컴퓨터 한글 코드 - 1. ASCII Code

인터넷 메타질라 Blog에서 발췌 정리

세벌식 자판의 우수성을 객관적으로 생각해보고자, 애국가를 분석한 일이 있었다. 결과는 좀 더 정리해서 올려보고자 한다. (다음 세벌식 사랑 모임 까페에 약간의 오류가 있고 정보가 부족해 보이는 애국가 분석자료를 HWP로 올려놓았습니다.)

분석 도중 애로사항은, '각'이라는 글자가 있다면, ㄱ+ㅏ+ㄱ 으로 나누어 분석해야 하는데, 600타 가까이 되는 애국가를 일일이 손으로 분석하는데 시간이 걸리며, 이보다 더 긴 글은 손으로 분석하는데 엄두도 내지 못한다는 것이다. 명색이 컴퓨터학과 졸업생이니 분석프로그램을 만들어 돌려보면 되지 않을까 생각도 했는데, 아뿔사 한글은 완성형으로 되어있고 이제까지 기껏 아스키(ASCII)코드로 알파벳이나 숫자정도만 끄적거려보던 내게는 한글 코드 자체를 배우지 않는 이상 무리였다. 그래서 직접 웹사이트 자료를 찾아다니며 한글코드를 조사했으며, 알게된 현재 표준코드와 실제 쓰이는 몇가지 한글 코드에 대해 정리해서 공개하고자 한다. 프로그래밍을 하는 분들이나, 그밖에 궁금했던 분들에게 유용한 정보가 되었으면 하는 바램이다.

* ASCII Code (American Standard Code for Information Interchange)

한글 코드를 설명하기 전에 먼저, 영문코드를 설명해야 할듯 싶다. 현재의 컴퓨터에 가장 기본이 되는 기반인 이 ASCII 코드부터 알아야 코드 매핑에 대한 이해가 쉬워질 듯하다. 원래 컴퓨터가 미국에서 개발되어 발달되었으니 어쩔 수 없는 현실이다.

컴퓨터가 원래는 0과 1 밖에 표현 못하는 바보라는 것은 알고 있는지. 컴퓨터는 전기로 움직이며, 전기가 흐르는 상태를 0, 흐르지 않는 상태를 1, 혹은 그 반대로 정해서 그것으로 숫자를 표현하기로 했다. 이렇게 0 또는 1을 나타내는 장치를 보통 '소자'라고 부르며, 단위는 비트(bit)라고 표현한다. 이러한 소자를 두 개를 붙인다면, 그 장치는 최대 4개의 숫자를 표현할 수 있고, 세 개라면, 8개의 숫자, 네 개라면 16개의 숫자. 이렇게, 여러 개의 소자를 붙여나감으로서 점차로 기하급수적인 숫자를 표현할 수 있고, 이는 곧 숫자를 표현하는 방법 가운데 하나인 2진법을 의미한다.

4bit-2진수의 예제
0 : 0000, 4 : 0100, 8 : 1000, 12 : 1100
1 : 0001, 5 : 0101, 9 : 1001, 13 : 1101
2 : 0010, 6 : 0110, 10 : 1010, 14 : 1110
3 : 0011, 7 : 0111, 11 : 1011, 15 : 1111

이렇게 4비트는 총 16(2의 4제곱)개의 수를 표현 할 수 있다. 이렇게 4비트를 묶어서 하나의 수로 표현하는 것이 컴퓨터에서 많이 쓰이는 16진법이다.

2진수와 16진수
0 : 0000, 4 : 0100, 8 : 1000, C : 1100
1 : 0001, 5 : 0101, 9 : 1001, D : 1101
2 : 0010, 6 : 0110, A : 1010, E : 1110
3 : 0011, 7 : 0111, B : 1011, F : 1111

또한 8비트를 묶어서 1 Byte라는 단위를 쓰며, 이것이 컴퓨터에서 저장되는 최소의 단위가 된다. 1 바이트는 0~FF(255)까지의 256개의 정보를 표현할 수 있다.

16비트 컴퓨터니, 64비트 컴퓨터니 하는 것은 "한 번에 처리할 수 있는 비트의 수"를 뜻한다고 하면 이해하기 쉬울 듯 하다. 즉, 계산을 할 때 16비트는 65,536(2의 16제곱)의 숫자를 구분 할 수 있다는 뜻이다. 현재의 컴퓨터는 64비트가 쓰이니 한번에 2의 64제곱인 18,446,744,073,709,551,616의 숫자를 구분하여 표시할 수 있다는 뜻이 된다. 엄청난 양의 정보다.

컴퓨터가 본격적으로 쓰여지기 시작할 때는 불과 8비트밖에 되지 않는 정보 처리량을 가졌다. 미국에서는 이 중 **7비트를 이용하여, 총 128개의 출력 문자신호를 매핑**하였으며 이중 표준으로 정해진 것이 **ASCII 코드**이다. 이후에, 사용되지 않던 **나머지 1비트를 추가하여 영문에 존재하지 않는 128개의 유럽의 알파벳이나 문자표를 추가**하였는데, 이것이 **확장 ASCII Code**이다. 영문 DOS시절 HEX Editor등을 통해 쉽게 들여다 볼 수 있던 코드들이며, 현재의 윈도라든지, 리눅스 등에서도 이 코드들이 기반이 되어 움직인다. (확장 ASCII 코드는 이제 세계적으로 거의 안 쓰이지만)

ASCII Code에서 처음의 32개는 시스템문자로 쓰이며, 48~57까지는 0 부터 9 까지의 숫자가 배치되어있고 65~90까지 A~Z의 대문자, 97~122까지 a~b의 소문자, 그 외의 영역은 우리가 흔히 볼 수 있는 특수 문자들로 채워져 있다.

만약, 전세계의 사람들 모두가 알파벳만을 쓴다면 이 ASCII 코드만으로도 아무 문제없이 컴퓨터를 이용할 수 있다. 그러나 동아시아만 보더라도 중국의 漢字가 있고 일본의 かな가 있고 우리나라의 한글이 있다. 문제는 이러한 글들을 컴퓨터에서 어떻게 표현하는가이다.

컴퓨터 한글 코드 - 2. 한글 코드 표준 제정, 조합형과 완성형

먼저 한글의 표현 방법부터 알아보자.

