串列輸入 / 輸出 5

方波風琴

本章節提供給你很多值回票價的內容。在一切都完成之前，你將學會如何在 AVR 和桌上型電腦或筆記型電腦之間的通訊，並且製作一個時尚的電腦控制樂器。

在此，你將會學到串列通訊協定，以及如何透過 AVR 製作快速簡潔的音頻。串列通訊是微處理器與桌上型電腦或筆記型電腦之間最簡單的一種通訊方式，這是你第一步替真實的世界和虛擬的世界之間搭起溝通的橋樑。

我們需要些什麼？

除了基本的工具套件外，你還會需要：

- 一個使用大約 10 uF 的直流－阻隔電容的喇叭。
- 一個 USB 轉串列的轉接器。
- （可選擇）一個放大器，如果你想大聲播放音頻。

串列通訊

電腦或積體電路元件之間到底是如何互相通訊呢？這個（過於簡短的）答案幾乎與我們在第 3 章節所做的方式一模一樣，其中一端透過連接到其他裝置導線所輸出的高、低電壓脈衝來發出訊息。另一端，以輸入模式，在導線上取得電壓。

將編碼數據轉換成電壓脈衝以及將電壓脈衝解碼成數據的規則，又稱為**串列通訊協定**。我們在第 16 和第 17 章節將探討更多有關其他串列通訊協定的真實情境。現在我們將以最常用到的串列模式為討論的範圍—萬用非同步串列的接收和發送（UART）。

為了了解 UART 串列通訊到底是什麼，先想像一下兩個人透過幾條導線所發出的電壓訊號相互交談。比方說，Alice 想透過導線發送數字 10 給她的朋友 Bob。具體來說，導線上有個上拉電阻，因此它能夠固定保持在 5 伏特電壓。在 Alice 這一端的導線，有個接地處理的開關，而在 Bob 這一端，另外還有一個 LED，藉由該 LED 是否亮起可以讓他看到導線上的電壓是多少。

Alice 會透過按鍵將數字 10 發送給 Bob，把導線接地處理，並關閉 Bob 在另一側的 LED。現在，她可以透過讓 LED 閃爍 10 次來傳送數字，但當她想傳送數字 253，或更慘的，傳送數字 64123 時，系統會造成當機。因此她倒不如寫出 10 的二進制，`0b00001010`，並且傳送一個相對應的閃爍模式。

要讓這個作業能夠順利進行，Bob 和 Alice 必須事先約定一些事情 －串列通訊協定。首先，它們需要決定編碼方式：事先約定好按壓按鈕（導線的 0 伏特訊號）代表 0，沒有按壓按鈕（5 伏特）代表 1，而且它們會先傳送該數字的最低有效位元。

接下來，需要約定好 Alice 按鈕按壓或不按壓的頻率。比方說，它們選擇每秒傳送訊號一次。這就是**波特率**—導線上的電壓能夠被改變的頻率有多頻繁，反過來說，接收器讀取新電壓的頻率有多頻繁。

那麼 Bob 是如何知道傳輸開始和結束的時間呢？ 他們已經約定好用來包裝八個位元的是兩個額外的位元：一個是**啟始位元**，它永遠是 0，這樣你就可以分辨什麼時候開始傳輸，以及一個**停止位元**，它是一個 1。

當 Bob 看到 LED 閃爍時，他坐了下來，眼睛盯著 LED。當他看到 LED 閃爍了一下—啟始位元！ 現在啟始位元之後每秒鐘一次，他會將 LED 是否開啟或關閉全部記錄下來。在第一次閃爍之後，他會看到關閉、開啟、關閉、開啟、關閉、關閉、關閉、關閉的情境，然後 LED 停留在開啟狀態一會兒。他寫下它的八位元數值，`01010000`。然後切換位元的排序，並且看到 Alice 傳送的是數字 10 ！

圖 5-1 裡的示波器軌跡是 AVR 傳送數字 10 到電腦的一個真實範例。我用每秒 9,600 位元（波特），而不是每秒一位元，因此每一位元約需要用到 104 微秒。

但是你可以完成模式：1111100101000011111。記得第一個低電平位元為啟始位元、然後計數八個位元、檢查第九位元是高電平、切換順序，然後以二進制來讀取。

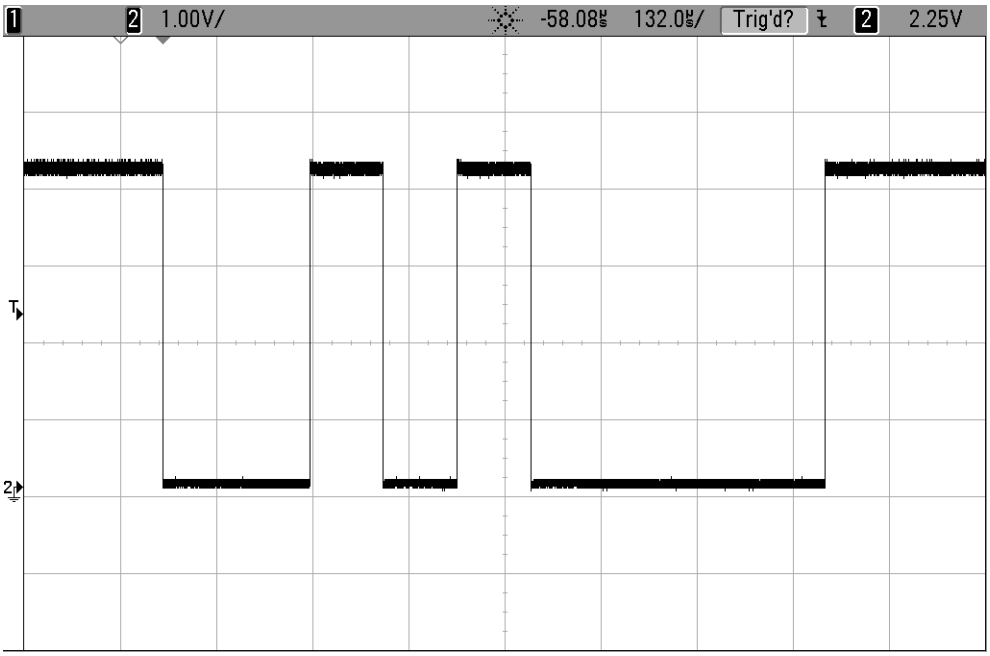


圖 **5-1** 串列方式傳送 10

現在如果 Bob 想回覆，最簡單的解決辦法就是只重複一樣的實際設定與通訊協定，但反過來說—就是給 Bob 一個按鈕以及給 Alice 一個 LED。這樣的話，Bob 和 Alice 就可以在任何後相互傳送訊號，並且往返傳送數字。完成這項作業需要用到兩條訊號線，但是使用專屬導線的優點就是因為具備簡便與數據的流通量。

圖 5-2 是傳送一列兩個位元組的軌跡圖：9，接下來是 10。請注意中間的部分，在兩個數字之間的停止位元比較長一 AVR 發揮作用在接下來多出的幾微秒不會傳送數據。這並不會干擾接收端的電腦，它只有在接收到下一個啟始位元時，才會再次開始計時。

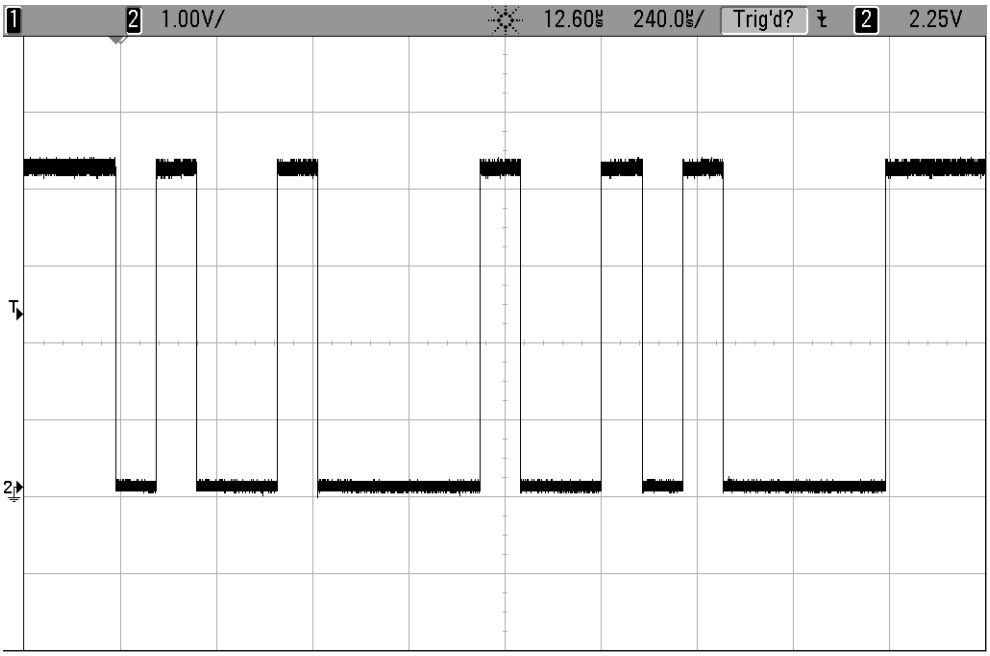


圖 **5-2** 以串列方式傳送 9 和 10

參考圖 5-3 訊號的邏輯說明，如果你有意願請嘗試從示波器的軌跡中讀取這些位元。

開始	0	1	2	3	4	5	6	7	結束	開始	0	1	2	3	4	5	6	7	結束
0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1

