ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

차량 통신 기술을 위한 OFDM 모듈레이션의 64-비트 스크램블러 설계

The 64-Bit Scrambler Design of the OFDM Modulation for Vehicles Communications Technology

이 대 식1*
Dae-Sik Lee

요 약

WAVE 시스템은 IEEE 802.11p표준으로 지능형 교통시스템 서비스에 응용되는 새로운 개념 및 차량 통신 기술이다. 또한 WAVE 시스템은 도로상의 트래픽의 효율과 안전을 높인다. 그러나 WAVE 시스템의 OFDM 모듈레이션에서 스크램블러 비트 연산 알고리즘은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하므로 효율성이 떨어지게 된다. 본 논문에서는 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성하는 알고리즘과 64비트 행렬 테이블과 입력 데이터를 병렬 연산하는 알고리즘을 제안하였다. 제안한 알고리즘은 64비트 행렬 테이블을 적용하여 실행한 결과 비트연산 스크램블러보다 1회와 10000회 처리 속도는 약 40.08%-40.27%가 향상되고, 초당 처리 횟수는 468.35회 더 수행할 수 있고, 32비트 스크램블러보다 1회와 10000회 처리 속도는 약 7.53%-7.84%가 향상되고, 초당 처리 횟수는 91.44회 더 수행할 수 있다. 따라서 64비트로 연산하는 스크램블러 알고리즘은 64비트를 처리할 수 있는 CPU를 사용한다면 32비트 스크램블러보다 40% 이상 성능을 향상시킬 수 있다.

☞ 주제어 : 지능형 교통시스템, WAVE System시스템, 64비트 행렬 테이블, 스크램블러, 차량통신기술

ABSTRACT

WAVE(Wireless Access for Vehicular Environment) is new concepts and Vehicles communications technology using for ITS(Intelligent Transportation Systems) service by IEEE standard 802.11p. Also it increases the efficiency and safety of the traffic on the road. However, the efficiency of Scrambler bit computational algorithms of OFDM modulation in WAVE systems will fall as it is not able to process in parallel in terms of hardware and software. This paper proposes an algorithm to configure 64-bits matrix table in scambler bit computation as well as an algorithm to compute 64-bits matrix table and input data in parallel. The proposed algorithm on this thesis is executed using 64-bits matrix table. In the result, the processing speed for 1 and 1000 times is improved about 40.08% ~ 40.27% and processing rate per sec is performed more than 468.35 compared to bit operation scramble. And processing speed for 1 and 1000 times is improved about 7.53% ~ 7.84% and processing rate per sec is performed more than 91.44 compared to 32-bits operation scramble. Therefore, if the 64 bit-CPU is used for 64-bits executable scramble algorithm, it is improved more than 40% compare to 32-bits scrambler.

☞ keyword : ITS : Intelligent Transportation Systems, Wireless Access for Vehicular Environment System, 64Bit Matrix Table, Scrambler, Vehicles Communications Technology

1. 서 론

WAVE 시스템은 차량 탑재장치(OBE, On-Board Equipment)를 보유한 운전자에게 교통정보, 위치정보 및 안전에 관한 정보 등 다양한 서비스를 제공하기 위한 차세대 차량용 통신 기술로 차량 탑재장치와 노변기지국(RSE, Road Side Equipment) 및 관련 어플리케이션으로 구성된 시스템이다[1].

현재 추진하고 있는 WAVE 시스템은 IEEE 802.11p 표준 규격의 무선기술을 기반으로 12 (필수규격) - 27(선택규격) Mbps의 고속 데이터를 전송하고, 낮은 전송응답대기(low latency), 지역적인 서비스 특화 및 고속 이동시 필요한 다양한 정보서비스 제공에 적합하게 설계되었다. 따라서 교통 혼잡을 효율적으로 조정하고 안정성을 획기적으로 증진시키므로 차세대 교통체제 시스템인 지능형 교통시스템(ITS)을 구현할 수 있다[2,3].