현대 한글을 자음, 모음으로 나누면
자음 ㄱ,ㄴ,ㄷ,ㄹ,ㅁ,ㅂ,ㅅ,ㅇ,ㅈ,ㅊ,ㅋ,ㅌ,ㅍ,ㅎ (14개)
모음 ㅏ,ㅑ,ㅓ,ㅕ,ㅗ,ㅛ,ㅜ,ㅠ,ㅡ,ㅣ (10개)
총 24자로 나뉜다.
이들만으로 코드를 만들어 한글을 표현할 수도 있으나 이는 대단히 비효율적이고 치명적인 문제가 있다. 자음의 코드가 1~2, 모음의 코드가 1~3개가 붙어서 표현될 경우가 생기기 때문에 한 글자를 표현하는데 최대 7개의 코드가 필요하게 된다. (꽦:ㄱ+ㄱ+ㅗ+ㅏ+ㅣ+ㄱ+ㄱ) 또한, ㅂ+ㅏ+ㄱ+ㄱ+ㅣ 라는 데이터가 있다면 이를 '바끼'로 읽어야 할까, 아니면 '박기'로 읽어야 할까?

컴퓨터에서의 한글처리의 어려움은 바로 이 모아쓰기를 하여 조합이 되어야 한 글자가 된다는 한글의 특성에 있다. 따라서 컴퓨터에서는 한글을 초성, 중성, 종성의 세 부분으로 나누고 이를 표현하는 것이 알맞게 된다.

현대 한글의 모든 초성, 중성, 종성을 나열해 보면
초성 ㄱ,ㄲ,ㄴ,ㄷ,ㄸ,ㄹ,ㅁ,ㅂ,ㅃ,ㅅ,ㅆ,ㅇ,ㅈ,ㅉ,ㅊ,ㅋ,ㅌ,ㅍ,ㅎ (19개)
중성 ㅏ,ㅐ,ㅑ,ㅒ,ㅓ,ㅔ,ㅕ,ㅖ,ㅗ,ㅘ,ㅙ,ㅚ,ㅛ,ㅜ,ㅝ,ㅞ,ㅟ,ㅠ,ㅡ,ㅢ,ㅣ (21개)
종성 <없음>,ㄱ,ㄲ,ㄳ,ㄴ,ㄵ,ㄶ,ㄷ,ㄹ,ㄺ,ㄻ,ㄼ,ㄽ,ㄾ,ㄿ,ㅀ,ㅁ,ㅂ,ㅄ,ㅅ,ㅆ,ㅇ,ㅈ,ㅊ,ㅋ,ㅌ,ㅍ,ㅎ (28개)
이들의 조합으로 총 11172(19*21*28)개의 한글을 표현할 수 있다.

이렇게 조합된 **한글을 저장하는 방법**은 다시 두 가지로 나눌 수 있다.
첫 번째는 위의 **초성, 중성, 종성으로 나뉜 코드를 그대로 저장하는 방식이며 이를 조합형**이라 부른다. 조합형의 장점은 적은 코드의 개수로 11172개의 한글을 모두 표현할 수 있다는 것이며, 단점은 이를 읽어 들일 때마다 한글자로 조합하는 처리를 해주어야 한다는 것이다.

두 번째로 조합된 11172자의 **글자에 모두 코드를 부여하는 방법이며 이를 완성형**이라 부른다. 장점은 읽어 들일 때 별다른 처리가 필요 없다는 것이나, 단점은 11172개의 모든 한글에 코드를 부여해야 하며, 이를 다시 초성, 중성, 종성으로 나누어 처리하고자 할 때에는 어쩔 수 없이 조합형을 다시 이용해야 한다는 것이다.

앞의 글에서 1바이트(8비트) 내에서 알파벳 문화권의 문자들을 표현하기 위한 표준으

로 ASCII Code가 사용되었다고 설명하였다. 우리나라에 PC가 퍼지기 시작한 80년대 초반의 컴퓨터들은 ASCII 코드를 따른 컴퓨터들이었으며, 이를 수입, 조립, 판매하던 국내 기업들에서는 각기 다른 저들만의 한글 코드를 개발하여 운영체제에 넣어 판매하기 시작했다. 이때 사용된 한글코드를 부여하는 방법의 공통점은, **표준 7바이트 ASCII 코드만을 사용하고 나머지 확장 ASCII 영역에 한글 코드를 부여**하여 이를 한글 처리에 필요한 코드로 사용하는 것이다. 즉 첫 비트가 0이면 영어, 1이면 한글 표현에 필요한 연산을 하게끔 처리하는 것이다. 그리고 이들 **128개의 절반 바이트짜리 확장 영역에서는 한글을 모두 표현 할 수 없기 때문에, 2바이트 이상을 조합하여 한글을 처리**한다. 그 방법 중 몇 개를 소개하자면,

※ N 바이트 조합형 한글코드

각각의 자음과 모음을 한 바이트씩 할당하여 이들의 조합으로 한글을 표현한다. 한 글자를 표현하는데 필요한 길이는 2바이트에서 5바이트까지 사용되었다. 그러나 데이터의 측면에서는 초성, 중성, 종성이 뒤섞여있는 꼴이므로, 한 글자 단위의 처리 예를 들어 **정렬이나 글자단위 비교에서는 적합하지 않다**. 현재에는 형태소 분석, 맞춤법 검사 등에 국한되어 이용되고 있다.

N바이트 한 음절 구성 예
가 = ㄱ + ㅏ → 2바이트
열 = ㅇ + ㅕ + ㄹ → 3바이트
권 = ㄱ + ㅜ + ㅓ + ㄴ → 4바이트
삯 = ㅅ + ㅏ + ㄱ + ㅅ → 4바이트
괎 = ㄱ + ㅗ + ㅏ + ㄱ + ㄱ → 5바이트

※ 3 바이트 한글 코드

한 글자를 초성, 중성, 종성으로 나누어 각각 한 바이트씩 사용한다. 2바이트 조합형에 비해 1바이트를 더 표현하기 때문에 당시에는 곧 자취를 감추게 되지만, UNICODE가 나온 이후 첫가끝 조합형코드의 기본이 되며 이는 몇몇 어플리케이션에서 사용하고 있다. (UNICODE는 각 자모가 2바이트를 차지하기 때문에 UNICODE 첫가끝 조합형은 엄밀히 따지면 3바이트 한글코드에 포함이 되지는 않는다.)

