RFC 20 — ASCII: 네트워크 문자 교환의 기초

> **선수 지식**: 이진수와 16진수 표기법, 비트/바이트 개념
> **RFC 계보**: ANSI X3.4-1963 → ANSI X3.4-1968 → **RFC 20 (1969, STD 80)** → ISO 646 (1991) → Unicode/UTF-8 (RFC 3629)
> **현행 상태**: Internet Standard (STD 80) — 1969년 발행 이후 현재까지 유효한 현행 표준

학습 목표

이 문서를 완료하면 다음을 할 수 있다

1. ASCII 코드 테이블의 구조를 설명하고, 임의의 문자의 비트 패턴을 계산할 수 있다.
2. 33개 제어 문자의 분류(CC/FE/IS)와 현대 시스템에서의 용도를 설명할 수 있다.
3. CR/LF 문제의 역사적 원인과 플랫폼별 차이를 이해하고, 실무에서 진단/변환할 수 있다.
4. ASCII와 UTF-8의 호환 관계를 바이트 수준에서 설명할 수 있다.
5. 리눅스 시스템에서 인코딩 관련 문제를 진단하고 해결할 수 있다.

1. 배경과 문제 정의

1.1 이 표준이 해결하려는 문제

1969년 ARPANET에 연결된 컴퓨터들은 서로 다른 문자 체계를 사용했다. IBM 메인프레임은 EBCDIC(Extended Binary Coded Decimal Interchange Code), DEC 미니컴퓨터는 자체 문자 집합, CDC 슈퍼컴퓨터는 6비트 Display Code를 사용했다. 워드 크기도 12비트, 16비트, 18비트, 36비트, 60비트 등으로 제각각이었다.

이 환경에서 텍스트 데이터를 교환하려면 모든 시스템이 합의하는 공통 문자 인코딩이 필요했다. 각 시스템은 자체 문자 체계와 공통 인코딩 사이의 변환만 구현하면, N개 시스템 간의 통신이 가능해진다. 이것이 RFC 20이 해결하려는 문제다.

1.2 선행 표준: ANSI X3.4

RFC 20이 채택한 문자 집합은 RFC 자체가 새로 만든 것이 아니다. 미국 표준 협회(ASA, 이후 ANSI)가 1963년에 처음 발행하고 1968년에 개정한 ANSI X3.4-1968(USA Standard Code for Information Interchange)을 네트워크 교환용으로 채택한 것이다.

RFC 20의 핵심 기여는 두 가지다. 첫째, 7비트 문자 코드를 8비트 바이트에 담되 최상위 비트를 항상 0으로 설정하도록 규정했다. 둘째, 이 인코딩을 ARPANET의 Host-Host 통신(NCP 위)에서 사용하는 기본 문자 교환 형식으로 지정했다.

1.3 설계 결정과 트레이드오프

7비트 선택: 128개의 문자를 표현할 수 있다. 당시 영어 알파벳(대소문자), 숫자, 기본 기호, 제어 문자를 담기에 충분했다. 8비트로 확장하면 256개까지 가능하지만, 당시 네트워크 대역폭과 일부 시스템의 7비트 데이터 경로를 고려하면 7비트가 실용적이었다.

8비트 바이트에 7비트 코드: 최상위 비트를 0으로 고정함으로써 7비트 코드를 8비트 바이트 환경에 자연스럽게 임베딩했다. 이 결정은 이후 UTF-8이 ASCII 바이트를 그대로 유지하도록 설계되는 근거가 되었다.

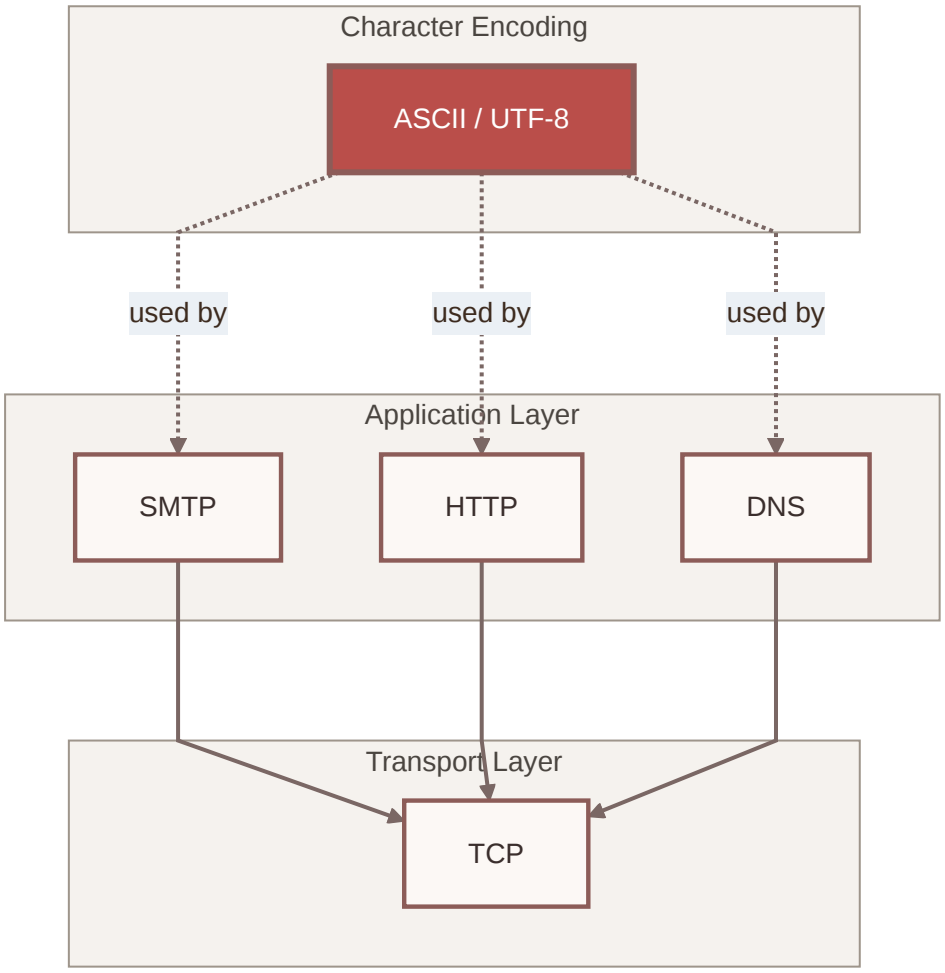
비목표(Non-goal): RFC 20은 물리적 매체에서의 기록 방법, 에러 제어, 코드 확장 기법, 제어 문자의 그래픽 표현을 정의하지 않는다. 이 범위 제한은 의도적인 것으로, 물리적 구현의 다양성을 허용하면서 논리적 문자 집합만 표준화하는 접근이다.

> 핵심 정리: RFC 20은 새로운 문자 집합을 만든 것이 아니라, 기존 미국 표준(ANSI X3.4-1968)을 네트워크 교환용 기본 인코딩으로 지정한 것이다. 7비트 코드를 8비트 바이트에 담되 최상위 비트를 0으로 고정하는 규칙이 핵심이다.

