Chuck Baird
http://www.zbaird.net

Timeout Timer

As every parent knows, young children sometimes need to take a break from terrorizing the world and sit quietly to settle down. This serves two purposes – the kid gets a cooling off period, and the parent doesn't go to jail for acting out their impulses. Most experts agree that a timeout should be presented not as punishment but rather as quiet time, and many recommend timeouts of one minute per year of the child's age.

This article describes a simple timeout timer with a user interface of one pushbutton, eight LEDs, and some noise (some would generously say "music") for marking certain events. As well as being a useful device, it also provides a look at several software techniques and a chance to play with a versatile yet inexpensive development system.

Operation

Initially the adult turns the power on, then presses and releases the pushbutton. This initiates a repeating sequence of slowly lighting one to eight LEDs. When the number of lit LEDs matches the number of minutes of timeout desired, the button is pressed again. When the button is released, the timing cycle for the chosen number of minutes begins.

All eight LEDs then light up with the rightmost one flashing. It will flash faster and faster until it finally goes out, then the process repeats using seven LEDs, then six, and so on. When all the LEDs have turned off, a tune plays, and the timeout period is over.

If the time chosen is one minute, each LED goes through its flashing sequence in 7.5 seconds for a total of 60 seconds. At the other extreme, if the time chosen is eight minutes, each LED's flashing sequence is one minute long. In theory the errant child will find some diversion in watching the changing rate of flashing LEDs.

As a reminder to the adult to turn off the power, when the timer is not in use but left on it will beep four times every 15 seconds. Although there is a low power sleep mode that could be utilized, it was assumed that this device might sit for weeks between uses, and that zero power consumption would be better than a sleep mode.

If the button is pressed during the timout period, the MCU is reset and the process repeats.

MCU Overkill

The Timeout Timer is built around the Atmel AVR Butterfly, a $21 (Digikey) "demonstration and evaluation kit" that is extreme overkill for this project. It has an 8 MHz Atmega169 processor, a 6 character LCD, a piezo element for noise making, a 4 megabit dataflash memory, a joystick, a temperature sensor, 16K flash, 1K SRAM, and 512 bytes of EEPROM. It also has RS-232 level converters and is shipped with a bootloader installed that allows its flash memory to be programmed via a Windows serial port with no additional hardware other than a DB-9 connector. The programming software, including an assembler, and a full featured C compiler (WinAVR), are free.

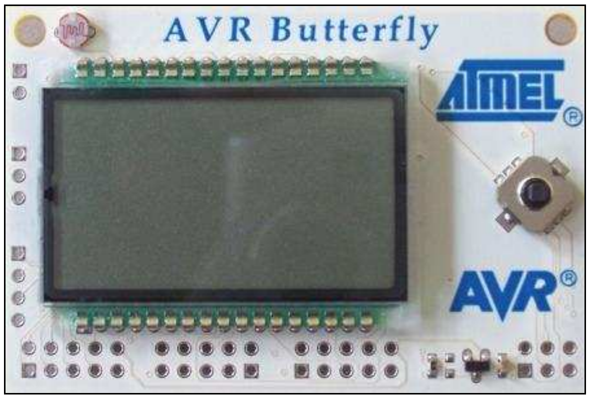

Figure 1 – The AVR Butterfly
Newer models no longer include the photo resistor (upper left)

With such an array of peripherals most of the general I/O pins on the Butterfly serve dual or triple purposes. We need one pin for the pushbutton input, eight pins for the LEDs, and one pin for the piezo element mounted on the Butterfly. In addition we use an I/O output as an interrupt "heartbeat" to allow monitoring (via oscilloscope) to verify the time base, although this could be omitted. We sacrifice some of the LCD lines to get these eleven I/O pins, but we are not using them anyway for this project so that is not a problem. There will be some random LCD display activity when we use the shared I/O pins, but it can be ignored.

The Butterfly's bootloader allows it to be programmed (read, write, and verify flash memory) through its serial port. The Windows programming software ("AVR Studio") may be downloaded free from Atmel, and your PC will need a serial port or a USB to serial converter. No additional hardware is needed, and we assume in the following discussion that this will be the programming method used. However, the Butterfly may also be programmed in a variety of other ways by using a separate hardware programmer. These alternate methods will usually erase the bootloader, but both the bootloader and the application that ships with the Butterfly are available as hex files on the Atmel site and may be reloaded. The bootloader cannot modify itself, so unless you use one of the other programming methods it should remain intact.

