(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2009년05월07일
(11) 등록번호 10-0895932
(24) 등록일자 2009년04월24일

(51) Int. Cl.
*H04N 7/24* (2006.01) *H04N 7/12* (2006.01)
*H04N 7/46* (2006.01)
(21) 출원번호 10-2004-7016815
(22) 출원일자 2004년10월19일
심사청구일자 2008년01월16일
번역문제출일자 2004년10월19일
(65) 공개번호 10-2004-0099461
(43) 공개일자 2004년11월26일
(86) 국제출원번호 PCT/US2003/012226
국제출원일자 2003년04월18일
(87) 국제공개번호 WO 2003/090470
국제공개일자 2003년10월30일
(30) 우선권주장
60/374,192 2002년04월19일 미국(US)
(56) 선행기술조사문헌
EP0886398 A1
US5955977 A
WO0199956472 A1
전체 청구항 수 : 총 32 항

(73) 특허권자
**마이크로소프트 코포레이션**
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이
(72) 발명자
**슬리반,게리,제이.**
미국 98053 워싱턴주 레드몬드 노쓰이스트 227번 애비뉴 6239
(74) 대리인
**백만기, 이중희, 주성민**

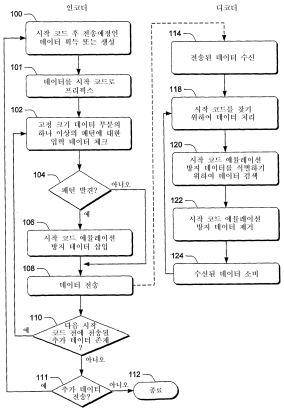
심사관 : 박상철

# (54) 비-바이트 정렬되거나 비트 시프팅된 위치를 포함하는위치에서 시작 코드 에뮬레이션을 방지하기 위한 방법 및시스템

## (57) 요 약

본 방법 및 시스템은 비트 레벨 보다 높은 입도(granularity)에서 시작 코드 에뮬레이션 방지(start code emulation prevention)에 대한 접근법을 제공한다. 다양한 실시예들에서, 바이트 등과 같은 데이터 경계들을 제외한 위치에서 시작 코드 에뮬레이션 방지가 발생할 수 있다. 처리된 데이터에서 데이터 정렬 경계들(data alignment boundaries)을 항상 보존하지 않는 시스템과 관련하여 이러한 실시예들이 사용될 수 있다. 일부 시스템에서, 기술된 기술들은 기준을 제공할 수 있는데, 데이터 경계들이 손상된 경우에 이러한 기준으로부터 디코더 시스템들이 복구될 수 있다.

*대 표 도* - 도1



## *특허청구의 범위*

**청구항 1**

인코딩된 데이터를 디코딩 준비하는 방법으로서,

데이터 스트림내에 적어도 하나의 시작 코드를 포함하는 인코딩된 데이터를 수신하는 단계와,

상기 수신된 데이터의 바이트 경계에서의 시작 코드 패턴의 위치에 기초하여 바이트 정렬을 복구하는 단계 - 상기 인코딩된 데이터는, 상기 시작 코드 패턴에 대하여, 검색 패턴으로서 특징적 패턴을 이용하여 상기 인코딩된 데이터에 에뮬레이션 방지 데이터 삽입을 수행하는 인코더로부터 발생하고, 상기 특징적 패턴은 상기 시작 코드 패턴의 복수의 비트 단위 쉬프트 중의 하나 내에서마다 발생하고 하나 이상의 바이트 정렬된 위치에 대하여 비트 쉬프트된 모든 위치에서의 상기 시작 코드 패턴의 에뮬레이션을 방지하도록 식별되는 복수의 바이트를 포함함 - 와,

상기 수신된 데이터의 상기 하나 이상의 바이트 정렬된 위치에서, 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴을 검색하는 단계와,

상기 수신된 데이터에서 상기 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴을 찾은 것에 응답하여, 상기 수신된 데이터로부터 상기 에뮬레이션 방지 바이트를 제거하는 단계

를 포함하는 방법.

**청구항 2**

제1항에 있어서,

상기 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴은, 상기 에뮬레이션 방지 데이터의 삽입 동안 상기 인코더에 의해 이용되는 치환 패턴과 동일한 방법.

**청구항 3**

전달하기 위한 데이터를 인코딩된 데이터 스트림으로 포맷하는 방법으로서,

인코딩되고 있는 데이터에서, 각 검색 패턴마다 복수의 바이트를 포함하는 하나 이상의 검색 패턴을 검색하는 단계와,

상기 하나 이상의 검색 패턴 중 하나의 검색 패턴이 상기 인코딩되고 있는 데이터에서 발견되면, 인코딩되는 데이터 스트림 내에서 상기 발견된 검색 패턴을 치환 패턴으로 치환하는 단계 - 상기 치환 패턴은 에뮬레이션 방지 바이트와 함께 상기 발견된 검색 패턴의 데이터를 포함함 -

를 포함하고,

특징적 패턴이, 시작 코드 패턴에 대하여, 인코딩되고 있는 상기 데이터에 에뮬레이션 방지 데이터 삽입을 수행하기 위한 상기 하나 이상의 검색 패턴 중 하나로서 사용되고, 상기 특징적 패턴은 상기 시작 코드 패턴의 복수의 비트 단위 쉬프트 중의 하나 내에서마다 발생하고 하나 이상의 바이트 정렬된 위치에 대하여 비트 쉬프트된 모든 위치에서의 상기 시작 코드 패턴의 에뮬레이션을 방지하도록 식별되는 복수의 바이트를 포함하여, 이에 따라 바이트 경계에서의 상기 시작 코드 패턴의 위치에 기초하여 디코더에서 바이트 정렬의 복구가 촉진되는,

전달하기 위한 데이터를 인코딩된 데이터 스트림으로 포맷하는 방법.

**청구항 4**

제3항에 있어서,

상기 디코더에서의 바이트 정렬의 복구는, 바이트 경계에 대하여 하나 이상의 비트 시프트된 위치에서, 제로인 1 바이트와 상기 제1 바이트에 바로 후속하는 시작 코드 프리픽스의 3 바이트로 이루어지는 4 바이트 패턴을 검색하는 것을 포함하는, 방법.

**청구항 5**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 인코딩된 데이터는 비디오 데이터를 포함하는, 방법.

**청구항 6**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 인코딩된 데이터는 오디오 데이터를 포함하는, 방법.

**청구항 7**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 에뮬레이션 방지 바이트는 0x03인, 방법.

**청구항 8**

제1항 내지 제3항 중 어느 한 항에 있어서,

상기 시작 코드 패턴은 시작 코드 프리픽스를 포함하는, 방법.

**청구항 9**

제8항에 있어서,

상기 시작 코드 프리픽스는 복수의 연속되는 0의 스트링을 포함하는, 방법.

**청구항 10**

제8항에 있어서,

상기 시작 코드 프리픽스는 복수의 연속되는 0과 바로 후속하는 하나의 1의 스트링을 포함하는, 방법.

**청구항 11**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 시작 코드 패턴의 위치는 31개의 연속되는 0과 바로 후속하는 하나의 1의 스트링에 의해 표시되는, 방법.

**청구항 12**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 특징적 패턴은 제로값인 3개의 바이트를 포함하는, 방법.

**청구항 13**

제1항 내지 제4항 중 어느 한 항에 있어서,

상기 특징적 패턴은 제로값인 복수의 바이트를 포함하고, 넌제로(non-zero) 값의 바이트를 포함하지 않는, 방법.

**청구항 14**

하나 이상의 프로세서에 의해 실행될 때, 상기 하나 이상의 프로세서가 제1항 내지 제4항 중 어느 한 항의 방법을 수행하게 하는 프로그램을 기록한 하나 이상의 컴퓨터 판독가능 매체.

**청구항 15**

인코딩된 데이터를 디코딩 준비하기 위해 하나 이상의 컴퓨터에

데이터 스트림내에 적어도 하나의 시작 코드를 포함하는 인코딩된 데이터를 수신하는 단계와,

상기 수신된 데이터의 바이트 경계에서의 시작 코드 패턴의 위치에 기초하여 바이트 정렬을 복구하는 단계 - 상기 인코딩된 데이터는, 상기 시작 코드 패턴에 대하여, 검색 패턴으로서 특징적 패턴을 이용하여 상기 인코딩된

데이터에 에뮬레이션 방지 데이터 삽입을 수행하는 인코더로부터 발생하고, 상기 특징적 패턴은 상기 시작 코드 패턴의 복수의 비트 단위 쉬프트 중의 하나 내에서마다 발생하고 하나 이상의 바이트 정렬된 위치에 대하여 비트 쉬프트된 모든 위치에서의 상기 시작 코드 패턴의 에뮬레이션을 방지하도록 식별되는 복수의 바이트를 포함함 - 와,

상기 수신된 데이터의 상기 하나 이상의 바이트 정렬된 위치에서, 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴을 검색하는 단계와,

상기 수신된 데이터에서 상기 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴을 찾은 것에 응답하여, 상기 수신된 데이터로부터 상기 에뮬레이션 방지 바이트를 제거하는 단계

를 실행시키기 위한 프로그램을 기록한 컴퓨터 판독가능 매체.

**청구항 16**

제15항에 있어서,

상기 에뮬레이션 방지 바이트를 포함하는 복수의 바이트의 패턴은, 상기 에뮬레이션 방지 데이터의 삽입 동안 상기 인코더에 의해 이용되는 치환 패턴과 동일한, 컴퓨터 판독가능 매체.

**청구항 17**

전달하기 위한 데이터를 인코딩된 데이터 스트림으로 포맷하기 위해 하나 이상의 컴퓨터에

인코딩되고 있는 데이터에서, 각 검색 패턴마다 복수의 바이트를 포함하는 하나 이상의 검색 패턴을 검색하는 단계와,

상기 하나 이상의 검색 패턴 중 하나의 검색 패턴이 상기 인코딩되고 있는 데이터에서 발견되면, 인코딩되는 데이터 스트림 내에서 상기 발견된 검색 패턴을 치환 패턴으로 치환하는 단계 - 상기 치환 패턴은 에뮬레이션 방지 바이트와 함께 상기 발견된 검색 패턴의 데이터를 포함함 -

를 실행시키기 위한 프로그램을 기록한 컴퓨터 판독가능 매체로서,

특징적 패턴이, 시작 코드 패턴에 대하여, 인코딩되고 있는 상기 데이터에 에뮬레이션 방지 데이터 삽입을 수행하기 위한 상기 하나 이상의 검색 패턴 중 하나로서 사용되고, 상기 특징적 패턴은 상기 시작 코드 패턴의 복수의 비트 단위 쉬프트 중의 하나 내에서마다 발생하고 하나 이상의 바이트 정렬된 위치에 대하여 비트 쉬프트된 모든 위치에서의 상기 시작 코드 패턴의 에뮬레이션을 방지하도록 식별되는 복수의 바이트를 포함하여, 이에 따라 바이트 경계에서의 상기 시작 코드 패턴의 위치에 기초하여 디코더에서 바이트 정렬의 복구가 촉진되는, 컴퓨터 판독가능 매체.

**청구항 18**

제17항에 있어서,

상기 디코더에서의 바이트 정렬의 복구는, 바이트 경계에 대하여 하나 이상의 비트 시프트된 위치에서, 제로인 1 바이트와 상기 1 바이트에 바로 후속하는 3 바이트의 시작 코드 프리픽스로 이루어지는 4 바이트 패턴을 검색하는 것을 포함하는, 컴퓨터 판독가능 매체.

**청구항 19**

제15항 내지 제18항 중 어느 한 항에 있어서,

상기 인코딩된 데이터는 비디오 데이터를 포함하는, 컴퓨터 판독가능 매체.

**청구항 20**

제15항 내지 제18항 중 어느 한 항에 있어서,

상기 특징적 패턴은, 제로값인 복수의 바이트를 포함하고, 넌제로값인 바이트를 포함하지 않는, 컴퓨터 판독가능 매체.

**청구항 21**

제15항 내지 제18항 중 어느 한 항에 있어서,

상기 에뮬레이션 방지 바이트는 0x03인, 컴퓨터 판독가능 매체.

**청구항 22**

제15항 내지 제17항 중 어느 한 항에 있어서,

상기 시작 코드 패턴은 시작 코드 프리픽스를 포함하는, 컴퓨터 판독가능 매체.

**청구항 23**

제22항에 있어서,

상기 시작 코드 프리픽스는 복수의 연속되는 0의 스트링을 포함하는, 컴퓨터 판독가능 매체.

**청구항 24**

제22항에 있어서,

상기 시작 코드 프리픽스는 복수의 연속되는 0과 바로 후속하는 하나의 1의 스트링을 포함하는, 컴퓨터 판독가능 매체.

**청구항 25**

제15항 내지 제18항 중 어느 한 항에 있어서,

상기 시작 코드 패턴의 위치는 31개의 연속되는 0과 바로 후속하는 하나의 1의 스트링에 의해 표시되는, 컴퓨터 판독가능 매체.

**청구항 26**

제15항 내지 제18항 중 어느 한 항에 있어서,

상기 특징적 패턴은 3개의 제로값 바이트를 포함하는, 컴퓨터 판독가능 매체.

**청구항 27**

제3항에 있어서,

상기 치환하는 단계 후에, 상기 바이트 정렬의 복구를 촉진시키기 위하여, 시작 코드 프리픽스가 뒤따르는 패딩 바이트를 상기 시작 코드 패턴으로서 송신하는 단계

를 더 포함하는 방법.

**청구항 28**

제27항에 있어서,

상기 치환하는 단계 후 상기 송신하는 단계 전에, 시작 코드 타입에 따라 상기 패딩 바이트의 송신 여부를 결정하는 단계

를 더 포함하는 방법.

**청구항 29**

제2항 내지 제4항 중 어느 한 항에 있어서,

상기 특징적 패턴의 복수의 바이트 각각은 0x00의 값을 갖고, 상기 특징적 패턴이 인코딩되고 있는 상기 데이터에서 발견된다면, 상기 치환 패턴은

각각의 값이 0x00에 해당하는 N-1 바이트, 상기 에뮬레이션 방지 바이트, 값이 0x00에 해당하는 N 번째 바이트 순으로 이루어지는, 방법.

**청구항 30**

제17항에 있어서,

상기 치환하는 단계 후에, 상기 바이트 정렬의 복구를 촉진시키기 위하여, 시작 코드 프리픽스가 뒤따르는 패딩 바이트를 상기 시작 코드 패턴으로서 송신하는 단계를 더 실행시키기 위한 프로그램을 기록한 컴퓨터 판독가능 매체.

**청구항 31**

제30항에 있어서,

상기 치환하는 단계 후 상기 송신하는 단계 전에, 시작 코드 타입에 따라 상기 패딩 바이트의 송신 여부를 결정하는 단계를 더 실행시키기 위한 프로그램을 기록한 컴퓨터 판독가능 매체.

**청구항 32**

제16항 내지 제18항 중 어느 한 항에 있어서,

상기 특징적 패턴의 복수의 바이트 각각은 0x00의 값을 갖고, 상기 특징적 패턴이 인코딩되고 있는 상기 데이터에서 발견된다면, 상기 치환 패턴은

각각의 값이 0x00에 해당하는 N-1 바이트, 상기 에뮬레이션 방지 바이트, 값이 0x00에 해당하는 N 번째 바이트 순으로 이루어지는, 컴퓨터 판독가능 매체.