1 Development Dept, Tricomtek, 157-030, Korea
* Corresponding author (daesik@tricomtek.com)
[Received 22 October 2012, Reviewed 13 November 2012(R2 3 January 2013, R3 18 January 2013), Accepted 20 February 2013]

IEEE 802.11p 표준규격에 규정된 OFDM 모듈레이션은 신호처리과정에서 송신데이터의 고속화를 위해 다양한 디지털 데이터 신호처리 기능을 규정하고 있지만 스크램블러의 비트 연산 처리 방식은 병렬 처리가 불가능하므로 소프트웨어나 하드웨어 설계의 효율성이 떨어지게 된다.

본 논문에서는 스크램블러를 비트 연산으로 64비트 행렬 테이블을 구성하는 알고리즘과 64비트 행렬 테이블과 입력 데이터를 64비트 단위로 병렬 처리하는 알고리즘을 제안한다. 특히, 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성하여 64비트 행렬 테이블과 입력 데이터를 병렬 처리하는 알고리즘을 비교 분석해 보고자 한다.

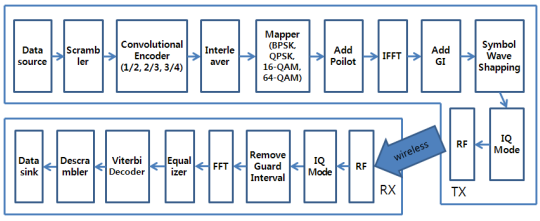
본 논문의 구성은 2장에서 관련연구인 WAVE 시스템의 물리계층과 IEEE 802.11p 표준 규격의 스크램블러를 살펴보고 3장에서 본 논문에서 제안하는 64비트 연산하는 스크램블러를 설명한다. 그리고 4장에서 실험 결과를 설명하고 5장에서 결론을 맺는다.

2. 관련 연구

2.1 WAVE 시스템의 물리계층

OFDM(Orthogonal Frequency Division Multiplexing) [4-6] 모듈레이션은 순서대로 스크램블러, 콘볼루션, 인터리버, 주파수 변조 기법에 따른 심볼 값 매핑, 파일롯 추가, IFFT(Inverse Fast Fourier Transform) 연산, GI(Guard Interval) 추가, 심볼 정형, IQ 모드 과정을 거친 후 RF (Radio Frequency)를 통해 데이터가 전달된다.

WAVE 물리계층에서 OFDM 모듈레이션은 (그림 1)과 같다[7,8].

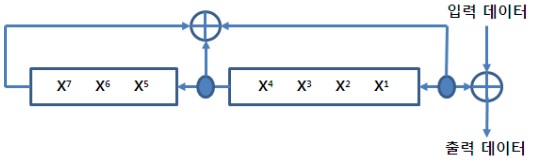


(그림 1) OFDM 모듈레이션
(Fig. 1) OFDM Modulation

(그림 1)에서 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 Rate로 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 FFT 연산을 한다. FFT 연산을 통해 생성된 각각의 심볼의 앞부분에 심볼간 간섭을 억제하기 위해 FFT 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 디모듈레이션은 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[9]. 스크램블러의 하드웨어 설계는 (그림 2)와 같다.



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

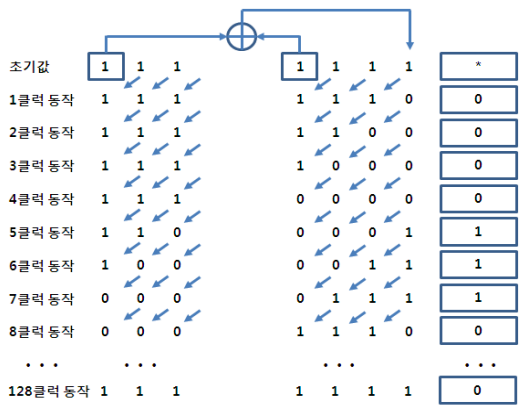
(그림 2)를 구현하는 알고리즘을 제시하면 알고리즘 1과 같다.

알고리즘 1을 보면 Scrambler_Bit_Calculation 함수로 스크램블러의 비트 연산을 한다. 스크램블러의 비트 연산에서 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 1을 적용하여 초기값 seed “1111111”을 7개의 레지스터에 입력하면 (그림 3)과 같다.