2. 프로토콜 아키텍처

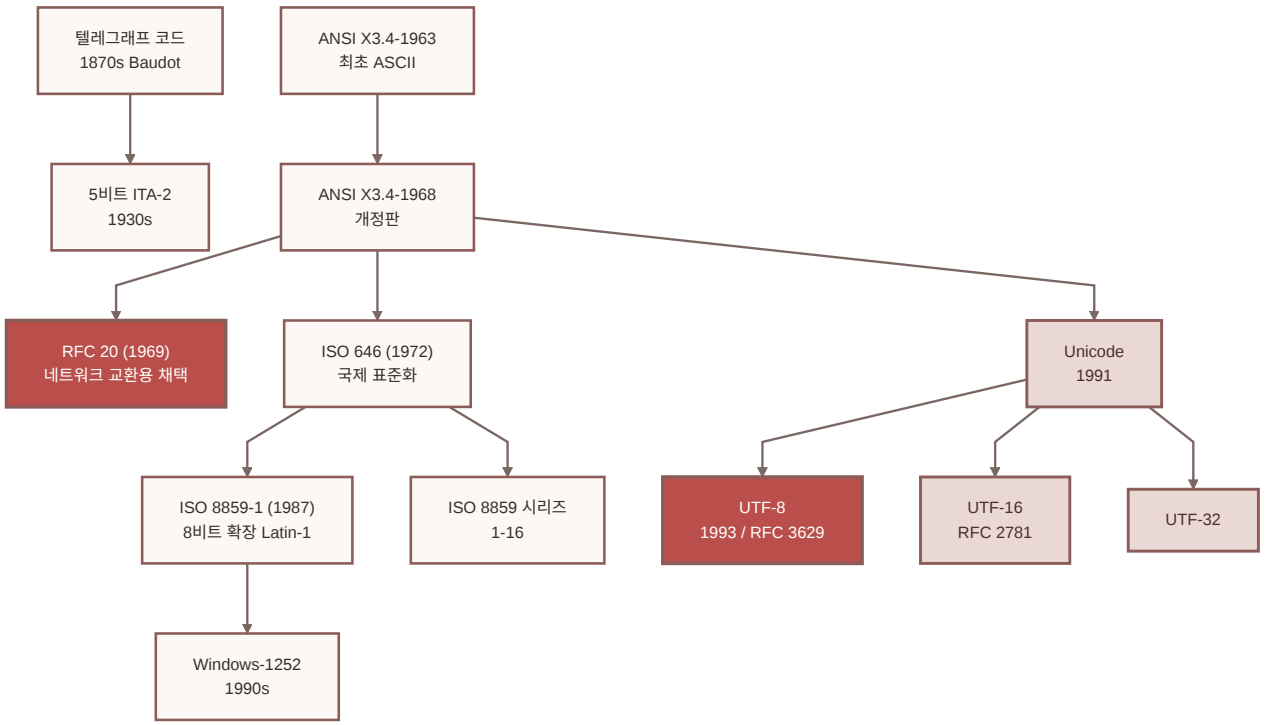
2.1 계층 내 위치

ASCII는 프로토콜 스택의 특정 계층에 속하는 프로토콜이 아니다. 모든 계층에서 텍스트 데이터를 표현할 때 사용하는 문자 인코딩 표준이다. HTTP 헤더, SMTP 명령어, DNS 도메인 이름, FTP 제어 채널, TLS 핸드셰이크의 SNI 필드 등 인터넷 프로토콜 스택 전반에 걸쳐 사용된다.



2.2 인접 표준과의 관계

ASCII는 단독으로 존재하지 않는다. 다음 표준들과 밀접한 관계를 맺고 있다.



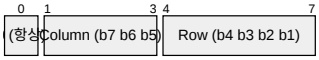
ASCII가 이 계보에서 차지하는 위치는 특별하다. UTF-8이 ASCII의 0x00~0x7F를 1바이트로 그대로 보존하도록 설계되었기 때문에, 유효한 ASCII 텍스트는 동시에 유효한 UTF-8 텍스트이기도 하다. 이 호환성이 UTF-8이 웹의 기본 인코딩이 된 핵심 이유 중 하나다.

> 핵심 정리: ASCII는 특정 프로토콜 계층에 속하지 않고 모든 계층에서 텍스트를 표현하는 기반이다. UTF-8은 ASCII의 상위 호환으로 설계되어, ASCII 텍스트는 변환 없이 UTF-8로 해석할 수 있다.

3. 메시지 구조: ASCII 코드 테이블

3.1 비트 레이아웃

ASCII는 7비트 코드다. 각 문자는 b7(최상위)부터 b1(최하위)까지 7개 비트로 표현된다. RFC 20은 이를 8비트 바이트에 담을 때 b8(최상위 비트)을 항상 0으로 설정하도록 규정했다.



Column 번호는 b7, b6, b5의 이진값으로 결정되고(0~7), Row 번호는 b4, b3, b2, b1의 이진값으로 결정된다(0~15). 따라서 128개(8×16) 코드 포인트가 존재한다.

예를 들어 문자 'K'의 코드 포인트는 Column 4, Row 11이다:

- b7=1, b6=0, b5=0 → Column 4
- b4=1, b3=0, b2=1, b1=1 → Row 11
- 7비트 이진: 1001011
- 8비트 바이트: 01001011 = 0x4B = 75(10진수)

3.2 코드 테이블 전체 구조

128개 코드 포인트는 다음과 같이 영역별로 나뉜다.

	Column: 0	1	2	3	4	5	6	7
Row 0	NUL	DLE	SP	0	@	P	`	p
Row 1	SOH	DC1	!	1	A	Q	a	q
Row 2	STX	DC2	"	2	B	R	b	r
Row 3	ETX	DC3	#	3	C	S	c	s
Row 4	EOT	DC4	$	4	D	T	d	t
Row 5	ENQ	NAK	%	5	E	U	e	u
Row 6	ACK	SYN	&	6	F	V	f	v
Row 7	BEL	ETB	'	7	G	W	g	w
Row 8	BS	CAN	(	8	H	X	h	x
Row 9	HT	EM	)	9	I	Y	i	y
Row 10	LF	SUB	*	:	J	Z	j	z
Row 11	VT	ESC	+	;	K	[	k	{
Row 12	FF	FS	,	<	L	\	l	¦
Row 13	CR	GS	-	=	M	]	m	}
Row 14	SO	RS	.	>	N	^	n	~
Row 15	SI	US	/	?	O	_	o	DEL

```
          CC    CC    기호  숫자  대문자 기호  소문자 기호
                            기호        대문자        소문자
```

영역별 구성:

Column	범위 (16진)	내용	개수
0	0x00~0x0F	제어 문자 (CC, FE, IS) 전반부	16
1	0x10~0x1F	제어 문자 후반부	16
2	0x20~0x2F	공백(SP) + 기호	16
3	0x30~0x3F	숫자 0~9 + 기호	16
4	0x40~0x4F	@ + 대문자 A~O	16
5	0x50~0x5F	대문자 P~Z + 기호	16
6	0x60~0x6F	` + 소문자 a~o	16
7	0x70~0x7F	소문자 p~z + 기호 + DEL	16

이 배치에는 의도적인 설계가 반영되어 있다.

대소문자 변환이 1비트 연산이다. 대문자 'A'(0x41=01000001)와 소문자 'a'(0x61=01100001)의 차이는 b6 비트 하나뿐이다. 대문자→소문자는 `c | 0x20`, 소문자→대문자는 `c & ~0x20` 또는 `c & 0xDF`로 변환된다. 이 비트 연산은 현대 C 코드에서도 case-insensitive 비교에 사용된다.

숫자 문자와 수치 값의 관계가 직관적이다. 문자 '0'의 코드는 0x30(=48), '9'는 0x39(=57)이다. 문자를 수치로 변환하려면 `c - '0'`(또는 `c - 0x30`)만 하면 된다. 이 관계 때문에 C 표준 라이브러리의 `atoi()`, `strtol()` 등이 단순하게 구현된다.

정렬 순서가 이진값으로 결정된다. RFC 20 §6.3에서 명시한 규칙이다. 두 문자의 상대적 순서는 이진값으로 결정되며, 이로 인해 숫자(0x30~) < 대문자(0x41~) < 소문자(0x61~) 순서가 된다. `strcmp()`의 바이트 비교 결과가 ASCII 내에서 일관된 순서를 보장하는 것은 이 설계 덕분이다. 다만 이 순서는 대소문자를 구분하지 않는 사전 순서와는 다르다. case-insensitive 정렬이 필요한 경우 `strcasecmp()`를 사용해야 한다.