**청구항 33**

삭제

**청구항 34**

삭제

**청구항 35**

삭제

**청구항 36**

삭제

**청구항 37**

삭제

**청구항 38**

삭제

**청구항 39**

삭제

**청구항 40**

삭제

**청구항 41**

삭제

**청구항 42**

삭제

**청구항 43**

삭제

**청구항 44**

삭제

**청구항 45**

삭제

**청구항 46**

삭제

**청구항 47**

삭제

**청구항 48**

삭제

**청구항 49**

삭제

**청구항 50**

삭제

**청구항 51**

삭제

**청구항 52**

삭제

**청구항 53**

삭제

**청구항 54**

삭제

**청구항 55**

삭제

**청구항 56**

삭제

**청구항 57**

삭제

**청구항 58**

삭제

**청구항 59**

삭제

**청구항 60**

삭제

**청구항 61**

삭제

**청구항 62**

삭제

**청구항 63**

삭제

**청구항 64**

삭제

**청구항 65**

삭제

**청구항 66**

삭제

**청구항 67**

삭제

**청구항 68**

삭제

**청구항 69**

삭제

**청구항 70**

삭제

**청구항 71**

삭제

**청구항 72**

삭제

**청구항 73**

삭제

**청구항 74**

삭제

**청구항 75**

삭제

**청구항 76**

삭제

**청구항 77**

삭제

**청구항 78**

삭제

**청구항 79**

삭제

**청구항 80**

삭제

**청구항 81**

삭제

**청구항 82**

삭제

**청구항 83**

삭제

**청구항 84**

삭제

**청구항 85**

삭제

**청구항 86**

삭제

**청구항 87**

삭제

**청구항 88**

삭제

**청구항 89**

삭제

**청구항 90**

삭제

**청구항 91**

삭제

**청구항 92**

삭제

**청구항 93**

삭제

**청구항 94**

삭제

**청구항 95**

삭제

**청구항 96**

삭제

**청구항 97**

삭제

**청구항 98**

삭제

**청구항 99**

삭제

**청구항 100**

삭제

**청구항 101**

삭제

**청구항 102**

삭제

**청구항 103**

삭제

**청구항 104**

삭제

**청구항 105**

삭제

**청구항 106**

삭제

**청구항 107**

삭제

**청구항 108**

삭제

**청구항 109**

삭제

**청구항 110**

삭제

**청구항 111**

삭제

**청구항 112**

삭제

**청구항 113**

삭제

**청구항 114**

삭제

**청구항 115**

삭제

**청구항 116**

삭제

**청구항 117**

삭제

**청구항 118**

삭제

**청구항 119**

삭제

**청구항 120**

삭제

**청구항 121**

삭제

# *명 세 서*

## *기 술 분 야*

<1> 본 출원은 2002년 4월 19일자로 출원된 미국 가출원 제60/374,192호에 대한 우선권을 주장한다.

<2> 본 발명은 시작 코드 에뮬레이션을 방지하기 위한, 그리고 데이터 스터핑(data stuffing)을 위한 방법 및 시스템에 관한 것이다.

## *배 경 기 술*

<3> 디지털 데이터는 전형적으로 소정 유형의 송신기로부터 소정 유형의 수신기로 전송된다. 송신기는 전형적으로, 전송을 위한 데이터를 인코딩하는 인코더를 포함하고, 수신기는 전형적으로, 수신하는 데이터를 디코딩하는 디코더를 포함한다. 비디오 데이터, 오디오 데이터, 오디오/비디오 데이터, 텍스트 데이터, 컴퓨터 실행 가능 프로그램 데이터, 보관 데이터, 데이터베이스 정보 등과 같은 상이한 유형의 디지털 데이터가 있다. 디지털 데이터가 전송될 때, 이는 전형적으로 소정 유형의 채널에서 전송된다. 마찬가지로, 컴퓨터 메모리 또는 임의의 저장 장치 또는 저장 매체는 본 명세서에서의 목적을 위한 전송 채널로 간주될 수 있다.

<4> 디지털 데이터가 전송될 때, 채널에서 데이터 내의 특정 포인트를 찾을 수 있는 것이 중요하다. 이는, 채널을 통한 데이터의 전송에서의 에러 또는 손상으로부터의 복구를 가능하게 하는 포인트나, 전체 스트림의 시작이 아닌 위치에서 프로세스의 디코딩을 시작하는 것을 가능하게 하는 포인트나, 상이한 목적을 위하여 이용되는 상이한 유형의 데이터를 검색하는 것을 가능하게 하는 포인트의 위치를 찾는 등의 다양한 목적을 위하여 행해진다. 따라서, 예를 들어, 디코더 측에서, 데이터가 적절히 처리될 수 있기 위해서는, 디코더 및 디지털 데이터를 처리하는 다른 컴포넌트는 종종 그 데이터의 콘텍스트(context)를 알 필요가 있다. 만일 송신된 제1 비트부터 시작할 수 있고 디코더가 어떤 에러도 없이 동작할 수 있다면, 이는 그다지 중요하지 않을 것이다. 이러한 상황에서 이상적으로 디코더는, 데이터의 포맷이 무엇인지 앎에 따라서 송신되고 있는 정보를 간단히 추적할 수 있을 것이다. 그러나, 이러한 이상적인 환경은 자주 발생하지 않는다. 디지털 데이터를 송신 및 수신하는 시스템을 설계 및 사용하는 사람들에게 도전을 제공하는 에러 및 다른 우발적인 사건이 발생한다. 진행 중인 데이터 방송 스트림으로 변환하는 때와 같은 일부 경우에, 디코더는 데이터 전송의 처음에서 시작할 수 없다. 데이터 포맷 파싱(parsing)에 의해 포인트의 위치를 알아내는 것은 또한 디코더에서 상당한 양의 복잡한 처리를 요구할 수 있다.

<5> 많은 유형의 채널 환경에서, 이러한 이슈는, 데이터 내에 소위 재동기화 마커(resynchronization markers)를 제공함으로써 처리된다. 재동기화 마커는, 시스템이 디코딩 프로세스를 시작하거나 에러로부터 복구할 수 있게 하는 메커니즘을 제공한다. 예를 들어, 디지털 데이터가 일련의 비트 또는 바이트로서 스트림화될 때, 스트림 내에 재동기화 마커를 갖는 것은, 전송에서 에러가 발생할 경우에 복구할 기준 포인트를 디코더에 제공한다.

<6> 재동기화 마커가 채용될 수 있는 한 방법은 시작 코드의 콘텍스트 내에 있다. 시작 코드는 특정 값을 갖는 비트들 또는 바이트들의 스트링이다. 일반적으로, 많은 시스템은 바이트들을 캐리하는 경향이 있기 때문에(예를 들어, H.222.0/MPEG-2 Systems), 시작 코드가 고유 값을 갖는 바이트들의 스트링으로서 정의될 수 있다. 고유한 바이트들의 스트링은 패턴을 제공하는데, 그 패턴의 존재는 재동기화 포인트를 가리킨다. 재동기화 포인트는 전형적으로, 독립적으로 디코딩 가능한 소정의 양의 데이터의 시작 및 경계를 지시한다. 예를 들어, H.262/MPEG-2 비디오 데이터에서, 재동기화 포인트는 슬라이스(slice; 즉, 픽처(picture)의 독립적으로 디코딩 가능한 영역)의 시작, 픽처의 시작, GOP(즉, "Group of Pictures" 또는 픽처의 독립적으로 디코딩 가능한 시퀀스)의 시작, 또는 새로운 비디오 시퀀스의 시작을 지시할 수 있다. 디지털 비디오 스트림은 또한, 시작 코드에 후행할 수 있는 소위 보조 또는 보충 데이터(ancillary or supplemental data)를 포함할 수 있다.

<7> 가끔, 시작 코드는 비디오 스트림과 같은 데이터 스트림 내에서 뿐만 아니라 시스템의 멀티플렉스 레벨에 의해서도 사용된다. H.222.0/MPEG-2 시스템 사양은, 시작 코드를 사용하고, 시스템 레벨 정보 및 오디오 정보로 인터리빙되는 비디오 데이터 스트림을 운반하는 시스템의 한 예이다.

<8> 시작 코드는 데이터 스트림 내에 재동기화 포인트를 제공하는 한 중요하므로, 사실 시작 코드를 나타내도록 의도되지 않은 곳에서 데이터 스트림 내에 시작 코드를 애뮬레이션하는 것을 피하는 것은 좋은 생각이다.

<9> 예를 들어, 다음에 대하여 생각해보라. 시작 코드는, 새로운 데이터 유닛의 시작을 식별할 수 있는 비트들 또는 바이트들의 특정 패턴을 정의한다. 만일 시작 코드들 사이에 임의의 데이터를 송신하고 있는 경우, 그 임의의 데이터가 그 자체 내에 시작 코드로서 사용되는 것과 동일한 패턴을 포함할 수 있다. 예를 들어, 운반되고 있는 데이터가 완전히 랜덤하다고 가정한다면, 시작 코드의 길이가 K 비트라고 하면 소정의 특정한 비트 위치에서 시작하는 비트들에서 시작 코드를 우발적으로 애뮬레이션할 확률은 $1/2^k$이다.

<10> 일부 경우, 시작 코드의 비트들의 수가 크면, 시작 코드가 우발적으로 애뮬레이션될 가능성은 매우 낮다고 판단할 수 있다. 이러한 상황에서, 우발적인 시작 코드 에뮬레이션의 결과가 그다지 심각하지 않다면, 우발적인 시작 코드 에뮬레이션의 방지를 보장하기 위하여 조치를 취하는 것이 불필요하다고 판정될 수 있다. 일부 오디오 데이터 포맷에 관한 경우가 이러한 경우이다. 전형적으로, 이러한 포맷은 초당 비트수로 측정하였을 때 매우 높은 비트 레이트를 이용하지 않으므로, 시작 코드가 임의의 특정 시간 구간 동안 우발적으로 에뮬레이션될 가능성이 매우 낮다. 비트 레이트는 보통 비디오 데이터의 전송에 대하여 훨씬 높으므로, 비디오 데이터에 있어서는 일반적으로 그렇지 않다.

<11> 과거의 주요한 비디오 코딩 표준(아마도 하나의 예외를 가짐)에서, 데이터 페이로드(data payload) 내의 비디오 구문 포맷은 시작 코드 에뮬레이션을 피하도록 설계되었다. 즉, 어떤 종류의 데이터 구성요소가 비디오 구문을 구성할지를 안다면, 우발적인 시작 코드가 발생하지 않도록 신중하게 구문을 설계할 수 있을 것이다. 예를 들어, 종래의 비디오 코딩 표준에서의 시작 코드는 0 비트들의 긴 스트링과, 이에 후속하는 하나의 1-비트로 시작한다. 이러한 긴 스트링은 23개의 0-비트와 그에 후속하는 하나의 1-비트를 포함할 수 있다. 송신되는 대부분의 데이터가 (종종 비공식적으로 호프만 코드라고 불리우는) 가변 길이 코드를 이용하여 엔트로피 코딩된다고 가정하자. 가변 길이 코드(veriable length codes; VLCs)는, 본 명세서에서 예를 위하여, 표현된 심볼들의 세트 중 선택하기 위하여 사용되는 가변 깊이 트리 구조 코드로서 정의된다. 이진 트리 VLC를 이용하는 하나의 기법은, 트리에서 루트(root)로부터 유효한 심볼을 표현하는 모든 리프(leaf)로의 경로가 언제나 그 내부의 어디엔가 "1"을 가지며, 트리 구조가 너무 깊지 않은 것을 보장하는 것이다.

<12> 따라서, 예를 들어, 모든 가변 길이 코드 스트링이 10 비트 보다 길지 않고 이러한 모든 스트링이 그 내부에 적어도 하나의 1-값 비트를 가질 것이라는 것을 안다면, VLC로부터의 코딩된 데이터의 시퀀스가 18개보다 많은 연속적인 0-값 비트를 포함할 수 있는 방법이 없다는 것을 알 것이다. 즉, 최악의 경우의 시나리오는 1000000000 다음에 0000000001이 오는 것일 것이다. 따라서, 구문을 신중하게 설계하고 모든 0-값 및 모든 1-값 비트의 위치를 조사하여 한 행에 얼마나 많은 0이 발생할 수 있는지를 확인하면, 구문에서 발생할 수 있는 0의 스트링보다 긴 스트링을 포함하는 시작 코드를 사용할 수 있다. 예를 들어, 유효한 구문이라면 시작 코드가 아닌 위치에서 23개의 0을 포함하는 일이 결코 없도록 구문이 설계될 수 있다. 따라서, 23개의 0이 발생하는 모든 경우는 시작 코드이어야 하고 디코더는 정확하게 시작 코드를 검출할 수 있어야 한다.

<13> 상술된 동작은 간단해 보이지만, 시작 코드 패턴이 우발적으로 송신될 수 없는 것을 보장하기 위하여, 송신될 가능한 모든 순서에서, (비트 레벨에서) 송신될 가능한 모든 데이터를 조사해야 하므로, 이러한 동작은 상당히 어려운 고려가 될 수 있다. 이는, 실수가 발생하기 쉬운 힘든 구문 설계 방법이다.

<14> 비트 레벨 조사 설계 처리는, 일반적으로, 많은 비디오 코딩 사양이 과거에 설계된 방법을 기술한다(즉, H.261, MPEG-1, H.262/MPEG-2, 대부분의 H.263, 및 MPEG-4). 이들의 하나의 예외는, 수학적인 사양으로부터 산술적인 방식으로 압축된 비트를 생성하기 위하여 산술 코딩이라 불리우는 기법을 사용하는 ITU-T Recommendation H.263의 Annex E이다. 여기서, 엔트로피 인코더의 끝에 생성되는 비트를 조사하는 추가 처리가 있고, 인코더 측에서, 한 행에 너무 많은 0이 있으면, 소정의 수의 0을 만나기 전에 "마커" 비트(하나의 1-비트)가 삽입된다. 디코더 측에서, 디코더는 0을 카운트하고, 임계수의 0을 만나면, 실제 시작 코드를 만났음을 알게 된다. 디코더가 임계수보다 하나 적은 0을 만나면, 후속하는 1 비트는 시작 코드 에뮬레이션을 피하기 위하여 삽입된 마커 비트라는 것을 알게 되고, 그 비트를 폐기하고, 후속하는 비트를 실제 데이터의 연속으로서 받아 들인다.

<15> 이러한 방법의 문제점은, 인코더 및 디코더에 의해 비트 레벨에서 들어오는 데이터를 조사 및 처리하게 한다는 것이다. 싱글 비트 위치마다 처리되고 있는 데이터의 위치를 분석 및 시프팅하는 것은 어렵고 디코더를 바람직하지 않게 혹사시킬 수 있다. 비트 단위(bit-wise) 시프팅은 또한 프로세서 집약적인 연산이다.

<16> 따라서, 본 발명은 시작 코드 에뮬레이션을 방지하기 위한 개선된 방법 및 시스템을 제공하는 것과 연관된 관계로부터 발생하였다.

## *발명의 상세한 설명*

<17> 시작 코드 에뮬레이션 방지를 위한 접근 방식들을 제공하는 방법들 및 시스템들을, 비트 레벨보다 높은 입도(granularity)에서 행해지는 동작들에 의해 설명한다. 비트 레벨과는 다른 레벨에서 동작함으로써, 처리 효율이 향상될 수 있다. 하나 이상의 실시예들에 따르면, 시작 코드 에뮬레이션 방지 방법은 싱글 비트들보다 큰 고정된 크기의 데이터 부분들에 대한 데이터 패턴들을 찾는다. 특정 패턴이 발견되면, 시작 코드 에뮬레이션 방지 데이터가 시작 코드 에뮬레이션을 방지하기 위해 삽입된다. 삽입된 데이터는 싱글 비트보다 더 크고,

어떤 실시예들에서는 바이트를 포함한다. 디코더는, 시작 코드 에뮬레이션 방지 데이터가 삽입된 데이터를 디코딩하면, 적법한(legitamate) 시작 코드들을 쉽게 식별한 다음 시작 코드 에뮬레이션 방지 데이터를 제거하여, 전달 예정이었던 본래 데이터를 제공할 수 있게 된다.

<18> 다양한 실시예들에서, 바이트 경계들 등과 같은 데이터 경계들과는 다른 위치들에서 시작 코드값들의 우발적인 에뮬레이션을 방지하는 시작 코드 에뮬레이션 방지가 발생할 수 있다. 이들 실시예들은 처리중인 데이터에서 데이터 정렬 경계들을 항상 유지하지는 않는 시스템들과 연관하여 이용될 수 있다. 어떤 실시예들에서, 설명된 기술들은 데이터 경계들이 손상되는 경우에 디코더 시스템들이 복구할 수 있게 하는 기초를 제공할 수 있다.

