Chapter 1

A Metronome

This project demonstrates the use of a very low-end microcontroller to implement an application that has relatively little interaction with the outside world, but is required to maintain several accurate timers. The application is an electronic metronome - a device that produces a “click” sound at a regular interval, mimicing a conventional mechanical metronome.

1.1 Requirements

Compared to its mechanical equivalent, the electronic metronome should offer these advantages:

- light weight
- elimination of winding (mechanical metronomes are clockwork powered)
- accuracy
- low cost
- ease of use

To achieve these without compromising portability, this device will obviously have to be battery powered. This then means its power consumption must be extremely low, both while operating and while not. To maintain ease of use a power switch should not be necessary.

1.2 Microcontroller Choice

These features constrain the choice of microcontroller - it must be low power, cheap but able to do the job with a minimum of external components (which add cost, weight and power consumption). There are several families of microcontrollers that could do the job, but the one chosen here is the Microchip PIC series, and the particular member of that family chosen is the smallest and cheapest, the PIC 16C54.

The 16C54 has the following characteristics that make it suitable for the job:

- Low cost
- Very low power consumption, and a sleep mode
- Small footprint (18 pin DIP)
- Good drive capability
- An on-board timer

The features of the chip that make the task a little challenging are related to its low-end nature:

- Limited ROM size (512 words)
- Limited RAM size (25 bytes)
- No interrupts
- Stack depth is two levels only

Of these, the only one that turns out to be a major problem is the RAM size. By choosing a realistic set of features to implement, and an appropriate software approach, the other limitations never become issues.

1.3 Hardware Design

The input/output requirerements of this device follow from its application - clearly it must have some way of generating a noise; the simple approach we take is to drive a small loudspeaker with a square wave from a port pin. Because the loudspeaker has a low impedance (8 ohms) and requires more current than a port pin can supply, some kind of amplifier will be necessary. For this example, a pair of transistors connected as emitter followers, and thus providing current amplification only will be used.

To allow the user to start and stop the metronome, we will need at least one input device. We also need some way of setting the desired rate, expressed in beats per minute. Since the range of rates can be quite wide (we have selected 10-200 beats/minute) it's out of the question to use a multi-position switch for this purpose. Instead the rate will be displayed on a set of 7-segment displays, with push-button switches to increment and decrement the rate.

These same switches will also be used for the stop/start control - the software will be designed so that a short press will stop or start the metronome, while holding down a switch will increment or decrement the rate.

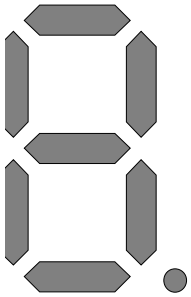

Figure 1-1: Seven-Segment Display

1.3.1 The Display

For the display, we will use three 7-segment LED displays. LED displays have good visibility in low light and are cheap. They are not very useful in bright conditions (e.g. sunlight) but for this application that is not important. A 7-segment display can display numbers and a limited range of letters by individually controlling 7 light-emitting diodes each consisting of a bar shape, and arranged in a figure-8 pattern as shown in figure 1-1 on page 2. There is also a decimal point, but the metronome display does not require this and it will not be used.

To display a number, the appropriate segments must be turned on to form the shape for the number. While there are special 7-segment driver chips available which can convert a number in binary format to a 7-segment pattern, this would be inconsistent with the ain of using a minimum of components. Instead, the microcrontroller will be used to drive the segments directly and the conversion to 7-segment form performed in software. This is done with a lookup table mapping the digits 0-9 into the equivalent 7-segment pattern

Each segment is actually a light-emitting-diode (LED) which requires a certain current (typically 20mA) passed through it to produce light. If we were to drive all segments simultaneously this would represent a very large current (7 segments by 3 digits by 20mA is nearly half an amp!), and in any case would be beyond the capabilities of the microcontroller (a single output can drive 20mA, but the total current sourced or sunk by the device is limited). So instead a multiplexing scheme will be used where only one segment at a time will be lit, but each one will be lit in turn at a rate much faster than the human eye can perceive, creating the illusion of continuous illumination.

It might be though that each segment will appear very dim if it is lit for only 5% of the time - in fact this is not the case, because the human eye is sensitive more to the peak brightness of an object than to its average brightness. Strobing the segments at a 5% duty cycle reduces the apparent brightness only by about half.

LED 7-segment displays come in two basic forms - common cathode, where all the cathodes (or negative terminals) of each diode are connected together, or common anode where all the anodes or positive terminals of each diode are connected. In this case the display used is a common cathode type, the FND359. The circuitry used to drive the displays will also accomodate common anode types, but the software would have to be changed.