※ 자소형 한글 코드

N바이트와 3바이트 한글 코드의 절충형이다. 즉 받침이 없을 때는 2바이트, 받침이 있을 때는 3바이트를 사용한다. 당시에는 거의 사용되지 않았지만, 현재에는 인터넷 정보검색, 맞춤법 검사, 자동 색인 등에서 사용한다.

※ KS 표준 완성형 코드

표준이란 말이 붙은 것은 **1987년에 KSC5601-1987이 발표**되면서부터이다. 완성된 **한글을 순서대로 배치하여 각각에 코드값을 부여**한다. 이때 **2바이트가 쓰이며**, 두 바이트 모두 A1~FE까지만 사용하기 때문에 실제로 사용 가능한 문자는 94*94(8836)자이므로 현대 한글 11172자를 모두 표현할 수가 없다. 따라서 많이 사용하는 한글 2350자, 한자 4888자, 특수문자 1128자, 나머지 470문자만을 순서대로 배정하였다. 이렇게 사용영역이 제한된 이유는 **당시 국제표준이었던 ISO-2022를 따르기 위해**서였다.

KSC5601-1987 예제

가(B0A1), 각(B0A2), 갂(없음), 갃(없음), 간(B0A3), 갅(없음), 갆(없음), 갇(B0A4),

갈(B0A5), 갉(B0A6), 갊(B0A7), 갋(없음), ……, 힡(C650), 힢(C651), 힣(C652)

※ 상용 조합형 코드

역시 **2바이트를 사용**하며, **맨 첫번째 비트를 1로 두고 나머지 비트를 다섯개씩 나누어 각각을 초성, 중성, 종성으로 나눈다.** 즉, MSB(most significant bit)가 0이면, 1바이트만을 읽어서 7bit ASCII 코드로 처리하고, MSB가 1이면 두 바이트를 읽어서 한글로 조합하여 표현한다. 처음에는 회사마다 제각각 다른 코드를 조합하여 사용했으나, 1987년 표준완성형 제정 이후에 삼보조합형코드만이 살아남았으며 이후 이를 상용조합형이란 이름으로 부르게 된다.

조합형 코드 테이블

비트값		초성	중성	종성
10진	2진			
0	00000	<사용불가>	<사용불가>	<사용불가>
1	00001	<채움>0x8400	<사용불가>	<채움>0x8001
2	00010	ㄱ 0x8800	<채움>0x8040	ㄱ 0x8002
3	00011	ㄲ 0x8C00	ㅏ 0x8060	ㄲ 0x8003
4	00100	ㄴ 0x9000	ㅐ 0x8080	ㄳ 0x8004
5	00101	ㄷ 0x9400	ㅑ 0x80A0	ㄴ 0x8005
6	00110	ㄸ 0x9800	ㅒ 0x80C0	ㄵ 0x8006
7	00111	ㄹ 0x9C00	ㅓ 0x80E0	ㄶ 0x8007
8	01000	ㅁ 0xA000	<사용불가>	ㄷ 0x8008
9	01001	ㅂ 0xA400	<사용불가>	ㄹ 0x8009
10	01010	ㅃ 0xA800	ㅔ 0x8140	ㄺ 0x800A
11	01011	ㅅ 0xAC00	ㅕ 0x8160	ㄻ 0x800B
12	01100	ㅆ 0xB000	ㅖ 0x8180	ㄼ 0x800C
13	01101	ㅣ 0xB400	ㅗ 0x81A0	ㄽ 0x800D
14	01110	ㅈ 0xB800	ㅘ 0x81C0	ㄾ 0x800E
15	01111	ㅉ 0xBC00	ㅙ 0x81E0	ㄿ 0x800F
16	10000	ㅊ 0xC000	<사용불가>	ㅀ 0x8010
17	10001	ㅋ 0xC400	<사용불가>	ㅁ 0x8011
18	10010	ㅌ 0xC800	ㅚ 0x8240	<ㄻ> 0x8012
19	10011	ㅍ 0xCC00	ㅛ 0x8260	ㅂ 0x8013
20	10100	ㅎ 0xD000	ㅜ 0x8280	ㅄ 0x8014
21	10101	<한자>0xD400	ㅝ 0x82A0	ㅅ 0x8015
22	10110	<한자>0xD800	ㅞ 0x82C0	ㅆ 0x8016
23	10111	<한자>0xDC00	ㅟ 0x82E0	ㅇ 0x8017
24	11000	<한자>0xE000	<사용불가>	ㅈ 0x8018
25	11001	<한자>0xE400	<사용불가>	ㅊ 0x8019
26	11010	<한자>0xE800	ㅠ 0x8340	ㅋ 0x801A
27	11011	<한자>0xEC00	ㅡ 0x8360	ㅌ 0x801B
28	11100	<한자>0xF000	ㅢ 0x8380	ㅍ 0x801C
29	11101	<한자>0xF400	ㅣ 0x83A0	ㅎ 0x801D
30	11110	<한자>0xF800	<ㆍ> 0x83C0	<ㅇ> 0x801E
31	11111	<사용불가>	<ㆍㅣ> 0x83E0	<사용불가>

http://metalliza.kr

조합 방법 예제 - OR 연산(|)을 사용한다. (0x는 16진수라는 C언어에서의 표기)
가 : ㄱ+ㅏ+<채움> : 0x8800 | 0x8060 | 0x8001 = 0x8861
각 : ㄱ+ㅏ+ㄱ : 0x8800 | 0x8060 | 0x8002 = 0x8862
훤 : ㅎ+ㅝ+ㄴ : 0xD000 | 0x82A0 | 0x8005 = 0xD2A5

분해 방법 예제 - AND 연산(&)으로 마스킹 하는 방법을 쓴다.
각-초성 : 0x8862 & 0xFC00 = 0x8800 : ㄱ
각-중성 : 0x8862 & 0x83E0 = 0x8060 : ㅏ
각-종성 : 0x8862 & 0x801F = 0x8002 : ㄱ

분해된 코드는 단독으로 출력할 수 없고, <채움>코드와 다시 조합되어야 출력.
ㄱ : ㄱ+<채움>+<채움> : 0x8800 | 0x8040 | 0x8001 = 0x8841

※ 표준 조합형 코드

1992년, 정부는 완성형의 한계를 인정하고 조합형의 필요성을 인식하여 KSC5601-1992란 이름으로, 기존 완성형코드에 조합형코드를 추가하여 공동표준으로 제정한다. 그러나 완성형코드로 작성된 문서와의 호환성과, 기존 상용조합형과의 약간의 차이점을 가지고 있는 문제가 겹쳐서 사실상 거의 쓰이지 않게 되었다. 다만 표준이라는 이름으로 현재 조합형을 쓰고 있는 몇몇 어플리케이션은 이 표준 조합형 방식을 채택하고 있다고 한다.

