对 HL 中。8080 中的乘法子程序如下：

```
Multiply:   PUSH    PSW          ; 将要修改的寄存器的原内容保存至堆栈
            PUSH    BC

            SUB     H, H         ; 将 HL（即乘积）置为 0000h
            SUB     L, L

            MOV     A, B         ; 乘数送至累加器 A
            CPI     A, 00h       ; 如果累加器中的值是 0，则结束
            JZ      AllDone

            MVI     B, 00h       ; 将 BC 的高字节置为 0

MultLoop:   DAD     HL, BC       ; 将 BC 的内容加到 HL
            DEC     A            ; 乘数减 1
            JNZ     MultLoop     ; 如果不为 0，则跳转

AllDone:    POP     BC           ; 将堆栈中保存的数据恢复至寄存器
            POP     PSW
            RET                  ; 返回
```

注意，上述子程序的第一行有一个标志 Multiply。当然，实际上这个标志对应着子程序在存储器中的起始地址。该子程序在开始处使用了两个 PUSH 指令，这是因为通常在子程序的起始处要保存程序用到的寄存器。

保存寄存器后，子程序下面要做的是把寄存器 H 和 L 置 0。尽管可以使用 MVI（转移立即数）指令代替 SUB 指令来实现该操作，但这样会用到 4 个字节而不是 2 个字节的指令。子程序执行成功后，运算结果会保存到寄存器对 HL 中。

接下来子程序把寄存器 B 中的数（即乘数）转移到累加器 A，并判断该数是否为 0。如果为 0，则乘法子程序结束，因为乘数为 0。由于寄存器 H 和 L 已经为 0，所以子程序可以使用 JZ（Jump If Zero）指令跳转到程序最后的两条 POP 指令。

如果乘数不是 0，子程序会把寄存器 B 置为 0。现在寄存器对 BC 存放的是 16 位的被乘数，而累加器中存放的是乘数。接下来 DAD 指令会把 BC（被乘数）加到 HL（运算结果）中。A 中的乘数减 1，如果结果不为 0，则执行 JNZ（非零跳转）指令，该指令会使 BC 再次加到 HL。这个循环会继续执行，直到循环的次数等于乘数为止（当然也可以

利用 8080 的移位指令编写一个更有效率的乘法子程序）。

在程序中使用如下指令来调用这个乘法子程序，例如，把 25h 和 12h 相乘：

```
MVI     B, 25h
MVI     C, 12h
CALL    Multiply
```

CALL 指令把 PC 的值保存到堆栈中，被保存的这个值是 CALL 指令的下一条指令的地址，然后 CALL 指令将使程序跳转到标志为 Multiply 的指令，即子程序的起始处。当子程序得到计算结果后，执行 RET（返回）指令，该指令使保存在堆栈的 PC 的值弹出，并重新设置到 PC，之后程序将继续执行 CALL 指令后面的指令。

8080 指令集包括条件 CALL 指令和条件 Return 指令，但它们使用的频率比条件跳转指令小得多。下面的表格完整地列出了这些指令。

条件	操作码	指令	操作码	指令	操作码	指令
None	C9	RET	C3	JMP aaaa	CD	CALL aaaa
Z not set	C0	RNZ	C2	JNZ aaaa	C4	CNZ aaaa
Z set	C8	RZ	CA	JZ aaaa	CC	CZ aaaa
C not set	D0	RNC	D2	JNC aaaa	D4	CNC aaaa
C set	D8	RC	DA	JC aaaa	DC	CC aaaa
Odd parity	E0	RPO	E2	JPO aaaa	E4	CPO aaaa
Even parity	E8	RPE	EA	JPE aaaa	EC	CPE aaaa
S not set	F0	RP	F2	JP aaaa	F4	CP aaaa
S set	F8	RM	FA	JM aaaa	FC	CM aaaa

正如你大概所了解的，存储器并不是连接在微处理器上的唯一设备。一个完整的计算机系统通常需要输入/输出设备（I/O）以实现人机交互。输入/输出设备通常包括键盘和显示器等。