Since the decimal point is of no use in this application, the seven other segments will be paralleled and driven by seven outputs from the microcontroller, and the three common cathodes will be driven by three other outputs, via current limiting resistors. The software will drive one digit output low, and one segment output high at any time to light one segment.

1.3.2 Switch Sca*nnnin*g

The display driver uses a total of 10 I/O pins - the speaker requires one more, leaving only one free from the 12 available for the switches. To scan two switches with one input requires that each switch be

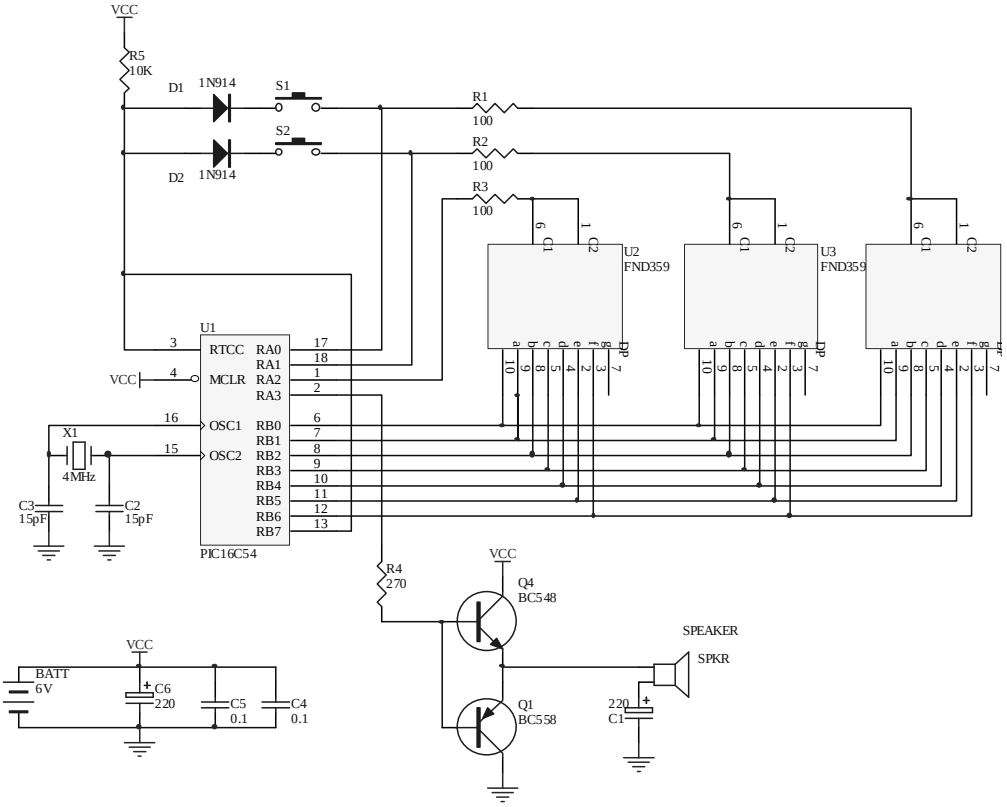

Figure 1-2: Metronome Circuit Diagram

strobed by one of the digit select outputs, and via an isolating diode pulls down the spare input. This means only one switch is enabled at any time, so it is possible to distinguish which switch is currently pulling the input pin low. If the input pin is high, neither switch is pressed.

1.4 The C*om*plete Circuit

tThe complete circuit diagram for the electronic metronome is shown in figure 1-2 on page 4. As well as the switches, LED displays and the speaker already discussed, the crystal and associated capacitors which provide the microcontroller's clock frequency are shown, as is the battery and filter capacitors.

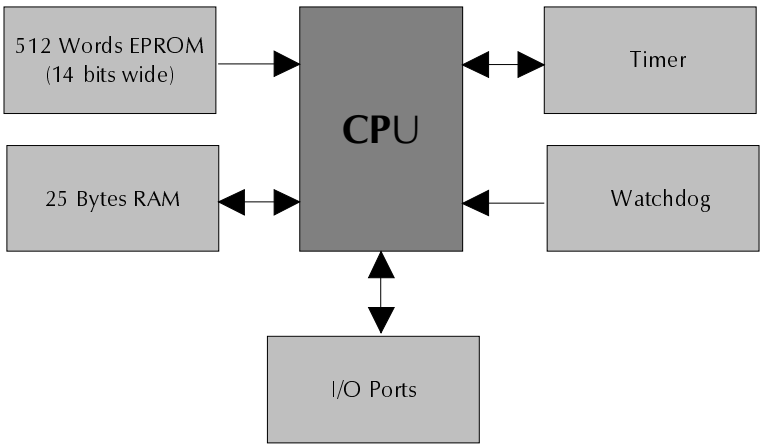

Figure 1-3: PIC 16C54 Overview

The */MCLR* input is tied high (to Vcc) as we only require resetting the processor on powerup. The PIC chips detect poweron and automatically reset themselves without any special circuitry. The *RTCC* input is not used, but is tied to the switch scanning input pin as it should not be left floating.

