DBpia

논문번호 12-37A-09-11

한국통신학회논문지 '12-09 Vol.37A No.09
http://dx.doi.org/10.7840/kics.2012.37A.9.799

WAVE 시스템에서 스크램블러의 속도 향상을 위한 연구

이 대 식[◆°], 유 영 모[*], 이 상 윤[*], 오 세 갑[**]

Research for Improving the Speed of Scrambler in the WAVE System

Dae-sik Lee[◆°], Young-mo You[*], Sang-Youn Lee[*], Se-Kab Oh[**]

요 약

WAVE(Wireless Access for Vehicular Environment) 시스템에서 스크램블러의 비트 연산은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하여 효율성이 떨어지게 된다. 본 논문에서는 행렬 테이블에서 시작 위치를 찾는 알고리즘을 제안한다. 또한 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘, 행렬 테이블에서 시작 위치를 찾는 알고리즘을 8비트, 16비트, 32비트 단위로 처리하여 성능을 비교 분석한 결과 초당 처리 횟수는 8비트는 2917.8회, 16비트는 5432.1회, 32비트는 10277.8회 더 수행할 수 있었다. 따라서 행렬 테이블에서 시작 위치를 찾는 알고리즘이 WAVE 시스템에서 스크램블러의 속도를 향상시키고, 지능형 교통 체계(ITS)에서 노변장치와 차량(V2I) 또는 차량 사이의 통신(V2V)으로 다양한 정보 수집의 수신 속도와 정밀도를 향상시킬 수 있다.

Key Words : 스크램블러(Scrambler), WAVE System(Wireless Access for Vehicular Environment System), IEEE 802.11p, OFDM(Orthogonal Frequency Division Multiplexing), 행렬 테이블(Matrix Table)

ABSTRACT

Bit operation of scrambler in the WAVE System become less efficient because parallel processing is impossible in terms of hardware and software. In this paper, we propose algorism to find the starting position of the matrix table. Also, when bit operation algorithm of scrambler and algorithms for matrix table, algorithm used to find starting position of the matrix table were compared with the performance as 8 bit, 16bit, 32 bit processing units. As a result, the number of processing times per second could be done 2917.8 times more in an 8-bit, 5432.1 times in a 16-bit, 10277.8 times in a 32 bit. Therefore, algorithm to find the starting position of the matrix table improves the speed of the scrambler in the WAVE and the receiving speed of a variety of information gathering and precision over the Vehicle to Infra or Vehicle to Vehicle in the Intelligent Transport Systems.

Ⅰ. 서 론

WAVE 시스템[1]은 차량 단말기를 보유한 이용자에게 교통정보, 위치정보 및 안전에 관한 정보 등 다양한 서비스를 제공하기 위한 시스템으로 단거리 고속 무선패킷통신 분야를 담당하는 통신 기술이다. 현재 IEEE 802.11p의 표준화가 진행 중이다. 이러한 표준화 작업은 통신 기술발달로 차량에서 운송수단의 역할뿐만 아니라 위성항법장치(Global Positioning System, GPS)를 이용한 위치기반의 실시간 교통정보,

◆ 주저자 겸 교신저자 : 트라이콤텍(주) 부설 연구소, daesik@tricomtek.com, 정회원
* 트라이콤텍(주) 부설 연구소, youngmo@tricomtek.com, sangyoon@tricomtek.com
** 대전테크노파크 IT융합산업본부, skoh@djtp.or.kr
논문번호 : KICS2012-05-251, 접수일자 : 2012년 5월 15일, 최종논문접수일자 : 2012년 8월 8일

www.dbpia.co.kr

DMB(Digital Multimedia Broadcasting)를 통한 방송 서비스 등의 다양한 정보 제공이 가능하게 되었다. 그러나 IEEE 802.11p의 표준 규격에서 스크램블러의 비트 연산은 한비트씩 연산되어 병렬 처리가 불가능하므로 소프트웨어나 하드웨어 설계의 효율성이 떨어지게 된다.

본 논문에서는 WAVE 시스템의 속도 향상을 위하여 스크램블러에서 행렬 테이블에서 시작 위치를 찾는 알고리즘을 제안한다. 특히, 효율적인 성능분석을 하기 위하여 IEEE 802.11p의 표준 규격에서 스크램블러의 비트 연산 알고리즘(이하 "스크램블러의 비트 연산 알고리즘"이라 칭함)과 기존 논문에서 제안한 스크램블러의 행렬 테이블 구성 알고리즘(이하 "행렬 테이블 구성 알고리즘"이라 칭함), 그리고 행렬 테이블에서 시작 위치를 찾는 알고리즘을 실제로 구현하여 비교 분석해 보고자 한다.

본 논문의 구성은 2장에서 관련연구인 WAVE 시스템의 물리계층과 IEEE 802.11p 표준 규격의 스크램블러, 기존 논문에서 제안한 행렬 테이블을 구성하는 스크램블러를 살펴보고 3장에서 본 논문에서 제안하는 스크램블러 알고리즘을 설명한다. 그리고 4장에서 실험 결과를 설명하고 5장에서 결론을 맺는다.

Ⅱ. 관련연구

2.1. WAVE 시스템의 물리계층

직교 주파수 분할 다중(OFDM : Orthogonal Frequency Division Multiplexing)[2,3] 모듈레이션은 순서대로 스크램블러(Scrambler), 길쌈 부호기(Convolutional Encoder), 인터리버(Interleaver), 주파수 변조 기법에 따른 심볼 값 매핑(Modulator), 파일롯 신호 추가(Add Pilot), 퓨리에 역변환(IFFT : Inverse Fast Fourier Transform) 연산, 심볼 보호 구간 추가(Add Guard Interval), 심볼 정형과정(Symbol Wave Shapping), 위상변조기(IQ) 과정을 거친 후 무선주파수(RF : Radio Frequency)를 통해 데이터가 전달된다.

WAVE 물리계층에서 OFDM 모듈레이션[4,5]은 그림 1과 같다.