<19> 또한, 페이로드 데이터의 크기를 바이트 단위의 정수배 크기로 올림(rounded up)한 다음, 필러 데이터(filler data)를 디코더에 의해 용이하게 검출할 수 있는 방법으로 부가할 수 있게 하는 데이터 스터핑(data stuffing) 방법을 설명한다.

## *실 시 예*

<27> **개요**

<28> 비트 레벨보다 높은 입도에서 시작 코드 에뮬레이션 방지를 위한 접근 방식들을 제공하는 방법 및 시스템들을 이하에 설명한다. 비트 레벨보다 높은 레벨에서 동작함으로써 처리 효율이 향상될 수 있다. 이 문헌의 콘텍스트에서, 비트 레벨보다 높은 레벨에서 동작한다는 것은 싱글 비트들보다 큰 고정 크기의 데이터 부분들에 대한 데이터 패턴들을 찾는 프로세스를 참조한다는 것을 의미한다. 예를 들어, 고정 크기 데이터 부분들은 바이트(즉, 8 비트), "워드"(즉, 16 비트), "더블 워드"(32 비트) 등을 포함할 수 있다. 따라서, 본 기술들은 바이트들, 워드들 등 내에 그리고 바이트들, 워드들 간에 패턴들을 찾을 수 있다.

<29> 더우기, 페이로드 데이터 크기를 바이트 양과 같은 데이터 유닛의 정수배 크기로 올림할 수 있게 하고, 필러 데이터를 디코더에 의해 용이하게 검출할 수 있는 방법으로 부가할 수 있게 하는 데이터 스터핑 방법을 설명한다.

<30> 다양한 실시예들에서, 바이트 경계들 등에서와 같은 데이터 경계들과는 다른 위치들에서 시작 코드값들의 우발적인 에뮬레이션을 방지하는 시작 코드 에뮬레이션 방지가 발생할 수 있다. 이들 실시예들은 처리 중인 데이터에서의 데이터 정렬 경계들을 항상 유지하지는 않는 시스템들과 연관하여 이용될 수 있다. 어떤 실시예들에서, 설명된 기술들은 데이터 경계들이 손상되는 경우에 디코더 시스템들이 복구할 수 있게 하는 기초를 제공할 수 있다.

<31> 또한, 이하에 제공된 예들은 비디오 데이터의 콘텍스트에서 논의되지만, 본 발명의 기술들은, 전형적으로 인코드 및 디코드되고 시작 코드 에뮬레이션 방지가 바람직하거나 필요한 임의의 타입의 데이터와도 연관하여 채용될 수 있음을 주지하기 바란다. 이러한 데이터의 예들은 오디오 데이터, 오디오/비디오 데이터 등을 포함한다.

<32> 도 1은 일 실시예에 따른 방법에서의 단계들을 설명하는 흐름도이다. 이 방법은 임의의 적당한 하드웨어, 소프트웨어, 펌 웨어 또는 이들의 조합으로 구현될 수 있다. 도시되고 설명된 실시예에서, 방법의 적어도 일부는 소프트웨어로 구현된다. 또한, 당업자는 이 방법이 두개의 다른 분기들- 하나는 "인코더"로 지정되고 하나는 "디코더"로 지정됨- 을 갖는 것으로 도시됨을 인지할 것이다. "인코더" 분기는 인코더에 의해 또는 인코더와 연관하여 실행되는 단계들을 설명한다. 마찬가지로, "디코더" 분기는 디코더에 의해 또는 디코더와 연관하여 실행되는 단계들을 설명한다.

<33> 단계(100)는 시작 코드 이후 전송 예정인 데이터 양을 획득하거나 생성한다. 이 데이터는 어떠한 적합한 데이터도 포함할 수 있다. 데이터 타입의 예들은 제한없이 비디오 데이터, 오디오 데이터, 오디오/비디오 데이터 등을 포함한다. 단계(101)는 데이터를 시작 코드로 프리픽스(prefix)한다. 이 단계는 단계(100)에서 획득되거나 생성된 데이터에 시작 코드들을 유효하게 부가한다. 단계(102)는 고정된 크기의 데이터 부분들의 하나 이상의 패턴들에 대한 입력 데이터를 체크하거나 검색한다. 도시되고 설명된 실시예들에서, 검색된 패턴(들)은 적어도 두개의 고정 크기 데이터 부분들을 포함하고 각 개별 데이터 부분은 적어도 2 비트를 포함한다. 단계(104)는 패턴이 발견되었는지 여부를 판정한다. 패턴이 발견되지 않았다면, 방법은 단계(108)로 진행하고 여기서 데이터가 전송될 수 있다.

<34> 한편으로, 패턴이 발견되면, 단계(106)에서 패턴을 포함하는 데이터에 대해 시작 코드 에뮬레이션 방지 데이터를 삽입한다. 도시되고 설명된 실시예에서, 시작 코드 에뮬레이션 방지 데이터의 개별 인스턴스들은 1 비트보다 많은 비트를 포함한다. 바람직하게는, 시작 코드 에뮬레이션 방지 데이터는 개별 고정 크기 데이터 부분과

비트수가 동일한 데이터 양으로 구성된다. 따라서, 고정 크기 데이터 부분이 8 비트(1 바이트로서 언급되는 데이터 양)로 구성될 때, 시작 코드 에뮬레이션 방지 데이터는 8 비트로 구성된다. 시작 코드 에뮬레이션 방지 데이터가 삽입된 후, 단계(108)에서 그 데이터를 전송한다. 단계(110)에서 다음 시작 코드 전에 전송될 추가 데이터가 있는지 여부를 판정한다. 만약 있다면, 방법은 단계(102)로 복귀하고 상술한 바와 같이 처리한다. 데이터가 없다면, 방법은 단계(111)에서 추가 데이터가 전송될 것인지 여부를 판정할 수 있다. 만약, 추가 데이터가 있다면, 방법은 단계(100)로 복귀한다. 없다면, 방법은 단계(112)에서 종료할 수 있다.

<35> 이외에, 다음을 고려하자. 이 특정한 기술을 사용하는 일 예는 시작 코드를 "시작 코드 프리픽스"와 "시작 코드 타입" 서픽스(suffix)로 분리하고, 여기서 프리픽스는 값들의 하나의 고유 스트링이고 서픽스는 시작 코드에 후속하는 데이터의 타입을 나타낸다. 특히, 이것은 MPEG-2 시작 코드들의 구조이다. 프리픽스/서픽스 구조를 특별한 경우로서 포함하는 좀 더 일반적인 사용 형태는 하나 이상의 시작 코드 패턴들을 갖는 일반적인 개념이다. 그러면, 하나 이상의 에뮬레이션 방지 패턴들을 가질 수도 있다. 각종 시작 코드 패턴들이 각종 에뮬레이션 방지 패턴들과 구별되고 모든 시작 코드 패턴들이 페이로드 데이터 처리에서 회피되는 한, 이러한 방식은 적절하게 기능할 것이다. 또한, 반드시 시작 코드 패턴들이 모두 동일한 길이를 갖는 것으로 가정할 필요는 없다.

<36> 디코더 측에서, 다음을 고려하자. 일단 시작 코드 에뮬레이션 방지 데이터가 인코더에 의해 삽입된다면, 이는 식별되고, 다른 데이터의 적절한 해석(interpretation)을 위하여 어떤 포인트에서 제거 또는 무시될 수 있다. 또한, 디코더는 전송된 데이터를 수신한 때에, 적법한 시작 코드들을 찾을 수 있다는 점도 고려하자. 일단 적법한 시작 코드들을 찾으면, 시작 코드로 한정된 데이터 경계들이 위치하는 곳을 알게 된다. 이제, 디코더는 시작 코드 에뮬레이션 방지 데이터를 찾고 제거하기 위해 진행할 수 있으므로 실제 데이터를 더 처리할 수 있다.

<37> 구체적으로, 단계(114)는 시작 코드들의 에뮬레이션을 방지하기 위해 인코더에 의해 처리되었던 전송 데이터를 수신한다. 단계(118)는 시작 코드를 찾기 위하여 데이터를 처리한다. 일단 시작 코드가 발견되고 적절히 처리되면(예를 들어, 판독 및 폐기), 단계(120)에서 시작 코드 에뮬레이션 방지 데이터를 식별하기 위하여 데이터를 검색한다. 일단 시작 코드 에뮬레이션 방지 데이터가 발견되면, 단계(122)에서 시작 코드 에뮬레이션 방지 데이터를 제거한다. 일단 시작 코드 에뮬레이션 방지 데이터가 제거되면, 데이터는 수신되었던 데이터 타입에 대한 전형적인 방식으로 처리될 수 있다. 예를 들어, 데이터는 단계(124)에서와 같이 소비자 장치에 의해 소비될 수 있다.

<38> **제1 예시적 방법**

<39> 기술될 방법은 도 1에 도시되고 설명된 방법의 일 특정예만을 설명한 것이다. 설명하려 하는 방법에서, N+1 바이트의 페이로드 데이터 스트링이 전체 시작 코드 프리픽스와 매치하거나 또는 시작 코드 프리픽스의 첫번째 N 바이트들 + 에뮬레이션 방지 바이트의 값과 매치할 때마다 1 바이트의 에뮬레이션 방지 데이터가 삽입된다. 이 방법은 "제2 예시적 방법"이라는 섹션에서 설명되는 방법보다는 데이터를 덜 부가하고, 따라서 페이로드 데이터 송신에 필요한 전송 용량을 감소시킨다.

<40> MPEG-2 시작 코드 프리픽스 구조는 바이트 정렬된 위치에서 시작하고 23개의 0과 후속하는 하나의 1을 갖는다. 이 시작 코드 프리픽스는 아래와 같다.

<41> 00000000 00000000 00000001

<42> 이 구조는 같은 값을 갖는 소정의 수의 바이트 N 과, 다른 값을 갖는 후속의 소정의 다른 바이트를 포함하는 패턴으로서 일반화될 수 있다. MPEG-2에서, N=2이고, 첫번째 두개 바이트들은 0 (이하에서 "W"라고 함)이고, 마지막 바이트는 1 (이하에서 "X"라고 함)이라고 할 수 있다. 따라서, 시작 코드 프리픽스는 다음의 패턴을 갖는다.

<43> WWX

<44> 이들 3개 바이트들 다음에, MPEG-2에서, 다른 바이트가 후속하여 어떤 종류의 시작 코드인지 식별한다. 이와 같이 후속하는 바이트를 "Y"라고 한다. 본질적으로, 시작 코드는 시작 코드 프리픽스 WWX와, 이에 후속하는 시작 코드 타입을 식별하는 바이트 Y로 구성된다. 전체 MPEG-2 시작 코드는 다음과 같이 표현될 수 있다.

<45> WWXY

<46> 시작 코드 프리픽스(WWX)는 고정값을 갖는 한편, Y는 시작 코드 타입(예를 들어, 슬라이스, 픽처, GOP, 시퀀스,

시스템 등)을 나타내는 많은 서로 다른 값들을 갖는다.

<47> 일 실시예에 따르면, 패턴 WWX를 찾기 위해 데이터가 처리된다. WWX 패턴이 발견되면, 시작 코드 에뮬레이션 방지 데이터가 시작 코드 에뮬레이션을 방지하기 위해 삽입된다. 여기서, 시작 코드 에뮬레이션 방지 데이터는 W와 X 바이트들의 값들과 구별되는 값을 갖는 바이트 Z를 포함한다. 따라서, 인코더가 데이터 바이트들을 검사하고 패턴 WWX를 통지한다고 가정하자. 데이터에서 이러한 패턴을 발견하는 데 응답하여, 인코더는 값 Z를 갖는 바이트를 삽입하여 다음 패턴을 제공한다.

<48> WWZX

<49> 이 포인트에서, 인코더는 디코더에 의해 전송되고 처리될 페이로드 데이터가 시작 코드 또는 시작 코드 프리픽스를 우발적으로 에뮬레이션하지 않음을 보장한다. 이제 다음을 고려하자. 페이로드 데이터가 WWX 패턴을 임의로 포함하여 시작 코드 프리픽스를 에뮬레이션할 가능성이 있기 때문에, 페이로드 데이터는 또한 시작 코드 에뮬레이션 방지 데이터를 포함하는 데이터를 임의로 에뮬레이트할 가능성도 갖는다. 즉, 페이로드 데이터는 본질적으로 패턴 WWZX를 포함할 수도 있다. 만약 이러한 경우 인코더가 아무것도 하지 않으면, 디코더가 시작 코드 에뮬레이션 방지 데이터를 제거하려 시도할 때, 이 경우 실제 데이터인 Z 바이트를 제거할 것이다.

<50> 따라서, 설명된 실시예들에서, 인코더는 페이로드 데이터가 시작 코드들 또는 시작 코드 프리픽스들을 에뮬레이션하는 것을 방지하도록 구성될 뿐만 아니라, 데이터가 시작 코드 에뮬레이션 방지 데이터 사용으로부터 발생하는 데이터 패턴들을 에뮬레이션하는 것을 방지하도록 구성된다. 특히, 본 예에서, 인코더가 패턴 WWZ를 식별한다면, 두번째 W와 Z 사이에 값 Z를 갖는 바이트를 삽입하여 다음의 패턴(삽입된 바이트 Z는 아래에서 첫번째로 나타나는 Z이다)을 제공한다.

<51> WWZZ

<52> 이제, 처리된 데이터를 디코더 시각에서 고려하자. 만약 디코더가 Z 또는 X로 이어지는 WWZ를 포함하는 바이트들의 어떤 패턴을 본다면, 첫번째 Z는 인코더에 의해 삽입되었던 에뮬레이션 방지 바이트임을 알게 된다. 따라서, 디코더는 첫번째 Z를 폐기할 수 있다. 따라서, 본 예에서, 에뮬레이션 방지 바이트가 삽입될 수 있는 두 개 상황들이 있다. 제1 상황은 데이터가 시작 코드 또는 시작 코드 프리픽스를 우발적으로 에뮬레이션하는 경우이다. 제2 상황은 그 데이터가 에뮬레이션 방지 바이트 삽입된 데이터를 우발적으로 에뮬레이션하게 되는 경우이다.

<53> 어떤 경우에서도, 디코더는 적절한 패턴을 간단히 찾을 수 있고, 에뮬레이션 방지 바이트를 폐기할 수 있으며, 데이터를 통상적인 방법으로 처리할 수 있다.

<54> 이하에서는 상술한 프로세싱을 좀 더 프로그램 형식으로 설명한다. 인코더 측에 있어서, 동일한 값 W의 N개 이상의 바이트와 다른 값 X의 최종 바이트로 이루지는 시작 코드 프리픽스와, 후속하는 값 Y의 1 바이트의 식별 시작 코드 타입 서픽스로 시작되는 B 바이트들의 패킷 P[]을 전송하기 위해, 아래와 같은 값 Z을 갖는 에뮬레이션 방지 바이트들을 삽입하는 의사 코드 프로세스를 행하며(여기서, W, X, Y, Z는 서로 상이한 값을 갖고, P[B-1]은 W와 동일하지 않다), 여기서 채널을 채우기 위해 전송하는 추가 데이터의 양은 E로 표시된다.

```
int  B, N, E, i, j;
byte *P, W, X, Y, Z;
for(j=0; j<N+E; j++) /* start code prefix (SCP) */
  send_byte( W );    /* jth byte of SCP  */
send_byte( X );      /* last byte of SCP */
send_byte( Y );      /* start code type suffix */
for(i=j=0; i<B; i++) {
  if(j >= N && (P[i] == X || P[i] == Z)) {
    send_byte( Z );
    j = 0;
  }
  send_byte( P[i] ); /* a byte of data payload */
  if (P[i] == W) j++;
  else           j = 0;
}
```

<55>

<56> 상술한 의사 코드에서, 함수 "send_byte()"는 단위 데이터의 전송을 행하는 것으로 가정한다(도 1의 프로세스(108) 참조).