※ 표준 한글 코드 제정과 문제점

1980년대 중반, 각 기업마다 저마다의 코드를 사용하기에 서로간의 문서의 호환성등, 문제가 불거지게 된다. 이에 표준한글 코드의 필요성이 대두되었는데, 정부는 1987년, 많은 논란에도 불구하고 KSC5601-1987 라는 이름으로 KS완성형을 표준으로 제정 발표하게 되었다. KS완성형이 표준이 된 가장 큰 이유는 당시의 국제 표준이었던 ISO2022를 준수하였다는 점이다. 지금에 와서는 이 ISO2022가 거의 폐지된거나 다름없으며, UNICODE로 대체되어가는 상황을 고려해볼 때 이때의 표준 제정은 아직까지도 문제가 되는 커다란 오점이다.

KS표준 완성형 코드의 가장 커다란 문제점은 모든 한글을 표현할 수 없다는 점이다. 한때 인기있었던 '똠방각하'에서 '똠'이란 글자는 이 표준 내에 존재하지 않는다.

KS표준 완성형 한글코드의 두번째 문제점은 역시 완성형코드 고유의 문제점인, 한글자모로 분해하기가 어렵다는 점이다. 한글자모로 분해하는 작업을 위해서는 어차피 조합형을 써야 하며, 완성형을 조합형으로 바꿔주어야 하는 수고가 필요해진다.

세번째 문제점은, 국제 표준을 따랐다고는 하나, 이것이 국제 호환으로 이어지지는 않는다는 점이다. 이는 ISO2022의 문제점이며, 어차피 각국마다 자신들의 언어로 같은 코딩 영역을 쓰기 때문에 각국에서 쓰는 인코딩에 따라 글씨가 깨지기도, 보이기도 한다. 일본이나 중국, 유럽같은 웹사이트를 돌아다니며 인코딩이 맞지 않아서 글씨가 깨져서 엉뚱한 한글이나 네모모양이 나왔던 것을 한번쯤은 다 경험해 봤을 것이다. 이들 문서도 다 ISO2022를 따랐음에도 불구하고 말이다. 마찬가지로 저들 국가의 인코딩으로 KS5601문서를 보았을 때는 엉뚱한 글자가 나오게 되어있다.

어차피 한국 내에서만 쓸 수 있는, 국내용 코드라면, 굳이 국제표준을 따라야 했을까? 한글을 최대한 잘 표현할 수 있는 조합형을 놔두고 저런 먹다 남은 옥수수마냥 이가 빠져있는 완성형을 채택했어야 했을까? 1995년 MS Windows 95가 발표되면서 이 문제점은 더더욱 꼬이게 된다.

컴퓨터 한글 코드 - 3. MS와 한글코드

※ MS-DOS 시절의 한글

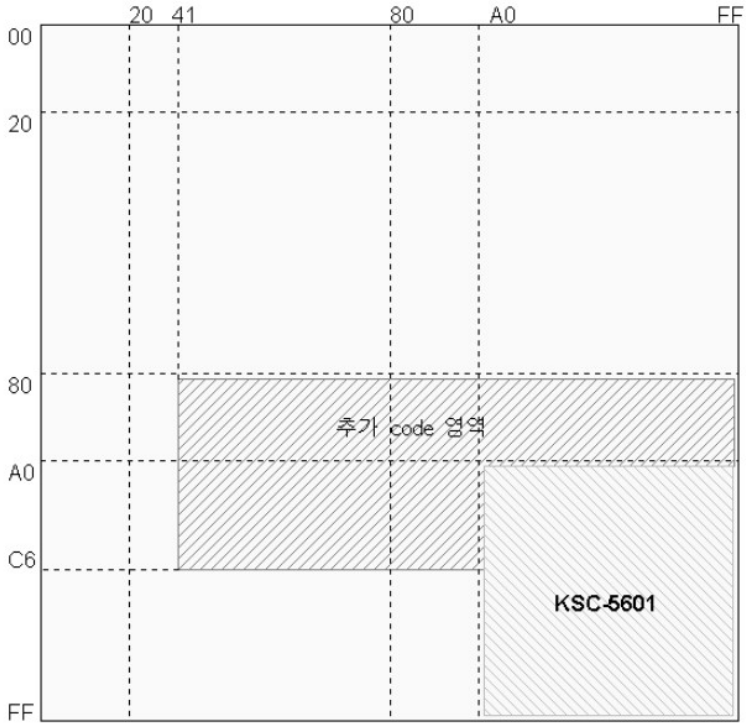
1980년대 중반, 컴퓨터를 구매하면 으레 DOS가 공짜로 깔려오던 시절, 90%이상의 컴퓨터에 MS-DOS가 깔려나오고 있던 때다. 당시는 영문도스 밖에 없었으며, 한글 지원을 위해서 각 컴퓨터 판매 업체와 프로그래머들이 램상주 프로그램(DOS는 멀티태스킹이 되지 않고 동시에 하나의 프로그램밖에 실행할 수 없는데, 메모리에 로딩을 해놓고 다른 프로그램이 돌아가는 동안에도 기능을 발휘하게끔 하는, 일종의 쉘 프로그램) 형식으로 각각 저마다의 한글을 지원, 배포하였다. 이중 **삼보조합형 한글**이 가장 효율이 좋았는데, 초기의 한글 MS-DOS는 이 조합형을 사용하여 한글을 지원했다.

