting pseudorandom data that could be detected by eye? Etc. The modified AES Fantastic code passed all of these checks with the exception of one (but possibly the most important one). A debugger was used to step through the modified AES Fantastic code line-by-line to determine what the first 16 bytes of ciphertext should be (the ciphertext produced after the first round of AES encryption). All signs from the debugger indicated that these 16 bytes of ciphertext should have been written correctly to EEPROM. However, the corresponding 16 bytes in EEPROM did not match what the debugger reported. It is possible that the aforementioned metadata that the JTAG device produces could have caused this discrepancy, but this has not yet been confirmed or denied. Further study will be required to determine the cause of the discrepancy.

Because all other checks of the modified AES Fantastic code passed, the pseudorandom data that was produced was run against the NIST tests. Below are the results (recall that a test passes if p is greater than 0.01):

- Frequency test produced a p-value of 0.434655 (SUCCESSFUL)
- Runs test produced a p-value of 0.083891 (SUCCESSFUL)
- Cumulative Sums (Forward) test produced a p-value of 0.445366 (SUCCESSFUL)
- Cumulative Sums (Reverse) test produced a p-value of 0.091001 (SUCCESSFUL)
- Linear Complexity test produced a p-value of 0.576198 (SUCCESSFUL)
- Overlapping Templates test produced a p-value of 0.246770 (SUCCESSFUL)

It should be noted that the results of the strong PRNG (AES) tests cannot be fully trusted until the discrepancy described above is resolved. These numbers were generated purely as a troubleshooting and/or curiosity measure.

Despite this setback, the lesson that the students were supposed to learn was to observe that the p-values of the strong PRNG (AES) should have been higher than those of the weak PRNG (LFSR). It is helpful to note, however, that the p-values of the weak PRNG (LFSR), which can be trusted, were all relatively high. This is quite reassuring considering the fact that the LFSR algorithm is still in use today as a PRNG algorithm.

Difficulties Experienced

The two main difficulties experienced in this project have been described in detail in the sections above. Namely, 1) the on-board thermistor of the AT90USB1287 did not produce sufficiently random data; 2) the code used to write to EEPROM the pseudorandom data generated by the AES Fantastic code did not pass all accuracy checks. More details on these difficulties can be found in the RNG and Strong PRNG sections of this report, respectively.

Additional difficulties experienced were the following: 1) coding a secure PRNG algorithm in Assembler; 2) verifying that it is working properly on the AT90USB1287. Although I do have experience coding cryptographic algorithms such as RC4 in high-level languages such as C++ and Java, I did not have any Assembler experience before enrolling in the Computer Security Architecture course. This situation is exacerbated by the fact that Assembler is so different in design and structure than languages such as C++ and Java.

Moreover, when selecting a strong PRNG algorithm, I had to perform further research to determine its most secure form. There are relatively secure cryptographic algorithms that have known weaknesses if not implemented properly. A good example of this is the RC4 algorithm. RC4 operates at a fast speed and has the capability to provide strong encryption with pseudorandom data. However, it has been shown that if RC4 is not implemented properly, a known plaintext attack can reveal information about the key.

Lastly, in my experience with coding cryptographic algorithms, I always had access to input/output systems that allowed me to view the values that my program generated. This allowed me to confirm with a high degree of confidence that my program was working as intended. The limited input/output interface of the AT90USB1287 made this process more challenging. The use of the JTAG device with the AT90USB1287 was absolutely essential with respect to this matter.

Suggestions for Improvement and Future Work

There are two main suggestions for future work: 1) improve the RNG to provide data that is more random; 2) revise the strong PRNG code (AES Fantastic) so that it works as intended. With respect to the former, this could consist of researching better ways to use the on-board thermistor so that it produces data with a higher level of unpredictability. This could also consist of finding an entirely different source of random data, such as a different hardware component of the AT90USB1287. With

respect to the revisions of the strong PRNG code, this will require further debugging of the code given in the Appendix of this report.

Conclusions

The purpose of this report is to provide students with a better understanding of programming in Assembler and of random and pseudorandom data. Both of these topics are covered in detail so as to demonstrate to the student their importance in modern practice. Sample code and detailed explanations are given to the student of how to program the following: 1) a RNG using the on-board thermistor of the AT90USB1287; 2) a weak PRNG using the LFSR algorithm; and 3) a strong PRNG using the AES algorithm. It is hoped that by the end of this report, students will understand all three components, how they fit together, and why they are important to the modern practice of coding and cryptography.

Unfortunately, not every component of this project went accordingly to plan. The on-board thermistor ultimately did not provide data that was sufficiently unpredictable, so random strings had to be hard-coded into the PRNG algorithms. In addition, the strong PRNG code (AES Fantastic) did not pass all checks for accuracy, so its results could not be trusted. Both of these discrepancies will be resolved by the time this report is used in an undergraduate course in the Fall 2008 semester.

References

"Security and Privacy". Lecture course offered at the Johns Hopkins University Information Security Institute by Dr. Aviel D. Rubin. Fall 2007 Semester.

http://en.wikipedia.org/wiki/Rc4. "RC4". Article published by Wikipedia.

http://csrc.nist.gov/rng. NIST Randomness Tests offered on the NIST website.

http://www.atmel.com/dyn/resources/prod_documents/doc7608.pdf

http://www.atmel.com/dyn/resources/prod_documents/doc7627.pdf

http://www.atmel.com/dyn/resources/prod_documents/doc7593.pdf

http://www.avr-asm-download.de/beginner_en.pdf

http://www.atmel.com/dyn/resources/prod_documents/7593S.pdf

http://www.avrbeginners.net/architecture/adc/m8_adc_example.html

http://en.wikipedia.org/wiki/Linear_feedback_shift_register

http://point-at-infinity.org/avraes/

http://www.analoglab.com/adc.html

Appendices

-Source Code

No comments have been included in this source code because each line of code has been thoroughly commented in the sections above.