> 핵심 정리: ASCII 코드 테이블은 단순한 문자-숫자 매핑이 아니다. 대소문자 변환을 1비트 연산으로 만드는 배치, 숫자 문자와 수치의 직관적 대응, 이진값 기반 정렬 순서 등 실용적 설계 결정이 반영되어 있다.

3.3 제어 문자 상세

128개 코드 포인트 중 33개가 제어 문자(0x00~0x1F + 0x7F DEL)이며, 1개가 공백(0x20 SP)이다. RFC 20은 제어 문자를 세 가지 기능 범주로 분류했다.

CC (Communication Control) — 통신 제어

통신 세션의 흐름을 제어하는 문자들이다. 텔레타이프와 모뎀 통신 시대의 산물이며, 현대 TCP/IP 기반 통신에서는 대부분 본래 용도로 사용되지 않는다.

코드	이름	16진	원래 용도	현대 용도
SOH	Start of Heading	0x01	메시지 헤더 시작	Ctrl+A (터미널: 줄 시작으로 이동)
STX	Start of Text	0x02	메시지 본문 시작	Ctrl+B (터미널: 커서 뒤로)
ETX	End of Text	0x03	메시지 본문 종료	**Ctrl+C (프로세스 인터럽트, SIGINT)**
EOT	End of Transmission	0x04	전송 종료	**Ctrl+D (EOF 신호, 셸 종료)**
ENQ	Enquiry	0x05	상대 식별 요청	Ctrl+E (터미널: 줄 끝으로 이동)
ACK	Acknowledge	0x06	수신 확인	거의 미사용
DLE	Data Link Escape	0x10	제어 문자 이스케이프	거의 미사용
NAK	Negative Acknowledge	0x15	수신 실패	Ctrl+U (터미널: 줄 삭제)
SYN	Synchronous Idle	0x16	동기 회선 유휴	거의 미사용
ETB	End of Transmission Block	0x17	전송 블록 종료	Ctrl+W (터미널: 단어 삭제)

이 중 현대 리눅스에서 매일 사용하는 것은 ETX(Ctrl+C)와 EOT(Ctrl+D)이다. Ctrl+C가 프로세스를 종료하는 것은 터미널 드라이버가 0x03을 수신하면 포그라운드 프로세스 그룹에 SIGINT 신호를 보내도록 설계되어 있기 때문이다. Ctrl+D는 터미널의 읽기 버퍼를 즉시 플러시하며, 버퍼가 비어있으면 EOF 조건이 되어 셸이 종료된다.

> **실습** — `stty -a` 명령으로 현재 터미널의 제어 문자 매핑을 확인해 보자.
>
> ```
> $ stty -a | grep -E 'intr|eof|erase|kill|susp'
> intr = ^C; quit = ^\; erase = ^?; kill = ^U;
> eof = ^D; eol = <undef>; ... susp = ^Z;
> ```
>
> `intr = ^C`는 "인터럽트 문자가 Ctrl+C(0x03, ETX)"임을 의미한다. `stty intr ^B`로 변경하면 Ctrl+B가 인터럽트 키가 된다.

FE (Format Effector) — 포맷 이펙터

출력 장치(프린터, 디스플레이)의 출력 위치를 제어하는 문자들이다. 현대 시스템에서 가장 활발히 사용되는 제어 문자 범주다.

코드	이름	16진	원래 용도	현대 용도
BS	Backspace	0x08	인쇄 위치 후퇴	**백스페이스 키, 터미널 문자 삭제**
HT	Horizontal Tab	0x09	수평 탭	**탭 문자 (\t)** — 코드 들여쓰기, TSV 파일 구분

코드	이름	16진	원래 용도	현대 용도
				자
LF	Line Feed	0x0A	종이 한 줄 전진	**Unix/Linux 줄바꿈 (\n)**
VT	Vertical Tab	0x0B	수직 탭	거의 미사용 (일부 프린터 제어)
FF	Form Feed	0x0C	다음 페이지로 이동	Ctrl+L (터미널 화면 클리어)
CR	Carriage Return	0x0D	인쇄 위치를 줄 시작으로	**Windows 줄바꿈의 일부 (\r\n), HTTP 헤더 구분**

이 중 LF와 CR은 현대 컴퓨팅에서 가장 많은 혼란을 일으키는 문자들이다. 이 주제는 §4에서 상세히 다룬다.

HT(수평 탭, 0x09)는 두 가지 논쟁의 원인이다. 첫째, 탭 폭이 표준화되어 있지 않다. 대부분의 시스템은 8칸을 기본으로 사용하지만, 에디터에서는 2칸 또는 4칸으로 설정하는 경우가 많다. 둘째, "탭 vs 스페이스" 논쟁은 프로그래밍 커뮤니티에서 반복적으로 등장하는 주제다. Python은 PEP 8에서 4칸 스페이스를 공식 권장하며, Go는 `gofmt`에서 탭을 강제한다.

IS (Information Separator) — 정보 분리자

데이터를 논리적 단위로 구분하기 위한 문자들이다. 계층적 관계가 정의되어 있다.

코드	이름	16진	계층	현대 용도
US	Unit Separator	0x1F	최소 단위	RFC 7464 JSON Text Sequences에서 사용
RS	Record Separator	0x1E	레코드	RFC 7464에서 JSON 시퀀스 구분자로 사용
GS	Group Separator	0x1D	그룹	일부 바코드 표준(GS1)에서 사용
FS	File Separator	0x1C	최대 단위	거의 미사용

계층 관계는 FS(가장 포괄적) > GS > RS > US(가장 세분화) 순이다. 이 분리자들은 원래 천공 카드와 자기 테이프의 데이터 구조화를 위해 설계되었다. 현대에서는 CSV, TSV, JSON, XML 등 더 풍부한 구조화 형식이 사용되므로 IS 문자들의 직접 사용은 드물다. 다만 RFC 7464(JSON Text Sequences)가 RS(0x1E)를 JSON 시퀀스의 각 항목 시작 구분자로 사용하는 것이 주목할 만한 현대적 사례다.

기타 제어 문자 — 장치 제어, 시프트, 취소

CC/FE/IS 어느 범주에도 속하지 않거나 위 테이블에서 누락된 제어 문자들이다.

코드	이름	16진	원래 용도	현대 용도
DC1	Device Control 1	0x11	장치 제어	**XON** — 소프트웨어 흐름 제어에서 전송 재개 (Ctrl+Q)
DC2	Device Control 2	0x12	장치 제어	거의 미사용

코드	이름	16진	원래 용도	현대 용도
DC3	Device Control 3	0x13	장치 제어	**XOFF** — 소프트웨어 흐름 제어에서 전송 중지 (Ctrl+S)
DC4	Device Control 4	0x14	장치 제어 (Stop)	거의 미사용
SO	Shift Out	0x0E	대체 문자 집합으로 전환	일부 터미널에서 그래픽 문자 모드 전환
SI	Shift In	0x0F	기본 문자 집합으로 복귀	SO의 반대 동작
CAN	Cancel	0x18	현재 데이터 취소	Ctrl+X (일부 에디터에서 잘라내기)
EM	End of Medium	0x19	매체(테이프) 끝 표시	Ctrl+Y (일부 셸에서 붙여넣기)
SUB	Substitute	0x1A	무효 문자 대체	**Ctrl+Z — Windows에서 EOF 신호, Unix에서 SIGTSTP(프로세스 일시 정지)**

DC1/DC3(XON/XOFF)는 현대 시스템에서도 실무적으로 중요하다. 터미널에서 `Ctrl+S`를 실수로 누르면 화면 출력이 멈추는 현상이 발생하는데, 이는 DC3(XOFF)가 전송 중지 신호를 보내기 때문이다. `Ctrl+Q`(XON)로 해제할 수 있다. 이 동작을 비활성화하려면 `stty -ixon`을 사용한다.