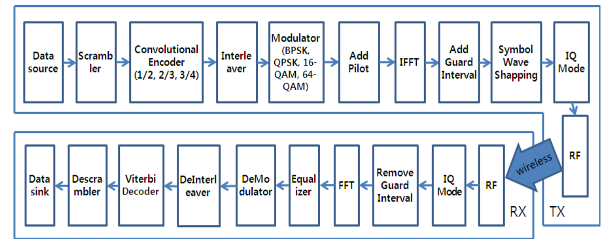


그림 1. OFDM 모듈레이션
Fig. 1. OFDM Modulation

그림 1에서 보면 데이터 소스가 전달되면 데이터 암호화, 데이터의 연속성이나 반복 패턴 등의 현상을 방지하고자 스크램블러 연산을 한다. 다음 콘볼루션 엔코더는 무선상의 산발적인 에러에 강한 코딩을 하고, 모듈레이션에 따라 1/2, 2/3, 3/4 부호화율로 구성된다. 콘볼루션 엔코딩이 완료된 데이터는 무선상 블록 에러를 산발 에러로 바꾸어 줄 수 있는 인터리버를 통과하고, 인터리버는 모듈레이션에 따라 레지스터 용량이 다르게 적용되어 데이터 비트가 섞이게 된다. 다음은 각각의 주파수 변조 기법에 따라 비트를 묶어 IQ 데이터로 매핑하고 도플러 현상과 신호의 세기를 통해 실시간으로 채널 조정을 해주기 위해 파일롯 신호를 추가하여 IFFT 연산을 한다. IFFT 연산을 통해 생성된 각각의 심볼의 앞부분에 심볼간 간섭을 억제하기 위해 IFFT 심볼 크기의 1/4에 해당하는 GI를 추가한다. 생성된 OFDM 심볼들은 심볼 값에 따라 주파수 위상이 크게 변하는 것을 막기 위한 정형과정을 거치고, IQ 모드에서 반송파와 곱해진다. 수신 OFDM 모듈레이션은 송신 OFDM 모듈레이션의 역으로 이루어진다.

2.2. IEEE 802.11p 표준 규격의 스크램블러

스크램블러는 데이터의 암호화, 반복적 패턴을 갖는 비트 블록 방지, 비트 값들 연속성(Null Data) 방지 등을 보장하기 위해 사용된다. WAVE 시스템의 스크램블러는 7개의 쉬프트 레지스터와 XOR 연산을 포함한다[6-8]. 스크램블러의 하드웨어 설계는 그림 2와 같다.

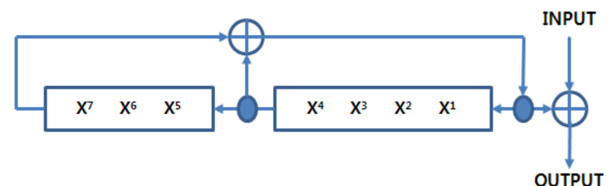


그림 2. 스크램블러의 하드웨어 설계
Fig. 2. Hardware Design of Scrambler

www.dbpia.co.kr

그림 2를 구현하는 알고리즘을 제시하면 알고리즘 1과 같다.

```
Void Gen_scrambler_calculation( ) {
 /* Bit arithmetic function */
Uint8 seed_src = seed, seed_val = 0; /* Initial value set */
int i, ii = 0;
Uint8* src, val;
src = input;
val = output;
for(ii = 0 ; ii < input_size ; ii++)  {
   for(i = 0 ; i < 8 ; i++)  {
   seed_val = ((seed_src & 0x08)/0x08) ^
   ((seed_src & 0x40)/0x40);  /* x4 and x7 XOR operation */
   seed_src = (seed_src * 0x02) + seed_val;
   /* Shift operation and x1 add */
   *val += (*src<<(i % 8)) ^
   (seed_val<<(i % 8));  }
   /* Input data and seed_val XOR operation */
   val++;   src++;  }  }
```

알고리즘. 1. 스크램블러의 비트 연산 알고리즘
Algorithm 1. Bit arithmetic algorithm of Scrambler

알고리즘 1을 보면 Scrambler_Calculation 함수로 스크램블러의 비트 연산을 한다. 스크램블러의 비트 연산은 7개의 레지스터에 입력되는 초기값 seed를 seed_src로 할당하고, 변수 seed_val, i, ii의 초기값은 0으로 할당한다. 또한 입력 데이터는 src, 출력 데이터는 val에 할당한다.

알고리즘 1에 초기값 seed “1111111”을 적용하여 7개의 레지스터에 입력하면 그림 3과 같다.

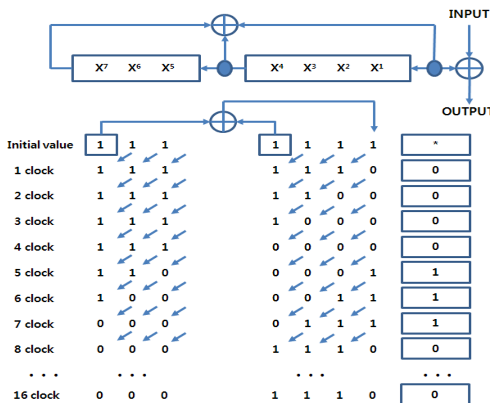


그림 3. 스크램블러의 비트 연산
Fig. 3. Bit Operation of Scrambler

그림 3에서 입력 데이터를 “10000010 01000010” 16비트로 예를 들어 설명하면 다음과 같다.

초기단계는 초기값 seed “1111111”을 seed_src에 할당하고 동작한다.

1클럭 동작은 초기값 “1111111”을 7개의 레지스터에 입력하여 x^4와 x^7의 XOR 연산으로 생성된 스크램블러 비트 “0”을 seed_val에 할당하고, 시프트 연산과 스크램블러 비트 “0”을 X^1에 추가하여 2클럭 동작을 위해 seed_src에 할당한다. 입력 데이터의 첫 번째 비트 “1”과 스크램블러 비트 “0”을 XOR 연산하여 출력 데이터 비트 “1”을 val에 저장한다.