(알고리즘 1) 스크램블러의 비트 연산 알고리즘
(Algorithm 1) Bit Calculation Algorithm of Scrambler

```
Void Scrambler_Bit_Calculation( ) {
Uint64 seed_src = seed, seed_val = 0;
int i, ii = 0;
Uint64* src = input, val = output;
for(ii = 0 ; ii < input_size ; ii++) {
  for(i = 0 ; i < 64 ; i++) {
  seed_val = ((seed_src & 0x08)/0x08) ^
  ((seed_src & 0x40)/0x40);
  seed_src = (seed_src * 0x02) + seed_val;
  *val += (*src<<(i % 64)) ^ (seed_val<<(i % 64)); }
  val++;   src++;  } }
```



(그림 3) 스크램블러의 비트 연산
(Fig. 3) Bit Calculation of Scrambler

(그림 3)의 스크램블러의 비트 연산에서 128비트 입력 데이터로 알고리즘 1을 설명한다. 128비트 입력 데이터는 (표 1)과 같다.

(표 1) 128비트 입력 데이터(16진수)
(Table 1) 128-Bit Input String(Hexadecimal)

28	14	8C	22	7A	26	2E	61
CF	7A	0F	F0	AA	3C	63	FF

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 "0"과 스크램블러 비트 "0"을 XOR 연산하여 출력 데이터 비트 "0"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 "1"과 스크램블러 비트 "0"을 XOR 연산하여 출력 데이터 비트 "1"을 val에 저장한다.

3클럭 동작으로 seed_src "1111100"을 7개의 레지스터에 입력하면 x^4와 x^7을 XOR 연산하여 생성된 스크램블러의 비트 "0"을 seed_ val에 할당하고, 시프트 연산으로 스크램블러의 비트 "0"을 X^1에 추가하여 4클럭 동작을 위해 seed_src에 할당한다. 입력 데이터의 세 번째 비트 "0"과 스크램블러의 비트 "0"을 XOR 연산하여 출력 데이터 비트 "0"을 val에 저장한다.

반복 동작으로 입력 데이터 비트 수만큼 128클럭까지 동작한다.

따라서 입력 데이터가 128비트이므로 스크램블러의 비트 연산으로 128클럭까지 동작하여 각 클럭마다 128비트의 입력 데이터와 한비트씩 XOR 연산을 하여 출력 데이터를 생성한다. 128비트 출력 데이터는 (표 2)와 같다.

(표 2) 128비트 출력 데이터(16진수)
(Table 2) 128-Bit Output String(Hexadecimal)

2F	E0	B5	26	3C	61	F8	62
7D	04	DD	B5	5F	94	B2	08

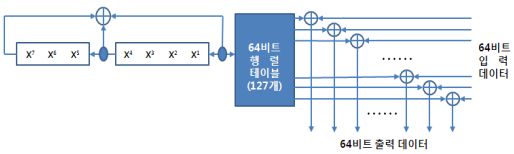
하드웨어나 소프트웨어 측면에서 스크램블러의 비트 연산은 입력 데이터의 비트 수만큼 반복문의 Loop 횟수가 비정상적으로 많아지고, 프로세서가 32비트 이상 지원되어도 8비트 프로세서와 처리속도는 같다. 결과적으로 스크램블러 처리속도를 높이기 위해 본 논문에서는 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성하는 알고리즘과 64비트 행렬 테이블과 입력 데이터를 64비트로 병렬 처리하는 알고리즘을 제안한다.

3. 제안한 64비트 스크램블러

IEEE 802.11p 표준 규격 스크램블러의 비트 연산은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하므로 효율성이 떨어지게 된다.

본 논문에서는 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성하여 입력 데이터를 64비트 단위로 병렬처리 할 수 있는 알고리즘을 제안한다.

제안한 스크램블러의 하드웨어 설계는 (그림 4)와 같으며 입력 데이터의 입력 단위를 64비트 기준으로 설계한 것이다.



(그림 4) 64비트로 연산하는 스크램블러의 하드웨어 설계
(Fig. 4) Hardware Design of 64-Bit Calculation Scrambler

(그림 4)는 초기값 "x^7 x^6 x^5 x^4 x^3 x^2 x^1"이 7개 레지스터에 입력되어 스크램블러의 비트 연산으로 메모리에 127개의 서로 다른 비트열로 64비트 행렬 테이블을 구성한다. 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블은 입력 데이터의 입력 단위 64비트와 병렬처리 할 수 있다. 따라서 (그림 4)와 같이 입력 데이터를 64비트 입력 단위로 병렬처리 할 수 있는 알고리즘을 제안한다.