微处理器是如何与外围设备（peripheral，除存储器外，与微处理器连接的所有设备都可以称为外围设备）互相通信的呢？外围设备配备了与存储器类似的接口，微处理器通过与某种外围设备对应的特定地址（即接口）对其进行读写操作。在某些微处理器中，外围设备实际上占用了一些通常用来寻址存储器的地址，这种结构称作内存映像 I/O（memory-mapped I/O）。但在 8080 中，除了常规的 65536 个地址外，另外增加了 256 个地

址专门用来访问输入/输出设备，它们被称作 I/O 端口（I/O ports）。I/O 地址信号标记为 $A_0 \sim A_7$，但 I/O 的访问方式与存储器的访问方式不同，两者的区分由 8228 系统控制芯片的锁存信号来标识。

OUT（输出）指令把累加器中的内容写入到紧跟该指令后的字节所寻址的端口（port）。IN（输入）指令把一个字节从端口读入到累加器。它们的格式如下所示。

操作码	指令
D3	OUT PP
DB	IN PP

外围设备有时候需要获得处理器的注意。例如，当你按下键盘的某个键时，处理器应该马上注意到这个事件。这个过程由一个称为中断（interrupt）的机制实现，这是一个由外围设备产生的信号，连接至 8080 的 INT 输入端。

但是，当 8080 复位后，就不再响应中断。程序必须执行 EI（Enable Interrupt）指令来允许中断，然后执行 DI（Disable Interrupts）禁止中断。这两条指令如下所示。

操作码	指令
F3	DI
FB	EI

8080 的 INTE 输出信号用来指明何时允许中断。当外围设备需要中断微处理器的当前工作时，它需要把 8080 的 INT 输入信号置为 1。8080 通过从存储器中取出指令来响应该中断，同时控制信号指明有中断发生。外围设备通常提供下列指令来响应 8080 微处理器。

操作码	指令	操作码	指令
C7	RST 0	E7	RST 4
CF	RST 1	EF	RST 5
D7	RST 2	F7	RST 6
DF	RST 3	EF	RST 7

上面列出的这些指令都称作 Restart（重新启动）指令，在其执行的过程中也会把当前 PC 中的数据保存到堆栈，这一点与 CALL 指令类似。但 Restart 指令在保存 PC 数据之后会立刻跳转到特定的地址，而且是根据参数的不同将跳转到不同的地址：比如 RST 0

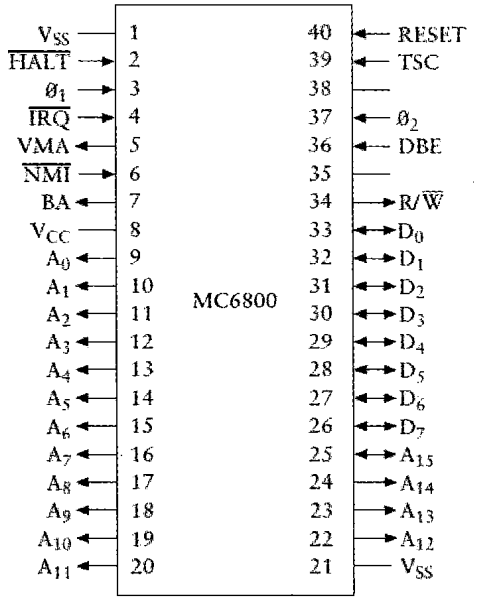
将跳转到地址 0000h 处，RST 1 将跳转到地址 00008h 处，依此类推，最后的 RST 7 将跳转到地址 0038h 处。这些地址存放的代码都是用来处理中断的。例如，由键盘引起的中断将执行 RST 4 指令，程序将跳转到地址 0020h 处，该地址存放的代码将负责从键盘读入数据（完整的过程将在第 21 章讲述）。

目前为止，我们已经介绍了 243 个操作码。在前 255 个数中，有 12 个没有作为操作码使用，它们是：08h，10h，18h，20h，28h，30h，38h，CBh，D9h，DDh，EDh 和 FDh。下面还需要讲到一个操作码。

操作码	指令
00	NOP

NOP 代表（即声明）no op（no operation，无操作）。NOP 指令使处理器什么操作也不执行。这样做有什么好处呢？填空，即保持处理器的运行状态而不做任何事情。8080 可以执行一批 NOP 指令而不会引起任何错误事件的发生。