SUB(0x1A)는 Unix와 Windows에서 완전히 다른 의미로 사용된다. Windows에서는 텍스트 파일의 EOF 표시로, Unix에서는 SIGTSTP 신호(프로세스 일시 정지, `fg`로 재개 가능)로 사용된다. 크로스 플랫폼 개발 시 이 차이를 인식해야 한다.

특수 문자

코드	이름	16진	설명
NUL	Null	0x00	모든 비트가 0. C 언어에서 문자열 종료자(`\0`). 천공 테이프에서는 "구멍 없음" 상태
BEL	Bell	0x07	터미널 경고음. `echo -e '\a'`로 확인 가능. 현대에서는 GUI 알림으로 대체
ESC	Escape	0x1B	코드 확장용 접두 문자. ANSI 이스케이프 시퀀스(`ESC[`로 시작)의 기반. 터미널 색상, 커서 제어에 현재도 사용
DEL	Delete	0x7F	천공 테이프에서 모든 구멍을 뚫어 기존 문자를 무효화. b1~b7 모두 1인 유일한 문자. 현대에서는 Delete 키에 매핑
SP	Space	0x20	공백. 기술적으로는 제어 문자가 아니라 인쇄 문자이나, "보이지 않는 문자"로서 제어 문자와 인쇄 문자의 경계에 있다

NUL(0x00)의 C 언어에서의 역할은 특히 중요하다. C 문자열은 NUL로 종료되며(`char *s = "hello"`는 메모리에 `68 65 6C 6C 6F 00`으로 저장), `strlen()`, `strcpy()` 등의 함수가 NUL을 문자열 끝으로 인식한다. 이 설계는 바이너리 데이터에 NUL이 포함될 경우 문자열 함수가 데이터를 잘라내는 문제를 야기하며, 이것이 바이너리 안전(binary-safe) 함수가 별도로 필요한 이유다.

ESC(0x1B)는 현대 터미널에서 여전히 핵심적으로 사용된다. ANSI 이스케이프 시퀀스는 `ESC[` (0x1B 0x5B)로 시작하며, 텍스트 색상 변경(`\033[31m` = 빨간색), 커서 이동(`\033[H` = 홈 위치), 화면 클리어(`\033[2J`) 등을 수행한다. `ls --color`, `git diff`, `htop` 등 색상이 있는 CLI 도구는 모두 이 메커니즘에 의존한다.

> 핵심 정리: ASCII의 33개 제어 문자 중 현대 시스템에서 일상적으로 사용되는 것은 LF, CR, HT, BS, NUL, ESC, ETX(Ctrl+C), EOT(Ctrl+D), SUB(Ctrl+Z, Windows EOF) 정도다. 나머지는 원래 용도와 다른 의미로 재활용되거나 사실상 사용되지 않는다.

4. 프로토콜 동작: CR/LF 문제

4.1 역사적 배경

CR/LF 문제는 ASCII에서 가장 실무적으로 중요한 주제다. 이 문제의 근원은 기계적 텔레타이프에 있다.

텔레타이프(특히 Teletype Model 33 ASR)에서 줄바꿈은 두 단계의 물리적 동작이 필요했다. **CR(Carriage Return, 0x0D)** 은 인쇄 캐리지를 줄의 시작 위치로 되돌리는 동작이다. **LF(Line Feed, 0x0A)** 는 종이를 한 줄 위로 올리는 동작이다. 두 동작을 합쳐야 비로소 "다음 줄의 시작"에 도달한다.

여기서 중요한 물리적 제약이 있었다. 캐리지가 오른쪽 끝에서 왼쪽 시작으로 되돌아오는 데 시간이 걸렸다. CR 직후 즉시 문자를 출력하면 캐리지가 아직 이동 중이므로 줄 중간에 문자가 찍히는 현상(smudge)이 발생했다. 이 때문에 CR 뒤에 LF를 보내어 캐리지 복귀 시간을 확보하거나, CR 뒤에 NUL(0x00)을 padding으로 삽입하는 관행이 있었다.

4.2 플랫폼별 줄바꿈 규칙

이 물리적 유산이 세 가지 서로 다른 줄바꿈 규칙으로 분화되었다.

플랫폼	줄바꿈 문자	16진	C 이스케이프	유래
Unix/Linux/macOS	LF	0x0A	`\n`	Multics → Unix 계보. 단일 문자로 단순화
Windows/DOS	CR+LF	0x0D 0x0A	`\r\n`	CP/M → DOS → Windows 계보. 텔레타이프 관행 유지
Classic Mac OS (pre-X)	CR	0x0D	`\r`	Apple의 독자적 선택. OS X에서 LF로 전환

네트워크 프로토콜(HTTP, SMTP, FTP, Telnet)은 CR+LF를 줄바꿈으로 사용한다. 이는 Telnet의 NVT(Network Virtual Terminal) 정의(RFC 854)에서 네트워크 줄바꿈을 CR+LF로 명시한 것에 기인하며, NVT의 문자 집합이 RFC 20의 ASCII를 기반으로 했기 때문이다. 이후의 모든 텍스트 기반 인터넷 프로토콜이 NVT의 줄바꿈 규칙을 계승했다.

4.3 실제 문제 시나리오

시나리오 1: Windows에서 작성한 스크립트를 Linux에서 실행

```
$ cat -A script.sh
#!/bin/bash^M
echo "hello"^M
```

`^M`은 CR(0x0D)의 표시다. bash가 이 스크립트를 실행하면 `#!/bin/bash\r`을 인터프리터 경로로 해석하여 `/bin/bash\r`이라는 파일을 찾으려 하고, 존재하지 않으므로 실패한다. 에러 메시지는 `bad interpreter: No such file or directory`로, CR 문자가 보이지 않으므로 원인을 파악하기 어렵다.

```
# 진단
$ file script.sh
script.sh: Bash script, ASCII text executable, with CRLF line terminators

# 수정
$ dos2unix script.sh
dos2unix: converting file script.sh to Unix format...

# 또는 sed로 직접 제거
$ sed -i 's/\r$//' script.sh
```

시나리오 2: Git에서의 줄바꿈 충돌

Windows와 Linux 개발자가 같은 저장소에서 작업할 때, Git이 체크아웃/커밋 시 줄바꿈을 변환하면서 불필요한 diff가 발생한다.

```
# Git 줄바꿈 설정 확인
$ git config --global core.autocrlf
# Linux에서 권장: input (커밋 시 CRLF→LF, 체크아웃 시 변환 없음)
# Windows에서 권장: true (커밋 시 CRLF→LF, 체크아웃 시 LF→CRLF)

# 프로젝트별 설정 (.gitattributes)
$ cat .gitattributes
* text=auto
*.sh text eol=lf
*.bat text eol=crlf
*.png binary
```

시나리오 3: HTTP 헤더 파싱

HTTP/1.1(RFC 7230)은 헤더 줄을 CR+LF로 구분하고, 헤더와 본문 사이를 빈 줄(CR+LF+CR+LF)로 구분한다. 일부 서버는 관용적으로 LF만으로도 헤더를 구분하지만, 엄격한 구현은 CR+LF만 인정한다.

```
GET / HTTP/1.1\r\n
Host: example.com\r\n
Accept: text/html\r\n
\r\n
(본문 시작)
```

이 4바이트 시퀀스(0x0D 0x0A 0x0D 0x0A)가 헤더 종료를 나타낸다. HTTP 파서에서 이 시퀀스를 감지하지 못하면 헤더와 본문을 구분할 수 없다.

시나리오 4: SMTP 스머글링

2023년에 발견된 SMTP smuggling 공격은 메일 서버들이 bare LF(CR 없는 LF)를 처리하는 방식의 차이를 악용했다. RFC 5321은 SMTP에서 줄바꿈이 반드시 CR+LF여야 한다고 규정하지만, 일부 서버는 bare LF도 줄바꿈으로 인정했다. 공격자는 이 불일치를 이용하여 SPF/DKIM 인증을 우회하는 이메일을 전송할 수 있었다.