2클럭 동작은 seed_src “1111110”을 7개의 레지스터에 입력하면 x^4와 x^7의 XOR 연산으로 생성된 스크램블러 비트 “0”을 seed_val에 할당하고, 시프트 연산과 스크램블러 비트 “0”을 X^1에 추가하여 3클럭 동작을 위해 seed_src에 할당한다. 입력 데이터의 두 번째 비트 “0”과 스크램블러 비트 “0”을 XOR 연산하여 출력 데이터 비트 “0”을 val에 저장한다.

반복 동작으로 입력 데이터 비트 수만큼 16클럭까지 동작한다. 따라서 입력 데이터 “10000010 01000010”은 16비트이므로 16클럭까지 동작하여 각 클럭마다 스크램블러의 비트를 생성하여 16개의 입력 데이터 비트와 한비트씩 XOR 연산을 하여 출력 데이터 16비트 “10001100 10110000”을 생성한다.

하드웨어나 소프트웨어 측면에서 스크램블러의 비트 연산은 입력 데이터의 비트 수만큼 반복문의 Loop 횟수가 비정상적으로 많아지고, 프로세서가 32비트 이상 지원되어도 8비트 프로세서와 처리속도는 같다는 단점이 있다.

2.3. 행렬 테이블을 구성하는 스크램블러

행렬 테이블은 초기값을 7개 레지스터에 입력하여 스크램블러의 비트 연산으로 구성하며 초기값이 변경되면 행렬 테이블을 재구성해야 한다. 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 행렬 테이블의 단위와 병렬 처리할 수 있다[9].

행렬 테이블을 구성하는 스크램블러의 하드웨어 설계는 그림 4와 같으며 입력 데이터의 입력 단위를 8비트 기준으로 설계한 것이다.

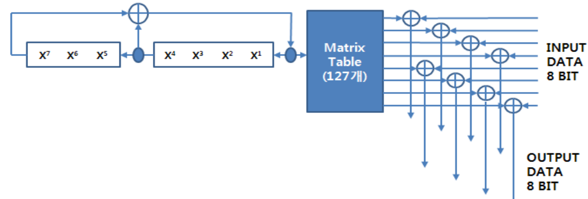


그림 4. 행렬 테이블을 구성하는 스크램블러의 하드웨어 설계
Fig. 4. Matrix Table to Configure the Hardware Design of Scrambler

www.dbpia.co.kr

그림 4는 초기값 "x^7 x^6 x^5 x^4 x^3 x^2 x^1"이 7개 레지스터에 입력되어 스크램블러의 비트 연산으로 127개의 서로 다른 비트열로 행렬 테이블을 구성한다. 구성된 행렬 테이블의 8비트 단위가 입력 데이터의 8비트 입력 단위와 병렬 처리할 수 있다. 따라서 그림 4와 같이 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬 처리할 수 있는 하드웨어를 설계할 수 있다.

그림 2에서는 입력 데이터가 16비트이면 각 클럭의 동작으로 생성된 스크램블러 비트와 한비트씩 XOR 연산을 16번 실행하여 출력 데이터를 생성한다. 하지만 그림 4에서는 스크램블러의 비트 연산으로 생성된 행렬 테이블의 8비트 단위와 병렬로 XOR 연산을 2번 실행하여 출력 데이터를 생성하므로 스크램블러의 성능이 향상된다.

그림 4를 구현하는 알고리즘을 제시하면 알고리즘 2, 알고리즘 3과 같다.

알고리즘 2는 스크램블러의 비트 연산으로 행렬 테이블을 구성하는 알고리즘이다. 입력 데이터 비트와 한비트씩 XOR 연산을 하여 출력 데이터 16비트 "10001100 1011000 0"을 생성한다.

알고리즘 2를 보면 Scrambler_Table 함수는 127개의 서로 다른 비트열로 행렬 테이블을 구성한다. 스크램블러의 비트 연산으로 행렬 테이블을 구성하기 위해 7개의 레지스터에 입력되는 초기값 seed를 seed_src로 할당하고, 변수 seed_val, i, ii의 초기값은 0으로 할당한다. memset 함수는 행렬 테이블을 초기화하는 함수이고, 행렬 테이블에 8비트씩 저장하기 위하여 8비트 정수형으로 정의한 Init_seq의 시작 포인터 위치를 ran_com에 할당한다.

알고리즘 2에 초기값 seed "1111111"을 적용하여 7개 레지스터에 입력하면 그림 3과 같이 스크램블러의 비트 연산으로 8클럭씩 동작하여 입력 데이터의 입력 단위 8비트와 같은 크기로 16개의 서로 다른 비트열이 생성된다. 스크램블러의 비트 연산으로 8클럭씩 동작하여 8비트 단위로 구성된 16개의 서로 다른 비트열은 표 1과 같다.

```
void Gen_scrambler_table(void) {
/* Matrix table configuration  Function */
Uint8 seed_src = seed, seed_val=0;
Uint8* ran_com = randomizer_table;
int i = 0, ii = 0;
memset(ran_com,0x00,1000);
/*  Mnitialization function of matrix table */
for(ii = 0 ; ii < 127 ; ii++)  {
/* 127 different bit calculation */
    for(i = 0 ; i < 8 ; i++)  {
    seed_val = ((seed_src & 0x08)/0x08) ^
    ((seed_src & 0x40)/0x40);
    /* x4 and x7 XOR operation */
    seed_src = (seed_src * 0x02) + seed_val;
    /* Shift operation and x1 add */
    *ran_com += seed_val<<(i % 8);  }
    /* Bit sequence stored in matrix table */
ran_com++;  }  }
```

알고리즘 2. 행렬 테이블 구성 알고리즘
Algorithm. 2. Matrix table construction algorithm

표 1. 16개 비트열
Table 1. 16 Bit String

1111 111	0000 1110	1111 0010	1100 1001	0000 0010	0010 0110	0010 1110	1011 0110
0000 1100	1101 0100	1110 0111	1011 0100	0010 1010	1111 1010	0101 0001	1011 1000

표 1과 같이 초기값 seed "1111111"을 seed_src에 할당하고, 7개 레지스터에 입력하여 그림 3과 같이 스크램블러의 비트 연산으로 16개의 서로 다른 비트열을 구성한다. 16개의 서로 다른 비트열은 행렬 테이블의 1바이트 단위 8비트 만큼 순환루프를 반복 동작하여 128(16*8 = 128)번째 7개 레지스터에 입력되는 비트열은 초기값 "1111111"이 되므로 127개의 서로 다른 비트열로 행렬 테이블을 구성할 수 있다. 따라서 행렬 테이블을 구성하여 입력 데이터의 입력 단위와 병렬 처리할 수 있다.