(그림 2)의 하드웨어 설계는 입력 데이터가 1284비트이면 각 클럭의 동작으로 스크램블러의 비트를 생성하여 한비트씩 입력 데이터와 XOR 연산을 128번 처리한다. 하지만 (그림 4)에서 제안한 하드웨어 설계는 64비트 행렬 테이블과 입력 데이터의 64비트 입력 단위가 병렬로 XOR 연산을 2번 처리한다.

입력 데이터의 입력 단위가 64비트인 (그림 4)를 구현하는 알고리즘을 제시하면 알고리즘 2, 알고리즘 3과 같다.

알고리즘 2는 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성하는 알고리즘이다.

알고리즘 2를 보면 Scrambler_Table 함수로 127개의 서로 다른 비트열로 64비트 행렬 테이블을 구성한다. 스크램블러의 비트 연산으로 64비트 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 비트열을 저장하기 위해 randomizer_table을 포인터 ran_com에 할당한다.

(알고리즘 2) 64비트 행렬 테이블 구성 알고리즘
(algorithm 2) 64-Bit Matrix Table Configuration Algorithm

```
void Scrambler_Table(void)  {
Uint64 seed_src = seed, seed_val=0;
Uint64* ran_com = randomizer_table;
int i = 0, ii = 0;
memset(ran_com,0x00,1000);
for(ii = 0 ; ii < 127 ; ii++)  {
    for(i = 0 ; i < 64 ; i++)  {
    seed_val = ((seed_src & 0x08)/0x08) ^
    ((seed_src & 0x40)/0x40);
    seed_src = (seed_src * 0x02) + seed_val;
    *ran_com += seed_val<<(i % 64);  }
ran_com++;  }  }
```

알고리즘 2를 적용하여 초기값 seed "1111111(0xFE)"을 7개 레지스터에 입력하면 (그림 3)과 같이 스크램블러의 비트 연산으로 64비트 행렬 테이블을 구성할려면 64클럭씩 동작하여 16개의 서로 다른 비트열이 생성된다. 64비트로 구성된 16개의 서로 다른 비트열은 표 3과 같다.

(그림 3)과 같이 초기값 seed_src "1111111(0xFE)"을 7개 레지스터에 입력하여 스크램블러의 비트 연산으로 (그림 3)과 같은 16개의 서로 다른 비트열을 구성하고, 16개의 서로 다른 비트열은 8바이트 메모리만큼 순환루프를 반복적으로 실시하면 128(16*8 = 128)번째 비트열은 처음에 생성된 비트열 "0x306D746440934F70"과 같다. 따라서 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블을 입력 데이터의 입력 단위와 64비트로 병렬 처리한다.

알고리즘 2에서 처럼 for 문을 127번의 수행하고, 1번의 for 문으로 64클럭 동작하여 127개의 64비트 행렬 테이블을 구성하여 입력 데이터의 전체 크기가 1016 바이

(표 3) 16개 비트열(16진수)
(Table 3) 16 Bit String(Hexadecimal)

306D746440934F70	7F1D8A5F542DE72B	9836BA322049A7B8	3F8EC52FAA16F395	CC1B5D191024D3DC
1FC76297D50B79CA	660DAE8C881269EE	0FE3B14BEA85BCE5	B306D746440934F7	87F1D8A5F542DE72
59836BA322049A7B	C3F8EC52FAA16F39	ACC1B5D191024D3D	E1FC76297D50B79C	5660DAE8C881269E
70FE3B14BEA85BCE				

(표 4) 127개의 64비트 행렬 테이블(16진수)
(Table 4) 127x 64-Bit Matrix Table(Hexadecimal)