<57> 디코더 측에 있어서, 패킷을 수신하기 위해, 디코더는 동일한 값 W의 N개 이상의 바이트와 다른 값 X의 최종 바이트로 이루어진 알고 있는 시작 코드 프리픽스를 이미 발견하여, 판독하고, 폐기하였다고 가정한다. 또한, 알려지지 않은 1 바이트 시작 코드 타입 서픽스를 변수 Y로 판독하고 페이로드 데이터의 패킷을 어레이 P[]로 판독하여, 페이로드 데이터의 양을 결정하고 변수 B에 그 양을 표시하는 한편, 값 Z을 갖는 에뮬레이션 방지 바이트들을 제거한다고 가정한다(여기서, W, X, Y, Z는 서로 상이한 값을 갖고, P[B-1]은 W와 동일하지 않다):

```
int  B, N, j, next;
byte *P, W, X, Y, Z;
/* assume start code prefix was already read */
Y = receive_byte( ); /* start code type suffix */
for(B=j=0, next=0; more_data() && !next; B++) {
  P[B] = receive_byte( );
  if(j >= N) {
    if(P[B] == W)
      j++;
    else{
      j = 0;
      next = (P[B] == X);
      if(P[B] == Z)
        B--;
    }
  }else
    if(P[B] == W) j++;
    else          j=0;
}
if(next) /* another start code found */
  B -= j+1;
```

<58>

<59> 상술한 의사 코드에서, 함수 "receive_byte()"는 단위 데이터의 수신을 행하는 것으로 가정하고, 함수 "more_data()"는 수신할 단위 데이터가 더 있는지 여부를 판별하는 것으로 가정한다(도 1의 프로세스(114)를 구성하는 두 개의 함수 참조).

<60> 상술한 방법에서는 임의의 양의 W 값을 시작 코드 앞에 채울 수 있으며, W값 프리픽스의 수를 N으로 고정하는 것도 동일하게 가능하다.

<61> **제2 예시적 방법**

<62> 도 1에 도시하고 설명한 방법의 다른 실시예를 설명한다. 여기서는 페이로드에서 N 바이트 데이터의 스트링이 시작 코드 프리픽스의 최소 N 바이트와 매칭하는 경우, 후속 페이로드 데이터의 값에 상관없이, 1 바이트의 에뮬레이션 방지 데이터를 삽입한다. 상술한 실시예에 따르면, 데이터가 임의의 것이 후속하는 패턴 "WW"를 포함하면 에뮬레이션 방지 바이트를 삽입한다. 따라서, 인코더가 패턴 WW를 식별하는 경우, 에뮬레이션 방지 바이트를 삽입하여 다음과 같은 패턴을 제공한다.

<63> WWZ

<64> 처음에 설명한 방법과 바로 전에 설명한 방법간의 차이점은, 처음 방법은 에뮬레이션 방지 바이트를 어디에 삽입할지를 확인하기 위해 최초 N+1 바이트를 관찰하는 반면, 바로 전에 설명한 방법은 최초 N 바이트를 관찰한다는 것이다.

<65> 처음 방법은 전송할 추가 데이터의 양을 감소시키며, 바로 전에 설명한 방법은 보다 간단한 규칙으로 동작한다. 그러므로, 두 방법을 조합하면 전송되는 데이터의 양을 줄이는 것과 규칙의 복잡성을 줄이는 것 사이의 선택이 가능하다. 처음 설명한 방법은 두 번째 설명한 방법에 비해 데이터의 양이 감소한다. 두 번째 설명한 방법은 보다 간단한 규칙을 사용한다.

<66> 이하에서는 상술한 프로세싱을 좀 더 프로그램 형식으로 설명한다. 인코더 측에 있어서, 동일한 값 W의 정확히 N개 바이트와 다른 값 X의 최종 바이트로 이루어진 시작 코드 프리픽스와, 후속하는 값 Y를 갖는 1 바이트의 식별 시작 코드 타입 서픽스를 갖는 B 바이트의 패킷 P[]을 전송하기 위해, 아래와 같은 값 Z을 갖는 에뮬레이션 방지 바이트를 삽입하는 의사 코드 프로세스를 행한다(여기서, W, X, Y, Z는 서로 다른 값을 갖고, P[B-1]은 W와 동일하지 않다):

```
int  B, N, i, j;
byte *P, W, X, Y, Z;
for(j=0; j<N; j++)    /* start code prefix */
  send_byte( W );     /* first byte of start code */
send_byte( X );       /* last byte of start code */
send_byte( Y );       /* start code type suffix */
for(i=j=0; i<B; i++) {
  send_byte( P[i] ); /* a byte of data payload */
  if( P[i] != W )
    j=0;
  else
    if(++j == N) {
      send_byte( Z );
      j = 0;
    }
}
```

<67>

<68> 상술한 의사 코드에서, 함수 "send_byte()"는 단위 데이터의 전송을 행하는 것으로 가정한다(도 1의 프로세스(108) 참조).

<69> 디코더 측에 있어서, 패킷을 수신하기 위해, 디코더는 동일한 값 W의 정확히 N개의 바이트와 다른 값 X의 최종 바이트로 이루어진 알고 있는 시작 코드 프리픽스를 이미 발견하여, 판독하고, 폐기하였다고 가정한다. 또한, 알려지지 않은 1 바이트 시작 코드 타입 서픽스를 변수 Y로 판독하고 페이로드 데이터의 패킷을 어레이 P[]로 판독하여, 페이로드 데이터의 양을 결정하고 변수 B에 그 양을 표시하는 한편, 값 Z을 갖는 에뮬레이션 방지 바이트들을 제거한다고 가정한다(여기서, W, X, Y, Z는 서로 상이한 값을 갖고, P[B-1]은 W와 동일하지 않다):

```
int  B, N, j, k;
byte *P, W, X, Y, Z;
/* assume start code prefix was already read */
Y = receive_byte( ); /* start code type suffix */
for(B=j=0, k=Z; more_data() && k != X; B++) {
  P[B] = receive_byte( );
  if(P[B] == W) {
    if(++j == N) {
      k = receive_byte( ); /* more_data() always */
      if(k != Z && k != X)
        declare_error( );
      j = 0;
    }
  }else
    j = 0;
}
if(k == X) /* another start code found */
  B -= N;
```

<70>

<71> 상술한 의사 코드에서, 함수 "receive_byte()"는 단위 데이터의 수신을 행하는 것으로 가정하고, 함수 "more_data()"는 수신할 단위 데이터가 더 있는지 여부를 판별하는 것으로 가정한다(도 1의 프로세스(114)를 구성하는 두 개의 함수 참조).

<72> 상기한 방법들은, 많은 양의 이상적인 랜덤 입력 페이로드를 제2 방법에 대하여 대략 $1/256^{N}$의 인수로 그리고 제1 방법에 대하여 $1/256^{(N+1)}$의 인수로 확장할 것으로 예상된다. 이러한 양은 N이 크면 작다(예를 들어, 2 이상이며, MPEG-2 시작 코드용으로 N=2임을 주목하길 바란다). 페이로드를 위한 최악의 확장 인수는 제2 방법에 대하여 1/N이고 제1 방법에 대하여 1/(N+1)이라 예상된다. N이 증가하면, 페이로드 확장 인수는 통계적 분석 및 최악의 경우의 분석 모두에 있어서 감소되지만 시작 코드에 의해 이용되는 데이터의 양 자체는 증가하게 된다.

<73> 상기한 에뮬레이션 방지 프로세스는 패킷을 전송하기 전에 얼마나 많은 데이터가 그 패킷에 존재하는지를 아는지에 의존하지 않는다는 것을 인식해야 한다. 따라서 이것은 어떠한 지연도 부가하지 않는다.

<74> 제2 방법은, 삽입된 에뮬레이션 방지 바이트가 Z값을 갖는 싱글 바이트임을 가정한다. 삽입된 데이터의 제1 바이트가 유효 시작 코드를 에뮬레이션하거나 프리픽스 시작의 연속으로 보일 W 또는 X가 아니기만 하다면, 에뮬레이션 방지 데이터용으로 값들의 하나 이상의 스트링이나 다수의 값 또는 임의의 값을 대신 이용할 수 있다.

<75> (예를 들어, H.263-스타일 GOB 프레임 ID/픽처 시퀀스 번호와 같이, 또는 단지 MSB를 '1'로 설정하고 나머지 7

개 비트를 이용하여 ASCII 문자를 전송하도록) 이러한 에뮬레이션 방지 바이트에 정보를 전달할 수도 있다.

<76> 디코더 측에서의 패킷의 끝에서 발생하는 것을 고려해 보면, 데이터 패킷 페이로드의 최종 바이트가 W가 아니면 동작을 제어하는 것이 보다 쉽다는 것을 인식하게 된다. 이것은, 시작 코드 전에 전송된 최종 바이트가 에뮬레이션 방지 바이트로 될 필요가 전혀 없음을 의미하며, 디코더가 그 다음 시작 코드용의 W과 동일한 바이트의 시퀀스의 시작과 페이로드 데이터의 끝 간에서 검출가능한 경계를 찾아낼 수 있음을 의미한다. 이것을 적용함으로써 그 페이로드의 끝이 어디인지를 추적하지 못하는 경우 없이 페이로드의 끝 후에 그리고 그 다음 시작 코드의 전에 W 바이트의 임의의 양(예를 들어, 0 바이트)으로 스터프를 행할 수 있다.

<77> **데이터 스터핑(Data Stuffing)**

<78> 일반적인 비디오 데이터에 있어서, 데이터 페이로드로서 전송되는 데이터는 바이트 유닛의 정수배가 아닐 수 있다. 예를 들어, 그 데이터는 2개의 시작 코드 간에 전송될 627비트를 가질 수 있다. 그러나, 시스템 멀티플렉스 레벨은 바이트로 동작할 수 있다. 이것은 MPEG-2 사양에 대한 것이다. 전송 에러에 의해 발생하는 일부 폴스(false) 시작 코드 패턴의 검출을 가능하게 하거나 페이로드 시작의 데이터 콘텐츠용으로 간단한 디코딩 프로세스를 가능하게 하는 것과 같은 다른 이유들은, 패킷이 바이트와 같은 데이터 유닛의 정수배를 포함하는 필요성을 정당화할 수 있다. 따라서, 데이터의 627비트를 전달하기 위해 더 많은 데이터를 전송해야 할 수 있다. 이에 따라, 데이터를 패드 아웃(pad out)하여 바이트 유닛의 정수배로 만드는 방식이 문제로 된다.

<79> 단지 추가 필러 데이터를 전송하는 것이 유용한 다른 상황들이 있다. 예를 들어, 채널이 1Megabit/sec의 용량을 갖고 전송되는 페이로드 데이터의 양이 단지 900kbits/sec이면, 그 채널을 필러 데이터(filler data)로 채울 필요가 있거나 채우길 원할 수 있다.

<80> 일 실시예에 따라, 데이터 스터핑 기술에 의해 추가 데이터를 그 채널에 부가하여 필수적으로 그 페이로드 데이터를 패딩할 수 있다.

<81> 도 2는, 시작 코드가 0 비트들의 스트링으로 시작하는 것으로 가정되는 일실시예에 따른 데이터 스터핑 방법의 단계들을 기술한 흐름도이다. 단계(200)는 전송 예정인 데이터의 끝의 위치를 확립한다. 단계(202)는 페이로드 데이터의 최종 비트 후에 "1" 비트를 삽입한다. 단계(204)는 정수개의 바이트들을 전송하는 데 필요한 추가 비트들의 수를 결정한다. 단계(206)는 삽입된 "1" 비트 후에 필요한 "0" 비트들의 수를 삽입한다. 단계(208)는 임의의 소망의 개수의 바이트의 필러 데이터를 추가한다. 필러 데이터는, 트루(true) 페이로드 데이터의 위치 및 고의 시작 코드의 위치에 대하여 혼란을 피하도록 설계된 임의의 데이터 패턴들로 구성될 수 있다. 이것은 전형적으로 "0" 값의 바이트들을 삽입함으로써 구현된다.

<82> 일례로서, 다음과 같은 상황을 고려한다. 627 비트가 전송된다고 가정한다. 여기서, 단계(202)는 627번째 비트 후에 "1"비트를 삽입할 것이다. 이후, 단계(204)는 정수개의 바이트들(여기서는, 79바이트)을 제공하는 데 4개 비트가 더 필요하다고 결정할 것이다. 이에 따라, 단계(206)는 4개의 "0" 비트, 즉 0000을, 삽입된 "1" 비트 후에 삽입할 것이다. 이제, 정수개의 바이트들을 확립하였으므로, 단계(208)는, 필요시에 임의의 필요로 하는 필러 데이터의 바이트들의 수를 추가할 수 있다. 이 특정 예에서, 0의 값을 갖는 바이트들을 삽입할 수 있다. 필러 데이터는 단순히 필러 데이터로서, 또는 디코더가 특정 목적을 위해 사용할 수 있는 정보를 포함하는 것 등의 특정한 다른 목적을 위해 사용될 수 있다.

<83> 삭제

<84> 디코더에 있어서의 상황을 고려해보자. 디코더는, 스터핑된 데이터를 수신하고, 데이터의 끝에서 시작하여 데이터를 통해 후방을 볼 수 있다. 디코더가 1 비트의 바이트에 도달할 때까지 초기에 보게 될 전체 바이트들은 0 바이트들일 것이다. "1" 비트는, 페이로드 또는 실제 데이터의 끝의 위치를 디코더에게 말해 준다. 따라서, 디코더가 일단 삽입된 "1"을 발견하면, 실제 데이터가 끝나는 곳을 정확히 결정할 수 있다.

<85> 따라서, 상술한 기술은 전송되는 비트들의 수가 정수개의 데이터 단위로 구성되도록, 전송될 비트 수를 올림하는데 사용된다. 부가적으로, 이 기술은 페이로드 데이터의 시작을 지정하는 시작 코드들 간에 필러 데이터를 스터핑하는데 사용될 수 있다.

<86> **비-경계 정렬 위치에 대한 시작 코드 에뮬레이션 방지(Start Code Emulation Prevention for Non-Boundary Aligned Positions)**

<87> 전술한 실시예는, 데이터의 경계 정렬 또는 바이트 정렬을 유지하도록 설계된 시스템에서는 잘 적용된다. 그러나, 데이터 경계 정렬을 항상 추적하지 않는 몇몇 시스템들이 있다.

<88> 예를 들면, ITU-T H.320으로 언급되는 이러한 하나의 시스템에서는, 데이터가 바이트들이 아닌 비트들의 시퀀스로서 전송되고 미공지량의 데이터 손상이 발생할 수 있기 때문에, 데이터 스트림의 시작과 관련하여 시스템의 데이터의 각 바이트가 시작되는 곳을 추적하지 못할 가능성이 있다. 시스템이 바이트 정렬을 추적하지 못하고 또한 에뮬레이션 방지가 바이트의 경계에서 시작하는 에뮬레이션만을 방지한다면, 바이트 정렬 손실로부터 복구되는 시스템 능력이 방해될 수 있다. 예를 들면, 시스템이 바이트 경계에서 시작 코드 에뮬레이션만을 방해하면, 시작 코드 에뮬레이션이 바이트 경계 이외의 위치에서 발생할 가능성이 있다. 즉, 바이트 경계에서의 시작 코드 에뮬레이션 방지를 위해 인코딩된 시스템 시스템은 바이트 경계 이외의 위치에서 발생한 비트-시프트 시작 코드 에뮬레이션에 대해서는 개의치 않을 수 있다. 바이트 경계 위치를 유지하거나 보존하는 시스템의 경우, 이는 문제가 되지 않는다. 왜냐하면, 이들 시스템들이 바이트 경계가 있는 곳을 아는 경우에, 이들은 바이트 경계의 위치를 쉽게 알아내고, 실제 시작 코드를 찾기 때문이다. 실제 시작 코드가 일단 발견되면, 시스템은 전술한 바와 같이 복구될 수 있다. 바이트 경계가 손실된 시스템에서는, 어떠한 이유에서건, 시스템이 비트-시프트 위치에서 발생하는 에뮬레이션된 시작 코드를 이용하여 복구하려고 시도할 수 있기 때문에 문제가 될 수 있다. 구체적으로는, 바이트 경계가 손실되면, 시스템은 원래의 바이트 경계에 대해 소정의 임의의 정렬 위치에서 시작하는 시작 코드 패턴에 맞는 임의의 데이터를 찾으려고 시도할 수 있다. 이러한 데이터가 실제 시작 코드가 아니고 오히려 비트 시프트 위치에서의 에뮬레이션된 시작 코드이면, 이 위치로부터의 복구는 시스템의 적법한 복구 능력을 심각하게 저해할 수 있다.