> 핵심 정리: CR/LF 문제는 1960년대 텔레타이프의 물리적 제약에서 시작되어, 운영체제 간 줄바꿈 규칙 차이, 네트워크 프로토콜의 줄바꿈 표준, 그리고 2023년의 SMTP 보안 취약점에까지 이르는 60년 역사의 문제다. 핵심은 "줄바꿈이 CR+LF인지 LF인지"를 항상 의식하는 것이다.

5. 리눅스 구현과 실무

5.1 커널 및 시스템 설정

리눅스 시스템에서 ASCII와 관련된 주요 설정:

로케일 설정: 시스템의 문자 인코딩은 로케일로 결정된다.

```
# 현재 로케일 확인
$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
...

# 사용 가능한 로케일 목록
$ locale -a ¦ grep -i utf
en_US.utf8
ko_KR.utf8
...

# C 로케일은 순수 ASCII만 사용
$ LC_ALL=C locale charmap
ANSI_X3.4-1968
```

`LC_ALL=C`(또는 `LANG=C`)로 설정하면 시스템이 순수 ASCII 모드로 동작한다. 이 상태에서는 0x80 이상의 바이트가 유효한 문자로 인식되지 않는다. 스크립트에서 바이트 단위 처리가 필요하거나, 로케일 의존적 동작을 피하려 할 때 `LC_ALL=C`를 명시적으로 사용한다.

터미널 제어 문자 매핑: `stty` 명령으로 확인/변경한다.

```
# 모든 제어 문자 매핑 확인
$ stty -a
speed 38400 baud; rows 24; columns 80;
intr = ^C; quit = ^\; erase = ^?; kill = ^U;
eof = ^D; eol = <undef>; start = ^Q; stop = ^S;
susp = ^Z; ...

# Ctrl+C의 매핑을 Ctrl+X로 변경 (권장하지 않지만 가능)
$ stty intr ^X
```

5.2 CLI 도구 실습

`xxd` — **바이트 수준 확인**: 파일의 실제 바이트 내용을 16진수로 확인한다.

```
# "Hello\n"의 실제 바이트 확인
$ echo "Hello" ¦ xxd
00000000: 4865 6c6c 6f0a                           Hello.

# Windows 줄바꿈 파일의 바이트 확인
$ printf "Hello\r\n" ¦ xxd
00000000: 4865 6c6c 6f0d 0a                        Hello..
```

0x0A는 LF, 0x0D는 CR이다. `xxd` 출력에서 `.`으로 표시되는 문자는 인쇄 불가능한 문자(제어 문자)를 나타낸다.

`file` — 파일 인코딩 및 줄바꿈 감지:

```
$ file *.txt
unix_file.txt:    ASCII text
windows_file.txt: ASCII text, with CRLF line terminators
utf8_file.txt:    UTF-8 Unicode text
binary_file.dat:  data
```

`od` — 8진수/16진수 덤프:

```
# 문자별 코드 포인트 확인
$ echo -n "AaBb" ¦ od -A x -t x1z
000000 41 61 42 62                                >AaBb<
```

`iconv` — 인코딩 변환:

```
# EUC-KR → UTF-8 변환
$ iconv -f EUC-KR -t UTF-8 input.txt > output.txt

# 변환 불가능한 문자를 '?'로 대체
$ iconv -f UTF-8 -t ASCII//TRANSLIT input.txt > ascii_only.txt
```

`tr` — 제어 문자 제거/변환:

```
# CR 문자 제거 (dos2unix 대체)
$ tr -d '\r' < windows.txt > unix.txt

# 인쇄 불가능 문자를 모두 제거
$ tr -cd '[:print:]\n' < dirty.txt > clean.txt
```

`hexdump` — HTTP 응답의 바이트 확인:

```
# HTTP 응답 헤더의 실제 줄바꿈 확인
$ curl -sI https://example.com ¦ xxd ¦ head -5
```

```
00000000: 4854 5450 2f31 2e31 2032 3030 204f 4b0d  HTTP/1.1 200 OK.
00000010: 0a44 6174 653a 2046 7269 2c20 3031 204d  .Date: Fri, 01 M
```

마지막 바이트 `0d`(CR)와 다음 줄 시작 `0a`(LF)가 HTTP 줄바꿈(CR+LF)을 구성하고 있음을 확인할 수 있다.

5.3 ASCII 관련 커널 메커니즘

TTY 라인 디시플린(Line Discipline): 리눅스 커널의 TTY 서브시스템은 ASCII 제어 문자를 해석하여 특정 동작을 수행한다. 이 처리는 `n_tty.c`(N_TTY 라인 디시플린)에서 이루어진다.

```
# cooked 모드(기본): 커널이 제어 문자를 해석
# raw 모드: 커널이 제어 문자를 해석하지 않고 그대로 전달
$ stty raw    # raw 모드 전환 (Ctrl+C도 작동하지 않게 됨)
$ stty cooked # 복원
```

cooked 모드에서 커널이 해석하는 주요 제어 문자:

- 0x03 (Ctrl+C) → SIGINT 전송
- 0x1A (Ctrl+Z) → SIGTSTP 전송 (프로세스 일시 정지)
- 0x04 (Ctrl+D) → EOF 조건 생성
- 0x08 (Ctrl+H) 또는 0x7F (DEL) → 문자 삭제 (erase)
- 0x15 (Ctrl+U) → 줄 삭제 (kill)

> 핵심 정리: 리눅스에서 ASCII는 로케일(`LANG`, `LC_*`), 터미널 드라이버(`stty`), TTY 라인 디시플린(`n_tty.c`)의 세 수준에서 처리된다. `xxd`, `file`, `iconv`, `tr` 등의 CLI 도구로 인코딩 문제를 진단한다.