알고리즘 2와 같이 1바이트 단위의 행렬 테이블은 127번의 for 문을 실행하고, 1번의 for 문으로 8클럭 동작하여 127개의 서로 다른 비트열로 행렬 테이블을 구성된다. 따라서 입력 데이터의 전체 크기가 127개 이상 되어도 스크램블러의 비트 연산을 하지 않고 행렬 테이블을 8비트 단위로 반복 순환 적용하여 병렬 처리할 수 있다.

스크램블러의 비트 연산으로 8클럭씩 동작하여 8비트 단위로 구성된 127개의 서로 다른 비트열은 표 2와 같다.

www.dbpia.co.kr

표 2. 8비트로 구성된 행렬 테이블
Table 2. 8-Bit Matrix Table

<table>
<tr><td>1111
111</td><td>0000
1110</td><td>1111
0010</td><td>1100
1001</td><td>0000
0010</td><td>0010
0110</td><td>0010
1110</td><td>1011
0110</td></tr>
<tr><td>0000
1100</td><td>1101
0100</td><td>1110
0111</td><td>1011
0100</td><td>0010
1010</td><td>1111
1010</td><td>0101
0001</td><td>1011
1000</td></tr>
<tr><td>1111
1110</td><td>0001
1101</td><td>1110
0101</td><td>1001
0010</td><td>0000
0100</td><td>0100
1100</td><td>0101
1101</td><td>0110
1100</td></tr>
<tr><td>0001
1001</td><td>1010
1001</td><td>1100
1111</td><td>0110
1000</td><td>0101
0101</td><td>1111
0100</td><td>1010
0011</td><td>0111
0001</td></tr>
<tr><td>1111
1100</td><td>0011
1011</td><td>1100
1011</td><td>0010
0100</td><td>0000
1000</td><td>1001
1000</td><td>1011
1010</td><td>1101
1000</td></tr>
<tr><td>0011
0011</td><td>0101
0011</td><td>1001
1110</td><td>1101
0000</td><td>1010
1011</td><td>1110
1001</td><td>0100
0110</td><td>1110
0011</td></tr>
<tr><td>1111
1000</td><td>0111
0111</td><td>1001
0110</td><td>0100
1000</td><td>0001
0001</td><td>0011
0001</td><td>0111
0101</td><td>1011
0000</td></tr>
<tr><td>0110
0110</td><td>1010
0111</td><td>0011
1101</td><td>1010
0001</td><td>0101
0111</td><td>1101
0010</td><td>1000
1101</td><td>1100
0111</td></tr>
<tr><td>1111
0000</td><td>1110
1111</td><td>0010
1100</td><td>1001
0000</td><td>0010
0010</td><td>0110
0010</td><td>1110
1011</td><td>0110
0000</td></tr>
<tr><td>1100
1101</td><td>0100
1110</td><td>0111
1011</td><td>0100
0010</td><td>1010
1111</td><td>1010
0101</td><td>0001
1011</td><td>1000
1111</td></tr>
<tr><td>1110
0001</td><td>1101
1110</td><td>0101
1001</td><td>0010
0000</td><td>0100
0100</td><td>1100
0101</td><td>1101
0110</td><td>1100
0001</td></tr>
<tr><td>1001
1010</td><td>1001
1100</td><td>1111
0110</td><td>1000
0101</td><td>0101
1111</td><td>0100
1010</td><td>0011
0111</td><td>0001
1111</td></tr>
<tr><td>1100
0011</td><td>1011
1100</td><td>1011
0010</td><td>0100
0000</td><td>1000
1001</td><td>1000
1011</td><td>1010
1101</td><td>1000
0011</td></tr>
<tr><td>0011
0101</td><td>0011
1001</td><td>1110
1101</td><td>0000
1010</td><td>1011
1110</td><td>1001
0100</td><td>0110
1110</td><td>0011
1111</td></tr>
<tr><td>1000
0111</td><td>0111
1001</td><td>0110
0100</td><td>1000
0001</td><td>0001
0011</td><td>0001
0111</td><td>0101
1011</td><td>0000
0110</td></tr>
<tr><td>0110
1010</td><td>0111
0011</td><td>1101
1010</td><td>0001
0101</td><td>0111
1101</td><td>0010
1000</td><td>1101
1100</td><td>0111
1111</td></tr>
</table>

표 2와 같이 행렬 테이블의 8비트 단위와 입력 데이터의 8비트 입력 단위를 병렬로 XOR 연산을 한다.

알고리즘 2에서 127개의 서로 다른 비트열로 구성된 행렬 테이블을 입력 데이터의 8비트 입력 단위와 병렬 처리하는 알고리즘은 알고리즘 3과 같다.

```
void Randomizer() {
/* Matrix table and Input data Parallel processing function */
Uint8* ran_com, com; /* Matrix table pointer */
int i = 0;
Uint8* src, val;
Uint8 counter = 0;
com = Init_seq;  /* Start pointer of  matrix table assign */
src = input;
val = output;
for(i = 0 ; i < input_size ; i++)  {
/* Matrix table and Input data Parallel XOR operation */
    *val = *src ^ *com;
    val++;  src++;  com++;  counter++;
    if(counter == 127)  {
        counter = 0;
        com = ran_com;  }  }  }
```

알고리즘 3. 행렬 테이블과 입력 데이터를 병렬 처리하는 알고리즘
Algorithm. 3. Matrix table and Input data Parallel processing algorithm

알고리즘 3을 보면 Randomizer 함수는 127개의 서로 다른 비트열로 구성된 행렬 테이블을 입력 데이터의 입력 단위와 병렬 처리한다. 변수 i, count, src, val을 지정하고, 행렬 테이블의 8비트 정수형으로 정의한 시작 포인터 Init_seq을 com에 할당하여 시작 위치를 지정한다. 입력 데이터 input은 src, 출력 데이터 output은 val에 할당한다.