<89> 바로 아래에서 설명하는 실시예는 반드시 바이트 경계 등의 데이터 경계에서 시작하지는 않는 위치에서의 시작 코드 에뮬레이션을 방지하는 방법 및 시스템을 제공하는 것에 관한 것이다. 따라서, 이 방법 및 시스템들은 데이터 경계 위치에서 데이터 정렬을 항상 유지하지 않는 시스템과 관련하여 채용되는 것이 유리할 수 있다. 또한, 특정한 이유로 데이터 경계를 추적하지 못하는 시스템에서, 발명의 방법 및 시스템은 적법한 시작 코드만을 이용하여 시스템이 정당하게 복구되는 방법을 제공할 수 있다.

<90> **조기 인터럽션(Early Interruption)**

<91> 설명하는 제1의 접근 방식은 "조기 인터럽션" 접근 방식이라고 불린다. 여기서, 조기 인터럽션 접근 방식은 시작 코드의 시프트 버젼이 발생할 수 있는 어떠한 가능성이라도 존재하기 전에 잠재적인 시작 코드 에뮬레이션을 인터럽트하는 것에 관한 것이다. 이것은 데이터 바이트의 시작에 대한 그 시작 코드의 각 시프트 위치에 대해, 시작 코드 설계의 비트들의 패턴을 상세히 분석함으로써 이루어진다. 다음으로, 바이트 단위로 정렬된 시작 코드가 발생될 수 있을 경우에만 특수 에뮬레이션 방지 바이트를 삽입하는 대신에, 본 방법은 이 경우에 바이트 경계들인 데이터 경계들에 대해 시프트된 위치에서의 시작 코드 패턴의 비정렬 에뮬레이션을 방지하는 방식으로 에뮬레이션 방지 바이트를 삽입한다.

<92> 트루 시작 코드의 검출은 데이터 경계 또는 바이트 경계에 대해 데이터의 트루 정렬을 검출하는 능력을 제공할 수 있다. 이는 시작 코드들이 통상적으로 데이터 또는 바이트 경계에서 항상 시작하도록 설계되기 때문이다. 데이터의 트루 시작 코드의 위치의 검출후, 수신기는 시작 코드로부터 그리고 시작 코드 에뮬레이션 방지 데이터로부터 페이로드 데이터를 분리하는 디코딩 프로세스시 도움이 되는 각 바이트의 시작 위치의 인식을 이용할 수 있다. 이는 디코더가 전송된 데이터의 시작 코드 패턴의 에뮬레이션을 방지하는데 사용되는 방법을 인식하도록 설계될 수 있기 때문이다.

<93> 도 3은 일 실시예에 따른 방법을 설명하는 흐름도이다. 도시된 방법은 시작 코드가 비트 시프트 위치에서 에뮬레이트되지 않도록 시작 코드 에뮬레이션 방지를 수행할 수 있는 방식의 일례를 제공한다.

<94> 단계(300)는 하나 이상의 데이터 경계에 대해 시작 코드의 적어도 일부분을 복수회 비트 단위 시프트한다. 일실시예에서, 비트 단위 시프트되는 시작 코드의 부분은 시작 코드 프리픽스로 구성된다. 단계(302)는 적어도 하나의 데이터 경계에 대해 매 비트 시프트된 위치에서 발생하는 시작 코드 부분내의 하나 이상의 특징적 패턴을 식별한다. 단계(304)는 단계(302)에서 식별된 특징적 패턴(들)을 시작 코드 에뮬레이션 방지 데이터 삽입을 수행하기 위한 검색 패턴으로서 사용한다.

<95> 이것을 수행할 수 있는 방식의 단지 일례로서 도 4에 관련하여 다음을 고려하자.

<96> 값 0x01을 갖는 1 바이트가 후속되는 값 0x00("0x"가 16진법 수치값을 가리키는 C-언어 협약을 이용함으로써, 0x00은 제로와 동일한 16진법 바이트 값을 가리킨다)을 갖는 N(여기서 N=2) 바이트로 구성되는 시작 코드 프리

픽스를 고려하자. 예를 들면, 도 4에서, 예시적인 시작 코드 프리픽스가 나타나고 그 시작은 실선 꺾음 괄호로 도시된 데이터 경계와 일치한다. 따라서, 시작 코드 프리픽스의 제1 바이트는 첫번째 8개의 제로로 구성되고, 제2 바이트는 다음 8개의 제로로 구성되며, 이러한 방식으로 계속된다. 선행하는 첫번째 8개의 제로들은 선행하는 데이터의 임의의 미지 값을 지칭하는 "x"값의 바이트이다. 이제, 이 시작 코드 프리픽스 또는 일부가 지정된 데이터 경계들(도 4의 최상부 바로 아래에 표시된 비트 단위 시프트들 중 하나)에 대해 비트 단위 시프트될 때, 점선 꺾음 괄호로 도시된 특징적 패턴이 나타난다. 바이트의 시작에 대해 시작 코드 프리픽스의 각 시프트 위치에 의해 생성된 비트들의 패턴을 조사함으로써, 이 경우에, 이 시작 코드의 시프트된 카피를 포함하는 임의의 데이터 시퀀스에서 값 0x00을 갖는 적어도 N 바이트가 항상 존재할 것이라는 것을 판정할 수 있다. 따라서, 에뮬레이션된 비트-시프트된 시작 코드 프리픽스에 대한 특징적 패턴은 값 0x00을 갖는 N 바이트의 존재이다. 임의의 단일 정렬 위치에서의 이 특징적 패턴의 방지는 모든 정렬 위치에서의 시작 코드의 발생을 방지하므로, 이 특징적 패턴은 시작 코드 에뮬레이션 방지 데이터 삽입을 수행하기 위한 검색 패턴으로서 사용될 수 있다.

<97> 보다 구체적으로, "제1 예시적 방법"하에서 상술한 방법을 고려하자. 만일 데이터가 값 0x01 또는 Z가 후속되는 값 0x00의 N 바이트의 패턴을 포함하고 있었다면 에뮬레이션 방지 바이트 Z가 삽입된다. 따라서, 새로운 패턴은, 값 0x00의 N 바이트, 값 Z를 갖는 바이트, 이어서, 값 0x01 또는 Z 중 어느 하나를 갖는 바이트를 포함할 것이다(길이 N+1의 2개의 패턴들 중 어느 한 패턴이 조우될 때마다 1 바이트씩 입력 데이터 길이의 확장).

<98> 지금 상술된 조기 인터럽션 방법에 따라, 원하는 바는 바이트-정렬된 위치의 시작 코드 프리픽스만이 아니라, 시작 코드 프리픽스 또는 일부의 모든 시프트된 버젼들을 방지하는 것이다. 그러므로, 이 방법에 따라 그리고 이 특정 예와 관련하여, 소정 값 Z를 갖는 에뮬레이션 방지 바이트는 값 0x00를 값는 N 바이트의 임의의 시퀀스의 마지막 바이트 앞에 삽입되어, 값 0x00의 N개의 바이트의 임의의 시퀀스를 방지하여, 임의의 바이트-정렬 포지셔닝에 대한 시작 코드들의 모든 시프트된 카피를 방지한다. 이 예에서, 패턴이 체크되고, 에뮬레이션 방지 바이트들이 바이트-정렬된 위치에만 삽입된다. 예로서, 비트-시프트된 시작 코드 프리픽스를 포함하는 시퀀스의 마지막 바이트 앞에 삽입된 에뮬레이션 방지 바이트(도시된 것처럼 "z"의 값을 가진 8개의 비트로 나타냄)를 가진 N=2에 대한 시퀀스를 나타내는 도 5를 고려해보자. 비록 도 5에 명확히 나타나진 않지만, 도 5에서 예시를 위해, "x" 값으로 심볼화된 마지막 4개의 비트들중 적어도 하나가 제로라고 가정한다(그러나, 제로라면, 에뮬레이션 방지 프로세스는 "x" 값으로 심볼화된 마지막 4개의 비트들을 포함하는 바이트 바로 뒤의 위치에 다른 에뮬레이션 방지 바이트를 삽입하였을 것이다).

<99> 그러므로, 이 방법은 시작 코드 프리픽스의 모든 비트-시프트된 위치들을 인식하고, 임의의 비트-시프트된 시작 코드 프리픽스 에뮬레이션을 방지하는 방식으로 시퀀스를 인터럽션한다. 이는 인코더가 바이트 경계에 할당된 위치에서만 특징적 패턴을 검색하여 전체 바이트 단위로만 에뮬레이션 방지 데이터를 삽입하므로, 상기 인코더에서의 비트 단위 프로세싱을 요구하지 않고 이루어진다.

<100> 부가적으로, 인코더는 또한 에뮬레이션 방지 바이트를 삽입하여 페이로드 데이터가 이미 삽입된 에뮬레이션 방지 데이터를 갖고 있는 데이터를 에뮬레이팅하는 경우를 피할 수 있다. 예를 들어, 만약 인코더가 에뮬레이션 방지 바이트(예를 들어, Z)의 값을 갖는 바이트가 후속하는, 0x00의 값을 갖는 N-1개의 바이트를 관측하면, 그 후 인코더는 에뮬레이션 방지 바이트를 삽입한다.

<101> 대안적으로, 만약 데이터 페이로드가, Z와 동일한 바이트가 후속하고, 또한 0x00와 동일한 바이트도 후속하는, 0x00와 동일한 N-1개의 데이터 바이트의 시퀀스를 포함하면, 인코더는 상술한 목적을 위해 바이트를 삽입할 수 있다. 비록 인코더 및 디코더에 가변 길이 시퀀스 프로세싱이 요구되지만, 이는 삽입된 에뮬레이션 방지 데이터의 양을 감소할 수 있다.

<102> 그러므로, 조기 인터럽션 방법은 이 특정 예에 대해서, 상술된 특정 시작 코드 프리픽스 값들에 대하여 다음의 표로 요약될 수 있다.

<103>

| 치환할 패턴 | 치환 패턴 |
|---|---|
| N 바이트 = 0x00 | N-1 바이트 = 0x00, 바이트 = Z,바이트 = 0x00 |
| N-1 바이트 = 0x00, 바이트 = Z | N-1 바이트 = 0x00, 바이트 = Z, 바이트 = Z |

<104> [제1 조기 인터럽션 방법]

<105> 이제, 디코더의 관점으로부터 상황을 고려해보자. 디코더에서, 디코더가 0x00의 값을 갖는 N개의 바이트를 관

측할 때마다, 디코더는, 시작 코드가 아닌 비트-시프트된 위치를 포함하는 임의의 위치에서 이러한 상황이 발생하는 것을 인코더가 방지해왔기 때문에, 이것이 트루인 시작 코드 프리픽스이어야 함을 알고 있다. 트루 시작 코드의 검출을 위한 특징적 패턴은 바이트 경계에 대한 임의의 정렬 위치에서 시작할 때 발견될 수 있으므로, 디코더는 비트 단위 프로세싱을 사용하지 않고 이 시작 코드 검출을 수행할 수 있다. 일단 디코더가 어디에 트루 시작 코드가 있는지를 알면, 그것은 바이트 정렬을 안다. 이는 트루 시작 코드가 바이트 정렬된 위치에서 시작하기 때문이다. 그러므로, 이 특정 예에서, 디코더는 0x00이 아닌 다음 바이트가 시작 코드 프리픽스의 끝을 포함함을 알고 있다. 디코더는 그후 시작 코드 프리픽스를 끝내는 비트들(이 경우에는 "1"비트)의 패턴을 찾고, 그후 트루 페이로드 데이터를 찾을 수 있다.

<106> 디코더가 일단 시작 코드 프리픽스의 위치를 알아냄으로써 바이트 정렬을 알게 되면, 삽입된 에뮬레이션 방지 데이터의 위치를 용이하게 알 수 있음을 부가적으로 고려해보자. 디코더는, 이를, 에뮬레이션 방지 데이터가 바이트-정렬된 위치에 삽입된 사실에 의해 행한다. 디코더가 바이트 정렬을 알기 때문에, 디코더는 바이트 경계들에서 에뮬레이션 방지 데이터 패턴들을 찾을 수 있다. 즉, 디코더는 위의 표에서 설명된 바이트 경계에서 치환 패턴을 찾을 수 있다. 그후 디코더는 에뮬레이션 방지 데이터를 제거하고 그것의 디코딩 프로세스로 진행한다.

<107> 전술한 방법이 유리한 이유는, 에뮬레이션 방지 프로세싱을 수행하는 인코더가 모든 비트-시프트된 위치에서 에뮬레이션을 방지하더라도, 바이트 레벨에서 데이터 시퀀스를 시험하고 상기 바이트 레벨에서만 그 데이터를 조작함으로써, 본 프로세스에서 비트 단위 연산 수행의 복잡도를 줄이기 때문이다. 바이트에 대하여 수행되는 연산에 의해 트루 시작 코드의 존재를 검출하고 그후 바이트 정렬을 복구하고 그후 바이트에 대하여 수행되는 연산에 의해 에뮬레이션 방지 데이터의 제거를 수행할 수 있는 디코더에 대해서도 마찬가지다.

<108> 이 제1 조기 인터럽션 방법과 "제1 예시적 방법"으로 언급된 섹션에서 설명된 방법을 비교 및 대조하면서, 다음을 고려해보자. 조기 인터럽션 방법의 한가지 단점은, 이 방법에서는 보다 짧은 패턴(예컨대, N+1 바이트가 아니라 N 바이트)을 검색하므로 데이터를 더 자주 삽입하게 된다. 그러나 이에 대해서는 대처하는 방법이 있다. 구체적으로, N의 값을 증가시킬 수 있다. 즉, 3 바이트의 시작 코드 프리픽스를 사용하는 대신 4바이트의 시작 코드 프리픽스를 사용할 수 있다. 그러나, 이는 시스템에 삽입되어야 하는 다른 종류의 데이터의 양을 증가시키게 된다. 즉, 인코더가 시작 코드를 삽입할 때마다 보다 긴 코드를 삽입하는 것이 되어 추가된 데이터의 양을 증가시킨다는 점에서 비용이 보다 높게 된다. 그러나, 이러한 조기 인터럽션 방법의 한가지 장점은 디코더가 바이트 정렬을 복구할 수 있도록 한다는 것이다.

<109> 삭제

<110> 제2 실시예의 조기 인터럽션 방법은 패턴 매칭에 대한 검색을 행하고 상기한 제1 실시예의 구성에서 치환된 복수의 패턴에서 공통적인 대안적인 패턴으로의 패턴의 치환을 행하는 것이다. 이 대안적인 실시예의 일례를 아래의 표에 도시한다.

<111>

| 치환할 패턴 | 치환 패턴 |
|---|---|
| N-1 바이트 = 0x00 | N-1 바이트 = 0x00, 바이트 = Z |

<112> [제2 조기 인터럽션 방법]

<113> 제2 조기 인터럽션 방법에서, 단일 검색 패턴(0x00과 동등한 N-1 바이트)은, 제1 조기 인터럽션 방법을 나타낸 표에서 2개의 행에 도시된 2개의 패턴에서 공통이므로, 모든 시프트된 위치에 대한 시작 코드 에뮬레이션을 방지하는데 충분하다.