1.4.1 Clock Generation

A 4MHz crystal has been used for this project - this is the maximum frequency that can be used with the standar 16C54 (a 10MHz version is also available). The clock signal can also be provided by an RC time constant, or by a ceramic resonator. The RC time constant would not be accurate enough for this application, but a ceramic resonator may be usable in place of the crystal. A lower clock frequency could also be used, which would have the advantage of reducing current consumption, but would reduce the timing resolution and could cause problems if the calculations performed by the software could not be completed in time for the next display update.

1.5 The PIC 16C54 Features

For full information on the PIC 16C54, consult the appropriate Microchip documentation. A great deal of this documentation is now available on-line via Microchip's World Wide Web site at **http://www.ultranet.com/biz/mchip**/. However a brief overview of relevant aspects of the chip will be presented here. The Microchip PIC 16C54 microcontroller has the internal structure shown in figure 1-3 on page 5. l

1.5.1 Program Memory

The EPROM holds the program, in 512 words, each 14 bits wide, of memory. Each word is one machine instruction. The software for this project will be written in C and translated to PIC machine instructions by the C compiler.

1.5.2 Data Memory

The RAM available is only 25 bytes - the 16C54 can address 32 bytes of data memory, but the first 7 bytes of this are occupied by special memory locations which allow the timer, I/O ports and other registers to be accessed. Because of the number of timer functions required, almost all this RAM will be used.

1.5.3 Timer

The timer subsystem of the this device is a simple 8 bit up-counter. It can be reloaded under software control to produce division by values less than 256, and a prescaler is provided that can extend the timer period by powers of 2, from 2 to 256. The timer is clocked either by an external input (*RTCC*) or by the CPU clock, which is one-quarter the crystal frequency.

1.5.4 Watchdog Timer

The watchdog timer is selectively enabled when programming the EPROM of the chip, and if enabled will reset the chip if it is allowed to time out. The period of the watchdog timer is independent of the crystal frequency, and is nominally 18mS, but varies with supply voltage, temperature etc. Particular use will be made of the watchdog to restart the processor periodically during a power-down state. The same prescaler used for the timer can be used for the watchdog, though not at the same time.

1.5.5 I/O ports

The 12 I/O pins are all bidirectional, and are switched between input and output under program control Each pin can source or sink up to 25mA current, enough for this application. When in input mode, the pins are high-impedance. The ports are organized into two 8 bit registers, however port A only implements the lower 4 bits. There are also two special registers that are used to individually define each I/O pin as input or output.

1.5.6 Special Registers

Some of the special registers in the 16C54 are shown in Table 1-1 on page 7 The important bits in these registers are the */TO* bit, which is used to detect when the chip has been reset by a watchdog timeout.,,

Table 1-1:

Register	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
STATUS	Not Used			/TO	/PD	Z	DC	C
OPTION	Not Used		RTS	RTE	PSA	Prescaler select		

and the bits in the OPTION register, which are used to select the clock source for the timer, and to assign the prescaler to the timer or watchdog.

1.6 C *Com*piler Suppo*r*t

The C compiler used for this project is the HI-TECH Software C cross compiler for the PIC series. A header file is provided for the 16C54 which defines the various registers that need to be accessed from C code.A copy of the file is in figure Table 1-4 on page 8. Note that most of the registers are simply memory mapped at specified absolute addresses, but the OPTION and similar registers are qualified *control* which tells the C compiler that access to those registers requires special instructions. This means that in the C code these registers can be assigned to like any other C variable, and the compiler will automatically take care of the correct code to perform the assignment. If the code was written in assembly language, it would be necessary to know how to access these registers.

1.7 The S*o*ftware

The complete program for controlling the metronome is given at the end of this chapter. Excepts from this will be presented and discussed here.