RNG and Weak PRNG (LFSR)

```
.include "usb1287def.inc"

.device AT90USB1287

.DEF RNGSeed = R19
.DEF PrevSeed = R20
.DEF TempSeed = R21
.DEF PRNGSeed = R22
.DEF PRNGCounter = R23
.DEF PRNGByteForEEProm = R24
.DEF ZeroCounter = R27
.DEF RNGCounter = R28
.DEF FullRegister = R29
.DEF OverFlowReg = R30
.DEF EightRegister = R31

.org 0x0000
rjmp main

.org 0x003A
call ADC_ISR

ADC_ISR:

        push r16
        in r16, SREG
        push r16
        push r17

        lds r16, ADCL
        lds r17, ADCH

        lsl RNGSeed

        andi r16, 0b00000001
        andi RNGSeed, 0b11111110
        or RNGSeed, r16

        mov PRNGSeed, RNGSeed

        inc RNGCounter
```

```
        cpse RNGCounter, EightRegister
        rjmp finishinterrupt

writerngeeprom:

        sbic EECR,EEPE
        rjmp writerngeeprom

        ldi RNGCounter, 0x00

        out EEARH, r26
        out EEARL, r25

        out EEDR, RNGSeed

        sbi EECR,EEMPE
        sbi EECR,EEPE

        cp r25, FullRegister
        breq overflowrng

writerngeepromii:

        cpse OverFlowReg, FullRegister
        inc r25

        ldi OverFlowReg, 0x00

finishinterrupt:

        pop r17
        pop r16
        out SREG, r16
        pop r16
        reti

overflowrng:

        inc r26

        ldi r25, 0x00

        ldi OverFlowReg, 0xFF

        rjmp writerngeepromii

main:
```

```
        wdr

        ldi r16, 0xFF
        out DDRD, r16

        ldi r16, 0xCF
        out DDRE, r16

        ldi r16, 0x1F
        out DDRB, r16

        ldi r16, 0x30
        out PortE, r16

        ldi r16, 0xE0
        out PortB, r16

        ldi r16, low(RAMEND)
        out SPL, r16
        ldi r16, high(RAMEND)
        out SPH, r16

        ldi r16, 0b01000000
        sts ADMUX, r16

        ldi r16, 0b11101111
        sts ADCSRA, r16

        ldi RNGSeed, 0x00
        ldi PrevSeed, 0x00
        ldi TempSeed, 0x00
        ldi PRNGSeed, 0x00
        ldi ZeroCounter, 0x00
        ldi RNGCounter, 0x00
        ldi FullRegister, 0xFF
        ldi r26, 0x00
        ldi r25, 0x00
        ldi OverFlowReg, 0x00
        ldi EightRegister, 0x08

        sei

loop:

        wdr

        in r17, PinB
```

```
        in r18, PinE

        sbrs r17, 6
        rjmp prng

        sbrs r18, 4
        rjmp prng

        sbrs r17, 7
        rjmp prng

        sbrs r18, 5
        rjmp prng

        sbrs r17, 5
        rjmp prng

        rjmp loop

prng:

        cli

        sts ADCSRA, ZeroCounter

        ldi r25, 0x00
        ldi r26, 0x00
        ldi OverFlowReg, 0x00

writeprngeeprom:

        sbic EECR,EEPE
        rjmp writeprngeeprom

        wdr

        ldi PRNGCounter, 0x08

        mov PRNGByteForEEProm, PRNGSeed

        out EEARH, r26
        out EEARL, r25

        out EEDR, PRNGByteForEEProm

        sbi EECR,EEMPE
        sbi EECR,EEPE
```

```
        cp r25, FullRegister
        breq overflowprng

writeprngeepromii:

        cpse OverFlowReg, FullRegister
        inc r25

        ldi OverFlowReg, 0x00

lfsr:

        mov PrevSeed, PRNGSeed

        mov TempSeed, PRNGSeed

        lsl TempSeed
        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed

        lsl TempSeed

        eor PrevSeed, TempSeed

        lsr PRNGSeed

        andi PrevSeed, 0b10000000
        andi PRNGSeed, 0b01111111
        or PRNGSeed, PrevSeed

        dec PRNGCounter

        cpse ZeroCounter, PRNGCounter
        rjmp lfsr

        rjmp writeprngeeprom

overflowprng:

        ldi RNGSeed, 0b00010000

        cp r26, RNGSeed
        breq endprogramloop
```

```
        inc r26

        ldi r25, 0x00

        ldi OverFlowReg, 0xFF

        rjmp writeprngeepromii

endprogramloop:

        rjmp endprogramloop
```

Strong PRNG (AES)