입력 데이터는 “10000010 01000010” 16비트와 행렬 테이블은 표 2로 예를 들어 설명하면 다음과 같다.

알고리즘 3에 적용하여 행렬 테이블의 8비트 단위와 입력 데이터의 8비트 입력 단위가 병렬로 XOR 연산을 한다. 행렬 테이블의 시작 포인터 위치 com으로부터 8비트 “00001110”과 입력 데이터의 입력 단위인 src 8비트 “10000010”은 병렬로 XOR 연산하여 출력 데이터 8비트 “10001100”을 val에 저장하고, count, com, src, val은 증가한다. 또한 127개의 서로 다른 비트열로 구성된 행렬 테이블은 count가 127과 같으면 행렬 테이블의 마지막을 지났으므로 count = 0, 행렬 테이블의 시작 포인터 위치 Init_seq를 다시 com으로 할당한다.

행렬 테이블에서 8비트 증가한 다음 포인터 위치 com으로부터 8비트 “11110010”과 다음 입력 데이터의 src 8비트 “01000010”을 병렬로 XOR 연산

www.dbpia.co.kr

하여 출력 데이터 8비트 "10110000"을 val에 추가하여 저장하고, count, com, src, val은 증가한다. 또한 행렬 테이블은 127개의 서로 다른 비트열로 count가 127과 같으면 행렬 테이블의 마지막을 지났으므로 count = 0, 행렬 테이블의 시작 포인터 위치 Init_seq를 다시 com으로 할당한다.

알고리즘 3을 적용한 결과 입력 데이터의 전체 비트가 16비트이므로 행렬 테이블의 8비트 단위와 병렬로 XOR 연산하여 출력 데이터 val에 "10001100 10110000"을 저장하고 프로그램을 종료한다.

따라서 스크램블러의 비트 연산 알고리즘에서 16번 실행하여 출력 데이터를 생성하였고, 행렬 테이블 구성하는 알고리즘에서는 2번 실행하여 출력 데이터를 생성하였다. 행렬 테이블 구성 알고리즘은 127개의 서로 다른 비트열로 구성된 행렬 테이블로 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬로 XOR 연산을 할 수 있으므로 WAVE 시스템에서 스크램블러의 처리 속도를 향상시켰다.

Ⅲ. 제안한 스크램블러 알고리즘

행렬 테이블 구성 알고리즘은 초기값이 변경되면 스크램블러의 비트 연산에 의해 행렬 테이블을 재구성하는 단점이 있다.

본 논문에서는 행렬 테이블이 8비트, 16비트, 32비트, 64비트에 따라 스크램블러의 비트 연산을 하여 127개의 서로 다른 비트열을 행렬 테이블로 미리 구성하여 배열로 저장한다. 따라서 행렬 테이블을 구성하는 알고리즘이 필요 없으며, 미리 구성된 행렬 테이블의 초기값에 따라 시작 위치를 선택해 입력 데이터의 입력 단위와 병렬 처리할 수 있는 알고리즘을 제안한다. 제안한 스크램블러의 하드웨어 설계는 그림 5와 같으며 입력 데이터의 입력 단위를 8비트 기준으로 설계한 것이다.

그림 5는 미리 구성된 8비트 단위의 행렬 테이블에서 초기값 "x^1 x^2 x^3 x^4 x^5 x^6 x^7"의 시작 위치를 찾아 입력 데이터의 8비트 입력 단위와 병렬로 XOR 연산을 한다.

따라서 그림 5와 같이 입력 데이터의 입력 단위가 8비트, 16비트, 32비트, 64비트이든 병렬 처리할 수 있는 하드웨어를 설계할 수 있다.

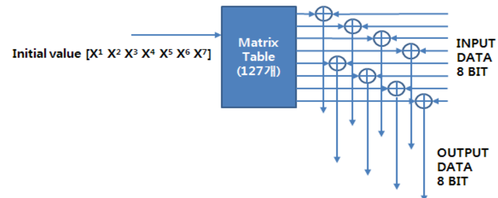


그림 5. 제안한 스크램블러의 하드웨어 설계
Fig. 5. Hardware Design of Proposed Scrambler

그림 2에서는 입력 데이터가 16비트이면 그림 2는 각 클럭의 동작으로 16번 실행하여 출력 데이터를 생성하고, 그림 4와 그림 5는 병렬로 XOR 연산을 2번 실행하여 출력 데이터를 생성한다. 하지만 그림 4에서는 스크램블러 비트 연산으로 행렬 테이블을 구성하여 병렬로 XOR 연산을 하고, 그림 5에서는 입력 데이터의 입력 단위에 따라 미리 행렬 테이블을 구성하여 시작 위치를 찾아서 병렬로 XOR 연산을 한다.

그림 5를 구현하는 알고리즘을 제시하면 알고리즘 3, 알고리즘 4와 같다.