This project consists of wiring a DB-9 to the Butterfly's RS-232 pins for use by the bootloader, connecting LEDs and resistors to eight of the output pins, and adding a pushbutton to ground to another I/O pin. I used two AA batteries to supply the 3v power since the Butterfly's coin battery isn't quite up to this job. There is a switch on the batteries, and you may wish to add an LED and resistor for a visual reminder. We then burn the hex file from this program into the Butterfly's flash memory, typically by using the Butterfly's bootloader and AVR Studio.

The code for this project is written in ImageCraft C ("ICCAVR"). Other C compilers will have slight syntactical differences but will be essentially the same. The ICCAVR convention that might be of concern to the casual reader is that char declarations are unsigned.

Besides the free WinAVR GCC port, the various commercial C compilers for the Atmel AVRs have free demo and trial versions for non-commercial use if you wish to modify the source code. To create your own timer using the compiled version, the $21 Butterfly and a few common parts are the only expense needed.

You may wish to download the Butterfly's documentation and the datasheet for the ATmega169 from the Atmel website. As mentioned, you will need to download and install the free AVR Studio software to communicate with the bootloader unless you use other software tools. You can find all these items with the Atmel site's search function.

Bootloader Considerations

As simple as this project is, there are several things that complicate it. We also have plenty of computing resources available here, so we will add to the complication for amusement and education, and use some interesting coding techniques.

The first consideration is that we may or may not be running code that was loaded via the Butterfly's bootloader. If the bootloader were not used, the power on reset jumps directly to our code. If the bootloader were used, the power on reset jumps to the bootloader, which then waits for an indication of whether it should accept input for burning flash, jump to the application (our program), or go to sleep.

The up and in joystick switches wake the bootloader out of sleep mode, and it jumps to the application if it sees the joystick pressed up. Therefore we will wire our pushbutton in parallel with the up joystick button, and that will cause the initial button push to entice the bootloader to run our code, in case the bootloader were used. In that case, pressing the button once will simply start our program running (it plays a startup tune). A second press starts the process of selecting the timeout minutes.

If a programming method other than the Butterfly bootloader is used, then the program (with its startup tune) will run immediately on power up or reset.

Another issue is that the bootloader calibrates the speed of the MCU when it starts up. The Butterfly's MCU runs off its internal R/C oscillator, which AVRs can fine tune by adjusting a register. The bootloader tweaks the MCU speed by comparison to the Butterfly's 32KHz clock crystal, thus insuring its RS-232 communications timing will be as accurate as possible. This adjustment will not be performed if we bypass the bootloader, although for this application it does not matter since exact timing is not necessary except for the bootloader's serial communications.

The bootloader leaves the MCU running at 2 MHz, which we will bump to 8MHz. It also leaves a few other little messes (at least in some versions of the bootloader) which we will clean up.

Timing is Everything

Since this is a timer, we need to have some sort of a timebase running. We'll use one of the Atmega169's 8 bit timers to generate an interrupt once every millisecond, and then derive our various timing needs from that. As mentioned, in this particular application exact timing isn't critical – it doesn't matter if a six minute timeout actually lasts 6.12 or 5.94 minutes. Not that we will be far off, but we won't worry about running off an R/C clock, or whether it has been calibrated. Most of the timing considerations are handled inside the 1ms interrupt handler.

One of the things we will do is multiplex the LEDs whether we need to or not. This will cause them to appear dimmer, but will limit the current drawn. The main reason for doing it here is to demonstrate one method of multiplexing outputs.

The program will view the LEDs as a single unsigned byte of eight bits, and if a bit is set, its corresponding LED lights up. The mechanism behind the scenes will be a second mask byte with exactly two bits set, and on each interrupt the mask and the main byte are ANDed together to determine the outputs. The mask bits are then circularly shifted in preparation for the next interrupt. Thus each LED is serviced every 4 interrupts.

Pushbutton debouncing is also handled by the interrupt handler. Every so many interrupts (currently 2) the pushbutton is read, and its single bit value is shifted into an unsigned byte variable. As it takes 8 shifts to "fill" the variable, at any point in time checking that variable gives the value of the last 8 reads. With the shift happening every 2 interrupts, that gives a debounce time of 16 milliseconds, probably considerably longer than necessary. If the variable equals 0 the pushbutton is not being pushed and is debounced. If the variable is 255 (0xff) then the pushbutton is pushed and debounced, and any other value means it is in transition.

The programming "gotcha" with the debouncing is that the pushbutton's variable must be declared volatile since its value changes in the interrupt handler. This is something the compiler wouldn't normally expect, so it needs to be told to reload the variable for each use. Otherwise the tough love of the optimizer could kill you, or at least your code.

The heartbeat output is not really a necessary part of this project, but sometimes it is useful to be able to verify the interrupts are working, the timer setup has been correctly performed, and the clock rate is correct. On every interrupt we toggle an output bit, so an oscilloscope should show a square wave with a half period of 1 ms.