306D746440934F70	7F1D8A5F542DE72B	9836BA322049A7B8	3F8EC52FAA16F395	CC1B5D191024D3DC
1FC76297D50B79CA	660DAE8C881269EE	0FE3B14BEA85BCE5	B306D746440934F7	87F1D8A5F542DE72
59836BA322049A7B	C3F8EC52FAA16F39	ACC1B5D191024D3D	E1FC76297D50B79C	5660DAE8C881269E
70FE3B14BEA85BCE	2B306D746440934F	B87F1D8A5F542DE7	959836BA322049A7	DC3F8EC52FAA16F3
CACC1B5D191024D3	EE1FC76297D50B79	E5660DAE8C881269	F70FE3B14BEA85BC	72B306D746440934
7B87F1D8A5F542DE	3959836BA322049A	3DC3F8EC52FAA16F	9CACC1B5D191024D	9EE1FC76297D50B7
CE5660DAE8C88126	4F70FE3B14BEA85B	E72B306D74644093	A7B87F1D8A5F542D	F3959836BA322049
D3DC3F8EC52FAA16	79CACC1B5D191024	69EE1FC76297D50B	BCE5660DAE8C8812	34F70FE3B14BEA85
DE72B306D7464409	9A7B87F1D8A5F542	6F3959836BA32204	4D3DC3F8EC52FAA1	B79CACC1B5D19102
269EE1FC76297D50	5BCE5660DAE8C881	934F70FE3B14BEA8	2DE72B306D746440	49A7B87F1D8A5F54
16F3959836BA3220	24D3DC3F8EC52FAA	0B79CACC1B5D1910	1269EE1FC76297D5	85BCE5660DAE8C88
0934F70FE3B14BEA	42DE72B306D74644	049A7B87F1D8A5F5	A16F3959836BA322	024D3DC3F8EC52FA
50B79CACC1B5D191	81269EE1FC76297D	A85BCE5660DAE8C8	40934F70FE3B14BE	542DE72B306D7464
2049A7B87F1D8A5F	AA16F3959836BA32	1024D3DC3F8EC52F	D50B79CACC1B5D19	881269EE1FC76297
EA85BCE5660DAE8C	440934F70FE3B14B	F542DE72B306D746	22049A7B87F1D8A5	FAA16F3959836BA3
91024D3DC3F8EC52	7D50B79CACC1B5D1	C881269EE1FC7629	BEA85BCE5660DAE8	6440934F70FE3B14
5F542DE72B306D74	322049A7B87F1D8A	2FAA16F3959836BA	191024D3DC3F8EC5	97D50B79CACC1B5D
8C881269EE1FC762	4BEA85BCE5660DAE	46440934F70FE3B1	A5F542DE72B306D7	A322049A7B87F1D8
52FAA16F3959836B	D191024D3DC3F8EC	297D50B79CACC1B5	E8C881269EE1FC76	14BEA85BCE5660DA
746440934F70FE3B	8A5F542DE72B306D	BA322049A7B87F1D	C52FAA16F3959836	5D191024D3DC3F8E
6297D50B79CACC1B	AE8C881269EE1FC7	B14BEA85BCE5660D	D746440934F70FE3	D8A5F542DE72B306
6BA322049A7B87F1	EC52FAA16F395983	B5D191024D3DC3F8	76297D50B79CACC1	DAE8C881269EE1FC
3B14BEA85BCE5660	6D746440934F70FE	1D8A5F542DE72B30	36BA322049A7B87F	8EC52FAA16F39598
1B5D191024D3DC3F	C76297D50B79CACC	0DAE8C881269EE1F	E3B14BEA85BCE566	06D746440934F70F
F1D8A5F542DE72B3	836BA322049A7B87	F8EC52FAA16F3959	C1B5D191024D3DC3	FC76297D50B79CAC
60DAE8C881269EE1	FE3B14BEA85BCE56			

트 이상이 되어도 스크램블러의 비트 연산을 하지 않고 64비트 행렬 테이블과 계속하여 병렬처리 할 수 있다. 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블은 (표 4)와 같다.

(표 4)와 같이 64비트 행렬 테이블과 입력 데이터를 64비트씩 병렬로 XOR 연산을 한다.

알고리즘 2에서 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블을 입력 데이터의 입력 단위 64비트와 병렬 연산하는 알고리즘은 알고리즘 3과 같다.