6. 코드 예제

6.1 C: ASCII 문자 분류기와 변환기

다음 프로그램은 ASCII 코드 테이블의 구조를 실제 코드로 확인한다. 문자의 분류, 대소문자 변환(비트 연산), 숫자 문자→정수 변환을 수행한다.

```
/* ascii_inspector.c
 * ASCII 문자 검사 및 변환 도구
 * 컴파일: gcc -o ascii_inspector ascii_inspector.c -Wall -Wextra
 * 사용:   echo "Hello, World! 123" | ./ascii_inspector
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

static const char *classify_control(unsigned char c) {
    /* RFC 20의 제어 문자 분류 */
    switch (c) {
        case 0x00: return "NUL (Null)";
        case 0x01: return "SOH (Start of Heading) [CC]";
        case 0x02: return "STX (Start of Text) [CC]";
        case 0x03: return "ETX (End of Text) [CC] — Ctrl+C";
        case 0x04: return "EOT (End of Transmission) [CC] — Ctrl+D";
        case 0x07: return "BEL (Bell)";
        case 0x08: return "BS  (Backspace) [FE]";
        case 0x09: return "HT  (Horizontal Tab) [FE]";
        case 0x0A: return "LF  (Line Feed) [FE] — Unix newline";
        case 0x0B: return "VT  (Vertical Tab) [FE]";
        case 0x0C: return "FF  (Form Feed) [FE]";
        case 0x0D: return "CR  (Carriage Return) [FE]";
        case 0x1B: return "ESC (Escape) — ANSI sequence prefix";
        case 0x1C: return "FS  (File Separator) [IS]";
        case 0x1D: return "GS  (Group Separator) [IS]";
        case 0x1E: return "RS  (Record Separator) [IS]";
        case 0x1F: return "US  (Unit Separator) [IS]";
        case 0x7F: return "DEL (Delete)";
        default:   return "(other control)";
    }
}

int main(void) {
    int ch;
    int count = 0;
```

```
    printf("%-6s %-4s %-8s %-10s %-30s\n",
           "Char", "Dec", "Hex", "Binary", "Classification");
    printf("------+----+--------+----------+------------------------------\n");

    while ((ch = getchar()) != EOF) {
        unsigned char c = (unsigned char)ch;

        /* 8비트 이진 표현 */
        char binary[9];
        for (int i = 7; i >= 0; i--) {
            binary[7 - i] = (c >> i) & 1 ? '1' : '0';
        }
        binary[8] = '\0';

        /* 표시용 문자 */
        char display[5];
        if (c < 0x20 || c == 0x7F) {
            snprintf(display, sizeof(display), "^%c", c < 0x20 ? c + 0x40 : '?');
        } else {
            snprintf(display, sizeof(display), "'%c'", c);
        }

        /* 분류 */
        const char *classification;
        if (c < 0x20 || c == 0x7F) {
            classification = classify_control(c);
        } else if (c == 0x20) {
            classification = "SP (Space)";
        } else if (c >= '0' && c <= '9') {
            /* 숫자→정수 변환: 0x30 빼기 */
            int val = c - '0'; /* 또는 c - 0x30 */
            static char buf[64];
            snprintf(buf, sizeof(buf), "Digit (numeric value: %d)", val);
            classification = buf;
        } else if (c >= 'A' && c <= 'Z') {
            /* 대→소 변환: bit 5 설정 (OR 0x20) */
            char lower = c | 0x20;
            static char buf[64];
            snprintf(buf, sizeof(buf), "Uppercase (lowercase: '%c' via |0x20)", lower);
            classification = buf;
        } else if (c >= 'a' && c <= 'z') {
            /* 소→대 변환: bit 5 클리어 (AND 0xDF) */
            char upper = c & 0xDF;
            static char buf[64];
            snprintf(buf, sizeof(buf), "Lowercase (uppercase: '%c' via &0xDF)", upper);
            classification = buf;
        } else if (c > 0x7F) {
```

```
            classification = "Non-ASCII (outside RFC 20 range)";
        } else {
            classification = "Symbol/Punctuation";
        }

        printf("%-6s %-4d 0x%02X     %s   %s\n",
               display, c, c, binary, classification);
        count++;
    }

    printf("\nTotal: %d bytes processed\n", count);
    return 0;
}
```

실행 예:

```
$ echo -n "Ab9\r\n" ¦ ./ascii_inspector
Char   Dec  Hex      Binary     Classification
------+----+--------+----------+------------------------------
'A'    65   0x41     01000001   Uppercase (lowercase: 'a' via ¦0x20)
'b'    98   0x62     01100010   Lowercase (uppercase: 'B' via &0xDF)
'9'    57   0x39     00111001   Digit (numeric value: 9)
^M     13   0x0D     00001101   CR  (Carriage Return) [FE]
^J     10   0x0A     00001010   LF  (Line Feed) [FE] — Unix newline

Total: 5 bytes processed
```

6.2 Python: 인코딩 변환 및 CR/LF 정규화 도구

```
#!/usr/bin/env python3
"""ascii_normalize.py — 파일의 인코딩과 줄바꿈을 검사하고 정규화한다.

사용법:
    python3 ascii_normalize.py [--fix] <filename>
    python3 ascii_normalize.py --fix --target-eol lf input.txt
"""
import sys
import argparse
from pathlib import Path


def analyze_file(filepath: str) -> dict:
    """파일의 인코딩 특성을 분석한다."""
    data = Path(filepath).read_bytes()

    result = {
```

```
        'total_bytes': len(data),
        'ascii_bytes': 0,
        'non_ascii_bytes': 0,
        'control_chars': {},
        'cr_count': 0,
        'lf_count': 0,
        'crlf_count': 0,
        'bare_cr_count': 0,
        'bare_lf_count': 0,
        'nul_count': 0,
        'is_pure_ascii': True,
    }

    i = 0
    while i < len(data):
        b = data[i]
        if b > 0x7F:
            result['non_ascii_bytes'] += 1
            result['is_pure_ascii'] = False
        else:
            result['ascii_bytes'] += 1

        if b == 0x00:
            result['nul_count'] += 1

        if b == 0x0D:  # CR
            result['cr_count'] += 1
            if i + 1 < len(data) and data[i + 1] == 0x0A:
                result['crlf_count'] += 1
                i += 1  # skip the LF that's part of CRLF
            else:
                result['bare_cr_count'] += 1
        elif b == 0x0A:  # LF (not preceded by CR)
            result['lf_count'] += 1
            result['bare_lf_count'] += 1
        elif b < 0x20 and b not in (0x09, 0x0A, 0x0D):
            # 비정상적 제어 문자 (HT, LF, CR 제외)
            name = {
                0x00: 'NUL', 0x01: 'SOH', 0x02: 'STX', 0x03: 'ETX',
                0x04: 'EOT', 0x07: 'BEL', 0x08: 'BS', 0x0B: 'VT',
                0x0C: 'FF', 0x1B: 'ESC', 0x1C: 'FS', 0x1D: 'GS',
                0x1E: 'RS', 0x1F: 'US',
            }.get(b, f'0x{b:02X}')
            result['control_chars'][name] = result['control_chars'].get(name, 0) + 1

        i += 1
```

```
    # 줄바꿈 방식 판정
    if result['crlf_count'] > 0 and result['bare_lf_count'] == 0:
        result['eol_style'] = 'CRLF (Windows)'
    elif result['bare_lf_count'] > 0 and result['crlf_count'] == 0:
        result['eol_style'] = 'LF (Unix)'
    elif result['bare_cr_count'] > 0 and result['crlf_count'] == 0 and result['bare_lf_count'] == 0:
        result['eol_style'] = 'CR (Classic Mac)'
    elif result['crlf_count'] > 0 and result['bare_lf_count'] > 0:
        result['eol_style'] = 'MIXED (CRLF + LF) — 문제 있음!'
    else:
        result['eol_style'] = 'None (단일 줄)'

    return result


def normalize_eol(filepath: str, target: str = 'lf') -> int:
    """줄바꿈을 지정된 형식으로 정규화한다."""
    data = Path(filepath).read_bytes()

    # 먼저 모든 CRLF를 LF로 통일
    normalized = data.replace(b'\r\n', b'\n')
    # 남은 bare CR도 LF로 변환
    normalized = normalized.replace(b'\r', b'\n')

    if target == 'crlf':
        normalized = normalized.replace(b'\n', b'\r\n')

    Path(filepath).write_bytes(normalized)
    return len(data) - len(normalized)


def main():
    parser = argparse.ArgumentParser(description='ASCII/인코딩 분석 및 정규화')
    parser.add_argument('filename', help='분석할 파일')
    parser.add_argument('--fix', action='store_true', help='줄바꿈 정규화 수행')
    parser.add_argument('--target-eol', choices=['lf', 'crlf'], default='lf',
                        help='정규화 대상 줄바꿈 (기본: lf)')
    args = parser.parse_args()

    result = analyze_file(args.filename)

    print(f"File: {args.filename}")
    print(f"Size: {result['total_bytes']} bytes")
    print(f"ASCII bytes: {result['ascii_bytes']}
({result['ascii_bytes']*100//max(result['total_bytes'],1)}%)")
    print(f"Non-ASCII bytes: {result['non_ascii_bytes']}")
    print(f"Pure ASCII: {'Yes' if result['is_pure_ascii'] else 'No'}")
```

```
    print(f"EOL style: {result['eol_style']}")
    print(f"  CRLF: {result['crlf_count']}, Bare LF: {result['bare_lf_count']}, "
          f"Bare CR: {result['bare_cr_count']}")

    if result['nul_count'] > 0:
        print(f"  WARNING: {result['nul_count']} NUL bytes found (binary file?)")
    if result['control_chars']:
        print(f"  Unusual control chars: {result['control_chars']}")

    if args.fix:
        diff = normalize_eol(args.filename, args.target_eol)
        target_name = 'LF' if args.target_eol == 'lf' else 'CRLF'
        print(f"\nNormalized to {target_name} (size change: {diff} bytes)")


if __name__ == '__main__':
    main()
```

실행 예:

```
$ python3 ascii_normalize.py mixed_eol_file.txt
File: mixed_eol_file.txt
Size: 1247 bytes
ASCII bytes: 1247 (100%)
Non-ASCII bytes: 0
Pure ASCII: Yes
EOL style: MIXED (CRLF + LF) — 문제 있음!
  CRLF: 12, Bare LF: 34, Bare CR: 0

$ python3 ascii_normalize.py --fix --target-eol lf mixed_eol_file.txt
Normalized to LF (size change: -12 bytes)
```

> 핵심 정리: C에서 ASCII의 대소문자 변환은 비트 연산(`| 0x20`, `& 0xDF`)으로 수행할 수 있다. 인코딩 문제 진단/정규화 도구는 바이트 수준에서 동작해야 하며, 텍스트 모드(`'r'`)가 아닌 바이너리 모드(`'rb'`)로 파일을 읽어야 플랫폼 의존적 줄바꿈 변환을 피할 수 있다.