My modifications to the original code have been clearly marked in the adjacent comments.

```
; Copyright (C) 2006 B. Poettering
;
; This program is free software; you can redistribute and/or modify
; it under the terms of the GNU General Public License as published by
; the Free Software Foundation; either version 2 of the License, or
; (at your option) any later version. Whenever you redistribute a copy
; of this document, make sure to include the copyright and license
; agreement without modification.
;
; This program is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
; GNU General Public License for more details.
;
; You should have received a copy of the GNU General Public License
; along with this program; if not, write to the Free Software
; Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
; The license text can be found here: http://www.gnu.org/licenses/gpl.txt

;                http://point-at-infinity.org/avraes/
;
; This AES implementation was written in November 2006 by B. Poettering.
; It is published under the terms of the GNU General Public License. If
; you need AES code, but this license is unsuitable for your project,
; feel free to contact me: avraes AT point-at-infinity.org


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
;                              RijndaelFantastic
;
; This is a microcontroller implementation of the Rijndael block cipher,
better
; known as AES. The target device class is Atmel's AVR, a family of very fast
; and very powerful flash MCUs, operating at clock rates up to 16 MHz,
; executing one instruction per clock cycle (16 MIPS). The implementation
```

```
; given here is optimized for RAM requirement, and achieves an encryption
; rate of about 63 kByte/sec (on a 16MHz MCU). Decryption is done with
; 55 kByte/sec.
;
; The implemented algorithm is restricted to block and key sizes of 128 bit.
; Larger key sizes can be obtained by altering the key scheduling code, which
; should be easy.
;
; This implementation makes extensive use of the AVR's "lpm" instruction,
; which loads data bytes from program memory at given addresses (the s-boxes
; are realized that way). Some members of the AVR family don't offer that
; instruction at all (e.g. AT90S1200), others only in a restricted way
; (forcing the target register to be r0). The code below requires the least
; restricted lpm instruction (with free choice of the target register).
; The ATmega161 devices meet the above mentioned requirements.
;
; Statistics:
;
; 16 MHz MCU | clock cycles | blocks per second | bytes per second
; -----------+--------------+-------------------+------------------
; encryption |    4059      |      3942         |      63070
; decryption |    4675      |      3422         |      54759
;
; (key preprocessing time is not considered)
;
; This source code consists of some routines and an example application,
; which encrypts a certain plaintext and decrypts it afterwards with the
; same key. Comments in the code clarify the interaction between the key
; expansion and the encryption/decryption routines.
;
; I encourage to read the following Rijndael-related papers/books/sites:
; [1] "The Design of Rijndael", Daemen & Rijmen, Springer, ISBN 3-540-42580-2
; [2] http://www.esat.kuleuven.ac.be/~rijmen/rijndael/
; [3] http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip
; [4] http://www.esat.kuleuven.ac.be/~rijmen/rijndael/atmal.zip
; [5] http://csrc.nist.gov/CryptoToolkit/aes/rijndael/
;
; [1] is *the* book about Rijndael, [2] is the official Rijndael homepage,
; [3] contains the complete Rijndael AES specification, [4] is another
; Rijndael-implementation for AVR MCUs (but much slower than this one,
; taking 3815 clock cycles per encryption), [5] is the official NIST AES
; site with further links.
;
; AVR and ATmega are registered trademarks by the ATMEL corporation.
; See http://www.atmel.com and http://www.atmel.com/products/avr/ for
; further details.


.include "usb1287def.inc"

.def H1           = r16
.def H2           = r17
.def Rcon   = r18
.def OverFlowReg = r21 ; THIS IS MY CODE
.def FullRegister = r22 ; THIS IS MY CODE
.def HighByteOverflow = r27; THIS IS MY CODE
```

```
main: cli               ; initialize stack
      ldi r31,high(RAMEND)
      out SPH,r31
      ldi r31,low(RAMEND)
      out SPL,r31

      ldi ZH, high(key<<1)    ; load key to RAM position $0060
      ldi ZL, low(key<<1)
      ldi YH, high($0060)
      ldi YL, low($0060)
main0:      lpm r16, Z+
      st Y+, r16
      cpi YL, low($0060+16)
      brne main0

      ldi ZH, high(text<<1)   ; load plaintext to r0-r15
      ldi ZL, low(text<<1)
      lpm r0, Z+
      lpm r1, Z+
      lpm r2, Z+
      lpm r3, Z+
      lpm r4, Z+
      lpm r5, Z+
      lpm r6, Z+
      lpm r7, Z+
      lpm r8, Z+
      lpm r9, Z+
      lpm r10, Z+
      lpm r11, Z+
      lpm r12, Z+
      lpm r13, Z+
      lpm r14, Z+
      lpm r15, Z+

; MY INSERTED CODE BEGINS HERE
; THIS CODE IS TO WRITE THE PLAINTEXT TO EEPROM BEFORE IT IS ENCRYPTED

      ldi r19, 0x00
      ldi r20, 0x00
