Tutorial 4: Cryptography with ARC4 (LPC2148)

Written by Brennen Ball, © 2007

Introduction

In this tutorial, I am going to go through the ARC4 stream cipher algorithm, and then show how to use it in a very simple application on the nRF24L01. I'm sure there are many of you who have had interest in securing your wireless connection, and here is a simple way to do it!

Please keep one thing in mind while reading this article – I am not a cryptography expert, so questions on the subject are not best directed towards me. I found out about ARC4 on the Spark Fun forum (http://forum.sparkfun.com) and proceeded to read into it more and develop a C language implementation.

With respect to the ARC4 description, virtually all of the information here is summarized or quoted directly from the Wikipedia article on ARC4 as of 8/13/2007 (http://en.wikipedia.org/wiki/ARC4), or from the Wikipedia articles that are linked to from the ARC4 page. For more information, please go directly to those sites, as the following is a very short summary of the algorithm.

What the Heck Is ARC4?

In 1987, Ron Rivest of the RSA developed a very simple stream cipher algorithm entitled "Rivest Cipher 4," or "RC4." For those unfamiliar with cryptography, one of the primary advantages of a stream cipher is that, unlike a block cipher, it allows the user to vary the length of the bytes that can be encrypted at a time (a block cipher works on a fixed length of bytes for any given encryption iteration). RC4 has been used in many different encryption schemes, including WEP, WPA, and SSL.

Until September 1994, RC4 was kept under lock and key by the RSA. However, around that time a post was made to the Cypherpunks mailing list that described the algorithm. The article then proliferated over the internet, and it was found that the implementation that was leaked actually had identical output to the implementations of the RSA. Since RC4 was and still is a trademarked name, the leaked implementation was dubbed "ARC4," standing for "alleged RC4."

The ARC4 Algorithm in Pseudocode

The actual implementation of ARC4 is extremely simple, and requires little computation and memory (about 258 bytes). You start with a key, the length of which can be between 1 and 256 bytes (inclusive), with typical values being between 5 and 16 bytes (40 to 128 bits). You then have an array "S", which is of length 256 bytes. This array is used to hold the values used for encryption.

To start the algorithm, we execute the key-scheduling algorithm (KSA). First, the array "S" is initialized to the "identity permutation," i.e. the value of a location x in the

array is x (S[x] = x for all bytes in S).   After this, S is processed to get the values of the new permutation (see pseudocode for the entire KSA as taken from Wikipedia in Figure 1 below).

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap(S[i],S[j])
endfor
```

Figure 1. KSA pseudocode

Once the KSA has completed, we are ready to encrypt data one byte at a time. For this, we use the pseudo-random generation algorithm (PRGA). The Wikipedia article states, "[f]or as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA increments i, adds the value of S pointed to by i to j, exchanges the values of S[i] and S[j], and then outputs the value of S at the location S[i] + S[j] (modulo 256). Each value of S is swapped at least once every 256 iterations." Pseudocode (as taken from Wikipedia) for the PRGA can be found below in Figure 2.

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
endwhile
```

Figure 2. PRGA pseudocode

The actual encryption is then accomplished by XOR'ing the byte that is returned from the PRGA with the desired byte to be encrypted. Interestingly enough, decryption is done the exact same way. The encrypted byte is XOR'ed with the byte that is returned from the PRGA, and the result is the original, unencrypted byte.

The Ins and Outs

In this tutorial, everything is setup exactly as in Tutorials 1, 2, and 3. This includes connecting one node (uC and 24L01) up to a serial port on your PC at 9600-N-1 and no flow control. The other node is totally standalone.

In this tutorial, our aim is to send bytes from the remote unit to the local unit. For this particular application, I am just going to send the alphabet over and over, and the characters that were sent in both encrypted and decrypted form will be displayed over the serial port through the UART. Since ARC4 is a stream cipher, we must keep the byte count up to date (missing a byte will result in an incorrect decryption from that point on), so I have also added a byte with each packet that keeps a packet count. This allows the

local unit to "catch up" to the correct byte in the PRGA (assuming you miss less than 256 packets in a row).

The Hardware I Used for this Tutorial

I used the exact same hardware in this tutorial as in Tutorials 1, 2, and 3. Nothing added, nothing subtracted, so as long as you were able to get those guys working, you should certainly be able to get this one going.

Tutorial Software Description/Requirements

Similar to the hardware required, the software for this tutorial requires the same support as in Tutorials 1, 2, & 3. If you were able to get them going, then you're in the clear for this tutorial.

Arc4.h and .c

These files form the library for the ARC4 routine. There are only 3 variables and 4 functions in the library, so it's extremely simple to use and implement. All that's in arc.h are function declarations, so we'll skip that and move to arc.c.

First, there are declarations for 3 variables: one for the "S" array and two index pointers. S is 256 bytes long (specified by the algorithm), and the pointers are one byte each. Requiring only 258 bytes in RAM, ARC4 is great for uC's without a large amount of memory.

Up next are the function implementations. The first is arc4_initialize_ksa(), which you call to initialize S with the key that you pass in. Once you execute the KSA, you are instantly able to encrypt data. Calling the arc4_encrypt() function will allow you to encrypt a string of bytes of arbitrary length. This function uses the arc4_get_prga_byte() call to encrypt the data that is passed to it. Finally, you can decrypt the data using the arc4_decrypt() function in the same way as arc4_encrypt().