<114> 이러한 2번째 상황을 디코더의 관점에서 살펴보면, 트루 시작 코드 프리픽스의 검출에는 변화가 없으며, 이는 제1 조기 인터럽션 방법에서와 같이 0x00의 값을 갖는 N 바이트의 존재를 인식하는 것으로 구성된다. 또한, 제1 조기 인터럽션 방법에서와 같이, 일단 디코더가 트루 시작 코드의 위치를 알게 되면, 디코더는 트루 시작 코드가 바이트 정렬 위치에서 시작되기 때문에 바이트 정렬을 알고 있으며, 이후, 디코더는 치환 패턴을 검색함으로써 삽입된 에뮬레이션 방지 데이터의 위치를 찾아낼 수 있게 되며, 이후, 이 에뮬레이션 방지 데이터를 제거하고 자신의 디코딩 프로세스를 진행할 수 있게 된다.

<115> 상기한 제2 조기 인터럽션 방법은 제1 조기 인터럽션 방법에 비해 패턴 검색 프로세스에 의한 인식에 필요한 패턴의 수를 줄이고 있다. 그 결과, 보다 짧은 패턴과의 매칭을 이용하고 있기 때문에 삽입된 에뮬레이션 방지 데이터의 양이 증가하게 되지만 검색 프로세스의 복잡도가 줄어들게 된다.

<116> 제1 조기 인터럽션 방법에서와 마찬가지로 추가된 에뮬레이션 방지 데이터의 양은 N의 값을 증대시킴으로써 줄어들 수 있지만, 이는 실제 시작 코드 데이터의 양에 더해지게 된다.

<117> 여기서 주목할 점은, 제2 조기 인터럽션은 (복수의 패턴을 사용하는 "제1 예시적 방법" 및 제1 조기 인터럽션 방법과 대조적으로) 복수의 보다 긴 패턴과의 패턴 매칭을 이용하는 것이 아니라 단일의 짧은 패턴과의 패턴 매칭을 이용한다는 점에서, "제2 예시적 방법"으로 명명한 섹션에서 설명한 방법과 유사한 구성을 갖는다는 것이다. 특히, 이들 방법을 검토해 보면, 제2 조기 인터럽션 방법에 있어서 바이트 경계에 대해 임의로 시프트된 위치에서 길이 N+1의 시작 코드 프리픽스의 에뮬레이션을 방지하기 위한 인코딩 패턴 검색 및 치환 프로세스는, "제2 예시적 방법"에 있어서 바이트 경계에 배열된 위치에서 길이 N의 시작 코드 프리픽스의 에뮬레이션을 방지하기 위한 인코딩 패턴 검색 및 치환 프로세스와 정확하게 일치한다. 이러한 특성은 다음 섹션에서 이용된다.

<118> **복수의 길이 시작 코드를 이용하는 에뮬레이션 방지(Emulation Prevention Using Multiple Length Start Codes)**

<119> 상술한 바와 같은 조기 인터럽션 방법은 시작 위치가 데이터 경계와 일치하지 않는 임의의 비트 시프트된 위치에서 시작 코드 에뮬레이션을 방지하는 것과 관련하여 바이트 정렬이 손상되는 경우에 복구를 가능하게 하는 문제에 대한 해결책을 제공한다. 그러나, 이는 정렬된 위치에서 발생하는 시작 코드의 에뮬레이션만을 방지하기 위하여 필요한 것보다 자주 에뮬레이션 방지 데이터를 부가하는 희생을 치루어 행해지는 것이다. 이하에 상술되는 실시예에서, 조기 인터럽션 방법과 유사한 방식으로 배열 복구를 가능하게 하고, 또한 삽입된 에뮬레이션 방지 데이터 및 시작 코드 데이터의 총량을 감소시키는 접근 방법이 제공된다. 상세 해결책을 논의하기 전에, 다음의 견해를 고려한다.

<120> 시작 코드가 송신되는 많은 다른 종류의 데이터가 있다. 예를 들어, 비디오 데이터에서, 각 GOP, 각 픽처, 및 각 픽처의 각각의 개별적인 디코드가능 영역(즉, 슬라이스 시작 코드) 내에 대한 시작 코드를 송신하는 것이 일반적이다. 또한, I-, P-, 및 B-픽처와 같은 상이한 종류의 픽처가 있다. 따라서, 임의의 하나의 특정 시스템 내에 상이한 종류의 시작 코드가 있을 수 있다. 이들 시작 코드의 몇몇은 다른 것보다 훨씬 자주 발생할 수 있다. 예를 들어, 슬라이스 상의 시작 코드는 GOP 또는 시퀀스 상의 시작 코드보다 훨씬 자주 발생한다.

<121> 이와 같이, 시스템을 설계하는 경우에, 덜 자주 발생하는 시작 코드 보다 더 자주 발생하는 시작 코드를 짧게 만드는 것이 바람직할 수 있다. 이하의 실시예에 따라서, 시작 코드 프리픽스의 2개 이상의 상이한 길이들 사이에 구분이 행해진다. 더 긴 길이를 갖는 시작 코드 프리픽스에 대하여, 페이로드 콘텐트 내의 비트 시프트된 에뮬레이션이 방지된다. 이는 더 긴 길이를 갖는 시작 코드 프리픽스에 대하여 조기 인터럽션 방법을 이용함으로써 이루어진다. 더 짧은 길이를 갖는 시작 코드에 대하여, 이러한 시작 코드의 바이트 정렬된 에뮬레이션만이 방지된다. 이러한 시스템에서, 평균 코드 길이가 더 짧은 시작 코드의 길이에 가깝도록 더 긴 시작 코드가 더 짧은 시작 코드보다 훨씬 덜 자주 송신된다면 효율을 더 향상시킬 수 있다.

<122> 이러한 해결책을 이용하게 되면, 바이트 정렬을 보전하지 않는 환경에서 데이터가 사용되는 경우, 바이트 정렬은 더 긴 시작 코드에 대한 데이터 내에서 검색함으로써 복구될 수 있다. 일단 더 긴 시작 코드의 위치가 찾아지고 바이트 정렬이 복구되면, 이를 따르는 더 짧은 시작 코드의 위치가 또한 바이트 지향의 검색만으로 찾아질 수 있다.

<123> 이를 구현할 수 있는 방식의 일례로서, 다음을 고려한다. 이 예에서, 더 긴 시작 코드 프리픽스의 길이는 $N_L$+1로 표시하고, 더 짧은 시작 코드 프리픽스의 길이는 $N_S$+1로 표시한다.

<124> 본 실시예에서 사용되는 시작 코드 프리픽스의 특정 구조로, 바이트 정렬을 보전하지 않는 시스템에서도, 더 긴 비트 시프트된 시작 코드는 0x00 값을 갖는 $N_L$ 바이트에 대한 바이트 지향의 검색을 이용하여 데이터 스트림내에서 검출될 수 있다. 이들 상세 시작 코드 값은 바이트 지향의 검출을 쉽게 가능하게 한다.

<125> 특히, 바람직한 예시의 방법은 $N_L$=$N_S$+1으로 설정된다. 그리고 하나의 에뮬레이션 방지 프로세스는 이하의 표에 나타내는 바와 같이 요약될 수 있다.

<126>

| 치환할 패턴 | 치환 패턴 |
|---|---|
| $N_S$ bytes=0x00, byte=0x01 | $N_S$ bytes=0x00, byte=Z, byte=0x01 |
| $N_S$ bytes=0x00, byte=0x00 | $N_S$ bytes=0x00, byte=Z, byte=0x00 |
| $N_S$ bytes=0x00, byte=Z | $N_S$ bytes=0x00, 2 bytes=Z |

<127> [복수의 길이 시작 코드를 이용하는 제1 에뮬레이션 방지 프로세스]

<128> 상기 표에서, 제1 행은 치환될 제1 패턴과, 바이트 단위로 정렬된 위치들에서 더 짧은 시작 코드의 에뮬레이션을 방지하는 치환 패턴을 설명한다. 제2 행은 제2 패턴과, 바이트 단위 및 바이트 단위외의 단위로 정렬된 위치들에서 더 긴 시작 코드의 에뮬레이션을 방지하는 치환 패턴이다. 제3 행은 제3 패턴과, 에뮬레이션 방지 처리에 의해 삽입된 Z 값의 바이트들과 데이터 페이로드내의 0x00 값의 $N_s$ 바이트들에 자연스럽게 이어지게 되는 Z값의 다른 바이트들 사이를 구분하는 방법을 제공하는 치환 패턴을 설명한다. 이는, 데이터 페이로드의 끝 위치 또는 다음 시작 코드의 위치를 식별하는 디코더의 능력에 악역향을 주지 않고도, 인코더가 임의의 시작 코드 앞에 임의의 개수의 제로 바이트들을 삽입할 수 있게 한다는 점에 유의한다. 실제로, 이러한 제로-삽입은, 인코더가 수 개의 더 짧은 시작 코드들 앞에 수 개의 제로 바이트들 삽입하도록 결정한다면, 바이트 정렬을 더욱 더 빨리 복구하는 데 일조한다.

<129> 대안적으로, 상기 프로세스의 제3 행은, Z값의 한 바이트가 후속하고 또한 0x00값의 입력 데이터 바이트가 후속하는 0x00 값의 입력 데이터의 Ns 바이트들의 경우에만 Z값의 추가 바이트가 삽입되도록 변경될 수 있다. 이는, 인코더 및 디코더 내에서 가변 길이의 시퀀스 처리를 필요로 하게 되는 것일 수 있지만, 삽입된 에뮬레이션 방지 데이터의 양을 줄이게 된다.

<130> 이러한 제1 방법은, 에뮬레이션 방지에 요구되는 데이터의 양을, 더 짧은 시작 코드 값들의 비정렬 에뮬레이션을 방지하는데 요구되는 양으로부터 약 171의 인수로 감소되는 양까지 감소시킨다 (약 $\frac{2}{256^{Ns}}$ 로부터 약 $\frac{3}{256^{Ns+1}}$ 로 에뮬레이션 방지 프로세스 데이터 확장 인수(expansion factor)를 감소시킴).

<131> 도 6은 서로 다른 길이들을 갖는 시작 코드들 또는 시작 코드 프리픽스들을 포함할 수 있는 데이터와 연계하여, 에뮬레이션 방지 데이터를 사용하는 일 실시예에 따른 방법의 단계들을 설명하는 흐름도이다. 본 방법은 하드웨어, 소프트웨어, 펌웨어, 또는 이들의 조합의 어떠한 적합한 형태로도 구현될 수 있다. 도시된 실시예에 있어서, 본 방법은 적당하게 구성된 인코더에 의해 구현된다.

<132> 단계(600)에서는, 서로 다른 길이들을 갖는 시작 코드들과 연관된 패턴들을 검색한다. 임의의 적절한 수의 서로 다른 길이들이 사용될 수 있다. 전술한 예에서, 두 개의 별도의 길이들을 갖는 시작 코드들이 사용된다. 제1 패턴 타입을 발견함에 응답하여, 단계(602)에서는, 에뮬레이션 방지 데이터를 삽입하여, 제1 길이들을 갖는 시작 코드들의 에뮬레이션을 방지한다. 여기서, 제1 패턴 타입은 더 짧은 시작 코드와 연관된 것일 수 있으며, 상기 표의 제1 행의 "치환할 패턴" 열에서 나타나는 패턴에 의해 예시될 수 있다. 또한, 이러한 특정 패턴 및 이러한 특정 예에 대한 해당 에뮬레이션 방지 데이터는 상기 표의 제1 행의 "치환 패턴" 열에서 나타나는 패턴에 의해 예시될 수 있다.

<133> 제2 패턴 타입을 발견함에 응답하여, 단계(604)에서는 에뮬레이션 방지 데이터를 삽입하여 제2 길이들을 갖는 시작 코드들의 에뮬레이션을 방지한다. 여기서, 제2 패턴형태는 더 긴 시작 코드와 연관된 것일 수 있으며, 상기 표의 제2 행의 "치환할 패턴" 열에서 나타나는 패턴으로 예시될 수 있다. 또한, 이러한 특정 패턴 및 이러한 특정 예에 대한 해당 에뮬레이션 방지 데이터는 상기 표의 제2 행의 "치환 패턴" 열에서 나타나는 패턴에 의해 예시될 수 있다.

<134> 이 방법은 또한 에뮬레이션 방지 프로세스에 의해 삽입된 값 Z의 바이트들과 데이터 페이로드에 값 0x00의 $N_s$ 바이트들을 자연스럽게 따르는 값 Z의 다른 바이트들을 구별하는 방법을 유익하게 제공할 수 있지만, 순서도가 이것을 특별히 설명하고 있지는 않다.

<135> 이제 디코더 쪽에 무엇이 발생할 수 있는지 고려한자. 디코더 쪽에, 바이트 정렬이 손상되면, 긴 시작 코드 프

리픽스를 찾는 디코더에 의해 복구가 실행될 수 있다. 인코더에서 실행하는 에뮬레이션 방지 프로세싱 때문에, 긴 시작 코드 프리픽스들의 에뮬레이션은 모든 비트-시프트된 위치들에서 방지됨을 주목한다. 그러므로, 일단 디코더가 트루인 긴 시작 코드 프리픽스에 해당하는 패턴을 발견하면, 디코더는 트루 시작 코드의 위치를 파악한 것이다. 트루인 긴 시작 코드의 위치를 파악함으로써, 바이트 정렬은 복구되는 것으로 이해된다. 즉, 그 트루인 긴 시작 코드들이 바이트 정렬된 위치들에 있기 때문에, 긴 시작 코드를 발견하는 것은 바이트 정렬을 재확립한다. 일단 디코더가 바이트 정렬을 복구하면, 디코더는 적절한 바이트 정렬된 위치들에서만 더 짧은 시작 코드들을 찾을 수 있다. 이 처리와 관련해서, 디코더는 상술된 방식으로 에뮬레이션 방지 데이터를 제거할 수 있다.

<136> 상술된 처리는 바이트 정렬이 손상된 이벤트에서 바이트 정렬 복구를 위해 메카니즘이 제공된다는 점에서 유익하다. 추가로, 상이한 길이의 시작 코드들 또는 시작 코드 프리픽스들이 사용된다는 사실로 인해, 가장 자주 사용되는 시작 코드 프리픽스들이 그보다 덜 자주 사용되는 시작 코드 프리픽스들보다 짧게 만들어 졌기 때문에, 상술된 방법은 추가 정보로 데이타 스트림에 부담을 주는 것을 피한다.

<137> 상술된 제2 조기 인터럽션 방법의 경우와 같이, 패턴들의 길이를 짧게 하여 인코더와 디코더에 의해 찾을 필요가 있는 패턴들의 수를 감소시키는 것이 가능할 것이다. 이 경우, 상이한 길이들을 갖는 시작 코드들의 에뮬레이션 방지를 위해 테스트될 패턴들은 프로세스의 특정 단계들의 병합을 하도록 해서 동일하게 될 것이다. 예를 들어, 단계(602)의 제1 패턴 타입의 검출을 위해 사용되는 패턴(들)과 단계(604)의 제2 패턴 타입의 검출을 위해 사용되는 패턴(들)이 동일 패턴이면, 단계(602)와 단계(604)는 병합될 수 있다. 단계들을 병합하는 이러한 능력을 설명하는 복수의 길이 시작 코드들을 사용하는 제2 에뮬레이션 방지 프로세스는 아래 표에 도시된다. 이 프로세스는 길이 $N_L$+1(길이 $N_S$+2임)의 비정렬된 시작 코드들의 에뮬레이션 방지를 위한 제2 조기 인터럽션 방법에 해당하고, 길이 $N_S$+1의 바이트 정렬된 시작 코드들의 에뮬레이션 방지를 위한 제2 방법의 예에 또한 해당한다. 그러므로, 단일 패턴의 검색 및 치환은 두 가지 타입들의 시작 코드 에뮬레이션을 방지하기 위해 충분할 수 있다.