圖 **5-3** 傳送 9 和 10

數據的編碼和解碼工作似乎相當繁重，但以每秒數以萬計位元的波特率傳送和接收數據可是一點都不會有困難。這就是為什麼所有的 AVR Mega 微控制器至少要有一個專屬的硬體周邊，該內建的裝置被稱為**萬用同步與非同步的收發器**（USART）。在本章節的其餘部分，我們將看到如何設定與使用 USART 硬體，以及如何設定讓電腦能與 AVR 通訊。

UART 與 USART

介紹時，我說過將會使用串列 *UART*（萬用非同步接收與傳送），但 *AVR* 的周邊裝置又被稱為 *USART*。這有什麼差別呢？為什麼會有額外的英文字母“*S*”？

AVR 的串列硬體可以在非同步模式下讓 *UART* 與 *USART* 兩者都能運作—使用有助於資料蒐集的時脈—以及在不用時脈下使用非同步模式，這就是“*USART*”（萬用同步與非同步接收器和傳送器）。但是到了第 *16* 章節之前，我們僅會討論非同步的串列通訊，因此現在開始你就可以先不去擔心 *AVR* 的 *USART* 裡英文字母“*S*”的問題。

在 AVR 實施串列通訊：迴圈循環專案

你想要做的第一件事就是確認電腦和微控制器之間的串列連接是開啟而且可以運作，同時熟悉它的作業方式。現在是讓晶片與電腦通訊的時候了。

我們在這裡的設定涉及三個不同的階段：

1. 設定 AVR。
2. 在電腦上安裝串列終端軟體。
3. 將它們都連接在一起

安裝：設定 AVR

因為 AVR 微處理器內部有專屬的 USART 電路，因此我們為了通訊目的唯一要設定的就是介面，然後把位元組傾印到用來傳送和接收的特殊硬體暫存器。硬體周邊會負責處理其他的事項。這是非常方便的作法。

首先，要確認這一切都能運作正常，我們會燒入一個快速示範程式。如果你在前面幾個章節所使用的 LED 仍然插在電路板上，真是太好的！ 如果不是這樣，將會錯過你所鍵入每個字元所代表的 ASCII 參數值。在深入探討 USART 硬體設定的細節之前，讓我們看看透過範例 5-1 可以如何使用。

範例 5-1 serialLoopback.c 列表

```
/*
串列埠功能的簡單測試。
一次抓取一個字元並且將它直接送出去
在 LED 上顯示 ASCII 值。
*/
// ------- 序言 -------- //
#include <avr/io.h>
#include <util/delay.h>
#include "pinDefines.h"
#include "USART.h"

int main(void) {
  char serialCharacter;

  // -------- 初始化 --------- //
  LED_DDR = 0xff;                                    /* 將 LED 設定為輸出 */
  initUSART();
  printString("Hello World!\r\n");                          /* 測試 */

  // ------ 事件迴圈 ------ //
  while (1) {

    serialCharacter = receiveByte();
    transmitByte(serialCharacter);
    LED_PORT = serialCharacter;
                                          /* 顯示字元的 ASCII/ 數字參數值 */

  }                                                  /* 結束事件迴圈 */
  return (0);
}
```

直接跳到事件迴圈的議題，你可以看到我們的程式只做了三件事情。它透過串列傳輸線用 `receiveByte()` 函數來接收一個位元組。然後，使用 `transmitByte()` 函數將相同的位元組直接傳送回去。最後，LED 以 ASCII 的方式將剛剛傳送的位元組顯示出來。

初始化的部分，正如我們在先前章節所做的方式把 LED 初始化為輸出一樣，你不應該感到驚訝。`initUSART()` 和 `printString()` 函數有些什麼新功能。它們必須在某些地方定義，但到底在哪裡呢？ 向上捲動到序文的位置，你將會看到我已經加入一個新檔案，*USART.h*。如果把檔案開啟，你會發現他們的原型，我

們將透過第 99 頁“設定 USART：實作細節”的設定子程式詳細了解，並包括第 106 頁“C 語言的模組”的那些模組，目前我們將專注在確認一切都能正常運作。

函數當作動詞

這是一個格式問題，但我發現，如果將我用動詞來替所有函數的功能命名，及所有的變數採用描述性的名詞，這麼一來 *C* 程式碼就會讀起來像一般的句子一樣。

當然，第一次你可能需要鍵入多一點的資訊，但試想一下自己半年後再回來看到這個程式碼的時候。你還能夠瀏覽程式碼並找到你想修改的部分？ 你可以辦得到，假設你的函數和變數的名稱都能將內容描述的很清楚。

這是一個總覽介紹。如果你還沒有閱讀，現在是把串列迴圈返回專案程式碼燒入 AVR 的好時機。

安裝：電腦

在你的電腦上，就是要安裝合適的軟體。特別的是，你需要連接到一個可以讓你透過 AVR 輸入並讀取回應訊息的**終端機應用程式**。如果你有最滿意的終端機應用程式，不用考慮太多直接拿來使用。要不然，在此我有幾個建議。

安裝：USB 串列轉接器

因此當 serialLoopback 程式碼被燒入 AVR 時，而且你的電腦載入全新的串列終端機軟體並且執行時。現在就是將它們連接在一起的時候了。

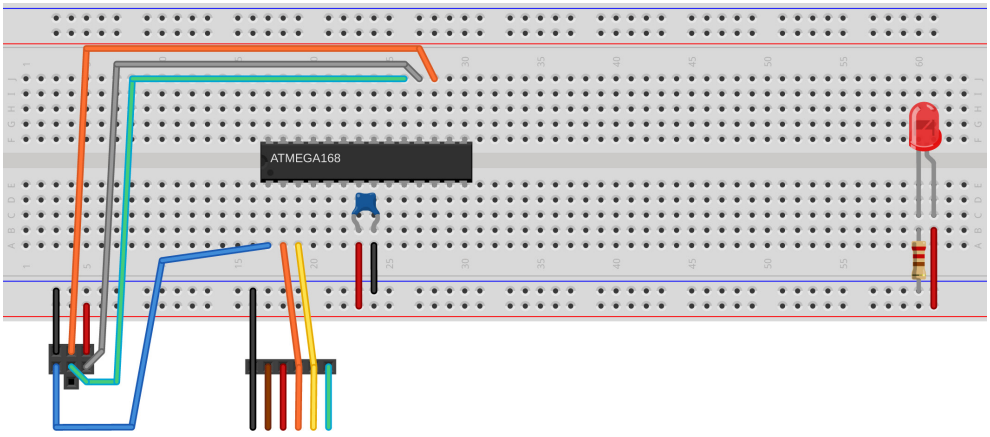
如果你擁有一條普通的 FTDI 傳輸線，它會有一個引腳輸出配置，如表 5-1 所示。

表 **5-1**　FTDI 電線引腳輸出配置

顏色	訊號
黑色	*GDN*
棕色	CTS#
紅色	*VCC*
橙色	*TXD*
黃色	*RXD*
綠色	RTS#

如果沒有 FTDI 傳輸線，你需要尋找的傳輸路會貼上類似 RX、TX、GND，以及可選擇的 VCC 標籤。你可以從一個 5V 的外部電源在專案的供電來源之間來做選擇，這是個正確的做法，或是透過 USB 傳輸線來供應整個電路的電源，這是一個方便的方式但它所需的電流可能比在規格範圍內，透過 FTDI 電路所允許的電流還多。