그러나 1989년, 결국 MS도 조합형을 포기하고, KS 완성형을 채택한 DOS를 발표한다. 주목할 것은, 이는 MS의 선택이 아닌 정부의 표준 완성형 보급정책이라는 것이다. 이 덕분에, PC통신의 황금기를 맞이하게 된 1990년대 초반, 2350개밖에 표현할 수 없는 한글로 우리는 KETEL(HiTEL), 천리안, 나우누리 등에서 울고 웃었던 것이다. 정부가 의도한대로…….

물론 당시 가장 많이 쓰여졌던 통신프로그램인 '이야기'에서는 조합형 한글을 지원했지만, 서버가 완성형을 지원하며 그에 따라 작성되는 게시판 글들 역시 완성형으로 저장되기 때문에, 조합형 한글은 점차 설 자리를 잃어갔다.

※ MS-Windows 확장 완성형 한글

1995년 MS는 WINDOWS 95를 발표하였다. 당시 정부는 UNICODE와 새로운 한글 표준을 준비하고 있을 때였는데, MS에서 발표한 이 운영체제는 신기하게도 '똠방각하'나 '펲시맨'등의 KS 완성형 2350개의 글자를 벗어난 글자를 표현할 수 있었다. 이는 MS에서 독단으로 KS완성형에서 쓰이지 않는 영역에 나머지 8822개의 글자를 구겨넣다시피 배치했기 때문이며, 이때 만들어진 코드가 현재까지도 쓰이고 있는 **확장 완성형 한글코드(MS949, 통합형 한글코드라고 불리기도 함)**이다.



MS 확장 완성형 한글의 추가된 한글 부호영역

확장 완성형 한글 역시 어찌보면 MS사의 어쩔 수 없는 선택일 수도 있다. 당시 추가 재정된 한글코드가 존재하지 않았으며(UNICODE 1.0은 KS완성형과 같았으며, UNICODE 1.1은 빈약했고, UNICODE 2.0은 발표직전이었을 무렵), 무엇보다도 기존 KS 완성형(KSC-5601)과의 호환성을 유지하기 위해서였다. 이 호환성 유지를 위해서 치명적인 문제가 발생한다.

앞에서 설명하였듯이, 또 위 그림에서 보듯이 KS완성형은 A1A1~FEFE까지 사용되며 이중 한글은 B0A1부터 시작되어 2350자가 연속적으로 배치된다. 이때 빠진 8822자는 사이사이로 끼어들지 못하고 확장영역으로 배치가 되게 되는데, 이때 한글 순서로 배치되어야 할 코드들의 연속성이 무너져 버리는 것이다.

가 - 0xB0A1
각 - 0xB0A2
갂 - 0x8141
갃 - 0x8142
간 - 0xB0A3
갅 - 0x8143
갆 - 0x8144
갇 - 0xB0A4
:

:
히 - 0xC8F7
힉 - 0xC8F8
힊 - 0xC59F
힋 - 0xC5A0
힌 - 0xC8F9
힍 - 0xC641
힎 - 0xC642
힏 - 0xC643
:

위에서 보듯, 검은색으로 표현된 표준영역과 붉은색으로 표현된 **확장영역 간에 연속성이 서로 제각각임**을 알 수 있다. 이렇게 되면 사전 검색이나 순차정렬에서 제대로 된 결과 값이 나오지 않는 것은 당연한 일이다. '돔(0xB5BC)'보다 '똠(0x8C63)'이 순서상 뒤여야 하는데 똠의 코드값이 돔의 코드값보다 앞으로 배치가 되어 있는 것이다. 이를 제대로 구현하기 위해서 조합형으로 변환하여 처리하는 방법이 있을 수 있지만, 비효율적임은 분명하다. (조합형으로 변환하기 위해서는 확장 완성형 코드 테이블의 데이터값을 모두 가지고 있어야 한다.

또 다른 문제는, 이 비효율적인 임시로 책정되다시피 한 한글코드가 아직까지도 쓰이고 있다는 점이다. MS 독단으로 결정한 이 코드는 어느 표준에도 속해있지 않다. 심지어 KS완성형이 준수하려던 ISO2022 마저 어기고 있다. 그러나 MS-WINDOWS에서 한번 사용하기 시작한 이 한글코드는 한글 OS의 99%를 차지하는 MS-WINDOWS에서 그 호환성을 유지하기 위하여 계속 사용되어 왔으며, 앞으로도 그럴 가능성이 높다. 엉뚱한 한글코드가 불완전한 KS완성형과 호환된다는 이유만으로 사실상의 표준으로 자리잡은 것이다.

컴퓨터 한글 코드 - 4. 유니코드와 인터넷 한글

마치 세벌식의 존재를 몰라도 두벌식으로도 뭐가 불편한지 모르고 타이핑하듯이 말이다. 그런데 그러한 일반 유저들도, 간혹 이러한 경험을 한번쯤은 했을 것이다.

상황1

"사진을 인터넷 게시판에 업로드 했는데 사진이 보이질 않아요. 왜 그렇죠?"

"혹시 파일 이름이 한글로 되어있나요?"

"네 그런데요?"

"가능하면 파일 이름에 한글이 들어가지 않도록 해주시고요.

한글로 된 파일을 보시려면 웹브라우저에서 도구-인터넷옵션-고급 메뉴로 들어가셔서요. 'URL을 항상 UTF-8로 보냄.' 이라고 쓰여 있는 옵션을 끄세요."

"왜 이렇게 복잡하죠?"

상황2

"메일을 하나 받았는데 글씨를 전혀 알아볼 수 없어요. 외국에서 보낸건가요?"

"그 메일을 보시려면 인코딩을 ……."

왜 한글로 된 파일 하나 인터넷으로 다운 받으려는데 평소 전혀 들어가 볼 일도 없는 '고급'메뉴까지 들어가야 할 정도로 복잡하고 어려운 것인가? 같은 한글로 된 이메일인데 왜 인코딩 변환까지 해가면서 봐야 한글을 읽을 수 있는 것인가?

그것은, 인터넷에서 사용하고 있는 한글 코드가 현재 알려진 2~3개가 모두 사용되고 있기 때문이다. 다행히 최근에 제작되는 인터넷 자료실이나 게시판, 이메일등은 이러한 것을 몰라도 알아서 처리해주니 큰 걱정은 하지 않아도 된다. 이러한 복잡한 상황은 고스란히 웹 프로그래머의 몫으로 돌아가는 것이고 이를 제대로 알아야만 중급 프로그래머로서 자리를 잡게 될 것이다.