本章不准备详细介绍 Motorola 6800 微处理器，因为在构造和功能方面它与 8080 非常相似。下面是 6800 的 40 个管脚的功能描述图。



在上图中，V_{SS} 表示接地，V_{CC} 代表 5V 的电源。同 8080 一样，6800 也有 16 个地址

输出信号端和 8 个数据信号端，其中数据信号端既可以用于输入信号也可以用于输出信号。它还有一个 RESET 信号端和一个 R/$\overline{W}$（read/write，读/写）信号。$\overline{IRQ}$ 信号代表中断请求。与 8080 相比，6800 的时钟信号较为简单，6800 没有设计独立的 I/O 端口，所有的输入/输出设备的地址都是存储器地址空间的一部分。

6800 有一个 16 位的程序计数器 PC、一个 16 位的堆栈指针 SP、一个 8 位的状态寄存器（用来保存标志位），以及两个 8 位的累加器 A、B。A 和 B 都可以用做累加器（而不是把 B 作为普通的寄存器），因为 A 和 B 的功能完全相同，任何用 A 做的工作都可以用 B 实现。与 8080 不同，6800 没有设置其他的 8 位寄存器。

6800 设置了一个 16 位的索引寄存器（index register），它可以用来保存 16 位的地址，其功能与 8080 的 HL 寄存器对相似。对于 6800 的大部分指令来说，它们的地址都可以由索引寄存器与紧跟在操作码后的字节相加得到。

尽管 6800 实现的操作与 8080 大致相同——加载、保存、加法、减法、移位、跳转、调用等，但对应的操作码和助记符是完全不同的。例如，下面列出了 6800 的转移（Branch）指令集。

操作码	指　令	意　义
20h	BRA	转移
22h	BHI	大于则转移
23h	BLS	相等或小于则转移
24h	BCC	进位为 0 则转移
25h	BCS	进位为 1 则转移
26h	BNE	不相等则转移
27h	BEQ	相等则转移
28h	BVC	溢出置 0 则转移
29h	BVS	溢出置 1 则转移
2Ah	BPL	为正数则转移
2Bh	BMI	为负数则转移
2Ch	BGE	大于或等于 0 则转移
2Dh	BLT	小于 0 则转移
2Eh	BGT	大于 0 则转移
2Fh	BLE	小于或等于 0 则转移

与 8080 不同，6800 没有设置奇偶标志位，而是设置了一个溢出标志位（Overflow flag）。上面的转移指令中有一些依赖于标志位的组合（combinations of flags）。

当然，8080 和 6800 的指令集是不同的，虽然这两款芯片于同一年发布，但它们是由属于不同公司的两组不同的工程师设计的。这就造成了它们之间的不兼容，因此它们不能执行对方的机器码，为一种芯片编写的汇编语言程序也不能在另一种芯片上执行。如何编写能在不同类型处理器上执行的计算机程序是第 24 章的主题。

8080 和 6800 的另一个有趣的区别是：在两个处理器中，LDA 指令都从存储器的特定地址将数据加载到累加器。例如，在 8080 中，下面的字节序列：

3Ah	8080 LDA 指令
7Bh	
34h	

将把存储器地址 347Bh 处的字节加载到累加器。现在对比一下 6800 的 LDA 指令，它使用 6800 扩展寻址模式（6800 extended addressing mode）：

B6h	6800 LDA 指令
7Bh	
34h	

上面的这组字节序列将把存储器 7B34h 地址处的字节加载到累加器。

两者的区别是很微妙的。当然，你可能已经注意到了操作码的不同：8080 的操作码是 3Ah，而 6800 的操作码是 B6h。但是两种微处理器对紧跟在操作码后的地址的处理方式是不同的，8080 假设低字节在前，高字节在后；而 6800 假设高字节在前，低字节在后。

Intel 和 Motorola 的微处理器在保存多字节数据问题上的根本区别从未得到解决。直到今天，英特尔的微处理器在保存多字节数据时，仍然把最低有效字节放在最前面（也就是说，在最低地址处），而 Motorola 的微处理器在保存多字节数据时，仍然把最高有效字节放在最前面。