仔細檢查你是否已經把電腦 / FTDI 傳輸線的 RX 引腳與 AVR 的 TX 引腳連接，反之亦然。在麵包板上，FTDI 傳輸線由彩虹顏色、六引腳標頭接片所代表，如圖 5-4 所示。



用 Fritzing.org 製作

圖 **5-4**　麵包板的串列輸入 / 輸出連接

終端機模擬軟體

Linux

`gtkterm` 是不錯且方便設定而且適用在大部分 Linux 版本。另一種選擇是 *screen*，它可以在 Mac OS 和 Linux 平台運行，但是有一種你可能意想不到的是另類串列終端機。透過 **`screen[portName]9600`** 觸發 *screen* 應該可以與你的 AVR 直接通訊。按壓 Ctrl-A 然後再按壓 Ctrl-? 會顯示輔助說明。按壓 Ctrl-A 然後再按壓 Ctrl-K 會刪除現有的對話。Linux 使用者透過鍵入 **`ls/ dev/ tty*`** 可以找出哪個 USB 串列埠被連接。

Mac OS

`CoolTerm` 和 `ZTerm` 是不錯的 Mac 串列終端機應用程式。如果沒有自動偵測出串列埠，你可以透過開啟一個普通的終端機並鍵入 `ls /dev/*usbserial*` 來尋找。將尋找類似 `/dev/cu.usbserial-XXXX` 的串列埠。

Windows

Windows 系統中有為數眾多的終端程式可用。早期，高速終端機 (Hyperterminal) 是一個標準配備並且在附代在零配件裡。現在，我使用過訴求簡潔的 Bray 終端機程式來完成作業。如果你不確定 USB 串列與哪個埠連接，透過連接埠→ USB 串列埠的裝置管理員找尋。

跨平台，*Python*

如果你已經安裝好 Python 和串列函式庫，就可以執行它所提供的串列終端機模擬程式。鍵入 **`python -m serial.tools.miniterm-cr [portName] 9600`** 應該可以讓你開始通訊。請注意，按壓 CTRL-] 會退出，按壓 Ctrl-T 然後再按壓 Ctrl-H 會顯示輔助說明。(如果有辦法，請為它製作一個快捷鍵。)

跨平台，*Arduino*

如果在電腦上已經安裝好 Arduino IDE，你可以透過工具→串列監視器找到串列終端程式。這個“終端機”程式對讀取 AVR 的作業很適合，但是為了傳送訊息，需要在你輸入的每個字元之後按壓 Enter 鍵。這對文字輸入來說，你可以用這種方式一次傳送一行文字，但是對於互動的事物，這個作法是不方便的。總之，即使你只需要透過 AVR 接收數據，我也不建議把 Arduino 串列監視器當作一般的終端機程式使用。

串列引腳的連接

將串列轉接器與 AVR 連接就像是連接三條導線一樣簡單，而第一個簡單的線路：它是接地處理 (GND)。其他兩條導線是“棘手的”。你需要把其中一個裝置的 RX 連接到另一個裝置的 TX，並且把一個裝置的 TX 連接到另一個裝置的 RX。

用這個方式“跳線”連接在電子學是說得通，但是要注意，它的命名規則與在編程中所使用 SPI 引腳的命名方式是完全不同的！ 在此，編程器上的 MOSI 連接到 AVR 上的 MOSI、SCK 連接到 SCK…等等，幾乎當今大部分的串列通訊協定規定都是這樣制訂的：具有相同名稱的引腳都會連接在一起。

較新的導線標籤規定，也讓導線之間的連接特別方便。將 MOSI 與 MOSI 連接的導線應該會標示“MOSI”，但連接 RX 與 TX 的導線你會如何貼標籤呢？ 或是連接 TX 與 RX 的導線會貼什麼標籤呢？通常情況下，答案就像是“TXA-RXB”和“RXA-TXB”，但除了不好記之外，你接著必需記住哪個裝置是“A”，哪個裝置是“B”，總知有點混亂。

在舊款串列埠的蹩腳命名規則推出之後所有工程師都學到了教訓，但在 USART 的領域內它並無法幫上任何忙。因此只要記住：使用（舊款）USART 串列通訊時，你會把 RX 連接到 TX，反之亦然。你所製作的電路示意圖，如圖 5-5 所示。

圖 **5-5** RX-TX 連接

再次提醒你，留意我們已經連接好 RX 和 TX 導線，並把 FTDI 傳輸線的接地線連接到 AVR 晶片和編程器之間的共用接地線。（如果你上一章節所連接的 LED 仍然存在，這是個額外的好處。）

全部都放在一起：測試你的迴圈循環

因此如果你已經把 serialLoopback 程式碼燒入到晶片，把 RX/TX 引腳（和 GND）連接到 USB 串列轉換器，並插入電腦，現在是測試這個成品的時候了。

開啟終端機程式，並確定連接埠（USB 轉換器）的設定是正確的，並且使用正確的波特率（9600）。如果有一個“本地回應”的選項，請將它設定為關閉—我們是要透過 AVR 晶片回傳訊號。如果“本地回應”的選項是開啟，你會看到你所輸入都會重複兩次，如“lliikkee tthhiiss”。

現在鍵入一些資料。LED 應該會顯示你所鍵入英文字母所對應的 ASCII 二進制參數值。成功了！ 如果你 AVR 電源（或暫時將 RESET 引腳接地裡）拔除再重新插入，你應該會被一個友善的問候“Hello World ！”如果你看到這個結果，表示你已經成功了。

ASCII

當透過串列傳輸路傳送 8 位元的位元組時，把它們都當作數字是一件很自然的事。但是，如果你想傳送英文字母，我們必須把數字映射成英文字母。請輸入美國標準訊息交換基本碼（ASCII）。

在網頁上查閱 ASCII 表格，或在 Linux 或 OS X 平台輸入 **man ascii**，這樣你就會了解“A”是 0x41，“B”是 0x42，“a”是 0x61，及“0”是 0x30，就可以運用自如。而其他所代表的參數值應該都不需要另外說明。

C 語言和 ASCII 可以一起配合運作。你可以在英文字母和數字之間轉換自如，而且因為字元是透過相對應的英文字母和數字的排序來定義，因此你可以用各種不同的技巧來使用 ASCII。例如 `'A'+2 ='C'`，`'0'+3 ='3'`。這代表，如果你有一個變數 `a=3`; 而且你要傳送一個代表它們的 ASCII 字元的數字，那麼你可以傳送 `'0'+a`。USART 函式庫裡的 `printByte()` 函數利用這事實傳送一個代表文字的位元組：

```
void printByte(uint8_t number){
    /* 百位數 */
    transmitByte('0'+ (number/100));
    /* 十位數 */
    transmitByte('0'+ ((number/10) %
10));
    /* 個位數 */
    transmitByte('0'+ (number % 10));
}
```

因此，載入 *serialLoopback.c* 示範程式，在你的鍵盤上鍵入英文字母，並觀察閃爍（ASCII 二進制數字）的燈光。你會看到機器上的鬼影！

用 printBinaryByte 替代 LED

如果你沒有連接 *LED*，但還是想看看個別字元的二進制 *ASCII* 位元是什麼樣子，可以用下列程式行取代：

```
printBinaryByte(serialCharacter);
```

在 *serialLoopback.c* 使用：

```
transmitByte(serialCharacter);
printString("\r\n");
```