현재 인터넷에서 사용되는 **한글 코드는 크게 세 가지가 존재**한다.

※ KS 완성형 한글 (KSC 5601, EUC-KR)

역시 2350개의 글자밖에 쓸 수 없는 코드이다. 다음 블로그 등에서 이 코드가 쓰이는데, 이때 표준을 벗어난 한글은 아래에서 설명할 UNICODE로 강제 변환시키는 코드를 사용하게 된다. '똠방각하'라고 저장을 하고자 할 때 서버의 데이터 값은 UNICODE 호출을 위해 '똠방각하'라고 변환되어 저장된다.

※ MS 확장 완성형 한글 (MS949, CP949)

바로 앞장에서 설명한, 어느 표준에도 포함되지 않은 MS 전용한글 코드이나, 통합 한

글코드라는 이름으로 거의 표준화되다시피 하여 JAVA 등에서 제공된다. 일부 MS이외의 OS에선 이 코드로 작성된 문서를 보지 못할 수도 있다. (내용 정정. JAVA에서는 내부적으로는 UNICODE의 케릭터 셋이 사용됩니다. 다만 입출력을 UNICODE로 하기 위해서는 UNICODE, 혹은 UTF에 관련된 입출력 메소드를 이용하여야 합니다.)

※ UNICODE (UTF-7, UTF-8, UTF-16)

엄밀히 말하면 유니코드와 UTF의 개념은 틀리지만 UTF란 UNICODE를 표기하기 위한 방법이므로 함께 설명하겠다.

먼저 **유니코드는** 국제 표준화 기구는 아니며 **컴퓨터 관련 업계를 중심으로, 다국어 언어를 효율적으로 처리에 필요한 코드 체계를 제정하기 위해서 1989년에 콘소시엄 형태로 설립된 회사**이다. 1991년 말, 유니코드 1.0을 제정하였는데, 이때 한글 코드는 KSC5601-1987만을 포함시켰다. 이후 제정된 유니코드 1.1에서는 조합형한글을 부분적으로만 수용하여 발표하였으나 이는 사용조차 할 수 없는 형태로 이루어졌다.

1995년이 되어서야 유니코드 2.0이 제정되었다. 이 유니코드는 한 글자가 차지하는 용량을 2바이트를 기본으로 하며, 한중일 통합 한자 확장 등의 몇몇 영역은 2바이트가 넘는 코드값이 쓰이기도 한다. 기존 7비트 ASCII영역 또한 전체적인 호환을 위해서 2바이트 영역에 배치되었다. 즉, 대문자 알파벳은 0041~005A에, 소문자 알파벳은 0061~007A에 배치되는 등 2바이트를 기본 구조로 사용한다. 2바이트를 넘는 문자를 제외하고 2바이트 문자만 고려한다면 유니코드는 약 6만 개의 글자를 표현할 수 있는데, 이중 한글을 위해서 만 개가 넘는 코드를 배정받은 것은 고무적인 일이다.

유니코드 2.0에서는 11172자의 완성형 한글코드를 담을 수 있는 영역(AC00~D7AF)과 조합형에 사용할 수 있는 초성, 중성, 종성의 옛한글을 포함한 한글 자모 영역(1100~11FF), 그리고 기존 완성형과의 호환성을 위한 한글 자모 영역(3130~318F)등을 한글을 위해 배정받았다. 이 유니코드는 발표되자마자 **국내표준으로도 채택되었으며, KSC 5700이라는 이름으로 제정**되어있다.

유니코드의 완성형 한글은 지금까지 나온 완성형 한글 중 가장 이상적이다. 11172자의 현대한글에서 조합가능한 모든 글자가 사전 순서대로 나열되어있다. 이 말은 즉 완성형과 조합형간의 변환이 별도의 변환테이블이 없이 코드값의 수식 계산만으로도 가능하다는 이야기다.

유니코드 완성형 한글의 계산방법은, 우선 한글 자모의 인덱스를 다음과 같이 둔다.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
초	ㄱ	ㄲ	ㄴ	ㄷ	ㄸ	ㄹ	ㅁ	ㅂ	ㅃ	ㅅ	ㅆ	ㅇ	ㅈ	ㅉ	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ									
중	ㅏ	ㅐ	ㅑ	ㅒ	ㅓ	ㅔ	ㅕ	ㅖ	ㅗ	ㅘ	ㅙ	ㅚ	ㅛ	ㅜ	ㅝ	ㅞ	ㅟ	ㅠ	ㅡ	ㅢ	ㅣ							
종	X	ㄱ	ㄲ	ㄳ	ㄴ	ㄵ	ㄶ	ㄷ	ㄹ	ㄺ	ㄻ	ㄼ	ㄽ	ㄾ	ㄿ	ㅀ	ㅁ	ㅂ	ㅄ	ㅅ	ㅆ	ㅇ	ㅈ	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ

그러면 조합된 완성형 한글의 코드값에 대한 공식이 바로 나온다.
(유니코드 완성형 한글코드) = 0xAC00 + 28*21*(초성인덱스) + 28*(중성인덱스) + (종성인덱스)

예를 들어, '한'이라는 글자의 코드값은
0xAC00 + 28*21*(18) + 28*(0) + (4) = 0xD55C 가 된다.

반대로, 완성된 코드값을 조합형으로 분해하는 것도 간단하다.
(int(A) : A의 소숫점 이하를 버린 값, 정수)
(mod(A,B) : A를 B로 나눈 나머지)

(초성 인덱스) = int(((유니코드)-0xAC00) / 28*21)
(중성 인덱스) = int(mod((유니코드)-0xAC00, 28*21) / 28)
(종성 인덱스) = mod((유니코드)-0xAC00, 28)

예를 들어, '특'이라는 글자의 유니코드 값은 0xD2B9 이며, 이를 계산하면
0xD2B9-0xAC00 = 0x26B9 → 9913 28*21 = 588
(초성인덱스) = int(9913 / 588) = 16 (ㅌ) mod(9913, 588) = 505
(중성인덱스) = int(505 / 28) = 18 (ㅡ)
(종성인덱스) = mod(9913, 28) = 1 (ㄱ)
이 됨을 알 수 있다.