알고리즘 3은 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘이고, 알고리즘 4는 행렬 테이블에서 시작 위치를 찾는 알고리즘이다.

```
Uint8 Scrambler_Table[127]={
    0xfe, 0x70, 0x4f, 0x93, 0x40,
    ……
    0xbe, 0x14, 0x3b };
Scrambler_Search() {
input_size=1080;
printf("Data Input : 0x");
scanf("%x", seed);
Switch(seed) {
    Case( 0xfe )
        Init_seq = &scrambler_table[0];
        Break;
    Case ( 0x70 ) :
        Init_seq = &scrambler_table[1];
        Break;
    ………
    Case ( 0x4f ) :
        Init_seq = &scrambler_table[126];
        Break;    }
Randomizer();
```

알고리즘 4. 행렬 테이블에서 시작 위치를 찾는 알고리즘
Algorithm. 4. Start position finding Algorithm in matrix table

www.dbpia.co.kr

알고리즘 4를 보면 Scrambler_Table 배열은 127개의 서로 다른 비트열로 행렬 테이블을 미리 구성하여 배열로 저장한다. Scrambler_Search 함수는 입력데이터 크기를 input_size로 정의하고, 시작 위치를 찾는 seed 데이터를 입력받아 Switch-Case 문으로 행렬 테이블에서 시작 위치를 찾는다. 또한 행렬 테이블에서 시작 위치로부터 8비트씩 입력 데이터와 병렬로 XOR 연산하는 알고리즘 3의 Randomizer 함수를 실행한다. 여기서 Switch-Case 문으로 seed 데이터를 비교하여 행렬 테이블에서 시작 위치를 선택해야 함으로 127개 Case 문 중 어떤 seed 데이터로 시작 위치를 선택하느냐에 따라 처리 속도가 달라지므로 처리 속도가 일정하지 않다. 하지만 시작 위치를 선택해야 할 seed 데이터가 127번째 Case 문에 있다고 하여도 행렬 테이블에서 시작 위치를 찾는 알고리즘의 처리 속도는 가장 빠르다.

따라서 행렬 테이블 구성 알고리즘은 초기값이 변경되면 행렬 테이블을 재구성해야 하지만, 본 논문에서 제안한 행렬 테이블에서 시작 위치를 찾는 알고리즘은 초기값이 변경되면 행렬 테이블을 재구성하지 않고 행렬 테이블에서 시작 위치를 다시 찾아서 사용하므로 행렬 테이블을 다시 생성하기 위한 스크램블러의 비트 연산 시간을 없기 때문에 행렬 테이블을 구성하는 알고리즘보다 WAVE 시스템에서 스크램블러의 처리 속도를 향상시킨다.

Ⅳ. 실험 결과

본 논문에서는 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘, 행렬 테이블에서 시작 위치를 찾는 알고리즘을 8비트, 16비트, 32 비트 단위로 처리하여 비교 분석하기 위해 실제 WAVE DSP(Digital Signal Processor) 보드로 실험하였고, 실험사양은 표 3과 같다.

표 3. 실험 사양
Table 3. Test Specifications

System	WAVE DSP BOARD
Operating System	DSP/BIOS
CPU	TMS320C6455
CPU Speed	1.2GHz
Memory	256MB
Language	C Language

본 논문에서는 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘, 행렬 테이블에서 시작 위치를 찾는 알고리즘의 타당성을 검증하기 위해 입력 데이터의 처리 단위를 8비트는 실험 1, 16비트는 실험 2, 32비트는 실험 3으로 실험하였다. 또한 입력 데이터의 처리 단위마다 추가적인 메모리 할당 부분이 필요하다.

실험 1, 2, 3에서는 입력 데이터의 크기는 1080Byte로 하고, 처리속도나 처리횟수는 10회 측정한 것을 평균값으로 표현하였다.

[실험 1]

실험 1에서는 표 2와 같이 행렬 테이블을 8비트 단위로 처리하였고, 처리속도나 처리횟수는 표 4와 같다.

표 4. 8비트 처리속도와 처리횟수
Table 4. 8-Bit Processing Speed and Processing Times

8Bit (Processing speed)	Bit arithmetic algorithms of Scrambler	Matrix table construction algorithm	Proposed algorithm
1Time	About 1.53ms	About 0.95ms	About 0.28ms
50000Times	About 76.77s	About 47.67s	About 14.14s
Per second	About 653.6Times	About 1052.6Times	About 3571.4Times
Memory	7Bit(Operation) +1Bit	127Byte+7Bit(Operation)	127Byte

[실험 2]

실험 2에서는 표 5와 같이 행렬 테이블을 16비트 단위로 처리하였고, 처리속도나 처리횟수는 표 6과 같다.

표 6. 16트 처리속도와 처리횟수
Table 6. 16-Bit Processing Speed and Processing Times

16Bit (Processing speed))	Bit arithmetic algorithms of Scrambler	Matrix table construction algorithm	Proposed algorithm
1Time	About 0.81ms	About 0.50ms	About 0.15ms
50000Times	About 40.64s	About 25.24s	About 7.48s
Per second	About 1234.6Times	About 2000.1Times	About 6666.7Times
Memory	7Bit(Operation) +1Bit	254Byte+7Bit (Operation)	254Byte

[실험 3]

실험 3에서는 표 7과 같이 행렬 테이블을 32비트 단위로 처리하였고, 처리속도나 처리횟수는 표 8과 같다.

www.dbpia.co.kr

표 5. 16비트로 구성된 행렬 테이블
Table 5. 16-Bit Matrix Table

0xfe3b	0x4f70	0x4093	0x7464	0x306d	0xe72b	0x542d	0x8a5f
0x7f1d	0xa7b8	0x2049	0xba32	0x9836	0xf395	0xaa16	0xc52f
0x3f8e	0xd3dc	0x1024	0x5d19	0xcc1b	0x79ca	0xd50b	0x6297
0x1fc7	0x69ee	0x8812	0xae8c	0x660d	0xbce5	0xea85	0xb14b
0x0fe3	0x34f7	0x4409	0xd746	0xb306	0xde72	0xf542	0xd8a5
0x87f1	0x9a7b	0x2204	0x6ba3	0x5983	0x6f39	0xfaa1	0xec52
0xc3f8	0x4d3d	0x9102	0xb5d1	0xacc1	0xb79c	0x7d50	0x7629
0xe1fc	0x269e	0xc881	0xdae8	0x5660	0x5bce	0xbea8	0x3b14
0x70fe	0x934f	0x6440	0x6d74	0x2b30	0x2de7	0x5f54	0x1d8a
0xb87f	0x49a7	0x3220	0x36ba	0x9598	0x16f3	0x2faa	0x8ec5
0xdc3f	0x24d3	0x1910	0x1b5d	0xcacc	0x0b79	0x97d5	0xc762
0xee1f	0x1269	0x8c88	0x0dae	0xe566	0x85bc	0x4bea	0xe3b1
0xf70f	0x0934	0x4644	0x06d7	0x72b3	0x42de	0xa5f5	0xf1d8
0x7b87	0x049a	0xa322	0x836b	0x3959	0xa16f	0x52fa	0xf8ec
0x3dc3	0x024d	0xd191	0xc1b5	0x9cac	0x50b7	0x297d	0xfc76
0x9ee1	0x8126	0xe8c8	0x60da	0xce56	0xa85b	0x14be	