这两种不同的方式分别称为 little-endian（Intel 方式）和 big-endian（Motorola 方式）。

争论两者之间哪一种方式更好是件有趣的事，但在这么做之前，先要知道 big-endian 这个术语出自乔纳森·斯威夫特（Jonathan Swift）的 *Gulliver's Travels*，指的是刘普特（Lilliput）和布鲁夫思科（Belfuscu）之间关于在吃鸡蛋之前应该把鸡蛋的哪一头敲碎的争论。因此，这种争论可能是没有意义的（另一方面，坦白地说，在本书第 17 章设计的计算机所采用的方式我个人并不喜欢）。尽管不能确定那一种方式本质上是“对的”，这种差别确实造成了附加的兼容性问题，这种问题通常会在采用 little-endian 和 big-endian 系统的机器共享信息时出现。

这两种微处理器后来的发展如何呢？8080 被应用在一些人所谓的第一台个人电脑（personal computer）上，更准确地说应该是用于第一台家用电脑（home computer）上。下图是 Altair 8800，它曾登上了 1975 年 1 月的 *Popular Electronics* 杂志的封面。

当你看到 Altair 8800 时，前面板上的灯泡和开关会让你感到似曾相识。这个界面和第 16 章介绍 64 KB RAM 阵列时的初始“控制面板”的界面是类似的。

在 8080 之后，Intel 又推出了 8085 芯片，而具有更重大意义的是 Z-80 芯片的出现。Z-80 是由 Zilog 公司制造的，该公司是英特尔公司的竞争对手，由英特尔的前雇员费德瑞克·菲戈金（Federico Faggin）创立，费德瑞克·菲戈金曾在 4004 芯片的研制过程中做出重要贡献。Z-80 和 8080 完全兼容，并且增加了许多非常有用的指令。1977 年，Z-80 曾被应用在 Radio Shack TRS-80 Model 1 上。

同样是在 1977 年，由斯蒂夫·乔布斯（Steven Jobs）和史蒂芬·沃兹内卡（Stephen Wozniak）创立的苹果计算机公司推出了新一代产品 Apple II。Apple II 既没有使用 8080

也没有使用6800，而是使用了基于MOS技术的更加便宜的6502芯片，它是6800的改进加强版本。

1978年6月，英特尔公司推出了8086芯片，这是一个16位的微处理器，可以寻址1MB的地址空间。8086的操作码与8080不兼容，但它包含了乘法指令和除法指令。一年后，英特尔推出了8088芯片，其内部结构与8086完全相同，但在外部仍以字节为单位（即外部接口为8位）访问存储器，所以该芯片能使用为8080设计的较为流行的8位的外围芯片（8-bit support chips）。IBM在5150个人计算机中使用了8088芯片，这种计算机通常称为IBM PC，于1981年秋季推出。

IBM大举进军个人计算机（Personal Computer，有时也简称为PC）市场对业界产生了重大影响，许多公司都推出了与个人计算机兼容的机器（兼容的含义将在随后的几章里详细讨论）。多年以来，“IBM PC兼容”也暗示了“Intel inside”（即内部使用了Intel微处理器），这里特指Intel x86系列微处理器。x86系列微处理器包括1982年发布的186芯片和286芯片，1985年发布的32位386芯片，1989年发布的486芯片。从1993年开始，英特尔公司推出Intel奔腾（Intel Pentium）系列微处理器，而这个系列如今被广泛地应用于PC兼容机。虽然这些处理器的指令集都在不断扩展，但是它们仍然支持始于8086的所有早期处理器的操作码。

苹果公司的Macintosh于1984年首次发布，它采用摩托罗拉的68000微处理器，68000是16位微处理器，是6800的下一代产品。68000及其后续产品（通常称为68K系列）是已发布的处理器中最受欢迎的一类。

从1994年开始，Macintosh计算机开始使用PowerPC微处理器，该处理器是由摩托罗拉，IBM以及苹果公司联合开发的。PowerPC是采用RISC（Reduced Instruction Set Computing，精简指令集计算机）微处理器体系结构来设计的，其目的是通过某些方面的简化来提高处理器的速度。在RISC计算机中，通常指令都是等长的（PowerPC中是32位），只有加载和保存两种指令能访问存储器，并且尽量简化指令的操作。RISC处理器设置了大量的寄存器，这样就能避免频繁访问存储器以提其高运行速度。

PowerPC拥有完全不同的指令集，因此不能执行68K系列微处理器的代码。然而，目前Macintosh计算机使用的PowerPC微处理器可以仿真（emulate）68K系列微处理器。运行在PowerPC上的仿真程序逐一检查68K程序的操作码，并执行相应的操作。它执行