이렇듯 유니코드는 완성형 한글임에도 불구하고 KS 완성형처럼 빠진 글자도 없으며, MS 완성형처럼 순서가 뒤죽박죽이 아닌 덕분에, 마치 조합형처럼 간단한 수식으로 한글 자모를 분해, 조합할 수 있는 것이다. 사전적인 배열이므로, 검색이나 정렬 등에서 처리가 훨씬 쉬워졌음은 물론이다.

그런데, 위에서 언급한 대로, 유니코드에서는 조합형을 위한 영역을 따로 두었다고 하였다.

유니코드 조합형을 위한 한글자모들.

초성 :
ᄀᄁᄂᄃᄄᄅᄆᄇᄈᄉᄊᄋᄌᄍᄎᄏᄐᄑᄒᄓᄔᄕᄖᄗᄘᄙᄚᄛᄜᄝᄞᄟᄠᄡᄢᄣᄤᄥᄦᄧᄨᄩᄪᄫᄬᄭᄮᄯᄰᄱᄲᄳᄴᄵᄶᄷᄸᄹᄺᄻᄼᄽᄾᄿᅀᅁᅂᅃᅄᅅᅆᅇᅈᅉᅊᅋᅌᅍᅎᅏᅐᅑᅒᅓᅔᅕᅖᅗᅘᅙᅚᅛᅜᅝᅞᅟᅠᅡᅢᅣᅤᅥᅦᅧᅨᅩᅪᅫᅬᅭᅮᅯᅰᅱᅲᅳᅴᅵᅶᅷᅸᅹᅺᅻᅼᅽᅾᅿᆀᆁᆂᆃᆄᆅᆆᆇᆈᆉᆊᆋᆌᆍᆎᆏᆐᆑᆒᆓᆔᆕᆖᆗᆘᆙᆚᆛᆜᆝᆞᆟᆠᆡᆢᆣᆤᆥᆦᆧᆨᆩᆪᆫᆬᆭᆮᆯᆰᆱᆲᆳᆴᆵᆶᆷᆸᆹᆺᆻᆼᆽᆾᆿᇀᇁᇂ
중성 :
ᅡᅢᅣᅤᅥᅦᅧᅨᅩᅪᅫᅬᅭᅮᅯᅰᅱᅲᅳᅴᅵᅶᅷᅸᅹᅺᅻᅼᅽᅾᅿᆀᆁᆂᆃᆄᆅᆆᆇᆈᆉᆊᆋᆌᆍᆎᆏᆐᆑᆒᆓᆔᆕᆖᆗᆘᆙᆚᆛᆜᆝᆞᆟᆠ
종성 :
ᄀᄁᄂᄃᄄᄅᄆᄇᄈᄉᄊᄋᄌᄍᄎᄏᄐᄑᄒᄓᄔᄕᄖᄗᄘᄙᄚᄛᄜᄝᄞᄟᄠᄡᄢᄣᄤᄥᄦᄧᄨᄩᄪᄫᄬᄭᄮᄯᄰᄱᄲᄳᄴᄵᄶᄷᄸᄹᄺᄻᄼᄽᄾᄿᅀᅁᅂᅃᅄᅅᅆᅇᅈᅉᅊᅋᅌᅍᅎᅏᅐᅑᅒᅓᅔᅕᅖᅗᅘᅙᅚᅛᅜᅝᅞᅟᅠᅡᅢᅣᅤᅥᅦᅧᅨᅩᅪᅫᅬᅭᅮᅯᅰᅱᅲᅳᅴᅵᅶᅷᅸᅹᅺᅻᅼᅽᅾᅿᆀᆁᆂᆃᆄᆅᆆᆇᆈᆉᆊᆋᆌᆍᆎᆏᆐᆑᆒᆓᆔᆕᆖᆗᆘᆙᆚᆛᆜᆝᆞᆟᆠᆡᆢᆣᆤᆥᆦᆧᆨᆩᆪᆫᆬᆭᆮᆯᆰᆱᆲᆳᆴᆵᆶᆷᆸᆹᆺᆻᆼᆽᆾᆿᇀᇁᇂ

비어있는 코드를 제외하면 위의 자모들은 유니코드에서 0x1100~0x11FF 안에 위의 순서대로, 초성 90자, 중성 66자, 종성 82자와, 중성 채움, 종성 채움의 모두 240자가 배치가 되어있다. 말 그대로 조합형 코드이기 때문에, 초성+중성+종성을 합한 유니코드 세 글자는 완성된 한글자로 인식될 수 있다. 예를 들어 'ㄱ(0x1100)' + 'ㅏ(0x1161)' + 'ㄱ(0x11A8)' 은 '각(0xAC01)'이란 글자와 같이 인식될 수 있다. 실제로, 유니코드 한글의 재정 원칙상, 위의 조합형 코드와 완성형 코드는 동일한 글자로 처리해야 한다.

헌데, **여기에는 중요한 문제가 몇가지 있다**.
첫째는 유니코드 한글조합형을 지원하는 시스템이 OS 차원에서는 아예 없으며, 응용프로그램에서도 거의 전무하다는 것이다. 0x1100116111A8 = 0xAC01 이어야 하는데 실제 이 조합형을 입력했을 경우 대부분 'ㄱㅏㄱ'이라고만 찍혀 나오지 '각'이라고 출력되어 나오는 소프트웨어가 거의 없다는 것이다. 현재 유니코드를 지원하는 대부분의 프로그램에서는 한글을 오직 완성형으로만 인식하며, 한글자모 단독출력도 기존 완성형과의 호환을 위해서 호환용 자모 영역(3130~318F)을 이용하고 있다.

둘째는 옛한글 지원의 문제이다. 유니코드에는 완성형 한글이 오로지 현대 한글로만 이루어져있으며, 옛한글을 사용하기 위해서는 조합형을 사용해야 한다. 사실 위의 옛한글 자모로 조합가능한 모든 글자가 무려 50만 자 가까이 되니 이를 완성형으로 표현한다는 것이 사실상 불가능하기도 하다. 그러나 금방 언급했듯이, 이 조합형을 지원하는 소프트웨어가 거의 없으며, 한글자모 자체가 현대한글과 옛한글로 나뉘어져 사전순서가 맞지 않는다는 문제도 있다. 위의 옛초성 중에서 ㅱ 과 ㅹ 은 ㅁ과 ㅂ사이에 들어와야 순서가 맞지 않겠는가. 또한 옛한글의 자모가 위의 240개가 전부가 아니며, 104자가 추가된 344자가 있어야 현재 알려진 모든 옛한글을 표현할 수 있다고 한다. 그러나 255자로 제한된 유니코드 조합형의 한글자모 영역에 이 104자를 추가할 수 있는 방법이 없다. 이는 유니코드를 준비하던 사람들의 사전 준비가 부족한 탓이라고 하니, 한글 코드표준화의 역사를 보면 한숨만 나올 뿐이다.