알고리즘 3을 보면 Parallel_Computation 함수로 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블을 입력 데이터와 64비트씩 병렬 처리한다. ran_com과 com으로 행렬 테이블의 포인터를 지정하고, 변수 i, count, src, val을 지정한다. 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블 randomizer_table을 ran_com에 할당하고, 행렬 테이블의 초기 포인터 위치 ran_com을 com에 할당한다. 또한 입력 데이터는 src, 출력 데이터는 val에 할당한다.

64비트 행렬 테이블 randomizer_table은 초기값 seed "1111111(0xFE)"로 생성된 (표 4)와 128비트의 입력 데이터 (표 1)로 예를 들어 설명하면 다음과 같다.

(알고리즘 3) 64비트 행렬 테이블과 입력 데이터를 병렬 연산하는 알고리즘
(Algorithm 3) 64-Bit Matrix table and Input data Parallel processing algorithm

```
void Parallel_Computation() {
Uint64* ran_com, com;
int i = 0;
Uint64* src, val;
Uint64 counter = 0;
ran_com = randomizer_table;
com = ran_com;
src = input;
val = output;
for(i = 0 ; i < input_size ; i++)  {
    *val = *src ^ *com;
    val++;  src++;  com++;  counter++;
    if(counter == 127)  {
        counter = 0;
        com = ran_com;  }  }  }
```

알고리즘 3을 적용하면 64비트 행렬 테이블과 입력 데이터를 64비트씩 나누어 병렬로 XOR 연산을 한다. 64비트 행렬 테이블의 64비트 "0x306D746440934F70"과 입력 데이터의 입력단위 src 64비트 "0x306D746440934F70"을 병렬로 XOR 연산하여 출력 데이터 64비트 "0x1879F8463AB56111"을 val에 저장하고, src, com, val, count는 64비트 행렬 테이블의 단위인 64비트씩 증가한다. 또한 행렬 테이블의 count가 127과 같으면 행렬 테이블의 마지막이므로 count = 0, 행렬 테이블의 초기 포인터 ran_com의 위치를 com으로 할당한다.

다음 64비트 행렬 테이블의 64비트 "0x7F1D8A5F542DE72B"와 입력 데이터의 입력 단위 src 64비트 "0xCF7A0FF0AA3C63FF"를병렬로 XOR 연산하여 출력 데이터 64비트 "0xB06785AFFE1184D4"를 val에 추가하여 저장하고, src, com, val, count는 64비트 행렬 테이블의 단위인 64비트씩 증가한다. 또한 행렬 테이블의 count가 127과 같으면 행렬 테이블의 마지막이므로 count = 0, 행렬 테이블의 초기 포인터 ran_com의 위치를 com으로 할당한다.

알고리즘 3을 적용한 결과 입력 데이터의 전체 비트가 128비트이므로 64비트 행렬 테이블의 64비트 단위와 병렬로 XOR 연산하여 출력 데이터 val에 "0x1879F8463AB56111 B06785AFFE1184D4"를 생성하고 프로그램을 종료한다.

따라서 IEEE 802.11p 표준 규격 스크램블러의 비트 연산 알고리즘에서 128번 실행하였고, 64비트로 연산하는 스크램블러 알고리즘에서는 2번 실행하였다.

본 논문에서는 127개의 서로 다른 비트열로 구성된 64비트 행렬 테이블과 입력 데이터를 병렬로 XOR 연산을 할 수 있으므로 WAVE 시스템의 처리 속도를 더욱 향상시킨다.

4. 실험 결과

본 논문에서는 비트연산 스크램블러와 32비트로 연산하는 스크램블러, 64비트로 연산하는 스크램블러를 비교분석하고, 실제 WAVE DSP 보드로 실험하였고, 실험사양은 (표 5)와 같다.

