```
// EncryptData_Initial.cpp
    __asm {
        // Copy the parameters before we set up our stack frame.
        mov esi,data
        mov edx,dataLength
        dec edx
        // Set up the main stack frame.
        push ebp
        mov ebp,esp
        /*
         *   Put every variable we will need into the stack.
         *   Easy Reference Table:
         *       -28     hop_count
         *       -24     starting_index
         *       -20     Key[]
         *       -16     pass Hash
         *       -12     gNumRounds
         *       -8      data[]
         *       -4      dataLength
         *       ebp     old ebp
         *
         *   Pushing 2 empty values at the end to reserve stack space
         *   for hop count and starting index
         *
         */
        push edx                // dataLength
        push esi                // data[]
        mov edx,gNumRounds
        dec edx
        push edx                // gNumRounds
        mov esi,gptrPasswordHash
        push esi                // passwordHash[]
        mov esi,gptrKey
        push esi                // Key[]
        xor eax,eax
        push eax                // reserve space on stack for starting index
        push eax                // reserve space on stack for hop count

        /*
         * Set up for the main looping structure
         * Using eax to get results from function calls
         * ebx will be used if needed to debug
         * Using ecx as the counter variable
         */
        xor eax,eax         // result storer
        xor ebx,ebx         // debug info (not used)
        xor ecx,ecx         // round counter = 0

/*
 * LOOP_ROUNDS
 *     1. Calls function to calculate the starting index,
 *      then stores the result in the reserved stack space.
 *     2. Calls function to calculate hop count,
 *      then stores the result in the reserved stack space.
 *     3. Saves the counter variable.
 *     4. Data Loop Structure
 */
lbl_LOOP_ROUNDS:
        call calculateStartingIndex // 1
        mov [ebp-24],eax            // 1
        call calculateHopCount      // 2
        mov [ebp-28],eax            // 2

        push ecx                    // 3
        xor ecx,ecx
```

```
/*
 * LOOP_DATA
 *     1. Calls function to xor the data byte with key file byte
 *     2. Calls function to increment the index by the hop count
 *     3. Calls step C
 *     4. Calls Step D
 *     5. Calls Step E
 *     6. Calls Step B
 *     7. Calls Step A
 */
lbl_LOOP_DATA:
        // DATA ENCRYPTION STEPS
        call xorByte            // 1
        call incrementIndex     // 2
        call stepC              // 3
        call stepD              // 4
        call stepE              // 5
        call stepB              // 6
        call stepA              // 7

        // LOOP_DATA control logic
        cmp ecx,[ebp-4]         // check if reached end of data
        je lbl_EXIT_LOOP_DATA   // end; exit data loop
        inc ecx
        jmp lbl_LOOP_DATA       // not end; keep looping

lbl_EXIT_LOOP_DATA:
        pop ecx
        // END data loop

        // LOOP_ROUND control logic
        cmp ecx,[ebp-12]        // check if current round = num rounds
        je lbl_EXIT_LOOP_ROUNDS // yes; exit round loop
        inc ecx
        jmp lbl_LOOP_ROUNDS     // no; keep looping

//                      FUNCTIONS
/*
 *   calculateStartingIndex
 *       1. Copy the password hash stored in the stack to esi
 *       2. Copy the [0+round*4]th byte of password hash to al
 *       3. Arithmatic shift left eax 8 times; equivalent to multiplying by 256
 *       4. Copy the [1+round*4]th byte of password hash to bl
 *       5. Add eax and ebx
 *       6. Return
 *
 *       Note:
 *        The result is in eax so the caller can obtain the value.