然而，倒不如使用一個簡單的回應，你會得到一個剛剛輸入並回傳給你的字元 *ASCII* 參數值。一個字元轉 *ASCII* 的轉換器！

串列連接的故障排除

有什麼地方故障了？讓我們逐一檢查吧！

1. 你已經安裝了終端機軟體，對不對？
2. 當你插入 USB 串列傳輸線時，你的電腦是否能識別？在 Linux 平台，輸入 **`lsusb`** 並且尋找“FT232 USB 轉串列埠”（在 *http://github.com/jlhonora/lsusb* 網址下載 Mac 的版本），在 Windows 平台檢查裝置管理員→連接埠。
3. 在 Windows 平台，你有沒有安裝正確的驅動程式？ 如果沒有，請嘗試拔除並重新插入 FTDI 傳輸線。假如驅動程式還沒安裝，它應該會帶你逐步安裝。
4. 連接埠的名稱正確嗎？ 在 Linux 和 Mac 平台，它就像是 */dev/ ttyUSB0* 或 */dev/XXX.usbserial-XXXXXXXX*。在 Windows 平台，點擊裝置管理員裡的 FTDI 裝置，並選擇硬體分頁。它應該會告訴你它是連接到哪個連接埠。
5. 你有沒有授權？在 Linux 和 Mac 平台，如果你遇到困難，請嘗試用終端機執行 sudo 指令。（sudo 是一個可以讓你以超級使用者 (最大權限的使用者) 暫時一次執行一個命令的命令。當你因為授權的問題無法執行時，你可以透過 sudo 命令快速避開這個問題。）長期來說，你可能需要把自己加入有 USB 串列轉接器讀取 / 寫入權限的群組。假如你還不知道如何做到這點，請搜尋“串列埠權限”加上作業系統的名字。

6. 你搞混了 TX 和 RX 傳輸線了嗎？ 這是一個切換它們的快速實驗，看看是否能將問題解決。

7. 最後，你可以從圖片中移除 AVR，並透過硬體迴圈循環確認不是電腦的問題。拿一條導線並連接 USB 串列轉接器 TX 和 RX 之間線路。現在開啟終端機程式並輸入一些東西。訊號應該透過 TX 傳送，再傳回 RX，並且列印在螢幕上。如果一切正常，有可能是因為連接到 AVR 的導線的跳線沒有安置妥當，或者某個位置的波特率設定錯誤。請仔細檢查。

8. 當你全部搞定後，拍一張電路的照片。當再次連接時，你可以稍後回來參考。

如果能夠開始執行 serialLoopback 程式，那麼你的電腦、AVR 晶片，以及它們之間的連接一切都能正常運作。現在，你準備好開始使用串列埠。如果你想直接到第 107 頁“AVR 方波風琴”裡的 AVR 風琴專案，請便。如果你想多知道一點 USART 函式庫到底是如何作業的細節，請繼續閱讀。

設定 USART：實作細節

在這個單元中，我將更詳細說明 `initUSART()` 程式碼的作業方式。你幾乎會在本書的每個章節裡看到硬體設定和初始化程式的類似範例。第一次作業時的確有點難度，但是無論是在本書中的這個章節或在後面的任何章節經過幾次的作業之後，你就會熟悉的。

每個 AVR 內建的周邊裝置都有少數的暫存器是硬體配置必須的基本概要。暫存器位元組的每個位元就是一個開啟或關閉周邊功能的開關。那麼透過硬體設定來完成作業，就是要弄清楚你需要設定哪個開關而已，並透過位元操作調整到妥適的狀態。

為了更深入的探討，讓我們來看看 *USART.h* 和 *USART.c* 檔案。*.h* 檔案是相當標準的檔案 — 如果你沒有指定波特率，它會定義一個基本的波特率，然後用描述性的說明來介紹功能。任何你在 *USART.c* 所需要的巨集定義，也可以在這裡找到。

因此在 *USART.c* 來源檔案中函數實際定義的地方。我摘錄了部分的檔案，如範例 5-2 所示。

範例 5-2 USART.c 部分的列表

```
/*
  讓串列通訊運作的快速和簡單功能
*/

#include <avr/io.h>
#include "USART.h"
#include <util/setbaud.h>

void initUSART(void) {                                    /* 波特率的要求 */
  UBRR0H = UBRRH_VALUE;                                 /* 在 setbaud.h 裡定義 */
  UBRR0L = UBRRL_VALUE;
#if USE_2X
  UCSR0A |= (1 << U2X0);
#else
  UCSR0A &= ~(1 << U2X0);
#endif
                                                  /* 開啟 USART 傳送器 / 接收器 */
  UCSR0B = (1 << TXEN0) | (1 << RXEN0);
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);      /* 8 個數據位元，1 個停止位元  */

}

void transmitByte(uint8_t data) {
                                                        /* 等待清空傳輸緩衝區 */
  loop_until_bit_is_set(UCSR0A, UDRE0);
  UDR0 = data;                                                /* 傳送數據 */
}

uint8_t receiveByte(void) {
  loop_until_bit_is_set(UCSR0A, RXC0);                    /* 等待接收的數據 */
  return UDR0;                                           /* 回傳暫存器參數值 */
}


// 一個有用的列印指令範例
void printString(const char myString[]) {
  uint8_t i = 0;
  while (myString[i]) {
    transmitByte(myString[i]);
    i++;
  }
}
```

閱讀 *USART.c* 檔案時，你會使用到 mega168 晶片的數據表，並開啟“USART0”的章節，請隨意閱讀數據表的部分，然後直接到“USART 初始化”的章節。

因為這是你第一次設定 AVR 硬體周邊，因此我會帶著你完成一些設定步驟的細節。看看的 `initUSART()` 函數的一般結構。這到底是怎麼回事？我們在程式碼的頂部把一些數據分別寫入暫存器 `UBRR0H` 和 `UBRR0L`。在程式碼的底部，我們則使用位元移位在 `UCSR0B` 和 `UCSR0C` 暫存器裡設定兩個位元。譬如說，特別是我會用 `UCSR0B=(1<<TXEN0)|(1 <<RXEN0)` 來設定 USART0 的“開啟傳送”位元和“開啟接收”位元。但我如何知道任何大寫英文字母的巨集定義代表什麼？ 它們在哪裡定義的？

所有加密的大寫英文字母巨集定義的意義可以在 AVR 數據表裡找到，而且它們都在包含在程式碼頂部的標頭檔案 *avr/ io.h* 裡透過 #defined 定義。類似於我們在第 3 章節中用在輸入和輸出之間切換引腳模式所指向的暫存器，DDRB 暫存器是如何定義的，UCSR0B 指向串列硬體的設定暫存器，其中包含當作開關的 8 個位元被用來控制多達八個不同的硬體功能。

為了弄清楚串列硬體設定暫存器的作業細節，我們將參考 AVR 的產品數據表。具體來說，找到“USART0”的“暫存器說明”單元，向下翻頁直到你找到 `UCSR0B`。（好吧，我會在圖 5-6 中摘錄暫存器的電路圖，但是你應該真的去練習如何查找數據表。）

USART 控制與狀態暫存器 0B

RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80
7	6	5	4	3	2	1	0

圖 **5-6** USCROB 暫存器：位元與它們的名稱

在類似 `UCSR0B` 標準設定暫存器裡，每個位元基本上都是一個開關，每個開關都有它自己的函數。這些函數都編列在數據表中，也可找到相關聯的“輔助記憶”名稱，而且與包括在程式碼頂部 AVR—專屬 *avr/io.h* 檔案裡的這些引腳編號，所指定的名稱相同。

如果控制暫存器的第 2 個位元或第 3 個位元是開啟傳輸的位元，這代表你不必記得它，但你可以用 `TXEN0` 來代表它並且預處理器會用 3 來取代，因為 *io.h* 裡的定義語句會對應到位元 3。可惜的是，你**必須**記得在眾多控制暫存器中需要存取哪個控制暫存器，可是有時候輔助記憶符號並不是真的那麼清楚，但有總

比沒有好。如果你的程式碼以 `UCSR0B|=(1<<4)` 來表示，你完全不知道發生了什麼事情，因此如果它以 `UCSR0B|=(1<<RXEN0)` 來表示，至少有機會了解開啟串列接收器硬體位元的導線。

avr/io.h

實際上，*avr/io.h* 檔案依序包含不同的子檔案，它是根據你所編寫的晶片而定，這樣一來，在更換晶片的時候不必改變大部分的程式碼。例如，使用一個 *ATmega168P*，而實際包括的檔案被稱為 *iom168p.h*。查看這個檔案，你可以看到每個暫存器位元組和位元的所有特殊暫存器的定義。

往下翻閱到 *USART* 的單元，我們會看到：

```
#define UCSR0B _SFR_MEM8(0xC1)
#define TXB80  0
#define RXB80  1
#define UCSZ02 2
#define TXEN0  3
#define RXEN0  4
#define UDRIE0 5
#define TXCIE0 6
#define RXCIE0 7
```

但是請記得，巨集定義並不能為你做所有的事—例如它所能做的是在每個有 `RXEN0` 的地方它會用 4 來替換。當你在正確的暫存器，`UCSR0B`，切換那個位元時，才能了解那位元數字真正的含義。這就是說，當你必須人工設定 AVR 週邊，沒有其他東西可以取代 AVR 數據表中“暫存器說明”的單元。