      ldi OverFlowReg, 0x00
      ldi FullRegister, 0xFF

      wdr

writetexteepromi:

      sbic EECR,EEPE
      rjmp writetexteepromi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r0

      inc r19

      sbi EECR,EEMPE
```

```
      sbi EECR,EEPE

      wdr

writetexteepromii:

      sbic EECR,EEPE
      rjmp writetexteepromii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r1

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromiii:

      sbic EECR,EEPE
      rjmp writetexteepromiii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r2

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromiv:

      sbic EECR,EEPE
      rjmp writetexteepromiv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r3

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromv:
```

```
      sbic EECR,EEPE
      rjmp writetexteepromv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r4

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromvi:

      sbic EECR,EEPE
      rjmp writetexteepromvi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r5

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromvii:

      sbic EECR,EEPE
      rjmp writetexteepromvii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r6

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writeeepromviii:

      sbic EECR,EEPE
      rjmp writeeepromviii

      out EEARH, r20
      out EEARL, r19
```

```
      out EEDR, r7

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromix:

      sbic EECR,EEPE
      rjmp writetexteepromix

      out EEARH, r20
      out EEARL, r19

      out EEDR, r8

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromx:

      sbic EECR,EEPE
      rjmp writetexteepromx

      out EEARH, r20
      out EEARL, r19

      out EEDR, r9

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromxi:

      sbic EECR,EEPE
      rjmp writetexteepromxi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r10

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE
```

```
      wdr

writetexteepromxii:

      sbic EECR,EEPE
      rjmp writetexteepromxii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r11

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromxiii:

      sbic EECR,EEPE
      rjmp writetexteepromxiii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r12

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromxiv:

      sbic EECR,EEPE
      rjmp writetexteepromxiv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r13

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromxv:

      sbic EECR,EEPE
```

```
      rjmp writetexteepromxv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r14

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

writetexteepromxvi:

      sbic EECR,EEPE
      rjmp writetexteepromxvi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r15

      inc r19

      sbi EECR,EEMPE
      sbi EECR,EEPE

; MY INSERTED CODE ENDS HERE

      ldi YH, high($0060)     ; encrypt the plaintext
      ldi YL, low($0060)

; MY INSERTED CODE BEGINS HERE
; THIS IS TO CONTINUOSLY RUN AES AND WRITE THE RESULTS TO EEPROM

      ldi FullRegister, 0xFF
      ldi OverFlowReg, 0x00

myloop:

      rcall AESEncrypt128
      rcall AESKeyRewind

writecipheepromi:

      sbic EECR,EEPE
      rjmp writecipheepromi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r0

      sbi EECR,EEMPE
      sbi EECR,EEPE
```

```
      wdr

      cp r19, FullRegister
      breq overflowprngi

writeciphitwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromii

overflowprngi:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramone

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphitwo

endprogramone:

      jmp endprogram

writecipheepromii:

      sbic EECR,EEPE
      rjmp writecipheepromii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r1

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngii

writeciphiitwo:

      cpse OverFlowReg, FullRegister
      inc r19
```

```
      ldi OverFlowReg, 0x00

      rjmp writecipheepromiii

overflowprngii:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramtwo

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphiitwo

endprogramtwo:

      jmp endprogram

writecipheepromiii:

      sbic EECR,EEPE
      rjmp writecipheepromiii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r2

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngiii

writeciphiiitwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromiv

overflowprngiii:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramthree
```

```
      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphiiitwo

endprogramthree:

      jmp endprogram

writecipheepromiv:

      sbic EECR,EEPE
      rjmp writecipheepromiv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r3

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngiv

writeciphivtwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromv

overflowprngiv:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramfour

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphivtwo