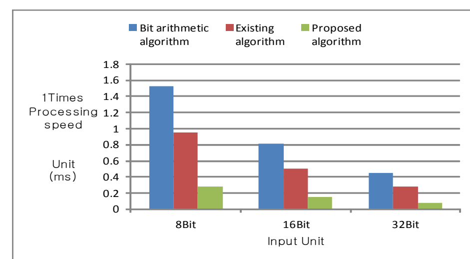
표 7. 32비트로 구성된 행렬 테이블
Table 7. 32-Bit Matrix Table

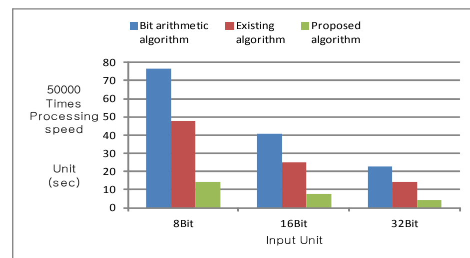
0xfe3b14be	0x40934f70	0x306d7464	0x542de72b	0x7f1d8a5f	0x2049a7b8	0x9836ba32	0xaa16f395
0x3f8ec52f	0x1024d3dc	0xcc1b5d19	0xd50b79ca	0x1fc76297	0x881269ee	0x660dae8c	0xea85bce5
0x0fe3b14b	0x440934f7	0xb306d746	0xf542de72	0x87f1d8a5	0x22049a7b	0x59836ba3	0xfaa16f39
0xc3f8ec52	0x91024d3d	0xaoc1b5d1	0x7d50b79c	0xe1fc7629	0xc881269e	0x5660dae8	0xbea85bce
0x70fe3b14	0x6440934f	0x2b306d74	0x5f542de7	0xb87f1d8a	0x322049a7	0x959836ba	0x2faa16f3
0xdc3f8ec5	0x191024d3	0xcacc1b5d	0x97d50b79	0xee1fc762	0x8c881269	0xe5660dae	0x4bea85bc
0xf70fe3b1	0x46440934	0x72b306d7	0xa5f542de	0x7b87f1d8	0xa322049a	0x3959836b	0x52faa16f
0x3dc3f8ec	0xd191024d	0x9cacc1b5	0x297d50b7	0x9ee1fc76	0xe8c88126	0xce5660da	0x14bea85b
0x4f70fe3b	0x74644093	0xe72b306d	0x8a5f542d	0xa7b87f1d	0xba322049	0xf3959836	0xc52faa16
0xd3dc3f8e	0x5d191024	0x79cacc1b	0x6297d50b	0x69ee1fc7	0xae8c8812	0xbce5660d	0xb14bea85
0x34f70fe3	0xd7464409	0xde72b306	0xd8a5f542	0x9a7b87f1	0x6ba32204	0x6f395983	0xec52Faa1
0x4d3dc3f8	0xb5d19102	0xb79cacc1	0x76297d50	0x269ee1fc	0xdae8c881	0x5bce5660	0x3b14bea8
0x934f70fe	0x6d746440	0x2de72b30	0x1d8a5f54	0x49a7b87f	0x36ba3220	0x16f39598	0x8ec52faa
0x24d3dc3f	0x1b5d1910	0x0b79cacc	0xc76297d5	0x1269ee1f	0x0dae8c88	0x85bce566	0xe3b14bea
0x0934f70f	0x06d74644	0x42de72b3	0xf1d8a5f5	0x049a7b87	0x836ba322	0xa16f3959	0xf8ec52fa
0x024d3dc3	0xc1b5d191	0x50b79cac	0xfc76297d	0x81269ee1	0x60dae8c8	0xa85bce56	

표 8. 32트 처리속도와 처리횟수
Table 8. 32-Bit Processing Speed and Processing Times

32Bit (Processing speed)	Bit arithmetic algorithms of Scrambler	Matrix table construction algorithm	Proposed algorithm
1Time	About 0.45ms	About 0.28ms	About 0.08ms
50000Times	About 22.58s	About 14.02s	About 4.10s
Per second	About 2222.2Times	About 3571.4Times	About 12500Times
Memory	7Bit(Operation)+1Bit	508Byte+7Bit(Operation)	508Byte

본 논문에서 제안한 행렬 테이블에서 시작 위치를 찾는 알고리즘을 처리 단위 8비트, 16비트, 32비트로 1회와 50000회 실행한 결과 스크램블러의 비트 연산 알고리즘보다 처리 속도는 약 81.5% ~ 82.2%가 향상되고, 행렬 테이블 구성 알고리즘보다 약 70.3% ~ 71.4%가 향상되었다. 따라서 행렬 테이블에서 시작 위치를 찾는 알고리즘이 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘보다 스크램블러의 성능이 향상된 것을 알 수 있다. 그림 6은 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘, 본 논문에서 제안한 행렬 테이블에서 시작 위치를 찾는 알고리즘의 성능 분석 결과를 나타낸 것이다.





www.dbpia.co.kr

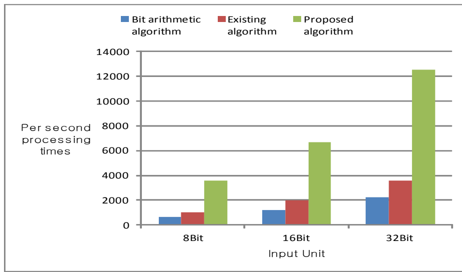


그림 6. 행렬 테이블에서 시작 위치를 찾는 알고리즘의 성능
Fig. 6. Performance of Algorithms to Find the Starting Position of the Matrix Table

V. 결 론

WAVE 시스템은 IEEE 802.11p를 기반으로 차량 단말기를 보유한 이용자에게 교통정보, 위치정보 및 안전에 관한 정보 등 다양한 서비스를 제공하기 위한 시스템으로 단거리 고속무선패킷통신 분야를 담당하는 통신 기술이다.