7. 트러블슈팅

7.1 "Bad interpreter" 에러

증상: Linux에서 셸 스크립트 실행 시 `bash: ./script.sh: /bin/bash^M: bad interpreter: No such file or directory` 에러 발생.

원인: 스크립트 파일이 Windows에서 작성되어 shebang 줄(`#!/bin/bash`)이 `#!/bin/bash\r`로 저장되었다. `\r`(CR, 0x0D)이 인터프리터 경로의 일부로 해석되어 존재하지 않는 파일을 찾는다.

진단:

```
$ file script.sh
script.sh: Bash script, ASCII text executable, with CRLF line terminators

$ head -1 script.sh | xxd
00000000: 2321 2f62 696e 2f62 6173 680d 0a         #!/bin/bash..
                                     ^^^^
                                     CR (0x0D) 이 문제
```

해결:

```
$ dos2unix script.sh
# 또는
$ sed -i 's/\r$//' script.sh
# 또는 vim에서
:set fileformat=unix
:wq
```

7.2 인코딩 불일치로 인한 한글 깨짐

증상: EUC-KR로 작성된 파일을 UTF-8 터미널에서 열면 한글이 깨져서 표시된다. 또는 그 반대.

원인: 파일의 실제 인코딩과 터미널/에디터의 기대 인코딩이 다르다.

진단:

```
# 파일 인코딩 추정 (정확하지 않을 수 있음)
$ file --mime-encoding document.txt
document.txt: euc-kr

# chardet으로 더 정확한 추정 (pip install chardet)
$ chardetect document.txt
document.txt: EUC-KR with confidence 0.99
```

```
# 실제 바이트 확인
$ xxd document.txt ¦ head -3
```

해결:

```
# EUC-KR → UTF-8 변환
$ iconv -f EUC-KR -t UTF-8 document.txt > document_utf8.txt

# 변환 불가능한 문자가 있으면 에러 발생. 무시하려면:
$ iconv -f EUC-KR -t UTF-8//IGNORE document.txt > document_utf8.txt
```

7.3 Git diff에서 모든 줄이 변경된 것으로 표시

증상: 내용을 변경하지 않았는데 `git diff`에서 파일 전체가 변경된 것으로 표시된다.

원인: `core.autocrlf` 설정에 의해 체크아웃 시 줄바꿈이 변환되었거나, 에디터가 줄바꿈 형식을 변경했다.

진단:

```
# 줄바꿈 변경 여부 확인
$ git diff --name-only
file.txt

$ git diff file.txt ¦ head -10
# 모든 줄에 줄바꿈 변경만 보이면 CR/LF 문제

# 현재 autocrlf 설정 확인
$ git config core.autocrlf
```

해결:

```
# .gitattributes로 프로젝트 수준에서 해결 (권장)
$ cat > .gitattributes << 'EOF'
* text=auto
*.sh text eol=lf
*.py text eol=lf
*.bat text eol=crlf
*.png binary
*.jpg binary
EOF

# 이미 커밋된 파일의 줄바꿈 정규화
$ git add --renormalize .
$ git commit -m "Normalize line endings"
```

7.4 curl/wget 응답 파싱 시 헤더 끝 감지 실패

증상: HTTP 응답을 파싱할 때 헤더와 본문의 경계를 찾지 못한다.

원인: 헤더 종료를 `\n\n`으로 검색하고 있지만, HTTP 표준은 `\r\n\r\n`을 사용한다.

진단:

```
# 실제 헤더 종료 바이트 확인
$ curl -sI https://example.com ¦ xxd ¦ grep "0d0a 0d0a"
```

해결: 헤더 파싱 시 `\r\n\r\n`을 사용하되, 관용적으로 `\n\n`도 허용하는 것이 실무적이다 (RFC 7230 §3.5 "robustness principle" 참조).

```
# 관용적 헤더/본문 분리
def split_http_response(raw: bytes) -> tuple:
    # 먼저 표준(CRLF) 시도
    sep = b'\r\n\r\n'
    idx = raw.find(sep)
    if idx >= 0:
        return raw[:idx], raw[idx + len(sep):]
    # 폴백: bare LF
    sep = b'\n\n'
    idx = raw.find(sep)
    if idx >= 0:
        return raw[:idx], raw[idx + len(sep):]
    # 구분자 없음
    return raw, b''
```

> 핵심 정리: ASCII 관련 트러블슈팅의 핵심 도구는 `xxd`(바이트 수준 확인), `file`(인코딩/줄바꿈 감지), `dos2unix`(줄바꿈 변환), `iconv`(인코딩 변환)이다. 문제의 원인은 대부분 "보이지 않는 문자"(CR, BOM, NUL)이므로 바이트 수준 확인이 필수다.

7.5 UTF-8 BOM으로 인한 파싱 에러

증상: UTF-8로 저장한 설정 파일이나 셸 스크립트가 올바른 내용인데도 파싱 에러를 일으킨다. JSON 파일의 경우 `Unexpected token` 에러가 발생하고, 셸 스크립트의 경우 shebang이 인식되지 않는다.

원인: Windows 메모장(Notepad)이 UTF-8 파일 저장 시 파일 시작에 BOM(Byte Order Mark, 0xEF 0xBB 0xBF)을 자동 삽입한다. UTF-8에서 BOM은 바이트 순서 표시의 기능이 없으므로(UTF-8은 바이트 순서가 고정) 불필요하지만, 일부 Windows 프로그램이 인코딩 식별 목적으로 삽입한다. Unix 도구 대부분은 BOM을 기대하지 않으며, 이를 일반 데이터로 처리하여 파싱에 실패한다.