The playing of sounds is also handled by the interrupt handler. This is explained in greater detail below.

The Main Event

For the general timing needs an event list was implemented. An event in this program consists of a function that will be executed at some future time, measured from the present time. When the time for the event arrives the function is called.

Event times are specified in tenths of a second. Every 100 interrupts the interrupt handler updates the active events' pending times, eventually decrementing them to zero. When the main task sees an event with zero time remaining, it removes the event from the list and executes the event's function.

For example, when the program starts out it sets an event for 1 minute, or 600 time units. If this event executes, it will start the beeping as a power down reminder mentioned above, sort of the poor person's watchdog timer interrupt. In the meantime, other events are loaded and executed which display the initial sequence of lights so the user can select a timeout value. When and if the selection is made, the shutdown event is cancelled.

Because the event list is modified by both the interrupt handler and the main task, a locking mechanism was necessary to prevent conflicts. The main program sets and later clears a variable to indicate when the event list is busy. If an interrupt needs to access the locked list, it sets a flag for itself and performs the action on the first interrupt following the unlocking of the list.

While there are a myriad of ways to implement the timing needs of this project, the event list is versatile, interesting, and effective. It also simplifies the interrupt handler's concerns about managing pending tasks.

Play Misty For Me Again, Sam

Finally there is the matter of the "music." As written the sound routines can play about anything, although making a few simple sounds would have sufficed for this project. It was implemented in the more complete form to allow for code reuse and to demonstrate the general principles.

The technique used is that a sound or song is encoded as a series of notes, each identified by common notation. The tempo, note duration, volume, and a repeat count may also be included with and within each song.

The ATmega169’s 16 bit timer is used in one of its PWM modes to generate each note, and conveniently (not an accident) its PWM output bit on the Butterfly is tied to the piezo element. All we have to do is feed the timer a PWM frequency count to start each note at the correct time, and the PWM duty cycle determines the volume (more or less).

To play a song or make a noise, the main task supplies the address of the song definition and sets a flag, and the interrupt handler does the rest. This involves determining the next note, starting the PWM output at the appropriate frequency, and sustaining for the note’s duration across subsequent interrupts. After that it either plays the next note, repeats the song, or shuts down the sound mechanism as appropriate. In other words, once a song is started it plays to completion with no further intervention outside the interrupt handler.

The Implementation

The application, written in ImageCraft C, is split into three source files and two header files. There are header and source files for the event list, header and source files for the music functions, and everything else is in the main source file. The processor dependent definitions are in an ICCAVR include file, as are some macros.

The frequency definitions for the sound generation were derived from the Butterfly’s demo application, although all code in this project is original. Atmel has published the source code for their demo, and if you are interested much can be learned from studying it.

Wiring the Butterfly

Figure 2 shows the connections for the Butterfly. Leads may be directly soldered to the board, or standard 0.1” headers may be used. The +3v supply is switched with an optional LED and resistor to ground, and the Butterfly’s coin battery should be removed. Actual port and pin assignments are shown in comments in the code. Resistor sizes will vary depending on the LEDs used, 220 ohms being typical.

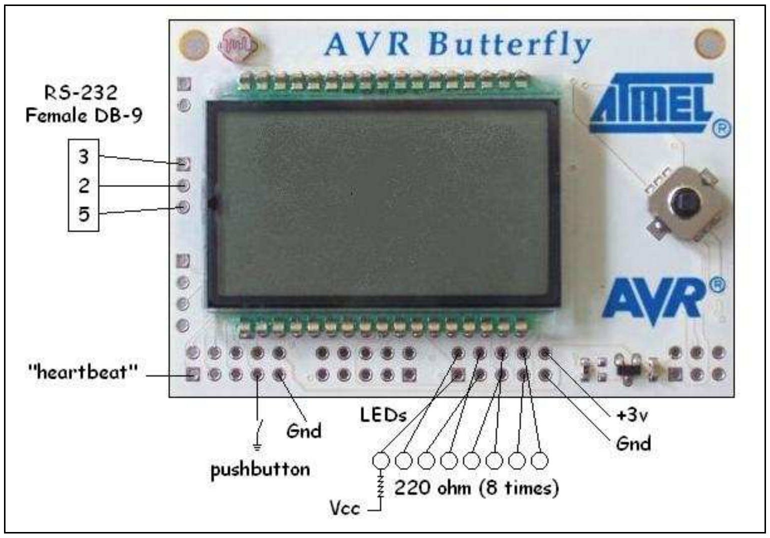

Figure 2 – Connections to the Butterfly

Figure 3 shows the prototype with the DB-9 wired to a header so it can be removed following programming. The red LED is used to show that the power is on.