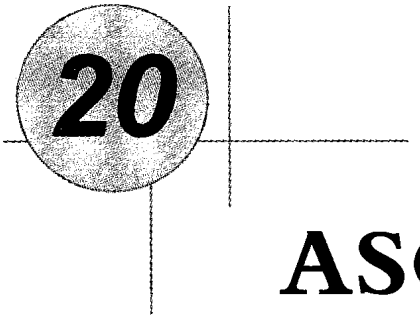
的速度没有 PowerPC 本身的代码那么快，但可以正常工作。

根据摩尔定律（Moore's Law），微处理器中的晶体管数量每 18 个月翻一倍，人们不禁要问：增加的这些大量的晶体管用来做什么呢？

一些晶体管用来适应处理器不断增加的数据宽度——从 4 位、8 位、16 位到 32 位；另一些新增的晶体管用来应对新的指令。例如，现在大部分微处理器都支持用于浮点数的指令（将在第 23 章详细介绍）；还有一些新增的指令用来执行重复计算，以便在计算机屏幕上呈现图片和电影。

现代处理器使用多种技术来提高其运行速度。其中一种就是流水线技术（pipelining），即处理器在执行一条指令的同时读取下一条指令，尽管 Jump 指令在一定程度上会改变这种流程。现代处理器还包括一个 Cache（高速缓冲存储器），它是一个设置在处理器内部，访问速度非常快的 RAM 阵列，用来存放处理器最近要执行的指令。由于计算机程序经常执行一些小的指令循环，使用 Cache 可以避免反复加载这些指令。上面提到的这些提高运行速度的策略都需要在处理器内部增加更多的逻辑组件和晶体管。

正如前面所提到的，微处理器只是整个计算机系统的一部分（尽管是最重要的一部分）。我们会在第 21 章构造这样一个系统，但首先要学习如何处理存储器中的数据，包括操作码和数字，我们要对这些数据进行编码。让我们从心态上回归到小学一年级，像孩子们学习读写一样学习如何编码吧。



ASCII码和字符转换

数字计算机中的存储器唯一可以存储的是比特，因此如果要想在计算机上处理信息，就必须把它们按位存储。通过先前的学习，我们已经掌握了如何用比特来表示数字和机器码。现在我们面临的一大挑战就是如何用它来存储文本。毕竟，人类所积累的大部分信息，都是以各种文本形式保存的。文本信息聚集最多的地方之一就是图书馆，数不清的书、杂志和报纸所提供的都是文本信息。当然，我们现在已经使用计算机来存放图像和影音信息了，不过为了易于理解，我们还是先从如何使用计算机存放文本开始讲解。

为了将文本表示为数字形式，我们需要构建一种系统来为每一个字母赋予一个唯一的编码。数字和标点符号也算做文本的一种形式，所以它们也必须拥有自己的编码。简而言之，所有由符号所表示的字母和数字（Alphanumeric）都需要编码。具有这种功能的系统被称为字符编码集（Coded Character Set），系统内的每个独立编码称为字符编码（Character Codes）。

许多疑问也随之而来，而要解决的第一个问题是：构成这些编码究竟需要多少比特？

要想回答这个问题就需要我们从长计议了。

当考虑用比特来表示文本的时候，切忌好高骛远。我们经常会看到书页上，或报刊和杂志的栏目上所有的内容被整齐地组织在一起。所有的段落都划分为宽度相等的文本行，但我们要注意，这种排版的形式永远只是文本之外的事物。当曾在某个杂志上细细品味过的一个故事，几年后与我们在另外一本书中重逢时，我们回忆起的往往是故事本身而不是文本的排版，没有人会因为行与行间距的不同而把它们当成两个故事。

我极力想阐述一个重要的观点，那就是文本与其印刷在纸上时采用的二维排版格式是两码事。充分发挥想象力，将文本看成是一维的由字母、数字和标点符号组成的数据流吧。当然，有时为了标明一句话的开始和结尾，还需要一些额外的编码。

还是先前所描述的例子，曾在某个杂志上细细品味过的一个故事，几年后出现在另外一本书中，但是文章的字体发生了变化，这算是一个问题吗？当年的杂志上是这样印刷的：

Call me Ishmael.