endprogramfour:

      jmp endprogram
```

```
writecipheepromv:

      sbic EECR,EEPE
      rjmp writecipheepromv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r4

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngv

writeciphvtwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromvi

overflowprngv:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramfive

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphvtwo

endprogramfive:

      jmp endprogram

writecipheepromvi:

      sbic EECR,EEPE
      rjmp writecipheepromvi

      out EEARH, r20
      out EEARL, r19

      out EEDR, r5
```

```
      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngvi

writeciphvitwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromvii

overflowprngvi:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramsix

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphvitwo

endprogramsix:

      jmp endprogram

writecipheepromvii:

      sbic EECR,EEPE
      rjmp writecipheepromvii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r6

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngvii

writeciphviitwo:

      cpse OverFlowReg, FullRegister
```

```
	inc r19

	ldi OverFlowReg, 0x00

	rjmp writecipheepromviii

overflowprngvii:

	ldi HighByteOverflow, 0b00010000

	cp r20, HighByteOverflow
	breq endprogramseven

	inc r20

	ldi r19, 0x00

	ldi OverFlowReg, 0xFF

	rjmp writeciphviitwo

endprogramseven:

	jmp endprogram

writecipheepromviii:

	sbic EECR,EEPE
	rjmp writecipheepromviii

	out EEARH, r20
	out EEARL, r19

	out EEDR, r7

	sbi EECR,EEMPE
	sbi EECR,EEPE

	wdr

	cp r19, FullRegister
	breq overflowprngviii

writeciphviiitwo:

	cpse OverFlowReg, FullRegister
	inc r19

	ldi OverFlowReg, 0x00

	rjmp writecipheepromix

overflowprngviii:

	ldi HighByteOverflow, 0b00010000

	cp r20, HighByteOverflow
```

```
      breq endprogrameight

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphviiitwo

endprogrameight:

      jmp endprogram

writecipheepromix:

      sbic EECR,EEPE
      rjmp writecipheepromix

      out EEARH, r20
      out EEARL, r19

      out EEDR, r8

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngix

writeciphixtwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromx

overflowprngix:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramnine

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphixtwo

endprogramnine:
```

```
      jmp endprogram

writecipheepromx:

      sbic EECR,EEPE
      rjmp writecipheepromx

      out EEARH, r20
      out EEARL, r19

      out EEDR, r9

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngx

writeciphxtwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromxi

overflowprngx:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramten

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphxtwo

endprogramten:

      jmp endprogram

writecipheepromxi:

      sbic EECR,EEPE
      rjmp writecipheepromxi

      out EEARH, r20
      out EEARL, r19
```

```
      out EEDR, r10

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngxi

writeciphxitwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromxii

overflowprngxi:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogrameleven

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphxitwo

endprogrameleven:

      jmp endprogram

writecipheepromxii:

      sbic EECR,EEPE
      rjmp writecipheepromxii

      out EEARH, r20
      out EEARL, r19

      out EEDR, r11

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngxii

writeciphxiitwo:
```

```
	cpse OverFlowReg, FullRegister
	inc r19

	ldi OverFlowReg, 0x00

	rjmp writecipheepromxiii

overflowprngxii:

	ldi HighByteOverflow, 0b00010000

	cp r20, HighByteOverflow
	breq endprogramtwelve

	inc r20

	ldi r19, 0x00

	ldi OverFlowReg, 0xFF

	rjmp writeciphxiitwo

endprogramtwelve:

	jmp endprogram

writecipheepromxiii:

	sbic EECR,EEPE
	rjmp writecipheepromxiii

	out EEARH, r20
	out EEARL, r19

	out EEDR, r12

	sbi EECR,EEMPE
	sbi EECR,EEPE

	wdr

	cp r19, FullRegister
	breq overflowprngxiii

writeciphxiiitwo:

	cpse OverFlowReg, FullRegister
	inc r19

	ldi OverFlowReg, 0x00

	rjmp writecipheepromxiv

overflowprngxiii:

	ldi HighByteOverflow, 0b00010000
```

```
	cp r20, HighByteOverflow
	breq endprogramthirteen

	inc r20

	ldi r19, 0x00

	ldi OverFlowReg, 0xFF

	rjmp writeciphxiiitwo

endprogramthirteen:

	jmp endprogram

writecipheepromxiv:

	sbic EECR,EEPE
	rjmp writecipheepromxiv

	out EEARH, r20
	out EEARL, r19

	out EEDR, r13

	sbi EECR,EEMPE
	sbi EECR,EEPE

	wdr

	cp r19, FullRegister
	breq overflowprngxiv

writeciphxivtwo:

	cpse OverFlowReg, FullRegister
	inc r19

	ldi OverFlowReg, 0x00

	rjmp writecipheepromxv

overflowprngxiv:

	ldi HighByteOverflow, 0b00010000

	cp r20, HighByteOverflow
	breq endprogramfourteen

	inc r20

	ldi r19, 0x00

	ldi OverFlowReg, 0xFF

	rjmp writeciphxivtwo
```

```
endprogramfourteen:

      jmp endprogram

writecipheepromxv:

      sbic EECR,EEPE
      rjmp writecipheepromxv

      out EEARH, r20
      out EEARL, r19