Local Main File: maintutorial4local.c

This is our main driver file for the local unit. It is nearly identical to the code used in Tutorial 1, so you should definitely be familiar with it before you really start digging in here.

The first addition is the key variable. It is initialized to a value of "Key", with the length being 3. It is very important that if you change the length of the key variable that you change the "keylen" define accordingly.

Upon entry into main(), there is a change to the Initialize() function. We have added a call to arc4_initiailize_ksa() to set up S for the ARC4 algorithm. Also, the nRF24L01 is set up as an RX with a payload width of 2. This is because each packet will consist of an encrypted byte (a letter), and a packet counter.

After Initialize() returns, we move into the main loop. Inside the main loop, we wait on a packet to be received. Once received, we print the raw data to the screen over

the UART. Next, we check to see if we have missed any characters, and if so, we catch the S array up by calling arc4_get_prsa_byte the number of packets that we've missed. Now, we decrypt the byte we received and print it out. If all goes well, it will show up as a letter. At this point, we increment our packet counter, toggle our LED, and return to the top of the loop.
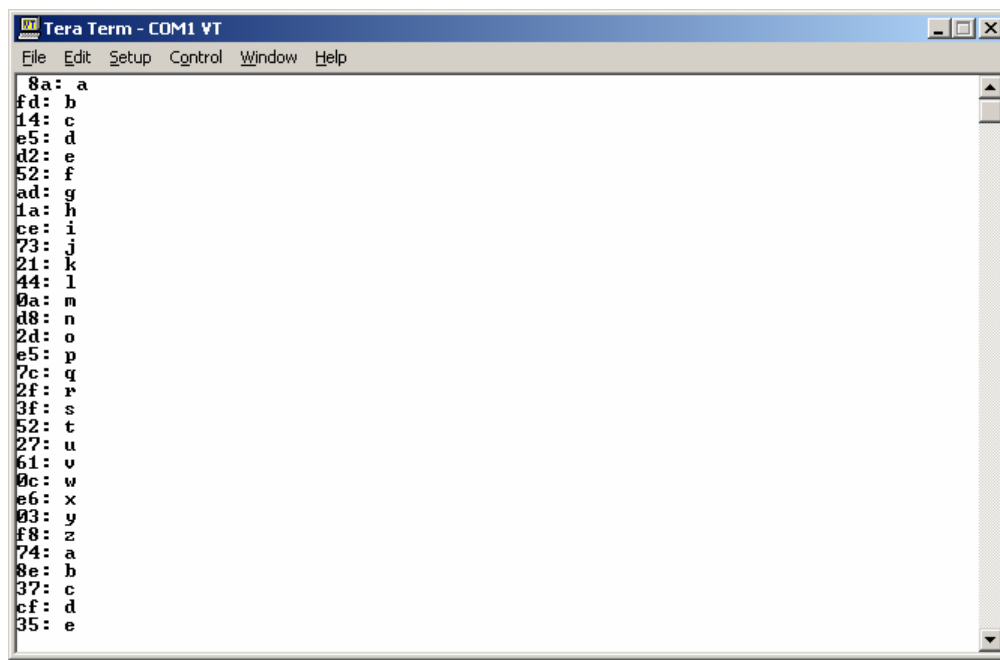
Remote Main File: maintutorial4remote.c

Similar to the local main file, maintutorial4remote.c is heavily based on the code used in Tutorial 1. Also similar to the local main file, we keep a key variable around. This key must match the one in the local file **exactly**. This includes case of letters and length of the key (keylen #define).

Once into the main routine, the Initialize() function here also calls arc4_initialize_ksa() to get S set up with the proper values, and the nRF24L01 is set up for TX and 2 data bytes.

In the main loop, we begin by waiting 1 second to allow the other device to get powered up (after each subsequent iteration, this will simply be a 1 second delay between letters). Then, the appropriate letter is sent over the RF link. Once we have sent the byte, we increment the letter to the next lower case letter in the alphabet (and start over at "a" if we are at "z"). Finally, we increment the packet counter, toggle our LED, and move back to the top of the loop.

Actual Program Output

In Figure 3, you can see the window in Tera Term. The display shows the entire alphabet and wraps back around.

```
 8a: a
fd: b
14: c
e5: d
d2: e
52: f
ad: g
1a: h
ce: i
73: j
21: k
44: l
0a: m
d8: n
2d: o
e5: p
7c: q
2f: r
3f: s
52: t
27: u
61: v
0c: w
e6: x
03: y
f8: z
74: a
8e: b
37: c
cf: d
35: e
```

Figure 3. Output window from Tera Term

Concluding Remarks

At this point, you should know how to encrypt data using ARC4 and send it over an nRF24L01 link. Remember that it is certainly possible to crack ARC4 (as well as RC4), so be very careful if you choose this algorithm for more sensitive projects. If you are particularly worried about the security of your data, you should probably be buying somebody's encryption algorithm anyway ☺. If you have any questions, feel free to email me at brennen@diyembedded.com.