而现在书中的写法变成了下面这样：

Call me Ishmael.

这些区别是我们所在意的吗？答案往往是否定的。字体改变了文本的表现形式，但故事本身的内容并没有因此而改变。字体是可以变来变去的。但这并无大碍。

还有一种简化问题的方法：我们可以总是使用毫无修饰的文本。没有斜体、粗体、下画线、颜色、空心体、上标、下标以及音标，同样的，这里没有元音字母标识等符号，只有赤条条的拉丁字母，这些字母组成了英语中 99%的文本。

在先前对莫尔斯码和布莱叶盲文的学习中，我们了解了如何将字母表中的字符以二进制的形式表现出来。这些系统在适合的场合很好用，但要想用到计算机中却是难上加难。就拿莫尔斯码来说，它是变量自适应长度（Variable-Width）编码：常用字符的编码较短，而不常用字符的编码较长。这样的编码非常适合电报系统，但并不适用于计算机。另外，莫尔斯码并不区分字母的大小写。

布莱叶盲文编码使用固定宽度，非常适合计算机使用。每一个字符对应着 6 比特的

编码，并且用到了转义（Escape）码对大小写进行了区分。转义码用来表明下一个字符为大写。这也就是说，每个大写字母都需要两组编码来表示。布莱叶盲文中用移位（Shift）码表示数字：移位码后紧跟的编码都被看做数字，直到遇到下一个移位码，此时系统又将后面的内容当做字母。

我们的目标是开发一个字符编码集，使用这个编码集，系统可以将如下的句子转换成为一系列的编码：

I have 27 sisters.

每一个字符的编码都会占据一定的比特。有的编码用来表示字母，有的用来表示标点符号，还有一些用来表示数字。甚至于单词间的空格也需要单独的编码。上面的句子中共18个字符（包括字间空格），对这样一个句子进行编码后得到的连续字符通常被称为文本字符串（string）。

我们需要对字符串中的数字进行编码，例如上面的句子中的 27。或许大家会感到疑惑，因为之前我们都是用比特来表示数字的。最简单的，也是最容易想到的做法就是使用二进制数 10 和 111 作为 2 和 7 的编码。但是这里却不适用。在这个句子中，可以像处理其他的字符一样来处理 2 和 7。它们的编码可以和本身表示的含义无关。

1874 年由法国电报服务公司（French Telegraph Service）职员埃米尔·波多（Emile Baudot）发明了可以打印的电报机，划时代的波多电传码也应运而生。即使在今天来看，这种编码十分“经济划算”，每一个文本字符都采用 5 位编码。这种编码 1877 年被法国电报服务公司采纳，后来经唐纳德·默里（Donald Murray）修改，最终在 1931 年被当年的 CCITT 组织（Comité Consultatif International Télégraphique et Téléphonique），即现在的国际电信联盟（ITU）定为标准。该编码的正式名称是国际电报字母表第二号（International Telegraph Alphabet No.2）或 ITA - 2，在美国常常被称为波多印字电报制（Baudot），不过更准确地说，叫做默里（Murray）编码。

随着 20 世纪的到来，Baudot 被广泛应用于电传打字机（teletypewriters）。Baudot 电传打字机配备了一个输入键盘，这款键盘有些像打字机，但只有 30 个键和一个空格键。电传打字机键盘上的每一个键实质上都起到了转换器的作用，它负责产生二进制编码并且通过输出电缆逐位传输出去。电传打字机也具备打印功能，通过输入电缆读取编码，

触发电磁铁，从而将字符打印在纸上。

由于 Baudot 对每个字符采用 5 位编码，整个系统由 32 个编码所组成，这些编码的十六进制取值范围从 00h 到 1Fh。下表给出了 32 个不同编码的十六进制形式及其所对应的字母表中的字符。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	E
01	T	11	Z
02	回车	12	D
03	O	13	B
04	空格	14	S
05	H	15	Y
06	N	16	F
07	M	17	X
08	换行	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	数字转义符号
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K
0F	V	1F	字符转义符号