      out EEDR, r14

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngxv

writeciphxvtwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp writecipheepromxvi

overflowprngxv:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogramfifteen

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphxvtwo

endprogramfifteen:

      jmp endprogram

writecipheepromxvi:

      sbic EECR,EEPE
      rjmp writecipheepromxvi

      out EEARH, r20
```

```
      out EEARL, r19

      out EEDR, r15

      sbi EECR,EEMPE
      sbi EECR,EEPE

      wdr

      cp r19, FullRegister
      breq overflowprngxvi

writeciphxvitwo:

      cpse OverFlowReg, FullRegister
      inc r19

      ldi OverFlowReg, 0x00

      rjmp myloop

overflowprngxvi:

      ldi HighByteOverflow, 0b00010000

      cp r20, HighByteOverflow
      breq endprogram

      inc r20

      ldi r19, 0x00

      ldi OverFlowReg, 0xFF

      rjmp writeciphxvitwo

endprogram:

      rjmp endprogram

; MY INSERTED CODE ENDS HERE

; I AM COMMENTING THIS OUT B/C WE DO NOT NEED DECRYPTION

/*
      ldi YH, high($0060)     ; prepare key for decryption
      ldi YL, low($0060)
      rcall AESKeyDecPreprocess
      rcall AESDecrypt128     ; decrypt the cipher text
*/

main1:      rjmp main1        ; stop

text:
.db $32,$43,$f6,$a8,$88,$5a,$30,$8d,$31,$31,$98,$a2,$e0,$37,$07,$34
key:
.db $2b,$7e,$15,$16,$28,$ae,$d2,$a6,$ab,$f7,$15,$88,$09,$cf,$4f,$3c
```

```
;***************************************************************************
**
; Encrypt the 16 byte block defined by r0-r15 under the 128 bit key [Y].
; Note that calling this function modifies [Y]. Therefore, before encrypting
a
; second block with the same key [Y] has to be restored. This can be done by
; calling the function AESKeyRewind, but it is faster to simply backup the
; original key somewhere to RAM and to restore it after calling
AESEncrypt128.
;
; Touched registers: Rcon, H1, H2, Z
;
AESEncrypt128:
      ldi Rcon, 1
AESEnc1:rcall AddRoundKey
      rcall RAMIncKey128
      rcall ShiftRowsSubBytes
      cpi Rcon, 0x6c
      breq AddRoundKey
      rcall MixColumns
      rjmp AESEnc1


;***************************************************************************
**
; Rewind the key given in [Y]. See AESEncrypt128 for more details.
;
; Touched registers: Rcon, H1, H2, Z
;
AESKeyRewind:
      ldi Rcon, 0x36
AESKeyR:rcall RAMDecKey128
      cpi Rcon, 0
      brne AESKeyR
      ret


;***************************************************************************
**
; Preprocess the key given in [Y] for use for decryption. See AESDecrypt128
; for more details.
;
; Touched registers: Rcon, H1, H2, Z
;

; I AM COMMENTING THIS OUT B/C WE DO NOT NEED DECRYPTION

/*
AESKeyDecPreprocess:
      ldi Rcon, 1
AESKeyF:rcall RAMIncKey128
      cpi Rcon, 0x6c
      brne AESKeyF
      ret
```

```
;***************************************************************************
**
; Decrypt the 16 byte block defined by r0-r15 under the 128 bit key [Y].
; The decryption key has to be preprocessed by AESKeyDecPreprocess before
; calling this funtion. Like in AESEncrypt128 [Y] is modified by this
; function, but the key can be restored by calling AESKeyDecPreprocess.
Again,
; backing up the key to RAM will be faster.
;
; Note that AESKeyRewind and AESKeyDecPreprocess are the inverses of each
; other. In other words: if encryption and decryption are performed in
; strictly alternating order, the calls to AESKeyRewind and AESKeyPreprocess
; can be ommitted.
;
; Touched registers: Rcon, H1, H2, Z
;
AESDecrypt128:
      ldi Rcon, 0x36
      rcall AddRoundKey
      rcall RAMDecKey128
      rcall ShiftRowsSubBytesInverse
AESDec1:rcall AddRoundKey
      rcall RAMDecKey128
      rcall MixColumnsInverse
      rcall ShiftRowsSubBytesInverse
      cpi Rcon, 0
      brne AESDec1
*/

;***************************************************************************
**
; The following subroutines are for internal use only. They shouldn't be
; called by any client application directly.
;***************************************************************************
**

.def ST11   = r0
.def ST21   = r1
.def ST31   = r2
.def ST41   = r3
.def ST12   = r4
.def ST22   = r5
.def ST32   = r6
.def ST42   = r7
.def ST13   = r8
.def ST23   = r9
.def ST33   = r10
.def ST43   = r11
.def ST14   = r12
.def ST24   = r13
.def ST34   = r14
.def ST44   = r15

AddRoundKey:      ; Touched registers: ST11-ST44, H1
      ld H1, Y
      eor ST11, H1
```

```
        ldd H1, Y+1
        eor ST21, H1
        ldd H1, Y+2
        eor ST31, H1
        ldd H1, Y+3
        eor ST41, H1
        ldd H1, Y+4
        eor ST12, H1
        ldd H1, Y+5
        eor ST22, H1
        ldd H1, Y+6
        eor ST32, H1
        ldd H1, Y+7
        eor ST42, H1
        ldd H1, Y+8
        eor ST13, H1
        ldd H1, Y+9
        eor ST23, H1
        ldd H1, Y+10
        eor ST33, H1
        ldd H1, Y+11
        eor ST43, H1
        ldd H1, Y+12
        eor ST14, H1
        ldd H1, Y+13
        eor ST24, H1
        ldd H1, Y+14
        eor ST34, H1