“暫存器說明”單元

當我遭遇 *AVR* 周邊設定的問題時，我幾乎都是先重新閱讀“特點”和“概述”的章節，然後直接到“暫存器說明”的章節。經過前兩個單元之後，我大概就能掌握到硬體應該有哪些功能。接下來“暫存器說明”單元就會介紹它的作業方式。

initUSART() 函數中的第一個區塊的程式碼，使用了 *util/setbaud.h* 的 Include 檔案中的一些巨集定義來安裝 USART 的設定暫存器。UBRR0L 和 UBRR0H 分別包含了決定 AVR 將使用什麼波特率，來進行串列通訊的 16 位元數字的低位元組和高位元組。為了在正確的時間點對串列傳輸線採樣，AVR 將系統 CPU 的時脈分頻，它通常在 1 MHz 與 20MHz 之間運作，將波特率調降到 9,600 Hz 到 115,200 Hz 之間的時脈運作。這個時脈的除數將儲存在 UBRR0 裡。

因為可用的系統時脈及適合波特率的範圍很廣，因此 AVR 也有一個倍速模式來處理較高的速度。一個在 *setbaud.h* 定義的 USE_2X 巨集可以知道我們應該使用正常或加倍速率的模式。#if、#else 和 #endif 命令就像是 #define 語句：它們都是用在預先處理器上。這是我唯一使用預先處理器 #if 語句的場合，因此，如果你不想要使用就不要花太多時間在它身上。

接下來，透過寫入傳送和接收的開啟位元，程式碼會啟動 USART 硬體。當我們這樣做的時候，AVR 會接管 USART 這些連接埠的正常作業，然後設定數據方向暫存器與上拉電阻，以及適用於串列通訊的其餘事項。最後，我們設定一些額外的設定位元，它能用來設定 USART 的 8 位元位元組及一個停止位元。

請隨意閱讀數據表中“暫存器說明”的單元，看看所有特殊暫存器的設定位元的功能。我們在此只會用到選項中很小的部分，因為基本上我們所要做的一就是把“一般的”硬體以串列方式連接電腦。特別是，如果你想改變每幀幅的位元數字、或包括一個奇偶校驗位元、或額外的停止位元時，你可以在暫存器的敘述中看到哪些位元需要在這裡設定。

一旦 USART 完成設定，使用時可是件很容易的事。查看 transmitByte() 和 receiveByte() 這兩個函數。它們兩個會檢查 UCSR0A（USART 控制和狀態暫存器 0 A）暫存器裡的位元，確認該硬體已經準備好，然後把目標位元組讀取或寫入一個特殊的硬體暫存器 UDR0（USART 數據暫存器），這是你用在透過 USART 硬體來讀取和寫入數據。

為了傳送數據，只需等到狀態暫存器 UDRE0（空的 USART 數據暫存器）的位元完成設定，然後載入你的位元組並傳送到 UDR0。USART 硬體邏輯會把數據傳送到另一個記憶體的位置，就如同在 Bob 和 Alice 範例中所描述的，透過串列傳輸線一個位元、一個位元將數據寫出來。

同樣的，狀態暫存器 UCSR0A 有一個完成接收位元（RXC0），它可以讓你知道何時能收到一個位元組。而當該位元被設定時，將可以讀出你剛剛才傳入的數據。事實上，萬一你不想一直循環作業直到數據透過 USART 傳送進來為止—譬如說你想要 AVR 這個時候處理其他事件—而不是使用 `loop_until_bit_is_set(UCSR0A, RXC0);` construct，你可以簡單地測試 RXC0 位元是否在你的主事件迴圈裡設定的，以及當數據傳入的時候是否對新數據做出回應。稍後，你將會看到這類型程式碼的範例。

暫存器：使用 USART

傳送

等到 *USART* 數據暫存器清空時，這表示透過 UCSR0A 裡的 UDRE0 位元已經被設定，然後再把數據寫入 UDR0。

接收

當數據傳入時，UCSR0A 裡的 RXC0 位元會被設定，並且你能從 UDR0 讀取輸入的數據。

printString() 函數只是依序顯示字串中的所有字元直到結束為止，並且每次傳送一個字元。如何能知道什麼時候是結束的時候呢？ C 語言字串以 NULL 字元當作結尾，它是一個包含 0 的特殊值的字元，你可以在 if() 或 while() 語句裡輕鬆測試該字元。

另外兩個函數 printByte() 和 printBinaryByte()，都是為了方便並協助你在電腦上完成漂亮的輸出。printByte() 把 8 位元位元組的參數值轉換成它所代表的三個 ASCII 數字，然後將它們傳送出去。因此，如果 a= 56, printByte(a) 透過串列傳輸線發送 “0”，接著是 “5”，接下來是 “6”：

```
void printByte(uint8_t byte){
  /* 將一個位元組轉換成十進制文字的字串，並傳送 */
  transmitByte('0'+ (byte/100));         /* 百位數 */
  transmitByte('0'+ ((byte/10) % 10));   /* 十位數 */
  transmitByte('0'+ (byte % 10));        /* 個位數 */
}
```

但 printBinaryByte() 只是以二進制方式完成類似的作業；printBinaryByte(56) 在你的終端機列印出“00111000”：

```
void printBinaryByte(uint8_t byte){
  /* 將位元組以一串 1 和 0 的方式來印出 */
  uint8_t bit;
  for (bit=7; bit <255; bit--){
    if ( bit_is_set(byte, bit) )
      transmitByte('1');
    else
      transmitByte('0');
  }
}
```

因為我們已經習慣以最高有效位元的順序來讀取二進制數據，printBinaryByte() 函數裡的 for() 迴圈從七至零，執行變數位元。在 if() 語句測試每個位元並相對應地發送 1 或 0 的字元。

如果你看過完整版的 *USART.h* 和 *USART.c* 檔案，你會發現我們把更多有用的函數安排在本書其他章節不同的地方。但是請先暫緩稍後再實施。現在是開始使用這兩個檔案的時候了。

現在你知道是什麼原因讓這一切能夠正常運作。再一次，因為 USART 程式碼全都包裹在 *.h* 和 *.c* 檔案裡，在程式碼裡使用串列方式你唯一要做的是，在序文中包含 *.h* 檔案，並且在 makefile 裡添加 *.c* 檔案。然後，你就可以讓 AVR 自由地與電腦通訊，而這巨大的力量就是你的了。

C 語言的模組

所有 USART 的程式碼被包裹在 *USART.h* 和 *USART.c* 檔案裡，它是以最方便匯入到自己程式碼內的方式編寫的。當你專案的複雜程度到達一個階段的時候，你也會想要這麼做。因此，讓我們花點時間看看程式開發者如何把他們的程式碼包裹成 C 語言可重複使用的模組。

一般來說 *.c* 檔案是程式碼儲存的地方，而 *.h*，或標頭檔案是保留給 #define 它是使用者可能希望會改變的定義，以及那些包括在 *.c* 檔案內的函數說明。這樣一來當閱讀別人的程式碼時，標頭檔案是首先要查看的地方—因為它是一個提供概略說明的地方。

此外，在 C 語言的嚴格規範下，每個函數在被定義或使用之前就應該製作它的原型，而傳統用來確認這種情況的方式就是把函數原型放入標頭檔案內。函數原型與函數定義是非常相似的，只是沒有程式碼而已。最重要的是你必須宣告每個函數是用什麼類型的變數當作參數，以及它會回傳什麼參數值。但是在這當下，你最好順便替所有事件都編寫一些有用的說明。這會讓標頭檔案非常有價值。

為了在程式碼裡使用特定模組裡的函數，譬如說 *module.c*，你首先須透過包括主檔案頂部的 *module.h* 來包括所有的函數的原型。（看看我在本章節頂部的程式碼如何使用 `#include "USART.h"`？來處理）接下來，你要告訴編譯器哪裡可以找到 *module.c* 檔案，這裡是函數的實際程式碼儲存的位置。大多數專案專屬的細節，如模組檔案的位置等等，被整合到專案的 makefile 裡。

Makefile 透過定義哪部分的專案與其他部分專案之間的關聯性的方式，讓原本重複且枯燥的軟體編譯自動化。因此，當你在程式碼添加新模組時，需要告訴 Makefile 在那裡可以找到 *module.c* 檔案，因此它可以與主程式的 *.c* 檔案一起編譯。