WAVE 시스템에서 스크램블러의 비트 연산 알고리즘은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하여 효율성이 떨어지게 된다. 따라서 기존 논문에서는 행렬 테이블을 구성하는 알고리즘과 입력 데이터와 행렬 테이블을 병렬 연산하는 알고리즘을 제안하였다.

본 논문에서는 행렬 테이블에서 시작 위치를 찾는 알고리즘을 제안하여 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘을 8비트, 16비트, 32비트로 처리하여 비교 분석하기 위해 실제 WAVE DSP 보드로 실험한 결과 행렬 테이블에서 시작 위치를 찾는 알고리즘이 스크램블러의 비트 연산 알고리즘과 행렬 테이블 구성 알고리즘보다 스크램블러의 성능이 향상된 것을 알 수 있었다.

따라서 본 논문에서 제안한 행렬 테이블에서 시작 위치를 찾는 알고리즘은 WAVE 무선통신 시스템에서 스크램블러의 성능을 더욱 업그레이드시켜 지능형 교통 체계에서 노변장치와 차량 또는 차량 사이의 통신으로 다양한 정보 수집의 수신 속도와 정밀도를 향상시키며 차세대 차량 시스템의 기본 기술로 적용될 수 있다.

참 고 문 헌

[1] Huynh Tronganh, Kim Jin Sang, Cho Won Kyung, “Hardware Design for Timing Synchronization of OFDM-Based WAVE Systems,” *J. Kor. Info. Comm. Soc. (J-KICS)*, Vol.33, No.4, pp.335- 486, 2008.

[2] T. M. Schmidl, D. C. Cox, “Robust frequency and timing synchronization for OFDM,” *IEEE Trans. Comm.*, Vol.45, No.12, pp.1613-1621, 1997.

[3] J. Chuang, N. Sollenberger, “Beyond 3G : Wideband Wireless Data Access Based on OFDM and Dynamic Packet Assignment,” *IEEE Comm. Mag.*, Vol. 38, No 7, 2000.

[4] Kwak Jae Min, “Implementation of Embedded System for IEEE802.11p based OFDM-DSRC Communications,” *Kor. Ins. Comm. Eng.*, Vol.10, No.11, pp.2062-2068, 2006.

[5] Jeongwook Seo, Jae-Min Kwak, Dong Ku Kim, “Simulation Performance of WAVE System with Combined DD-CE and LMMSE Smoothing Scheme in Small-Scale Fading Models,” *INTERNATIONAL JOURNAL OF KIMICS*, Vol.8, No.3, pp.281-288, 2010.

[6] ASTM Designation : E 2213-02e13, “Standard Specification for Telecommunications and Information Exchange Between Roadside and Vehicle Systems 5 GHz Band Dedicated Short Range Communications (DSRC) Medium Access Control (MAC) and Physical Layer (PHY) Specifications1,” 2003.

[7] IEEE std 802.11, “Wireless LAN Medium Access Control(MAC) and Physical Layer(PHY) specifications,” 2007.

[8] IEEE std 802.11pTM/D11.0, “Wireless LAN Medium Access Control(MAC) and Physical Layer(PHY) specifications Amendment 6: Wireless Access in Vehicular Environments,” 2010.

[9] Dae-sik Lee, Young-mo You, Sang-Youn Lee, Chung-ryong Jang, “The Algorithm Design and Implemention for Operation using a Matrix Table in the WAVE system,” *J. Kor. Info. Comm. Soc. (J-KICS),* Vol.37, No.4, pp.189-196, 2012.

www.dbpia.co.kr

이 대 식 (Dae-sik Lee)

1995년 2월 관동대학교 전자계산공학과 졸업
1999년 8월 관동대학교 전자계산공학과 석사
2004년 2월 관동대학교 전자계산공학과 박사
2005년~2007년 안동과학대학 사이버테러대응과 전임강사
2011년~현재 (주)트라이콤텍 연구소장
<관심분야> 무선/이동 전송, 데이터 통신, 차세대 이동통신, 네트워크 보안

유 영 모 (Young-mo You)

2010년 2월 경동대학교 멀티미디어 통신학과 졸업
2009년~현재 (주)트라이콤텍 연구원
<관심분야> 차세대 이동통신, 데이터 통신, 유비쿼터스, 네트워크 보안

이 상 윤 (SangYoon Lee)

1995년 2월 충북대학교 통계학과 졸업
1996년~1999년 텔넷코리아 연구원
1999년~현재 (주)트라이콤텍 엔지니어 총괄 팀장
<관심분야> 무선/이동 전송, 차세대 이동통신, 데이터 통신, 유비쿼터스, 네트워크 보안

오 세 갑 (Se-Kab Oh)

1999년 8월 한국항공대학교 항공통신정보공학과 석사
2010년 2월 목원대학교 IT공학과 박사
1999년~2001년 (주)세영통신 전파기술연구소 연구원
2001년~2006년 (주)벨웨이브 통신연구소 책임연구원
2006~현재 (재)대전테크노파크 IT 융합산업본부 팀장
<관심분야> 무선 멀티미디어 통신, 무선 이동 통신, IT 기반 융합 기술 등

www.dbpia.co.kr