(표 5) 실험 사양
(Table 5) Test Specifications

시스템	WAVE DSP 보드
운영체제	DSP/BIOS
CPU	TMS320C6455
CPU 속도	1.2GHz
메모리	256MB
언어	C언어

본 논문에서는 입력 데이터는 "0x306D746440934F70 CF7A0FF0AA3C63FF" 128비트를 적용하여 64비트로 연산하는 스크램블러 알고리즘의 타당성을 검증하기 위해 실험 1은 32비트 행렬 테이블을 적용하는 스크램블러, 실험 2는 64비트 행렬 테이블을 적용하는 스크램블러로 실험하였다. 또한 입력 데이터의 처리 단위마다 추가적인 메모리 할당 부분이 필요하다. 실험 1, 2에서는 입력 데이터의 크기는 1080Byte로 하고, 처리속도나 처리횟수는 10회 측정한 것을 평균값으로 표현하였다.

〔실험 1〕 실험 1에서는 표 6과 같이 32비트 행렬 테이블을 적용하는 스크램블러로 처리하였고, 처리속도나 처리횟수는 (표 7)과 같다.

(표 6) 127개의 32비트 행렬 테이블(16진수)
(Table 6) 127x 32-Bit Matrix Table(Hexadecimal)

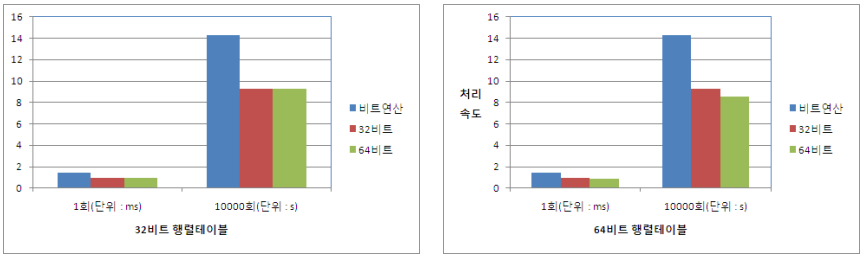
fe3b14be	40934f70	306d7464	542de72b	7f1d8a5f	2049a7b8	9836ba32	aa16f395	3f8ec52f	1024d3dc
cc1b5d19	d50b79ca	1fc76297	881269ee	660dae8c	ea85bce5	0fe3b14b	440934f7	b306d746	f542de72
87f1d8a5	22049a7b	59836ba3	faa16f39	c3f8ec52	91024d3d	aoc1b5d1	7d50b79c	e1fc7629	c881269e
5660dae8	bea85bce	70fe3b14	6440934f	2b306d74	5f542de7	b87f1d8a	322049a7	959836ba	2faa16f3
dc3f8ec5	191024d3	cacc1b5d	97d50b79	ee1fc762	8c881269	e5660dae	4bea85bc	f70fe3b1	46440934
72b306d7	a5f542de	7b87f1d8	a322049a	3959836b	52faa16f	3dc3f8ec	d191024d	9cacc1b5	297d50b7
9ee1fc76	e8c88126	ce5660da	14bea85b	4f70fe3b	74644093	e72b306d	8a5f542d	a7b87f1d	ba322049
f3959836	c52faa16	d3dc3f8e	5d191024	79cacc1b	6297d50b	69ee1fc7	ae8c8812	bce5660d	b14bea85
34f70fe3	d7464409	de72b306	d8a5f542	9a7b87f1	6ba32204	6f395983	ec52Faa1	4d3dc3f8	b5d19102
b79cacc1	76297d50	269ee1fc	dae8c881	5bce5660	3b14bea8	934f70fe	6d746440	2de72b30	1d8a5f54
49a7b87f	36ba3220	16f39598	8ec52faa	24d3dc3f	1b5d1910	0b79cacc	c76297d5	1269ee1f	0dae8c88
85bce566	e3b14bea	0934f70f	06d74644	42de72b3	f1d8a5f5	049a7b87	836ba322	a16f3959	f8ec52fa
024d3dc3	c1b5d191	50b79cac	fc76297d	81269ee1	60dae8c8	a85bce56			

(표 7) 32비트 처리속도와 처리횟수
(Table 7) 32-Bit Processing Speed and Processing Times

32비트	비트연산 스크램블러	32비트 스크램블러	64비트 스크램블러
1회 처리속도	약 1.44ms	약 0.93ms	약 0.93ms
10000회 처리속도	약 14.32s	약 9.31s	약 9.31s
초당 처리횟수	약 694.44회	약 1071.35회	약 1071.35회
메모리	7Bit+1Bit	508Byte+7Bit	508Byte