如果你在一個專案裡只使用一個特定的模組，而將標頭和程式碼檔案放在同一目錄中當作你的主要程式碼是有道理的。另一方面，若要在多個專案中重複使用相同的程式碼，你可能需要把模組儲存到其他的“函式庫”目錄裡，並且從那裡把它包括在內。

在我的 Makefile 裡，我提供兩種可能的選擇。如果你想在其他的專案裡的同一個目錄中包括 *.c* 檔案，你可以把檔案名稱添加到 `LOCAL_SOURCE` 變數裡。如果你想包括儲存在其他地方的函式庫函數，可以將目錄名稱傳給 `EXTRA_SOURCE_DIR` 變數，並把檔案名稱傳給 `EXTRA_SOURCE_FILES` 變數。其他 Makefile 會有類似的定義。

因此，簡單來說，如果想把一個 module 模組包含在你的主程式碼裡，你會需要：

1. 把 *module.h* 與 *module.c* 檔案複製到你的專案目錄裡，或者把你在 Makefile 所儲存的檔案包括在目錄中。
2. 在使用任何函數之前，把 `#include module.h` 放在程式碼的頂端。
3. 把 *.c* 檔案當作額外的來源添加到你的 Makefile 裡。

USART 的其他用途

除了與你的電腦通訊之外，在此 USART 有些其他的用途：

- 除錯。能夠透過串列傳輸線傳送 AVR 的狀態訊息是很有趣的事。`printByte()`、`printWord()` 和 `printBinaryByte()` 函數可以讓除錯變得很輕鬆。
- 使用具備串列輸入功能的 LCD 和 VFD 顯示。
- 製作自己的 GPS 數據記錄器是相當容易的，因為 GPS 模組通常使用 USART 模式的串列通訊協定來傳送它們位置的數據。
- 整合磁條和紅外線兩種卡片閱讀器到你的設計之中，因為它們通常用 USART 串列傳輸線傳送數據。
- 透過無線電通訊協定來傳送數據，因為大多數是使用各種不同的 USART 編碼方式。
- 在本書中製作其他範例：我們會在第 7 章節使用 ADC 來製作一個低速“示波器”。

AVR 方波風琴

現在是製造一些聲響的時候了，同時好好的利用我們的串列埠。我喜愛聲音的專案，並且認為 AVR 風琴是個很好的範例。不過，如果你要彈奏風琴，肯定需要用到鍵盤。與其打造一個烏檀木、象牙及 88 鍵的開關，我們將利用電腦的鍵盤，並使用串列埠來連接具備 AVR 的電腦。讓我們開始動手吧！

聽覺是很神奇地。如果你無法認同我奇妙的感受，請想一想：在你我周圍的空氣中週期性壓力波按壓在薄薄的皮膚膜上，它會讓類似槓桿設定的小型骨頭一樣，將動作放大並且按壓連接到感測振動神經的毛髮上的流體一填充袋。當較長的毛髮是透過流體來振動，你會聽到低音。當較短的毛髮振動時，你會聽到高音。

沒錯。聽覺是一種我們與生俱來的能力。因此唯一要做的就是在空氣中製造相對應的週期性壓力波。就是因為這一點，我們會使用綁上電磁鐵的紙製圓錐體，並透過電力將電磁鐵和圓錐體在永久磁鐵的磁場周圍往復移動。我們會把這個裝置稱為喇叭。現在需要做的就是傳送依時間變動電壓到喇叭的線圈，我們將會聽到聲音。成功了！

用你的微處理器來製作音樂

微控制器並不是用來驅動喇叭。喇叭對直流（DC）電來說有非常低的電阻值，大約是 8 歐姆（Ω）。譬如說，如果我們以 5 伏特（V）電壓來執行 AVR 並且直接連接到 8 歐姆（Ω）的喇叭，它會汲取 625 毫安培（mA）的電流（5V 除以 8Ω），這顯然已經超過原先 AVR 引腳所規範的 20 到 50 毫安培（mA）電流。解決的辦法就是在電路上添加一個阻擋電容，如圖 5-7 所示。

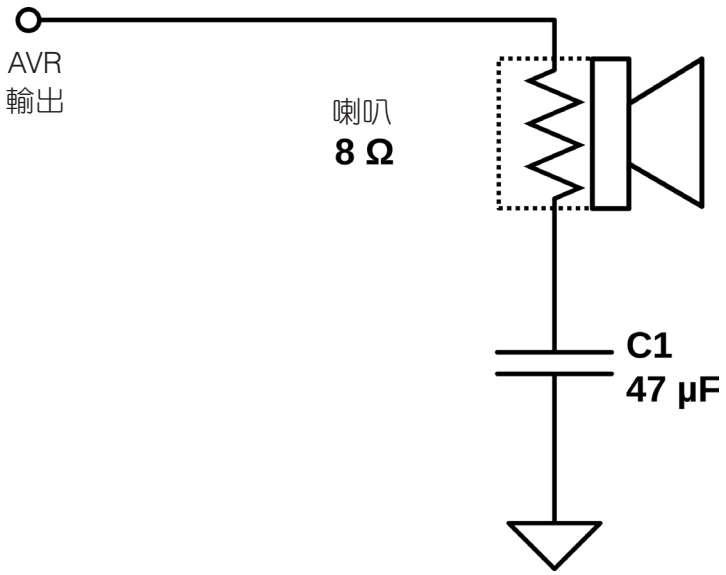
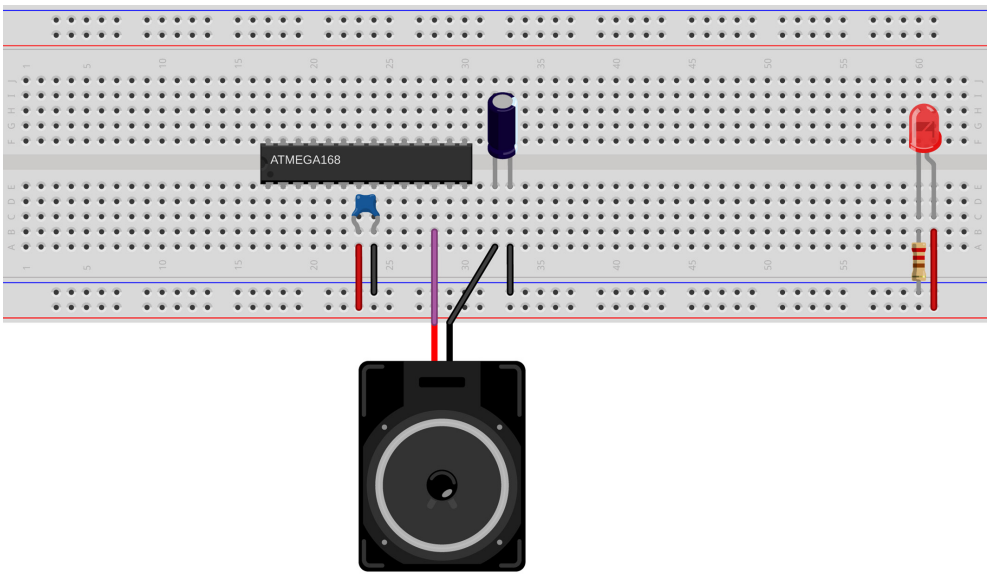


圖 **5-7** 具備阻擋電容器的喇叭

電容器的使用受限於直流電壓的規範，它會讓微小電流通過直到該電壓被“充飽”為止，接著它會阻擋電流進一步通過。這代表，電流僅在電壓有變化時才通過電容器。這很適合喇叭的使用！ 它所組成聲音透過訊號的變化來傳送，當電壓維持在高電平時，AVR 的輸出電流不會讓喇叭超載。

但是如果你使用的是電解電容（一個金屬管狀裝置）千萬要特別注意，它們通常都有正極和負極。雖然我們流經電容器的電流不至於破壞它，但建議你還是把有條紋標示（負極）的一側接地處理。在麵包板上的接線，如圖 5-8 所示。



用 Fritzing.org 製作

圖 **5-8** 麵包板上安裝具備阻擋電容的喇叭

究竟要使用多大的電容器完全取決於你所需要低音的程度、喇叭的零組件，以及你會允許 AVR 輸出的超載量。根據我的經驗，任何介於 10 到 100 微法拉（μF）之間的效果最好，而我喜歡用 47 微法拉（μF）。你可用任何參數值來實驗；而這樣實驗造成任何東西毀損的機率並不大。