셋째로, 조합 방법의 문제이다. 조합 방법에 대한 구체적인 표준이 불분명하다. 어느 문헌에서는 첫가끝 고정바이트 방식을 사용한다고 명시 되어 있으나 또 어느 문헌에서는 이에 대한 언급이 없다. 채움 코드가 있는 것으로 봐서는 첫가끝 고정바이트 방

식을 사용하는 것이 맞겠지만, 초성에 대한 채움 코드도 불명확하다. 공식 문서에서는 0x115F가 초성 채움코드라는 것을 확인했는데, 아래아 한글에서 유니코드표를 확인해보면 알겠지만, 이는 그루지야어(Georgian) 라는 문자 영역이며, 이는 국제 공식적으로 쓰일 경우 문제가 있을 수 있다. 그런데 사실 채움코드가 제일 첫 초성인 'ㄱ(0x1100)'의 앞으로 오게끔 하는게 정답일 것이다. 그러면 이것 또한 'ㄱ'앞을 비워놓지 않은, 유니코드를 준비하던 사람들을 탓해야 할 것 같다.

마지막으로, 미완성 한글의 처리이다. 미완성 한글이란, 한 글자에서 초성이나 중성이 빠졌거나, 한글 자모 단독으로 쓰이는 경우인데, 특히 종성자음만 쓰일 경우 받침임을 확실히 알 수 있도록 자음이 약간 아래쪽으로 표시되게끔 하는 것이다. 현재 이러한 표기방법은 아래아한글과, 날개셋 편집기에서만 지원한다.

이는 조합형 지원과, 조합 방법 문제, 그리고 완성형 한글영역의 복합적인 문제가 될 수 있다. 소프트웨어가 유니코드 조합형을 지원하고, 조합 방법에서 채움코드 등이 정확히 표준화 된다면 이를 표현하는데 문제가 되지 않겠지만, 현재로서는 역시 특정 프로그램에서만 쓸 수 있는 방법이다. 또 이를 표기하기 위해서 완성형은 11,172자가 아니라, 이들 미완성 조합을 포함한 12,319자가 되어야 한다는 이야기도 있다.

UTF 란 Unicode Transformation Format의 약자로서, 말 그대로 유니코드를 저장하거나, 보내고 받기위한 방법을 의미한다. UTF-7, UTF-8, UTF-16등이 있으며 각각 알고리즘이 틀리다. 뒤의 숫자는 코드 처리를 위한 bit수를 의미한다.

UTF-7은 이메일 시스템에 많이 쓰이는 코드이며, 최근에는 사용량이 줄어드는 추세이다.

현재 가장 많이 쓰이고 있는 것은 UTF-8인데, 그 이유는 7bit 표준 ASCII와 완벽하게 호환이 되기 때문이다. 즉, 2바이트가 기본이 되는 유니코드를 ASCII로만 이루어진 문서에 적용하게 되면 용량이 모조리 두 배가 되기 때문에, 이를 위해 고안된 것이 UTF-8이다. UTF-8은 가변길이 방식으로, ASCII등의 문자를 표현할 때는 큰 효과를 보지만, 동양권의 문자 등을 표현할 때는 많은 용량을 차지할 수 있다. 완성형 한글 영역은 3바이트를 차지하게 된다.

UTF-16은 유니코드를 2바이트가 기본이 되는 유니코드는 그대로 저장하면서, 2바이트가 넘는 코드는 4바이트로 확장하여 저장하는 방식이다.

코드 범위 (십육진법)	UTF-16BE 표현 (이진법)	UTF-8 표현 (이진법)	설명
000000-00007F	00000000 0xxxxxxx	0xxxxxxx	ASCII와 동일한 범위
000080-0007FF	00000xxx xxxxxxxx	110xxxxx 10xxxxxx	첫 바이트는 110 또는 1110으로 시작하고, 나머지 바이트들은 10으로 시작함
000800-00FFFF	xxxxxxxx xxxxxxxx	1110xxxx 10xxxxxx 10xxxxxx	
010000-10FFFF	110110yy yyxxxxxx 110111xx xxxxxxxx	11110zzz 10zzxxxx 10xxxxxx 10xxxxxx	UTF-16 서로게이트 쌍 영역 (yyyy = zzzzz - 1). UTF-8로 표시된 비트 패턴은 실제 코드 포인트와 동일하다.

처음에 언급한, 파일이름이 한글로 된 자료를을 볼 때 'URL을 항상 UTF-8로 보냄.'을 해제해야 하는 이유는 이때 표현된 한글을 유니코드로 변환시키기 때문에 KS완성형으로 되어있는 본래의 파일 이름과 매치가 되지 않기 때문이다. 요즘엔 이러한 점 때문에 업로드 파일을 저장할 때 파일이름을 utf-8로 변환하여 저장하는 서버가 점점 늘고 있는 추세이므로, 사용자들은 점점 신경을 쓸 필요가 없어지고 있다.

유니코드는 현재 5.0 버전까지 발표가 되었으며, 전세계의 모든 문자를 표현할 수 있기 때문에 점차 세계 표준으로 자리를 잡아가고 있다.(ISO 10646) 또한, 기존 KS완성형이 갖고 있던 약점이 많이 개선되었기 때문에, 한글을 처리를 위해서도 좋은 선택이 되고 있다. MS에서도 Window 98버전부터 유니코드를 내장하여 왔으며, 점차적으로 유니코드의 사용량을 늘려나가고 있다. 다만, 아직까지도 KS 완성형이 쓰이고 있는 이유인 기존 데이터와의 호환성, 그리고 유니코드 조합형 한글 표현의 OS 차원에서의 지원 등, 앞으로도 넘어야 할 과제들이 많다.