编码 00h 被保留了下来，没有指派给任何值。剩下的 31 个编码中，字母表中的字符占了 26 个，其余 5 个用来调整格式，如上表中的楷体排版的语句所示。

编码 04h 用来表示空格，通常用于分隔单词。编码 02h 和 08h 表示的是回车和换行。这些都是电传打字机中的专用术语。当使用电传打字机上打字，一旦到了一行的末尾时，我们通常会按下一个操作杆或按钮。这个操作其实包括两个动作：第一个动作是，使打印机的滑架回到起始位置，这样打印下一行时可以从纸的最左边开始，这就是回车。第二个动作是，将打印机的滑架移至正在使用中的位置的下一行，这就是换行。在 Baudot 编码系统中，这两个编码由专门的按键产生。Baudot 电传打字机在打印的时候会响应这两个编码以完成相应的操作。

Baudot 系统里怎么没有数字和标点符号呢？其实这是因为编码 1Bh 中暗藏玄机，它

的实际作用是数字转义（Figure Shift）。数字转义编码后的所有的编码都会被解释为数字或标点符号，直到遇到字符转义编码（1Fh），一切就又被解释为字符。下表展示了十六进制编码以及所对应的数字和标点符号。

十六进制码	Baudot 字符	十六进制码	Baudot 字符
00		10	3
01	5	11	+
02	回车	12	身份不明
03	9	13	?
04	空格	14	‘
05	#	15	6
06	,	16	$
07	.	17	/
08	换行	18	-
09	)	19	2
0A	4	1A	响铃
0B	&	1B	数字转义符号
0C	8	1C	7
0D	0	1D	1
0E	:	1E	(
0F	=	1F	字符转义符号

其实在 ITU 规范化的编码方案中，05h、0Bh 和 16h 是留做他用的，官方说法为“国内使用”。表中列出的是这几个编码在美国使用时的含义。某些欧洲国家将这些编码代表重音符号。响铃编码令电传打字机发出清脆的铃声。“Who Are You”编码用来让打字员激活身份识别机制。

像莫尔斯码一样，这种 5 位的编码并没有提供区分大、小写的方法。下面这个句子：

I SPENT $25 TODAY.

表示成编码的十六进制数据流就是：

0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08

请注意三个转义码的使用：1Bh 出现在数字之前，1Fh 出现在数字之后，而数字结束之后又出现了 1Bh。这一行编码以回车、换行符结尾。

问题出来了，如果把相同的数据流再一次输入到电传打印机，情况就大不一样了，

如下所示：

I SPENNT $25 TODAY.

8'03,5 $25 TODAY.

怎么会这样？这是由于在接收到第二行编码之前打印机接收到的最后一个转义码是数字转义码，所以当遇见第二行开头几个编码时，打印机将它们解释成数字。

这种问题产生的根源就是采用了转义码，这的确很让人头痛。尽管 Baudot 电传码是很简洁实用的编码，但是，我们更加希望采用能唯一表示字符、数字及标点符号的编码方案，如果还能对大、小写进行区分那就更好不过了。

如果想知道比 Baudot 更好用的编码系统中一个编码需要多少比特，我们需要做几个小加法：所有的大小写字母加起来共需 52 个编码，0～9 数字需要 10 个编码，加起来共有 62 个，如果算上一些标点符号，数量超过了 64 个，也就是说，一个编码至少需要 6 比特。但无论如何字符数应该不超过 128 个，而且应该远远不够 128 个，也就是说编码长度不会超过 8 位。

所以，答案就是 7。在采用 7 位编码时，不需要转义字符，而且可以区分字母的大小写。

这些字符编码是什么样子的呢？其实我们可以随意编码。如果我们要去打造一台自己的计算机，计算机硬件的每一个部分都要亲自制作，计算机内部的程序也要亲手编写，而且不打算把这台计算机与其他的进行连接，那么就完全可以构造自己的编码系统。其实也很简单，就是给每一个字符指派一个唯一的编码。

但是这种自己制造计算机，并且独立使用的情况实在是太少了，所以所有人都遵循并使用统一化的编码，计算机的存在才有意义。这样一来，使用不同方法制造出的计算机之间就可以互相兼容，甚至可以互相交流文本信息。

这样一来，随意的编码就显得不太合适了。当我们使用计算机来处理文本时，如果字母表中字母的编码是按顺序来的，就会给我们的工作带来很多便利，显而易见的优点就是，字母的排序和分类将变得简单易行。

幸运的是，这种标准已经存在并且被广泛使用，它被称为美国信息交换标准码