放大率

同樣地透過 AVR 讓喇叭發出震耳欲聾的聲音的機會也是很小，因此如果真正想惹惱鄰居，你可能需要透過手邊的裝置把音頻訊號透過放大率重新安排：例如電腦的擴音喇叭、立體聲音響、混音鍵盤等等。但首先，你必須先將訊號弱化。AVR 所輸出的 5 V 訊號是無法驅動喇叭輸出大的音頻，但將它直接輸入放大器則會變得聲音過大。

大多數的擴大裝置都是一種“線性位準”的訊號；也就是說，介於一到兩伏特的波峰至波峰的訊號。假設 AVR 的運作電壓是 5 V，你會希望在傳送訊號之前至少將它的強度減半。要做到這點最簡單的方法就是將分壓器電路放在電容器之前來限制它的輸出。電阻值介於 10 kΩ 至 100 kΩ 範圍內的電位器可以拿來當作很好的分壓器，以及啟動用的音量調節鈕。請參考圖 5-9 的電路圖。

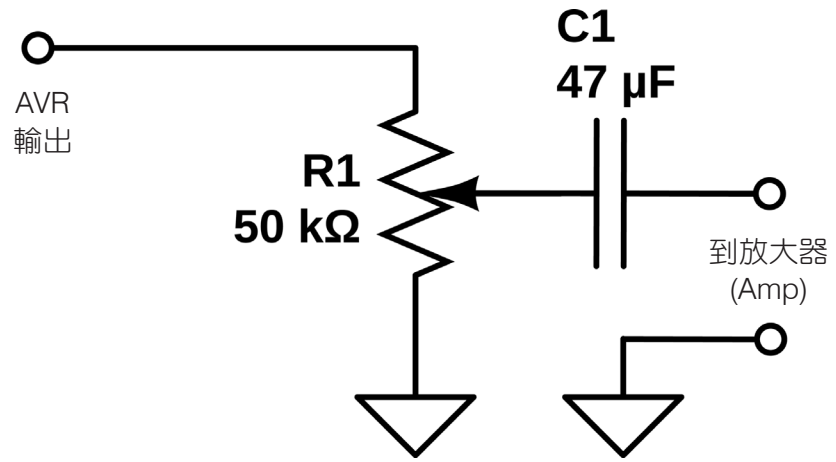


圖 **5-9** 使用電位計、阻擋電容器的音頻

該電路對自備電源“電腦顯示器”型式的喇叭是非常合適。事實上，你可能會想使用小巧且便宜而不是用高傳真的音響設備。我有一個舊式附電源的電腦喇叭；我將插頭切除並用三個鱷魚夾（用在立體聲）取代。很多的時候這是個方便的作法，而且也不用擔心它會損毀。由於我們將製作的方波不是非常細緻，除非你真的知道自己在做什麼，否則在直接插入到立體聲輸入傳輸線的時候我會特別小心。

一旦接好喇叭、附電源或是沒附電源，你就一切準備就緒。把 *serialOrgan.c* 程式碼燒入，然後啟動你的終端機程式，並且在主鍵盤上彈奏音符。

風琴函式庫

在這個程式裡我非用模組不可。因為大多數實際製造出聲音的程式碼都儲存在 *organ.h* 和 *organ.c* 函式庫裡，因此如果你願意，就可以在自己的程式碼裡再次使用。讓我們快速地瀏覽一下。

像往常一樣，先看看 *organ.h*。有兩個函數在此定義：`playNote(uint16_t wavelength,uint16_t duration)` 及 `rest(uint16_t duration)`。如果你觀察任何播放器的程式，它們也都包含 *scale16.h* 檔案，它會用 `#define` 提供 `playNote` 函數的波長方式的音階來替代。因此，讓我們來看看範例 5-3 中 `playNote()` 函數，並了解是怎麼回事。

範例 5-3　playNote 函數列表

```
void playNote(uint16_t wavelength, uint16_t duration){
  uint16_t elapsed;

  uint16_t i;
  for(elapsed = 0; elapsed < duration; elapsed += wavelength){
    /* 讓擁有可變延遲的 For 迴圈來選擇音調 */
    for (i = 0; i < wavelength; i++){
      _delay_us(1);
    }
    SPEAKER_PORT ^= (1 << SPEAKER);
  }
}
```

playNote() 函數以波長（基本上是頻率或音調的倒數）和持續時間為參數，並播放相對應音階與長度的音符。在特定頻率下在 0 和 1 之間切換 SPEAKER 位元，而且因為喇叭的引腳被設定為輸出，因此會先傳送 5 V 接著在傳送 0 伏特到喇叭的圓錐體，並往復移動及產生音頻。

最內層的 for 迴圈只會等待一段精確的時間。透過 i 迴圈每一次等待一微秒的時間，該迴圈造成了一個大約 wavelength 微秒的延遲。因此，對喇叭圓錐體來說 wavelength 較大的參數值相對應的喇叭圓錐體較慢的往復動作，因此為了降低頻率。如果選擇了這些時間恰到好處，我們會聽到音樂的音階。

最外層的 for 迴圈（透過 elapsed 的迴圈）有一個聰明並值得一提的方法。以音樂來說我們希望指定一段特定持續時間的音調，它就是 duration 參數。但是同時，透過更改每次最內層迴圈每步階所需要的時間—wavelength 微秒，我們播放出來的音符的音階是不一樣的。當內部迴圈採用可變的時間參數，這樣它就能夠知道內部迴圈到底循環多少次。我們要如何編寫外部迴圈呢？

雖然我們經常在 for() 迴圈中透過每次加 1 的方式來遞增計量變數，但我們沒必要這麼做。因透過內部迴圈的每次循環，經過了 wavelength 微秒的時間。外部迴圈透過每次循環迴圈在 elapsed 變數中增加 wavelength 微秒的方式，來追蹤全部經過的時間。因此，與其在每個循環中的 elapsed 變數中加 1，elapsed 變數的增加是根據內部迴圈所花費的時間來決定的：wavelength 微秒。現在只要 elapsed 時間不會少於 duration 變數的時間，它就可以很輕鬆離開該迴圈。

程式碼

如果你因為編寫這些雜亂無章的程式碼而感覺很煩，請燒入 *serialOrgan.c* 程式碼，開啟一個串列終端機，然後按下一些按鍵。如果你已經連接好喇叭，應該可以用電腦鍵盤彈奏一些簡單的歌曲。享受一下成果吧！然後讓我們開始搞清楚透過範例 5-4 是如何製造出這些聲音的。

範例 5-4 serialOrgan.c 列表

```
/*

serialOrgan.c

透過鍵盤讀取串列字元，演奏出音符

查看 organ.h 內的引腳定義和其他巨集
查看用於 playNote() 和 rest() 的 organ.c ( 並包括在 Makefile 裡 )

*/

// ------- 序言 -------- //
#include <avr/io.h>
#include <util/delay.h>
#include "organ.h"
#include "scale16.h"
#include "pinDefines.h"
#include "USART.h"

#define NOTE_DURATION 0xF000                    /* 決定長的音符長度 */

int main(void) {

  // -------- 初始化 --------- //
  SPEAKER_DDR |= (1 << SPEAKER);                 /* 用於輸出的喇叭 */
  initUSART();
  printString("----- Serial Organ ------\r\n");

  char fromCompy;                               /* 用於儲存串列輸入 */
  uint16_t currentNoteLength = NOTE_DURATION / 2;
  const uint8_t keys[] = { 'a', 'w', 's', 'e', 'd', 'f', 't',
    'g', 'y', 'h', 'j', 'i', 'k', 'o',
    'l', 'p', ';', '\''
  };
  const uint16_t notes[] = { G4, Gx4, A4, Ax4, B4, C5, Cx5,
    D5, Dx5, E5, F5, Fx5, G5, Gx5,
```

```
    A5, Ax5, B5, C6
  };
  uint8_t isNote;
  uint8_t i;

  // ------ 事件迴圈 ------ //
  while (1) {

                                                          /* 讀取音階 */
    fromCompy = receiveByte();                          /* 等到有輸入為止 */
    transmitByte('N');                          /* 通知電腦可以進行下一個的音符 */
                                                          /* 播放音符 */
    isNote = 0;
    for (i = 0; i < sizeof(keys); i++) {
      if (fromCompy == keys[i]) {                /* 在對照表格找到符合的數據 */
        playNote(notes[i], currentNoteLength);
        isNote = 1;                               /* 我們找到一個音符的記錄  */
        break;                                        /* 離開 for() 迴圈  */
      }
    }
                                          /* 處理非音符音階：拍子速度變化和停止 */
    if (!isNote) {
      if (fromCompy == '[') {                                /* 短音符的程式碼 */
        currentNoteLength = NOTE_DURATION / 2;
      }
      else if (fromCompy == ']') {                           /* 長音符的程式碼 */
        currentNoteLength = NOTE_DURATION;
      }
      else {                                              /* 無法判斷，就靜止不動 */
        rest(currentNoteLength);
      }
    }

  }                                                         /* 結束事件迴圈 */
  return (0);
}
```