1.7.1 *M*appi*n*g Table t*o* 7-Seg*me*n*t* Patter*n*s

Firstly, the table to map numbers to 7-segment patterns:

```
4: static const code unsigned char digits[10] =
5: {
6:      0x5F,       /* 0 */
7:      0x06,
8:      0x3B,
9:      0x2F,       /* 3 */
10:     0x66,
11:     0x6D,
```

```
/*
 *       Header file for the Microchip PIC 16c54 chip
 */

static unsigned charRTCC@ 0x01;
static unsigned charPC@ 0x02;
static unsigned charSTATUS@ 0x03;
static unsigned charFSR@ 0x04;
static unsigned charPORT_A@ 0x05;
static unsigned charPORT_B@ 0x06;
static unsigned charPORT_C@ 0x07;

static unsigned char controlOPTION@ 0x00;
static unsigned char controlTRIS_A@ 0x05;
static unsigned char controlTRIS_B@ 0x06;
static unsigned char controlTRIS_C@ 0x07;
```

Figure 1-4: C Header File for the 16C54

```
12:     0x7D,
13:     0x07,       /* 7 */
14:     0x7F,
15:     0x6F,
16: };
```

There are 10 entries in the table, mapping the digits 0-9 into equivalent 7-segment patterns. In this case since we are using common cathode displays, and thus a segment line will be driven high to turn it on, a one bit in the pattern will correspond to a lit segment. If the displays were common anode, the reverse would apply.

1.7.2 Constants

Although the code is not completely parameteric in terms of clock speed, most of the values needed are derived by comile-time calculation by specifiying the clock speed, desired tick rate etc. here:

```
18: #define TOUT    0x10                    // watchdog timeout bit in STATUS
19: #define XTAL    4000000                 // crystal freq
20: #define PRE     4                       // prescaler value
21: #define INTVL   1000u                   // 1000uS (1mS) per loop
```

```
22:
23: #define DIVIDE  (XTAL/PRE/INTVL/4)      // division ratio - must be < 256
24: #define SECS(x) ((x)*1000000/INTVL/21)  // convert secs to loop count
25:
26: #define DEBOUNCE        10              // debounce 10mS
27: #define COUNT           40              // this many debounce intervals
28: #define CYCLES          10              // this many cycles per “tick”
29: #define PWRDOWN         30              // power down time
```

The values defined here are:

TOUT A bit definition in the STATUS register (not provided in pic16c54.h) for the timeout flag

XTAL The crystal frequency in Hertz - other timing values are derived from this.

PRE Our chosen prescaler value

INTVL The duration of each inner loop of the main program - this value will determine the scanning rate for the display and the frequency of the tone generated for each click.

DIVIDE A calculated value from the preceding values, which will be used to reload the counter register each time around the loop

SECS A macro to convert a time in seconds into an outer loop count. The outer loop comprises 21 iterations of the inner loop - this is derived from the fact we have three digits each of seven segments to scan.

DEBOUNCE

We choose to wait this many inner loop counts after a switch closure to be sure that the switch is actually closed, and any switch bounce has settled. See the discussion below on switch debouncing.

COUNT This is the number of debounce intervals to wait to distinguish a short press of a switch from a long press - a short press stops or starts the metronome, a long press increments or decrements the programmed rate.

CYCLES This is the number of clock periods to toggle the speaker output to generate a click sound.

PWRDOWN

This value in seconds is how long the display will remain lit after the a switch is released.

1.7.3 Variables

The variables are defined in two places - some outside the main() function, and some inside main(). The variables defined outside main are there because they are either bit variables, which can only be defined outside a function, or because they are qualified *persistent* to ensure they are preserved across a reset. The variables at the outer level are:

```
31: persistent static unsigned char rate;   // current rate
32: static bit      btn_1, btn_2;           // push buttons
33: static bit      spkr;                   // speaker bit
34: static bit     counting;              // counting up or down
35: static bit     fast;                 // we are counting fast!
```

```
    36: static bit      display;                // display on
```


rate — This is the current rate that the metronome is set to. It is expressed in beats per minute, and an 8 bit variable gives a range of 0-255, which is adequate for our selected range of 10-200. It is qualified *persistent* which tells the compiler it should not be cleared on startup. This in conjunction with the continuous battery power provided to preserve RAM contents will ensure that the programmed rate is not lost if the metronome is shut off.

btn_1 — This is a bit variable, and represents the current state of button 1 (stop).

btn_2 — Another bit variable, representing the current state of the other button.

spkr — A bit variable which is set if the speaker is currently enabled, i.e. the metronome is operating.

counting — This bit variable is set while the display is counting up or down, i.e. while a button is held down.

fast — After 5 counts, the the rate will be incremented or decremented by 10 rather than 1. This bit variable is set to indicate the faster counting rate.

display — The LED display will be switched off after 30 seconds to reduce current consumption. This bit variable is set while the display is enabled.