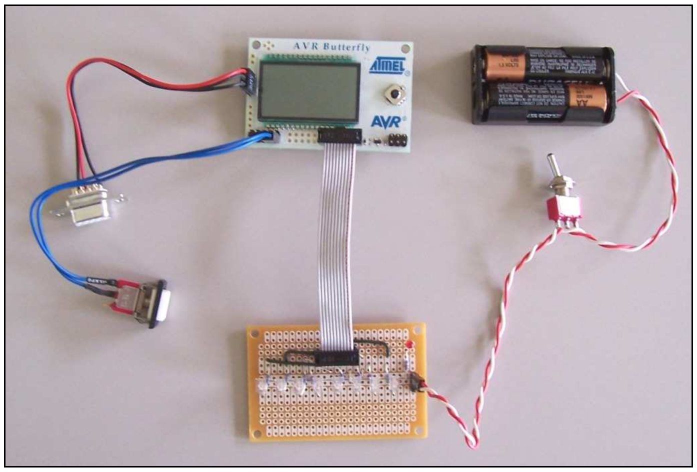

Figure 3 – Ready to go, but not dressed up yet

As for a case for your Timeout Timer, anything goes. Figure 4 shows a mounting in a rectangular plastic case. The square button on the top is power, and the round button is the input pushbutton.

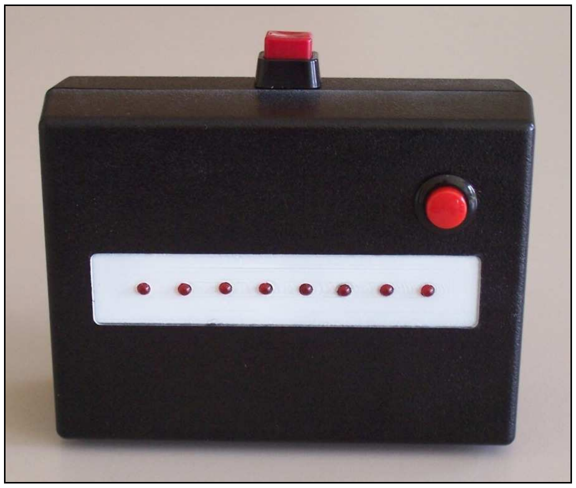

Figure 4 – Simple plastic case enclosure

I also had a soft plastic Lion King™ drink cup lid from a family trip to Disneyland some years ago, so I used it to make a case for a timer. Of course there are trademark issues so while I paid retail for the drink and cup, this particular timer could not be resold. The finished result is shown in Figure 5, with the pushbutton mounted in the hole where the straw protruded (in the back) and the power switch on the bottom.

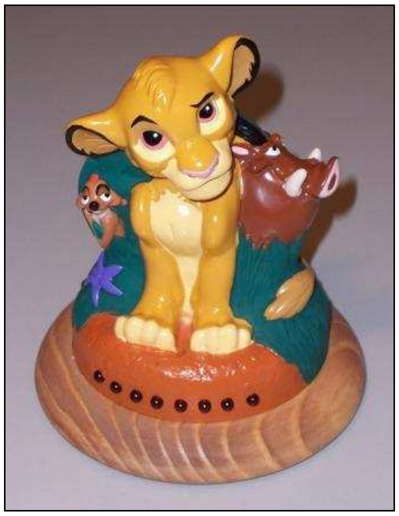

Figure 5 – The Lion Timer

In any mounting, you will want to make sure the piezo element's sounds can be heard. One option is to relocate it from the back of the Butterfly to a more audible position, but otherwise you may need to creatively locate the Butterfly within your case.

Conclusion

As mentioned, using a Butterfly for this application is extreme overkill, but the gains of having minimal hardware construction, no additional programming hardware, free development tools, and the Butterfly's reasonable price make up for the wasted resources. I believe you will find, as I did, that the Butterfly is an impressive board with great potential, and I hope you enjoy your encounter with it.

Resources

http://www.atmel.com
AVR Studio software
Butterfly documentation (also included in AVR Studio help files)
Atmega169 datasheet
Other AVR related application notes, documentation, etc.

http://www.digikey.com
Butterflies, and everything else but the kitchen sink

http://www.imagecraft.com
Free 45 day trial version of ICCAVR C compiler, size limited beyond that
AVR Studio plugin for integrated compatibility (optional; allows the normally standalone compiler to be used within AVR Studio)

http://www.smileymicros.com
Source for Butterfly and support materials, including books

http://www.avrfreaks.net
The end all and be all user community for all things AVR
You may contact the author through user name zbaird