 */
calculateStartingIndex:
        mov esi,[ebp-16]                    // 1
        xor eax,eax
        mov al,byte ptr[esi+ecx*4]          // 2
        sal eax,8                           // 3
        inc esi
        mov bl, byte ptr[esi + ecx * 4]     // 4
        add eax,ebx                         // 5
        ret
        // END calculateStartingIndex
```

```
/*
 *      calculateHopCount
 *      1. Copy the password hash from the stack to esi
 *      2. Copy the [2+round*4]th byte of the hash to al
 *      3. Arithmatic shift left eax 8 times; (eax * 256)
 *      4. Copy the [3+round*4]th byte of the hash to bl
 *      5. Sum eax and ebx
 *      6. Check if eax is 0
 *          6a. If eax is 0, set it to 65536 (0xFFFF)
 *
 *      Note:
 *       The result is in eax so caller can get value
 */
calculateHopCount:
        mov esi,[ebp-16]            // 1
        add esi,2
        xor eax,eax
        mov al,byte ptr[esi+ecx*4]  // 2
        sal eax,8                   // 3
        inc esi
        mov bl,byte ptr[esi+ecx*4]  // 4
        add eax,ebx                 // 5
        cmp eax,0                   // 6 (check)
        je lbl_FIX_HOP
        ret

    lbl_FIX_HOP:                    // 6a
        mov eax,0xFFFF
        ret
        // END calculateHopCount

xorByte:
        push ebx
        xor ebx,ebx
        mov ebx,[ebp-24]    // get index from stack frame
        mov esi,[ebp-8]     // get *data from stack frame
        mov edi,[ebp-20]    // get *gKey from stack frame
        mov al,byte ptr[edi+ebx]   // al = gKey[index]
        xor al,byte ptr[esi+ecx]   // al = al ^ data[x]
        mov byte ptr[esi+ecx],al   // update data buffer byte
        pop ebx
        ret

incrementIndex:
        mov eax,[ebp-24]    // eax = index
        add eax,[ebp-28]    // index += hopcount
        // fix index if necessary
        cmp eax,65537
        jae lbl_FIX_INDEX   // fix
        mov [ebp-24],eax    // set new index
        ret                 // dont fix

    lbl_FIX_INDEX:
        sub eax,65537
        mov [ebp-24],eax    // set new index
        ret
```

```
stepA:
        push ebx
        xor ebx,ebx
        mov esi,[ebp-8]             // get *data
        mov al,byte ptr[esi+ecx]    // get data[x]
        xor ebx,ebx
        mov bh,0xAA                 // 0xAA has all even bits 1 and odd 0,
                                    // bitwise and with this value will get all even bits
        and bh,al                   // bh is now all the even bits of our byte
        mov bl,0x55                 // 0x55 has all even bits 0 and odd 1,
                                    // bitwise and with this value will get all odd bits
        and bl,al                   // bl is not all the odd bits of our byte
        shr bh,1                    // shift all even bits right 1 time
        shl bl,1                    // shift all odd bits left 1 time
        or bl,bh                    // combine them back together
        mov al,bl                   // al is now out byte with even and odd bits swapped
        mov byte ptr[esi+ecx],al    // update data buffer byte
        pop ebx
        ret

stepB:
        push ebx
        xor ebx,ebx
        mov esi,[ebp-8]             // get *data
        mov al, byte ptr[esi+ecx]   // get data[x]
        xor al,00111100b            // this inverts the middle 4 bits
        mov byte ptr[esi+ecx],al    // update data buffer byte
        pop ebx
        ret

stepC:
        mov esi,[ebp-8]             // get *data
        mov al,byte ptr[esi+ecx]    // get data[x]
        ror al,4                    // rotate 4 to right
        mov byte ptr[esi+ecx],al    // update data buffer byte
        ret

stepD:
        mov esi,[ebp-8]             // get *data
        xor eax,eax
        push ebx
        xor ebx,ebx
        mov al,byte ptr[esi+ecx]    // get data[x]
        lea edi, gEncodeTable       // put the address of the first byte of gEncodeTable into esi
        mov bl, byte ptr[edi+eax]   // copy the value at the index al from gEncodeTable
        mov byte ptr[esi + ecx],bl  // update data buffer byte
        pop ebx
        ret
```

```
stepE:
        push ebx                    // save old ebx value
        push edx                    // save old ecx value
        xor eax,eax
        mov esi,[ebp-8]             // get *data
        mov al,byte ptr[esi+ecx]    // get data[x]
        xor ebx,ebx                 // set ebx to 0, will be our counter
        xor edx,edx                 // set ecx to 0, will be the new reversed value
        jmp lbl_ELOOP               // start looping

    lbl_ELOOP:
            shl al,1                    // shift the right most bit into the carry
            rcr dl,1                    // rotate the carry into dl
            cmp ebx,7                   // compare counter to 7
            je lbl_EEND                 // if counter is 7 end the loop
            inc ebx                     // else increment count
            jmp lbl_ELOOP               // and keep looping

    lbl_EEND:
            mov byte ptr[esi + ecx],dl  // update data buffer byte
            pop edx                     // restore ecx
            pop ebx                     // restore ebx
            ret                         // return

//                      END FUNCTIONS

lbl_EXIT_LOOP_ROUNDS:
        // Restore the previous stack
        add esp,28
        pop ebp
    }
```