<138>

| 치환할 패턴 | 치환 패턴 |
|---|---|
| $N_S$바이트들 = 0x00 | $N_S$바이트들 = 0x00, 바이트 = Z |

<139> [복수의 길이 시작 코드들을 사용한 제2 에뮬레이션 방지 프로세스]

<140> 복수의 길이 시작 코드들을 사용하는 이 제2 에뮬레이션 방지 프로세스는, 검색되고 치환될 필요가 있는 패턴들의 숫자를 감소시킴으로써, 패턴 검색 및 치환 프로세스를 단순화시킨다. 그러나, 더 짧은 패턴의 매칭을 사용하기 때문에, 삽입되는 에뮬레이션 방지 데이타의 양은 증가된다. 삽입되는 에뮬레이션 방지 데이타의 양은 상술된 복수의 길이 시작 코드들을 사용하는 제1 에뮬레이션 방지 프로세스에 비하여 약 16/3 = 5.33 인수만큼 증가된다 (에뮬레이션 방지 프로세스 데이타 확장 크기를 약 $\frac{3}{256^{N_S+1}}$ 로부터 약 $\frac{1}{256^{N_S}}$으로 증가시킴).

<141> 개선책들

<142> 얼마나 쉽게 특정 패턴들을 검출하는가에 관계되고 처리 효율을 향상시키기 위해 채택될 수 있는 몇몇 개선된 실시예들이 있다. 이하에서 예시적 목적으로, 시작 코드 프리픽스가 값 W=0x00을 갖는 특정 개수의 바이트들과, 뒤이어 X=0x01을 갖는 바이트를 포함하는 것으로 가정한다. 그러나, 이하에 설명된 원리들이 이 예들에서 주어진 것들과 다른 값들을 갖는 그밖의 시작 코드 프리픽스와 연계하여 활용될 수 있다는 점을 알아야 한다.

<143> 제1 개선책은 다음과 같다. 앞서 주어진 표기(즉, W, X, 및 Z)를 사용하여, Z 및 X, 또는 Z 및 W가 단지 일 비트만큼만 달라지도록 선택될 수 있다. 이는 인코더 및 디코더에서 데이터 스트림에 있는 두개의 값들 중에서 어떠한 하나의 값에 대해서도 용이한 검출을 가능하게 할 수 있다. Z 및 X가 단지 일 비트 값만큼 다르도록 하는 것의 유용성을 예시하기 위해서, 데이터 수신기에 의해 행해지는 처리를 고려해 보자. 디코더가 시작 코드들을 검출하고 에뮬레이션 방지 데이터를 제거하기 위해 데이터 스트림을 검색하고 있을 때, 값 W=0x00을 갖는 N개 이상의 바이트들을 검출한 후에, 입력 데이터의 다음 바이트가 Z 또는 X와 동등한 지의 여부에 대한 테스트를 수행할 수 있다. 만약 Z 및 X가 일 비트보다 많은 비트 값만큼 차이가 난다면, 처리 요구 사항들을 예시하기 위해 의사 C 프로그래밍 컨스트럭트(construct)를 사용하면, 이 테스트는 다음과 같이 실행될 수 있다.

```
if(in_byte != Z && in_byte != X) {
  etc.
}else{
  etc.
}
```

<144>

<145> 그러나, 만약 Z 및 X가 단지 일 비트값만큼 다르다면, 이후 의사 C 프로그래밍 컨스트럭트를 사용하고 X는 0x01이고 Z는 0x81이라고 가정하면, 이 테스트는 다음과 같이 실행될 수 있다.

```
if((in_byte & 0x7F) == 0x01) {
  etc.
}else{
  etc.
}
```

<146>

<147> 제2 개선책은, 앞서 설명한 에뮬레이션 방지 프로세스들 중의 몇몇에서, 세개의 패턴들을 구별하고 있다는 것을 인식한 것이다. 만약 이들 패턴들이 서로 최소 수의 비트만큼만 다르기를 원한다면, 이는 세개의 패턴들을 구별하기 위해 두개의 비트들이 필요하다. 그러나, 두개의 비트는 네개의 대상들 또는 패턴들을 식별하는 데에 충분할 것이다. 여기서, 이들 사이에서 구별하는 데에 필요한 단지 세개의 패턴들이 있으며, 그에 따라 시스템이 보통의 데이터에 대해서 사용하지 않는 넷째 패턴이 있는 것이 유리할 수 있다. 이제, 만약 인코더가 2 또는 3 값에 동등한 에뮬레이션 방지 바이트를 설정하도록 구성되어 있다면, 이는 이 시스템이 각각의 패턴 중에서 두개의 최하위 비트들을 단순히 참조함으로써 세개의 상기 패턴들을 구별할 수 있다는 것을 의미한다. 예를 들어, 만약 W가 0x00에 동등하고, X가 0x01에 동등하고, Z가 0x02에 동등한 경우, 만약 두개의 최하위 비트들이 0이라면, 이는 초기 바이트 값 W의 반복이다; 만약 두개의 최하위 비트들이 1 이라면, 이는 최종 바이트값 X이고; 만약 두개의 최하위 비트들이 2 라면, 이는 에뮬레이션 방지 값 Z이다. 그밖의 값, 즉 값 3은 보통의 값 이외의 특별값으로서 취급될 수 있다. 따라서, 인코더에서, 세개의 패턴들을 참조하는 대신에 네개의 패턴들을 참조할 수 있고, 이들이 2개의 최하위 비트들을 제외하고는 동일하기 때문 이들을 매우 쉽게 발견할 수 있다. 이는 인코더 및 디코더에서의 처리를 가속화하고 단순화할 수 있다.

<148> 따라서, 값들의 세개의 시퀀스를 에뮬레이션 방지 프로세스에 의해 치환하는 대신에, 본 시스템은 값들의 네개의 시퀀스들을 치환할 수 있다. 이런 네개의 시퀀스들은 단지 두개의 비트의 값에 의해 서로 구별될 수 있다. 바꾸어 말하면, 입력 데이터가 이들 특별 값들(우선, W=0x00, X=0x01, 및 Z=0x81임을 가정함) 중 하나를 가지는지 여부를 검출하기 위한 인코더 또는 디코더 검출 프로세스는 간단히,

```
if((in_byte & 0x7E) {
  etc.
}else{
  etc.
}
```

<149>

<150> 일 수 있다.

<151> 네번째 특별 바이트 값(이 경우 0x80)은 시작 코드의 또다른 타입과 같은 다양한 목적으로 사용될 수 있거나 또는 시스템에 의해 비사용되어 남겨질 수 있다.

<152> 세번째 개선책으로서, 다음을 고려한다. 데이터 페이로드의 끝 다음의 공간을 값 W를 갖는 바이트들로 패딩하는 것은 패딩 값들 사이에 에뮬레이션 방지 바이트들을 삽입할 필요없이 허용될 수 있는데, 이는 W의 시퀀스 이후 도착하는 다음 값이 X이고, 이것은 트루 시작 코드 프리픽스를 나타내기 때문이다. 이것은 임의의 정수개의 바이트의 "스터핑"이, 특별 경계(이를테면 32비트 DWORD 경계들 또는 고정 길이 패킷 경계들)에 대한 정렬과 같은 소정의 목적을 위해서 전송 채널을 채우게 하거나, 또는 어떤 정보도 전송되기를 희망하지 않을 경우 단순히 일정 비트 레이트 채널을 채우게 할 수 있다.

<153> 패딩의 또 다른 이점은, 시작 코드의 값들이 패딩을 허용하도록 선택된다면, 인코더가 바이트 정렬 손상의 경우에 바이트 경계들에게 재동기화를 제공하도록 패딩할 수 있다는 점이다. 따라서, 인코더는 얼마나 자주 하나

이상의 엑스트라 패딩 바이트를 전송하는지에 대한 재량권(discretion)을 가질 수 있어 디코더에 의한 바이트 정렬의 복구를 허용할 수 있다. 소정의 인코더들은 매 시작 코드에 앞서 엑스트라 패딩을 추가함으로써 매우 자주 바이트 정렬 복구를 가능하게 하고, 다른 인코더들은 덜 자주 그렇게 하도록 선택할 수 있다.

<154> Z의 몇몇 특별히 관심있는 값들은, W=0x00 및 X=0x01인 경우에, Z=0x80, 0x02, 0x81 및 0x03이다.

<155> 포맷 명세는 바이트 정렬 복구를 위해 추가된 엑스트라 패딩 바이트들을 위한 최소 주파수들을 특정할 수 있거나, 엑스트라 패딩을 데이터 스트림내의 특정 타입의 시작 코드와 연관시킬 수 있다(이를테면, MPEG-2 또는 H.263 비디오 내에 슬라이스(slice) 또는 GOB 시작 코드로 공지된 낮은 레벨 시작 코드 전에 패딩을 요구하지 않고 압축 비디오 데이터 스트림 내의 각 픽처의 시작을 패딩함).

<156> **예시적인 컴퓨팅 환경**

<157> 도 7은 후술하는 시스템 및 관련 방법들이 구현될 수 있는 적합한 컴퓨팅 환경(700)의 예를 도시한다.

<158> 컴퓨팅 환경(700)은 적합한 컴퓨팅 환경의 일 예일 뿐이며, 상술한 인코딩/디코딩 시스템의 사용 또는 기능의 범위에 대해 어떠한 제한을 두려는 의도가 아님을 유의하여야 한다. 컴퓨팅 환경(700)은 예시적인 컴퓨팅 환경(700) 내에 설명된 컴포넌트들의 임의의 하나 또는 그들의 조합에 관하여 어떠한 종속성을 가지거나 그들에 대한 요구 조건을 가지는 것으로 해석되어서는 안 된다.

<159> 다양한 상술한 실시예들은 다른 여러 가지 일반적인 목적 또는 특별한 목적의 컴퓨팅 시스템 환경들 또는 구성들과 동작할 수 있다. 매체 처리 시스템과 함께 사용하는데 적합할 수 있는 주지의 컴퓨팅 시스템들, 환경들 및/또는 구성들의 예들로는, 개인용 컴퓨터들, 서버 컴퓨터들, 씬 클라이언트들(thin clients), 씩 클라이언트들(thick clients), 핸드 헬드 또는 랩 톱 장치들, 멀티프로세서 시스템들, 마이크로프로세서 기반 시스템들, 셋톱 박스들, 프로그램 가능한 가전 제품들, 네트워크 PC들, 미니 컴퓨터들, 메인 프레임 컴퓨터들, 상기 시스템들 및 장치들의 임의의 것을 포함하는 분산형 컴퓨팅 환경들 등을 들 수 있으나, 이들에 제한되는 것은 아니다.

<160> 특정 구현 예에서, 시스템 및 관련 방법들은 컴퓨터에 의해 실행되는, 프로그램 모듈들 등의 컴퓨터 실행 가능한 명령어들의 일반적인 콘텍스트로 잘 설명될 수 있다. 일반적으로, 프로그램 모듈들은 특별한 임무를 수행하거나 특별한 추상적인 데이터 형태들을 구현하는 루틴들, 프로그램들, 객체들, 컴포넌트들, 데이터 구조들 등을 포함한다. 본 실시예들은 분산형 컴퓨팅 환경들에서 실행될 수 있으며, 이 환경들에서 임무들은 통신 네트워크를 통해 연결되어 있는 원격 처리 장치들에 의해 수행된다. 분산형 컴퓨팅 환경에서, 프로그램 모듈들은 메모리 저장 장치들을 포함하고 있는 로컬 및 원격 컴퓨터 저장 매체 내에 위치될 수 있다. 상술한 컴퓨팅 시스템의 컴포넌트들은 상술한 바와 같이 기능하는 인코더 및 디코더를 구현하는데 사용될 수 있다.

<161> 도 7의 상술한 예시적인 실시예들에 따르면, 컴퓨팅 환경(700)은 1 이상의 프로세서들 또는 처리 장치들(702), 시스템 메모리(704), 및 시스템 메모리(704)를 포함하는 다양한 시스템 컴포넌트들을 프로세서(702)에 결합시키는 버스(706)를 포함하는 것으로 도시되어 있다.

<162> 버스(706)는 메모리 버스 또는 메모리 제어기, 주변 버스, 가속화된 그래픽 포트, 및 다양한 버스 아키텍처의 임의의 것을 사용하는 프로세서 또는 로컬 버스를 포함하는 다양한 종류의 버스 구조들의 하나 이상을 나타내는 것이다. 예를 들자면, 이러한 아키텍처로는 ISA(Industry Standard Architecture) 버스, MCA(Micro Channel Architecture) 버스, EISA(Enhanced ISA) 버스, VESA(Video Electronics Standards Association) 로컬 버스, 및 메자닌(Mezzanine) 버스로도 알려진 PCI(Peripheral Component Interconnects) 버스가 있지만, 이들에 제한되는 것은 아니다.

<163> 컴퓨팅 환경(700)은 주로 다양한 컴퓨터 판독 가능 매체를 포함한다. 이러한 매체는 컴퓨팅 환경(700)에 의해 국부적으로 및/또는 원격적으로 액세스 가능한 임의의 이용 가능한 매체일 수 있으며, 휘발성 및 비휘발성 매체, 분리식 및 비분리식 불가능 매체도 포함한다.

<164> 도 7에서, 시스템 메모리(704)는 RAM(random access memory)(710)과 같은 휘발성 형태 및/또는 ROM(read only memory)(708)과 같은 비휘발성 형태의 컴퓨터 판독 가능 매체를 포함한다. 기동 시와 같이 컴퓨팅 환경(700) 내의 요소들 간에 정보를 전달하기 위해 사용되는 기본 루틴들을 포함하는 BIOS(basic input/output system)(712)는 ROM(708)에 저장된다. RAM(710)은 주로 즉시 액세스 가능하고 그리고/또는 처리 장치(702)에 의해 현재 동작되고 있는 데이터 및/또는 프로그램을 포함한다.

<165> 컴퓨팅 환경(700)은 다른 분리식/비분리식, 휘발성/비휘발성 컴퓨터 저장 매체를 더 포함할 수 있다. 예시를

하자면, 도 7은 비분리식, 비휘발성 자기 매체(도시하지 않음, 주로 "하드 드라이브"라 칭함)로부터 판독하거나 그곳에 기록하기 위한 하드 디스크 드라이브(728), 비분리식, 비휘발성 자기 디스크(732)(예컨대, "플로피 디스크")로부터 판독하거나 그곳에 기록하기 위한 자기 디스크 드라이브(730), CD-ROM, DVD-ROM 또는 다른 광 매체와 같이 비분리식, 비휘발성 광 디스크(736)로부터 판독하거나 그곳에 기록하기 위한 광 디스크 드라이브(734)를 나타내고 있다. 하드 디스크 드라이브(728), 자기 디스크 드라이브(730), 광 디스크 드라이브(734)는 각각 하나 이상의 인터페이스들(726)에 의해 버스(706)에 접속된다.

<166> 드라이브들 및 그들의 연관된 컴퓨터 판독 가능 매체는 컴퓨터 판독 가능 명령어들, 데이터 구조들, 프로그램 모듈들, 및 컴퓨팅 환경(700)을 위한 다른 데이터들의 비휘발성 저장소를 제공한다. 여기서 설명한 예시적인 환경은 하드 디스크(728), 분리식 자기 디스크(732) 및 분리식 광 디스크(736)를 사용하지만, 당업자라면 예시적인 동작 환경에서 자기 카세트들, 플래시 메모리 카드들, 디지털 비디오 디스크들, RAM들, ROM들 등의 컴퓨터 액세스 가능한 데이터를 저장할 수 있는 다른 형태의 컴퓨터 판독 가능 매체도 사용될 수 있음을 알아야 한다.

<167> 다수의 프로그램 모듈들은 운영 체제(714), 하나 이상의 애플리케이션 프로그램들(716)(예컨대, 멀티미디어 애플리케이션 프로그램(724)), 기타 프로그램 모듈(718) 및 프로그램 데이터(720)와 함께, 하드 디스크(728), 자기 디스크(732), 광 디스크(736), ROM(708) 또는 RAM(710)에 저장될 수 있지만, 이들에 제한되는 것은 아니다. 사용자는 키보드(738) 및 포인팅 장치(740)("마우스" 등) 등의 입력 장치들을 통해 컴퓨팅 환경(700)으로 커맨드들이나 정보를 입력할 수 있다. 다른 입력 장치들로는 오디오/비디오 입력 장치(753), 마이크, 조이스틱, 게임 패드, 위성 안테나, 직렬 포트, 스캐너 등(도시하지 않음)을 예로 들 수 있다. 이러한 입력 장치들은 버스(706)에 결합되거나 병렬 포트, 게임 포트 또는 USB(universal serial bus) 등의 기타 인터페이스 및 버스 구조들에 의해 연결될 수 있는 입력 인터페이스(742)를 통해 처리 장치(702)에 접속된다.