(표 8) 64비트 처리속도와 처리횟수
(Table 8) 64-Bit Processing Speed and Processing Times

	비트연산 스크램블러	32비트 스크램블러	64비트 스크램블러
1회 처리속도	약 1.44ms	약 0.93ms	약 0.86ms
10000회 처리속도	약 14.32s	약 9.31s	약 8.58s
초당 처리횟수	약 694.44회	약 1071.35회	약 1162.79회
메모리	7Bit+1Bit	508Byte+7Bit	1016Byte+7Bit



(그림 5) 스크램블러의 성능분석
(Fig. 5) Performance Analysis of Scrambler

[실험 2] 실험 2에서는 (표 4)와 같이 64비트 행렬 테이블을 적용하는 스크램블러로 처리하였고, 처리속도나 처리횟수는 (표 8)과 같다.

본 논문에서는 제안한 64비트로 연산하는 스크램블러 알고리즘은 실험 1에서 나타났듯이 32비트 행렬 테이블을 적용하여 실험한 결과 32비트로 연산하는 스크램블러와 처리 속도와 처리 횟수가 같았다. 하지만 64비트 행렬 테이블을 적용하여 실행한 결과 비트연산 스크램블러보다 1회와 10000회 처리 속도는 약 40.08%-40.27%가 향상되고, 초당 처리 횟수는 468.35회 더 수행할 수 있고, 32비트 스크램블러보다 1회와 10000회 처리 속도는 약 7.53%-7.84%가 향상되고, 초당 처리 횟수는 91.44회 더 수행할 수 있다. 따라서 64비트로 연산하는 스크램블러 알고리즘은 64비트를 처리할 수 있는 CPU를 사용한다면 32비트 스크램블러보다 40% 이상 성능을 향상시킬 수 있다.

(그림 5)는 비트연산 스크램블러와 32비트 스크램블러, 본 논문에서 제안한 64비트 스크램블러 알고리즘의 성능 분석 결과를 나타낸 것이다.

5. 결 론

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

WAVE 시스템에서 IEEE 802.11p 표준 규격 스크램블러의 비트 연산 알고리즘은 하드웨어나 소프트웨어 측면에서 병렬 처리가 불가능하므로 효율성이 떨어지게 된다.

본 논문에서는 64비트 행렬 테이블 구성 알고리즘과 64비트 행렬 테이블과 입력 데이터를 병렬 연산하는 알고리즘을 제안하였다. 제안한 알고리즘은 비트연산 스크램블러보다 1회와 10000회 처리 속도는 약 40.08%-40.27%가 향상되고, 초당 처리 횟수는 468.35회 더 수행할 수 있고, 32비트 스크램블러보다 1회와 10000회 처리 속도는 약 7.53%-7.84%가 향상되고, 초당 처리 횟수는 91.44회 더 수행할 수 있다. 따라서 64비트로 연산하는 스크램블러 알고리즘은 64비트를 처리할 수 있는 CPU를 사용한다면 32비트 스크램블러보다 40% 이상 성능을 향상시킬 수 있다.

따라서 본 논문에서 제안한 64비트 행렬 테이블을 구성하는 알고리즘과 64비트 행렬 테이블과 입력 데이터를 병렬 연산하는 알고리즘은 WAVE 무선통신 시스템 성능을 더욱 업그레이드시켜 시간과 장소에 구애 받지 않고 차내 운전자에게 지능형 교통 시스템 서비스의 활용가치를 높이고 교통상황 등 각종 정보수집의 속도와 정밀도를 향상시킬 수 있다.

참 고 문 헌(Reference)

[1] 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.

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

[3] 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.

[4] 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.*

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

[6] 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.

[7] 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.

[8] 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.

[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.

◐ 저 자 소 개 ◑

이 대 식

1995년 관동대학교 전자계산공학과 졸업(학사)
1999년 관동대학교 전자계산공학과 졸업(석사)
2004년 관동대학교 전자계산공학과 졸업(박사)
2011년~현재 (주) 트라이콤텍 연구소장
관심분야 : 데이터 통신, 차세대 이동통신 etc.
E-mail : daesik@tricomtek.com