        ldd H1, Y+15
        eor ST44, H1
        ret


MixColumnsInverse:      ; Touched registers: ST11-ST44, H1, H2, Z
        ldi ZH, high(xtime<<1)
        mov ZL, ST11            ; u = xtime(xtime(a[0] ^ a[2]))
        eor ZL, ST31
        lpm ZL, Z
        lpm ZL, Z
        eor ST11, ZL            ; a[0] ^= u
        eor ST31, ZL            ; a[2] ^= u
        mov ZL, ST21            ; v = xtime(xtime(a[1] ^ a[3]))
        eor ZL, ST41
        lpm ZL, Z
        lpm ZL, Z
        eor ST21, ZL            ; a[1] ^= v
        eor ST41, ZL            ; a[3] ^= v

        mov ZL, ST12
        eor ZL, ST32
        lpm ZL, Z
        lpm ZL, Z
        eor ST12, ZL
        eor ST32, ZL
        mov ZL, ST22
        eor ZL, ST42
        lpm ZL, Z
```

```
        lpm ZL, Z
        eor ST22, ZL
        eor ST42, ZL

        mov ZL, ST13
        eor ZL, ST33
        lpm ZL, Z
        lpm ZL, Z
        eor ST13, ZL
        eor ST33, ZL
        mov ZL, ST23
        eor ZL, ST43
        lpm ZL, Z
        lpm ZL, Z
        eor ST23, ZL
        eor ST43, ZL

        mov ZL, ST14
        eor ZL, ST34
        lpm ZL, Z
        lpm ZL, Z
        eor ST14, ZL
        eor ST34, ZL
        mov ZL, ST24
        eor ZL, ST44
        lpm ZL, Z
        lpm ZL, Z
        eor ST24, ZL
        eor ST44, ZL


MixColumns: ; Touched registers: ST11-ST44, H1, H2, Z
        ldi ZH, high(xtime<<1)
        mov H1, ST11            ; Tmp = a[0] ^ a[1] ^ a[2] ^ a[3]
        eor H1, ST21
        mov ZL, H1
        eor H1, ST31
        eor H1, ST41
        mov H2, ST11            ; save a[0] for later use
        lpm ZL, Z         ; Tm = xtime(a[0] ^ a[1])
        eor ST11, ZL            ; a[0] ^= Tm ^ Tmp
        eor ST11, H1
        mov ZL, ST21            ; Tm = xtime(a[1] ^ a[2])
        eor ZL, ST31
        lpm ZL, Z
        eor ST21, ZL            ; a[1] ^= Tm ^ Tmp
        eor ST21, H1
        mov ZL, ST31            ; Tm = xtime(a[2] ^ a[3])
        eor ZL, ST41
        lpm ZL, Z
        eor ST31, ZL            ; a[2] ^= Tm ^ Tmp
        eor ST31, H1
        mov ZL, ST41            ; Tm = xtime(a[3] ^ a[0])
        eor ZL, H2
        lpm ZL, Z
        eor ST41, ZL            ; a[3] ^= Tm ^ Tmp
        eor ST41, H1
```

```
mov H1, ST12
eor H1, ST22
mov ZL, H1
eor H1, ST32
eor H1, ST42
mov H2, ST12
lpm ZL, Z
eor ST12, ZL
eor ST12, H1
mov ZL, ST22
eor ZL, ST32
lpm ZL, Z
eor ST22, ZL
eor ST22, H1
mov ZL, ST32
eor ZL, ST42
lpm ZL, Z
eor ST32, ZL
eor ST32, H1
mov ZL, ST42
eor ZL, H2
lpm ZL, Z
eor ST42, ZL
eor ST42, H1

mov H1, ST13
eor H1, ST23
mov ZL, H1
eor H1, ST33
eor H1, ST43
mov H2, ST13
lpm ZL, Z
eor ST13, ZL
eor ST13, H1
mov ZL, ST23
eor ZL, ST33
lpm ZL, Z
eor ST23, ZL
eor ST23, H1
mov ZL, ST33
eor ZL, ST43
lpm ZL, Z
eor ST33, ZL
eor ST33, H1
mov ZL, ST43
eor ZL, H2
lpm ZL, Z
eor ST43, ZL
eor ST43, H1

mov H1, ST14
eor H1, ST24
mov ZL, H1
eor H1, ST34
eor H1, ST44
mov H2, ST14
```

```
        lpm ZL, Z
        eor ST14, ZL
        eor ST14, H1
        mov ZL, ST24
        eor ZL, ST34
        lpm ZL, Z
        eor ST24, ZL
        eor ST24, H1
        mov ZL, ST34
        eor ZL, ST44
        lpm ZL, Z
        eor ST34, ZL
        eor ST34, H1
        mov ZL, ST44
        eor ZL, H2
        lpm ZL, Z
        eor ST44, ZL
        eor ST44, H1
        ret


ShiftRowsSubBytes:      ; Touched registers: ST11-ST44, H1, Z
        ldi ZH, high(sbox<<1)
        mov ZL, ST11
        lpm ST11, Z
        mov ZL, ST12
        lpm ST12, Z
        mov ZL, ST13
        lpm ST13, Z
        mov ZL, ST14
        lpm ST14, Z
        mov H1, ST21
        mov ZL, ST22
        lpm ST21, Z
        mov ZL, ST23
        lpm ST22, Z
        mov ZL, ST24
        lpm ST23, Z
        mov ZL, H1
        lpm ST24, Z
        mov H1, ST31
        mov ZL, ST33
        lpm ST31, Z
        mov ZL, H1
        lpm ST33, Z
        mov H1, ST32
        mov ZL, ST34
        lpm ST32, Z
        mov ZL, H1
        lpm ST34, Z
        mov H1, ST44
        mov ZL, ST43
        lpm ST44, Z
        mov ZL, ST42
        lpm ST43, Z
        mov ZL, ST41
        lpm ST42, Z
```

```
      mov ZL, H1
      lpm ST41, Z
      ret


ShiftRowsSubBytesInverse:     ; Touched registers: ST11-ST44, H1, Z
      ldi ZH, high(isbox<<1)
      mov ZL, ST11
      lpm ST11, Z
      mov ZL, ST12
      lpm ST12, Z
      mov ZL, ST13
      lpm ST13, Z
      mov ZL, ST14
      lpm ST14, Z
      mov H1, ST21
      mov ZL, ST24
      lpm ST21, Z
      mov ZL, ST23
      lpm ST24, Z
      mov ZL, ST22
      lpm ST23, Z
      mov ZL, H1
      lpm ST22, Z
      mov H1, ST31
      mov ZL, ST33
      lpm ST31, Z
      mov ZL, H1
      lpm ST33, Z
      mov H1, ST32
      mov ZL, ST34
      lpm ST32, Z
      mov ZL, H1
      lpm ST34, Z
      mov H1, ST44
      mov ZL, ST41
      lpm ST44, Z
      mov ZL, ST42
      lpm ST41, Z
      mov ZL, ST43
      lpm ST42, Z
      mov ZL, H1
      lpm ST43, Z
      ret


RAMIncKey128:     ; Touched registers: Rcon, H1, H2, Z
      ldi ZH, high(sbox<<1)
      ldd H2, Y+12
      ldd ZL, Y+13
      lpm ZL, Z
      eor ZL, Rcon