首先最重要的，查閱一長串惱人的 include 檔案列表。我們要在 *organ.c* 和 *organ.h* 檔案中使用 `playNote()` 函數，因此我已經包括了這些檔案。我也會在 *scale16.h* 檔案裡使用預先定義的音階數據。（“16”，因為該音階善用 16 位元數字當作更精確的音高精準度。）最後，我包括了 *USART.h* 檔案，它包含我們已經看到的通用串列函數。

有關 USART 函式庫，繼續查看 `main()` 函數。請注意，在進入事件迴圈，於初始化單元之前，我會呼叫 `initUSART()` 函數，並在串列埠列印出一些友善的文字。正如先前所看到的一樣，`initUSART()` 選擇波特率，並且將 USART 硬體設定好。print 命令能夠讓我們在串列終端機視窗找到某些文字，藉以確認串列連接一切正常。在離開初始化單元之前，請注意，透過在 DDR 設定正確的位元我們已經把喇叭的引腳設定為輸出。

最後，在事件迴圈裡，程式一開始會等待電腦傳送串列位元組。一旦收到一個位元組，它會回傳到電腦確認，稍後它會讓你彈奏串列風琴的腳本，接下來該子程式會根據你所傳送的音階播放合適的音調。程式碼映射這些按鍵到音符的方式是比較進階的議題，如果你願意，可以隨時略過下面的側欄目。

把按鍵的按壓轉換成音符

風琴程式碼必須透過串列終端機程式讀取輸入，並且以你直覺的方式把這些按鍵映射到音符。為了能夠以靈活及易於維護的方式編寫這個 lookup，我使用兩個常數陣列，一個是按鍵的列表，而另一個是音高的列表。這其實是一個用來處理用戶輸入相當常見的情境，因此我想我會在這裡提出來。

`(i=0;i<sizeof(key);i++){}` 迴圈是 lookup 的“技巧”。該程式碼會查詢對應到音符的所有可能音階。如果與其中之一匹配，它會立即播放第 i 個音符，並退出 `for()` 迴圈。很帥吧！

在 `for` 迴圈內我想指出一個細微的差別。請注意 `i` 只有在 `i<sizeof(keys)` 條件下才會遞增。因為程式碼用 `i` 完成 `key[]` 陣列的索引，所以沒什麼好奇怪的。但是，我們應該仔細檢查 `i` 永遠不會大於該陣列的大小。因為每個 `key[]` 的輸入項目是一個位元組，而陣列的長度是 18 位元組，`sizeof(key)` 命令會回傳 18，而 `i` 的最大值是 17。也就是說，我們只希望在 `i <sizeof(keys)` 的條件下，執行 `for()` 迴圈。

C 語言程式初學開發者所犯的錯誤就像是編寫類似 `i <= sizeof(keys)` 的問題，因為我們有 18 位元組，而且我們想讓所有的位元組循環執行，對嗎？ 是的，但是這裡有個問題！由於 C 陣列是以位置 0 為索引的開始，因此最後一個元素是 `key[17]`，而不是 `key[18]`。也就是說，使用小於或等於，最後會導致超出陣列索引末端一個元素的排列方式是不可行的。

如果你仍然是 C 程式初學開發者，每次看到索引陣列變數時，你可能要仔細檢查一下。只要記住，最大陣列的索引值是該陣列長度減去 1，因為我們是從 0 開始而不是從 1 開始計數。

如果程式沒有接收到有效的音符命令，它會檢查你是否改變了音符的長度，如果有，它會設定合適的參數值。萬一這些都沒有發生，那麼標準的反應就是靜止不動（不播放音符）一段時間。

現在一切就緒。你可以透過筆記型電腦的鍵盤來彈奏 AVR，這是很酷的。但是，這僅是一個長期和成功友誼的開始而已；你的電腦和 AVR 是駭客界的組合簡直是天作之合。

額外超值的功能

我另外還在 serialOrgan 專案裡包括了兩個有趣的軟體套件。兩個檔案都是用 Python 編寫的，而不是用 C 語言編寫。Python 是一種解譯式的程式語言，並包括一些讓你輕鬆編寫任何標準功能的模組。我會用在整本書裡需要用到電腦 / AVR 的互動的專案裡使用，或是進行一些對 AVR 計算需求較為吃重的專案。

製作一個包含音符音高數據的 *scale16.h* 標頭檔案，這是一個你在大電腦上真正想做簡單的子程式範例，而不是在 AVR 上即席重新製作。在專案的軟體分配中，我已經包括了製作標頭檔案的 *generateScale.py* 檔案。

我還包含了一個簡單的腳本（*autoPlay.py*），它說明電腦與 AVR 之間的介面。該程式碼是使用 Python 及 `pyserial` 函式庫。該腳本傳送字元到風琴來播放音符，然後在傳送下一個音符之前等待 AVR 的回應—當 AVR 準備好繼續的時候，你可以重呼 AVR 的回應。

雖然要掌握到合適時序是有它的困難度，甚至要記住哪些音調是當下 AVR 風琴彈奏的音調，鍵入一串字元並使用 Python 子程式把它們傳送到 AVR 則容易許多。純粹為了好玩的目的，我讓示範程式透過網頁攫取包含“歌曲”數據的文字，並且在風琴上彈奏出來。

這些程式對你了解 AVR 串列周邊都是沒有必要的，因此你要自行決定是否要深入探討該程式碼—在此它們只是為了刺激你的學習慾望。如果你的電腦上已經安裝了 Python，請隨意執行它們兩者。如果你還沒有安裝 Python 或 `pyserial`，而且你已經等不及了，請先翻閱到第 134 頁“安裝 Python 和串列函式庫”並將它們兩者都安裝。如果你想獲得更多更好的 AVR 編程學習，不要氣餒；稍後我會讓你了解整個有關 Python 的事情。

總結

在深入探討類似如此複雜的應用程式之後，你可能已經忘記了我們來到這裡的初衷—為了學習 AVR 的串列周邊裝置。

使用 USART：

1. 選擇一個波特率，在此透過定義 `BAUD`，並且在波特率暫存器 `UBRRL` 和 `UBRRH` 寫入適合的參數值。（*setbaud.h* 函式庫會利用這個來協助你。）
2. 開起串列接收和發送的暫存器位元。
3. 如果你正在傳送中，等到硬體準備就緒 (`loop_until_bit_is_set (UCSR0A, UDRE0)`)，然後把位元組數據載入 `UDR0`。AVR 的硬體會自動處理其他一切事物。
4. 如果你在等待數據，請檢查數據接收位元 (`bit_is_set(UCSR0A, RXC0)`)，然後從 `UDR0` 讀取數據。自動地閱讀 `UDR0` 會清除數據接收位元，而 AVR 會準備好接收下一個位元組。

此外，我把這些簡單的語句包含到一些我覺得方便使用的函數裡。我一定會在程式碼裡使用 `initUSART()`、`transmitByte()` 和 `receiveByte()` 函數。如果你喜歡挑戰，你可以查閱 *USART.c* 檔案，並觀察我是如何製作其他實用的串列函數，用相當於二進制或十六進制的方式來列印位元組，並且一次列印出整個字串。

把你的大型桌上電腦連接到 AVR 內部的小型電腦提供了許多創新的可能性。你的電腦可以攫取及解析網站、查詢股票價格，並執行大量的數學運算。AVR 可以驅動馬達、點亮 LED、感應燈光亮度，即時回應按鈕的按壓、並且與 DIY 的電子世界通訊。把它們倆者整合在一起是一個致勝的關鍵，而串列埠就是你的連接方式。