진단:

```
# BOM 존재 여부 확인
$ xxd config.json ¦ head -1
00000000: efbb bf7b 0a20 2022 6e61 6d65 223a       ...{. "name":
          ^^^^^^^^
          UTF-8 BOM (0xEF 0xBB 0xBF)

$ file config.json
config.json: UTF-8 Unicode (with BOM) text
```

해결:

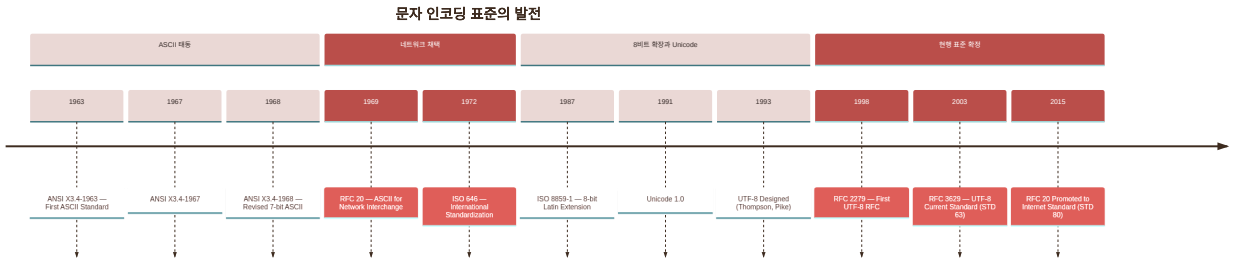
```
# BOM 제거 (sed)
$ sed -i '1s/^\xEF\xBB\xBF//' config.json

# BOM 제거 (tail — 3바이트 건너뛰기, 대용량 파일에 부적합)
$ tail -c +4 config.json > config_no_bom.json

# 디렉토리 내 모든 파일에서 BOM 일괄 제거
$ find . -type f -name "*.json" -exec sed -i '1s/^\xEF\xBB\xBF//' {} +
```

8. RFC 계보와 현대 발전

8.1 ASCII에서 UTF-8까지



ASCII의 한계와 확장 시도: ASCII는 영어 알파벳만 지원한다. 유럽어의 악센트 문자(é, ü, ñ), 아시아 문자(한글, 한자, 가나)를 표현할 수 없다. 이 문제를 해결하기 위해 여러 확장이 시도되었다.

ISO 8859 시리즈: 8비트로 확장하여 0x80~0xFF에 추가 문자를 배치했다. ISO 8859-1(Latin-1)은 서유럽어, ISO 8859-2는 동유럽어 등 지역별로 다른 코드 페이지를 정의했다. 문제는 하나의 파일에서 여러 언어를 동시에 사용할 수 없다는 것이다.

EUC-KR / Shift_JIS / Big5: 동아시아 문자를 위한 멀티바이트 인코딩. 각 언어별로 독자적인 인코딩이 존재하여 호환성 문제가 심각했다.

Unicode와 UTF-8: Unicode는 세계의 모든 문자를 하나의 코드 포인트 체계에 통합했다. UTF-8은 Unicode 코드 포인트를 가변 길이 바이트 시퀀스로 인코딩하며, ASCII 호환성을 보장한다.

8.2 UTF-8의 ASCII 호환성

UTF-8이 ASCII와 호환되는 구조:

바이트 수	코드 포인트 범위	바이트 패턴
1바이트	U+0000 ~ U+007F (ASCII)	0xxxxxxx
2바이트	U+0080 ~ U+07FF	110xxxxx 10xxxxxx
3바이트	U+0800 ~ U+FFFF	1110xxxx 10xxxxxx 10xxxxxx
4바이트	U+10000 ~ U+10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

1바이트 인코딩(0xxxxxxx)이 정확히 ASCII의 7비트 코드에 최상위 비트 0을 붙인 형태다. 따라서 유효한 ASCII 텍스트는 바이트 수준에서 변환 없이 유효한 UTF-8 텍스트이기도 하다.

이 설계에는 중요한 특성이 있다. 멀티바이트 시퀀스의 후속 바이트(10xxxxxx, 0x80~0xBF)는 ASCII 범위(0x00~0x7F)와 겹치지 않는다. 따라서 ASCII 문자를 찾는 바이트 검색(예: `strchr(s, '/')`)이 멀티바이트 문자 중간을 잘못 매칭하는 일이 없다. UTF-8이 Shift_JIS 같은 기존 멀티바이트 인코딩보다 안전한 결정적 이유다.

8.3 현대 프로토콜에서의 ASCII

현행 인터넷 프로토콜에서 ASCII가 사용되는 방식:

프로토콜	ASCII 사용 영역	비ASCII 데이터 처리
HTTP/1.1 (RFC 7230)	헤더 이름/값, 메소드, URI	본문은 Content-Type 인코딩 사용
SMTP (RFC 5321)	명령어, 주소	MIME(RFC 2045)로 본문 인코딩
DNS (RFC 1035)	도메인 라벨	IDN(RFC 5891)으로 유니코드 도메인 처리
FTP (RFC 959)	제어 채널 명령어	데이터 채널은 별도
TLS (RFC 8446)	SNI(Server Name Indication)	—
JSON (RFC 8259)	키/값(UTF-8 기본)	`\uXXXX` 이스케이프
URI (RFC 3986)	경로, 쿼리	비ASCII → %XX 퍼센트 인코딩

> 핵심 정리: ASCII는 UTF-8의 진부분집합이다. UTF-8은 ASCII 호환성을 핵심 설계 제약으로 삼았으며, 이것이 UTF-8이 웹의 기본 인코딩이 된 결정적 이유다. 현대 인터넷 프로토콜은 제어 채널/헤더에서 여전히 ASCII를 사용하며, 비ASCII 데이터는 별도의 인코딩 메커니즘(MIME, IDN, 퍼센트 인코딩)으로 처리한다.

용어집

용어	설명
ASCII	7비트 문자 인코딩 표준. 128개 코드 포인트로 영문 대소문자, 숫자, 기호, 33개 제어 문자를 정의한다. RFC 20(STD 80)으로 네트워크 교환 표준으로 채택되었다.
CR (Carriage Return)	ASCII 0x0D. 인쇄 위치를 줄 시작으로 되돌리는 제어 문자. Windows 줄바꿈(CR+LF)의 일부이자 HTTP/SMTP 헤더 줄바꿈의 구성 요소다.
LF (Line Feed)	ASCII 0x0A. 인쇄 위치를 다음 줄로 이동시키는 제어 문자. Unix/Linux의 기본 줄바꿈 문자이다.
NUL	ASCII 0x00. 모든 비트가 0인 문자. C 언어에서 문자열 종료자(`\0`)로 사용된다. 천공 테이프에서는 "구멍 없음" 상태를 의미했다.
ESC	ASCII 0x1B. 이스케이프 문자. 후속 바이트의 해석을 변경하는 접두 문자로, ANSI 이스케이프 시퀀스(`ESC[`)의 시작이다.
UTF-8	Unicode 코드 포인트를 가변 길이(1~4바이트) 바이트 시퀀스로 인코딩하는 방식. ASCII의 0x00~0x7F를 1바이트로 그대로 보존하므로 ASCII와 상위 호환된다.
Code Point	문자 집합에서 특정 문자에 할당된 번호. ASCII에서는 0~127, Unicode에서는 U+0000~U+10FFFF 범위다.
FE (Format Effector)	출력 장치의 출력 위치를 제어하는 ASCII 제어 문자 범주. BS, HT, LF, VT, FF, CR이 해당한다.
CC (Communication Control)	통신 세션의 흐름을 제어하는 ASCII 제어 문자 범주. SOH, STX, ETX, EOT, ENQ, ACK, DLE, NAK, SYN, ETB이 해당한다.
IS (Information Separator)	데이터를 계층적으로 구분하기 위한 ASCII 제어 문자 범주. FS > GS > RS > US 순의 계층 관계를 가진다.
BOM (Byte Order Mark)	Unicode 텍스트의 바이트 순서를 표시하는 문자(U+FEFF). UTF-8에서는 0xEF 0xBB 0xBF로 인코딩되며, 파일 시작에 있으면 인코딩 식별에 사용되지만 불필요한 문제를 일으키는 경우가 많다.

참고 자료

- RFC 20: ASCII format for Network Interchange — https://www.rfc-editor.org/rfc/rfc20
- RFC 3629: UTF-8, a transformation format of ISO 10646 — https://www.rfc-editor.org/rfc/rfc3629
- RFC 7230: HTTP/1.1 Message Syntax and Routing (§3.5 줄바꿈 관용성) — https://www.rfc-editor.org/rfc/rfc7230
- RFC 5321: SMTP (§2.3.8 줄바꿈 규칙) — https://www.rfc-editor.org/rfc/rfc5321
- RFC 7464: JSON Text Sequences (RS 문자 사용) — https://www.rfc-editor.org/rfc/rfc7464
- 관련 심화 문서: RFC_심화_UTF8_Unicode.md (예정)