      lsl Rcon
      brcc PC+2
      ldi Rcon, 0x1b
      rcall RAMInc1
      ldd ZL, Y+13
```

```
      lpm ZL, Z
      rcall RAMInc1
      ldd ZL, Y+13
      lpm ZL, Z
      rcall RAMInc1
      mov ZL, H2
      lpm ZL, Z
      rcall RAMInc1
      sbiw YL, 4
      ret
RAMInc1:ld H1, Y
      eor ZL, H1
      st Y+, ZL
      ldd H1, Y+3
      eor ZL, H1
      std Y+3, ZL
      ldd H1, Y+7
      eor ZL, H1
      std Y+7, ZL
      ldd H1, Y+11
      eor ZL, H1
      std Y+11, ZL
      ret

RAMDecKey128:     ; Touched registers: Rcon, H1, H2, Z
      ldi ZH, high(sbox<<1)
      ldi H1, 4
RAMDec1:ldd ZL, Y+12
      ldd H2, Y+8
      eor ZL, H2
      std Y+12, ZL
      ldd ZL, Y+4
      eor H2, ZL
      std Y+8, H2
      ld H2, Y+
      eor ZL, H2
      std Y+3, ZL
      dec H1
      brne RAMDec1
      ldd ZL, Y+8
      lpm ZL, Z
      ld H1, -Y
      eor H1, ZL
      st Y, H1
      ldd ZL, Y+12
      lpm ZL, Z
      ld H1, -Y
      eor H1, ZL
      st Y, H1
      ldd ZL, Y+12
      lpm ZL, Z
      ld H1, -Y
      eor H1, ZL
      st Y, H1
      ldd ZL, Y+12
      lpm ZL, Z
      ld H1, -Y
```

```
      eor H1, ZL
      eor H1, Rcon
      st Y, H1
      lsr Rcon
      cpi Rcon, 0x0d
      brne PC+2
      ldi Rcon, 0x80
      ret


;;;
*****************************************************************************
;;;
;;; SBOX and "xtime" tables
;;;
;;; The following tables have to be aligned to a flash position with lower
;;; address byte equal to $00. In assembler syntax: low(sbox<<1) == 0.
;;; To ensure the proper alignment the assembler directive .ORG should be
;;; used. The order of the tables is arbitrary. They even do not have to be
;;; allocated in adjacent memory areas.

.CSEG
.ORG $800               ; ensure proper alignement

sbox:
.db $63,$7c,$77,$7b,$f2,$6b,$6f,$c5,$30,$01,$67,$2b,$fe,$d7,$ab,$76
.db $ca,$82,$c9,$7d,$fa,$59,$47,$f0,$ad,$d4,$a2,$af,$9c,$a4,$72,$c0
.db $b7,$fd,$93,$26,$36,$3f,$f7,$cc,$34,$a5,$e5,$f1,$71,$d8,$31,$15
.db $04,$c7,$23,$c3,$18,$96,$05,$9a,$07,$12,$80,$e2,$eb,$27,$b2,$75
.db $09,$83,$2c,$1a,$1b,$6e,$5a,$a0,$52,$3b,$d6,$b3,$29,$e3,$2f,$84
.db $53,$d1,$00,$ed,$20,$fc,$b1,$5b,$6a,$cb,$be,$39,$4a,$4c,$58,$cf
.db $d0,$ef,$aa,$fb,$43,$4d,$33,$85,$45,$f9,$02,$7f,$50,$3c,$9f,$a8
.db $51,$a3,$40,$8f,$92,$9d,$38,$f5,$bc,$b6,$da,$21,$10,$ff,$f3,$d2
.db $cd,$0c,$13,$ec,$5f,$97,$44,$17,$c4,$a7,$7e,$3d,$64,$5d,$19,$73
.db $60,$81,$4f,$dc,$22,$2a,$90,$88,$46,$ee,$b8,$14,$de,$5e,$0b,$db
.db $e0,$32,$3a,$0a,$49,$06,$24,$5c,$c2,$d3,$ac,$62,$91,$95,$e4,$79
.db $e7,$c8,$37,$6d,$8d,$d5,$4e,$a9,$6c,$56,$f4,$ea,$65,$7a,$ae,$08
.db $ba,$78,$25,$2e,$1c,$a6,$b4,$c6,$e8,$dd,$74,$1f,$4b,$bd,$8b,$8a
.db $70,$3e,$b5,$66,$48,$03,$f6,$0e,$61,$35,$57,$b9,$86,$c1,$1d,$9e
.db $e1,$f8,$98,$11,$69,$d9,$8e,$94,$9b,$1e,$87,$e9,$ce,$55,$28,$df
.db $8c,$a1,$89,$0d,$bf,$e6,$42,$68,$41,$99,$2d,$0f,$b0,$54,$bb,$16

isbox:
.db $52,$09,$6a,$d5,$30,$36,$a5,$38,$bf,$40,$a3,$9e,$81,$f3,$d7,$fb
.db $7c,$e3,$39,$82,$9b,$2f,$ff,$87,$34,$8e,$43,$44,$c4,$de,$e9,$cb
.db $54,$7b,$94,$32,$a6,$c2,$23,$3d,$ee,$4c,$95,$0b,$42,$fa,$c3,$4e
.db $08,$2e,$a1,$66,$28,$d9,$24,$b2,$76,$5b,$a2,$49,$6d,$8b,$d1,$25
.db $72,$f8,$f6,$64,$86,$68,$98,$16,$d4,$a4,$5c,$cc,$5d,$65,$b6,$92
.db $6c,$70,$48,$50,$fd,$ed,$b9,$da,$5e,$15,$46,$57,$a7,$8d,$9d,$84
.db $90,$d8,$ab,$00,$8c,$bc,$d3,$0a,$f7,$e4,$58,$05,$b8,$b3,$45,$06
.db $d0,$2c,$1e,$8f,$ca,$3f,$0f,$02,$c1,$af,$bd,$03,$01,$13,$8a,$6b
.db $3a,$91,$11,$41,$4f,$67,$dc,$ea,$97,$f2,$cf,$ce,$f0,$b4,$e6,$73
.db $96,$ac,$74,$22,$e7,$ad,$35,$85,$e2,$f9,$37,$e8,$1c,$75,$df,$6e
.db $47,$f1,$1a,$71,$1d,$29,$c5,$89,$6f,$b7,$62,$0e,$aa,$18,$be,$1b
.db $fc,$56,$3e,$4b,$c6,$d2,$79,$20,$9a,$db,$c0,$fe,$78,$cd,$5a,$f4
.db $1f,$dd,$a8,$33,$88,$07,$c7,$31,$b1,$12,$10,$59,$27,$80,$ec,$5f
.db $60,$51,$7f,$a9,$19,$b5,$4a,$0d,$2d,$e5,$7a,$9f,$93,$c9,$9c,$ef
```

```
.db $a0,$e0,$3b,$4d,$ae,$2a,$f5,$b0,$c8,$eb,$bb,$3c,$83,$53,$99,$61
.db $17,$2b,$04,$7e,$ba,$77,$d6,$26,$e1,$69,$14,$63,$55,$21,$0c,$7d

xtime:
.db $00,$02,$04,$06,$08,$0a,$0c,$0e,$10,$12,$14,$16,$18,$1a,$1c,$1e
.db $20,$22,$24,$26,$28,$2a,$2c,$2e,$30,$32,$34,$36,$38,$3a,$3c,$3e
.db $40,$42,$44,$46,$48,$4a,$4c,$4e,$50,$52,$54,$56,$58,$5a,$5c,$5e
.db $60,$62,$64,$66,$68,$6a,$6c,$6e,$70,$72,$74,$76,$78,$7a,$7c,$7e
.db $80,$82,$84,$86,$88,$8a,$8c,$8e,$90,$92,$94,$96,$98,$9a,$9c,$9e
.db $a0,$a2,$a4,$a6,$a8,$aa,$ac,$ae,$b0,$b2,$b4,$b6,$b8,$ba,$bc,$be
.db $c0,$c2,$c4,$c6,$c8,$ca,$cc,$ce,$d0,$d2,$d4,$d6,$d8,$da,$dc,$de
.db $e0,$e2,$e4,$e6,$e8,$ea,$ec,$ee,$f0,$f2,$f4,$f6,$f8,$fa,$fc,$fe
.db $1b,$19,$1f,$1d,$13,$11,$17,$15,$0b,$09,$0f,$0d,$03,$01,$07,$05
.db $3b,$39,$3f,$3d,$33,$31,$37,$35,$2b,$29,$2f,$2d,$23,$21,$27,$25
.db $5b,$59,$5f,$5d,$53,$51,$57,$55,$4b,$49,$4f,$4d,$43,$41,$47,$45
.db $7b,$79,$7f,$7d,$73,$71,$77,$75,$6b,$69,$6f,$6d,$63,$61,$67,$65
.db $9b,$99,$9f,$9d,$93,$91,$97,$95,$8b,$89,$8f,$8d,$83,$81,$87,$85
.db $bb,$b9,$bf,$bd,$b3,$b1,$b7,$b5,$ab,$a9,$af,$ad,$a3,$a1,$a7,$a5
.db $db,$d9,$df,$dd,$d3,$d1,$d7,$d5,$cb,$c9,$cf,$cd,$c3,$c1,$c7,$c5
.db $fb,$f9,$ff,$fd,$f3,$f1,$f7,$f5,$eb,$e9,$ef,$ed,$e3,$e1,$e7,$e5
```