<168> 모니터(756) 또는 다른 종류의 디스플레이 장치도 비디오 어댑터 또는 비디오/그래픽 카드(744) 등의 인터페이스를 통해 버스(706)에 접속된다. 개인용 컴퓨터들은, 모니터 외에도, 통상적으로 출력 주변 인터페이스(746)를 통해 접속될 수 있는 스피커들 및 프린터들 등의 기타 주변 출력 장치들(도시하지 않음)을 포함한다.

<169> 컴퓨팅 환경(700)은 원격 컴퓨터(750) 등의 하나 이상의 원격 컴퓨터들에 대한 논리적 접속들을 사용하여 네트워크 환경에서 작동할 수 있다. 원격 컴퓨터(750)는 여기서 컴퓨터에 대해 설명한 요소들 및 특징들의 다수 또는 모두를 포함할 수 있다.

<170> 도 7에 나타낸 바와 같이, 컴퓨팅 환경(700)은 LAN(local area network)(751) 및 일반적인 WAN(wide area network)(752)를 통해 원격 장치들(예컨대, 원격 컴퓨터(750))에 통신 가능하게 결합된다. 이러한 네트워크 환경들은 사무실, 기업형 컴퓨터 네트워크들, 인트라넷들 및 인터넷 내에서 흔히 사용되는 것이다.

<171> 컴퓨팅 환경(700)은 LAN 네트워크 환경에서 사용될 때, 적합한 네트워크 인터페이스 또는 어댑터(748)를 통해 LAN(751)에 접속된다. 컴퓨팅 환경(700)이 WAN 네트워크 환경에 사용될 때는, 주로 WAN(752)을 통한 통신을 설정하기 위한 모뎀(754) 및 기타 수단을 포함한다. 내부 또는 외부에 설치될 수 있는 모뎀(754)은 사용자 입력 인터페이스(742) 또는 기타 적합한 기구를 통해 시스템 버스(706)에 접속될 수 있다.

<172> 네트워크 환경에서, 컴퓨팅 환경(700) 또는 그 부분에 관한 프로그램 모듈들은 원격 메모리 저장 장치에 저장될 수 있다. 도 7은 원격 애플리케이션 프로그램들(716)이 원격 컴퓨터(750)의 메모리 장치에 상주하는 것으로 예시하고 있지만, 이들에 제한되는 것은 아니다. 도시하고 설명한 네트워크 접속들은 예시적인 것이고, 컴퓨터들 간의 통신 링크를 설정하기 위한 다른 수단도 사용할 수 있다는 것에 대해 유의하여야 할 것이다.

## *산업상 이용 가능성*

<173> 상술한 방법들 및 시스템들의 일부는 비트 레벨 외의 레벨에서 시작 코드 에뮬레이션 방지(start code emulation prevention)를 제공할 수 있다. 이것은 처리를 단순화할 수 있기 때문에 이점이 있다. 또한, 일부 실시예들은 정수개의 바이트들이 필요에 따라 송신될 수 있도록 하는 데이터 스터핑(data stuffing)을 위한 직접적인 방법을 제공한다. 또한, 여러 실시예들은 반드시 데이터 경계들과 일치하지는 않는 비트 시프트된 위치들에서 시작 코드 에뮬레이션 방지를 제공할 수 있다. 다른 실시예들은 다양한 처리 효율을 달성하면서도, 손상되는 이벤트에서 데이터 경계 정렬을 복구할 수 있는 메커니즘을 제공할 수 있다. 당업자라면 기타의 이점들도 이해할 수 있을 것이다.

<174> 본 발명을 구조적 특징들 및/또는 방법론적인 단계들에 구속된 언어로 표현하였지만, 본 발명은 상술한 구체적인 특징들이나 단계들에 제한되는 것으로 해석되는 것은 아니며 단지 첨부한 청구의 범위에 의해서만 제한되는

것이다. 즉, 구체적인 특징들 및 단계들은 본 발명을 구현하는 바람직한 형태들로서 개시된 것일 뿐이다.

## 도면의 간단한 설명

<20> 도 1은 일 실시예에 따른 방법에서의 단계들을 설명하는 흐름도.

<21> 도 2는 일 실시예에 따른 방법에서의 단계들을 설명하는 흐름도.

<22> 도 3은 일 실시예에 따른 방법에서의 단계들을 설명하는 흐름도.

<23> 도 4는 일 실시예에 따른 처리 방법의 일 양태를 도시하는 도면.

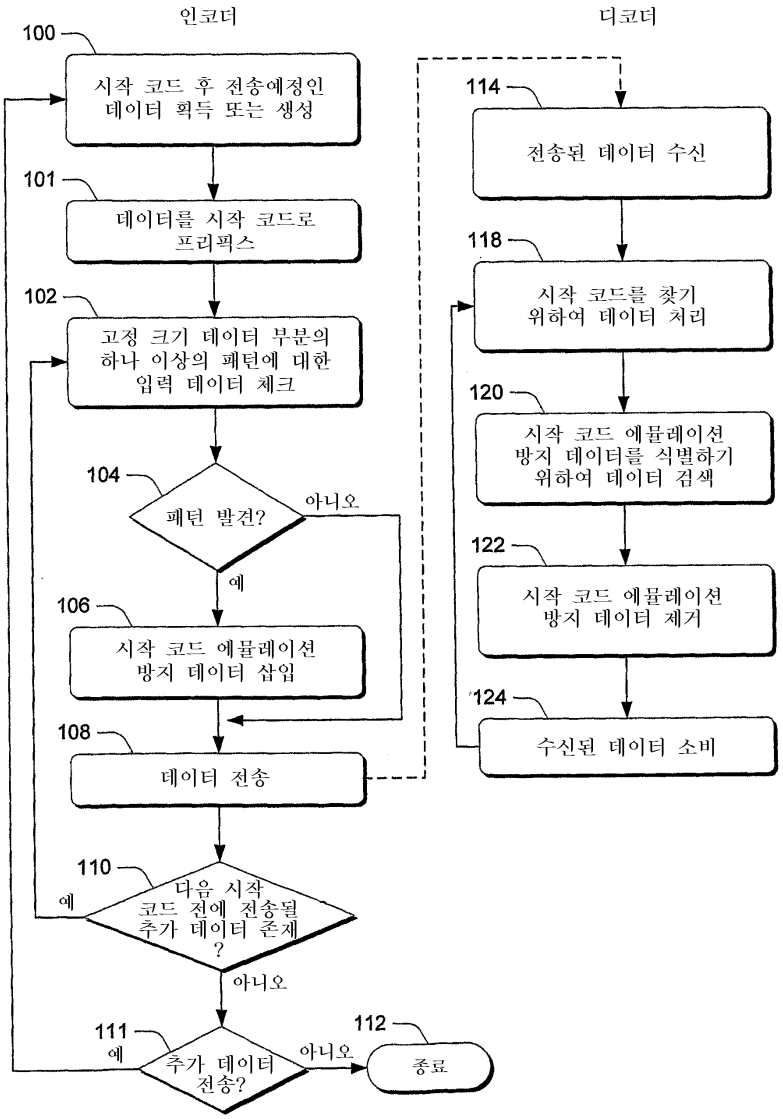
<24> 도 5는 일 실시예에 따른 처리 방법의 일 양태를 도시하는 도면.

<25> 도 6은 일 실시예에 따른 방법에서의 단계들을 설명하는 흐름도.

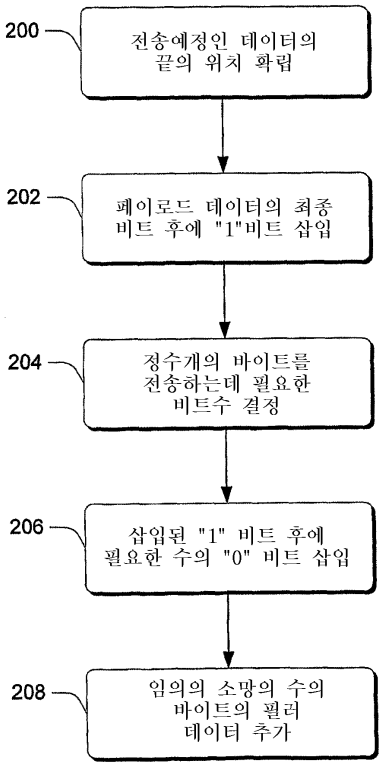
<26> 도 7은 하나 이상의 실시예들과 연관하여 구현될 수 있는 컴퓨팅 환경의 고 레벨 도.

## 도면

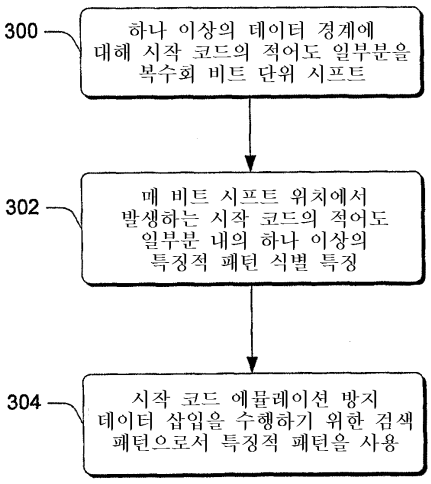
### 도면1



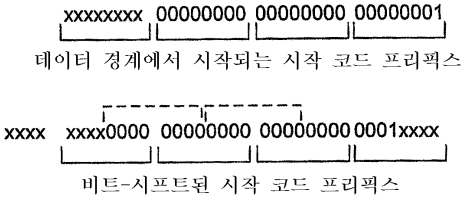
도면2


200
전송예정인 데이터의 끝의 위치 확립
202
페이로드 데이터의 최종 비트 후에 "1"비트 삽입
204
정수개의 바이트를 전송하는데 필요한 비트수 결정
206
삽입된 "1" 비트 후에 필요한 수의 "0" 비트 삽입
208
임의의 소망의 수의 바이트의 필러 데이터 추가

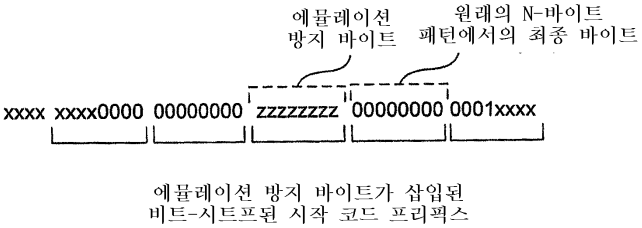
도면3


300
하나 이상의 데이터 경계에 대해 시작 코드의 적어도 일부분을 복수회 비트 단위 시프트
302
매 비트 시프트 위치에서 발생하는 시작 코드의 적어도 일부분 내의 하나 이상의 특징적 패턴 식별 특징
304
시작 코드 에뮬레이션 방지 데이터 삽입을 수행하기 위한 검색 패턴으로서 특징적 패턴을 사용

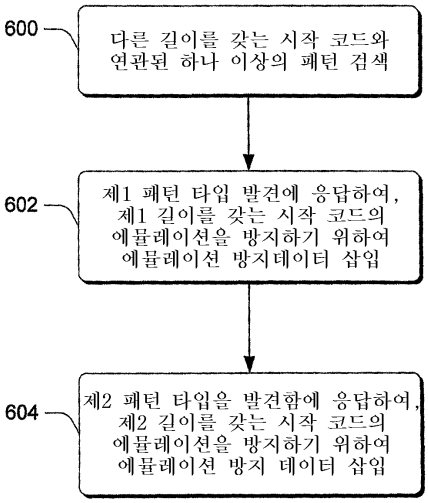
도면4


xxxxxxxx 00000000 00000000 00000001
데이터 경계에서 시작되는 시작 코드 프리픽스
xxxx xxxx0000 00000000 00000000 0001xxxx
비트-시프트된 시작 코드 프리픽스

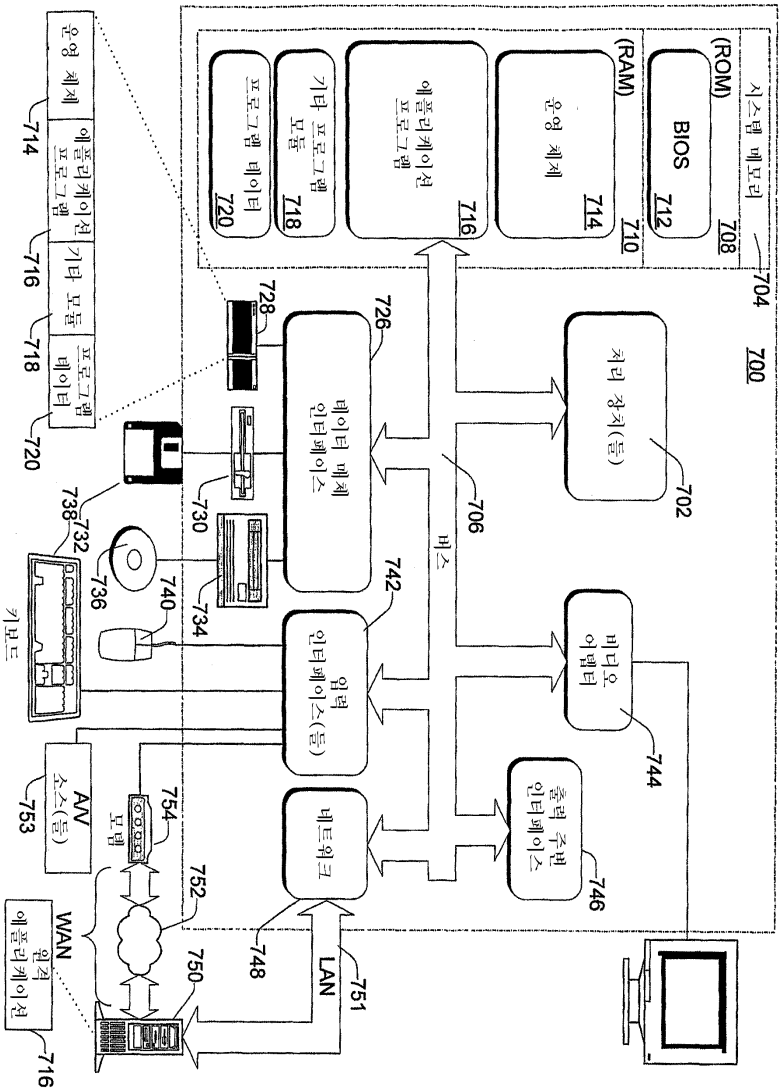
도면5


에뮬레이션 방지 바이트
원래의 N-바이트 패턴에서의 최종 바이트
xxxx xxxx0000 00000000 zzzzzzzz 00000000 0001xxxx
에뮬레이션 방지 바이트가 삽입된
비트-시프트된 시작 코드 프리픽스

도면6


600
다른 길이를 갖는 시작 코드와 연관된 하나 이상의 패턴 검색
602
제1 패턴 타입 발견에 응답하여, 제1 길이를 갖는 시작 코드의 에뮬레이션을 방지하기 위하여 에뮬레이션 방지데이터 삽입
604
제2 패턴 타입을 발견함에 응답하여, 제2 길이를 갖는 시작 코드의 에뮬레이션을 방지하기 위하여 에뮬레이션 방지 데이터 삽입

도면7


700
704
시스템 메모리
708
(ROM)
BIOS
712
(RAM)
710
운영 체제
714
에플리케이션 프로그램
716
기타 프로그램 모듈
718
프로그램 데이터
720
처리 장치(들)
702
706
버스
726
데이터 매체 인터페이스
728
730
734
742
입력 인터페이스(들)
732
738
736
740
키보드
744
비디오 어댑터
746
출력 주변 인터페이스
748
네트워크
LAN
751
750
752
754
모뎀
753
A/V 소스(들)
WAN
원격 에플리케이션
716
714
716
718
720
운영 체제
에플리케이션 프로그램
기타 모듈
프로그램 데이터