



[Maxim](#) > [App Notes](#) > [MICROCONTROLLERS](#)

Keywords: bicycle, maxq2000, heartrate

May 24, 2006

APPLICATION NOTE 3845

The MAXQ Microcontroller in Action: Designing a Bicycle Computer with the MAXQ2000

Abstract: Built for an avid cyclist who dreams of winning the Tour de France, the compact bicycle computer can help a rider through workouts. The bicycle computer is based on a MAXQ2000 microcontroller, and includes a speedometer, clock, thermometer, humidity sensor, heart-rate monitor, and stopwatch.



Many people do not know this, but I have been training for the Tour de France. Like another somewhat more famous Texan, I might even win the race someday. Sure, that other famous Texan has strength, determination, guts, courage, and the stamina to face the grueling climbs through the French Alps. But I have something on my side that he does not. I have the MAXQ2000 microcontroller.

The MAXQ2000 is the first of Dallas Semiconductor's MAXQ family of fast, quiet, low-power microcontrollers for embedded applications. (See **Appendix 2** for more information on the MAXQ microcontrollers.) The MAXQ2000 has an LCD controller, SPI port, real-time clock, two UARTs, a hardware multiply-accumulate unit (MAC), and enough general-purpose I/Os to handle many tasks. With its single-cycle core and low-power modes, the MAXQ is capable of doing a lot of work in a short amount of time and with very little battery power.

But how exactly, you ask, is a microcontroller going to help me win the Tour?

Getting Trained—Essentials for a Bicycle Monitor

There are two parts to me. There is the part that wants to get out and exercise, and then there is the engineer who would rather build gadgets to help improve my exercise. With the Tour de France as my goal, I decided to use the MAXQ2000 to build a battery-powered bike monitor. To improve my performance on the bike, I needed a speedometer, clock, thermometer, humidity sensor, heart-rate monitor, and a stopwatch. The MAXQ2000 has all the right tools to solve this problem.

The MAXQ2000's LCD controller supports up to 132 segments, which is far more than I needed for this application's display.^[1] I used the MAXQ2000's real-time clock (RTC) to track the current time and run my stopwatch. A DS1923 Hygrochron iButton managed my humidity and temperature measurements.^[2] (See **Appendix 1** for more information on the DS1923 Hygrochron iButton.) The two remaining bike monitor functions—measuring the bicycle's speed and measuring my heart rate—were more interesting (and difficult).

For the speed measurement, I used a strong magnet and a DS2423 1-Wire counter. The magnet was placed on the wheel of the bike, activating the DS2423's counter on every wheel revolution. The speed calculation was then just mathematics, if I could remember my high school geometry.^[3]

The heart-rate monitor was the last hurdle before I dove into the design. I have a Polar F1 heart-rate monitor, which straps around my chest. Although it has a watch to monitor your heart rate, it was more fun to design a heart-rate monitor into my application. After a bit of research, I found that the Polar transmitter generates short, 5kHz pulses on every heartbeat that can be picked up by an inductor coil. A few op-amp stages could filter and amplify the signal into something that the microcontroller could receive.^[4]

With the upper-level design decisions made, it was time to start prototyping the application using the MAXQ2000 Evaluation Kit.

Mapping the Route—Prototyping the Device

The MAXQ2000 Evaluation Kit is a good place to start the prototyping—it has the hardware to help me code and test many of my bike monitor's functions. It comes with a 4 and 1/2 7-segment LCD display, two pushbuttons, and all the I/Os that needed to connect the Hygrochron, DS2423, and heart-rate circuit. Once I prototyped the application, I could do my own custom board—something that fit conveniently on my bike handlebar.

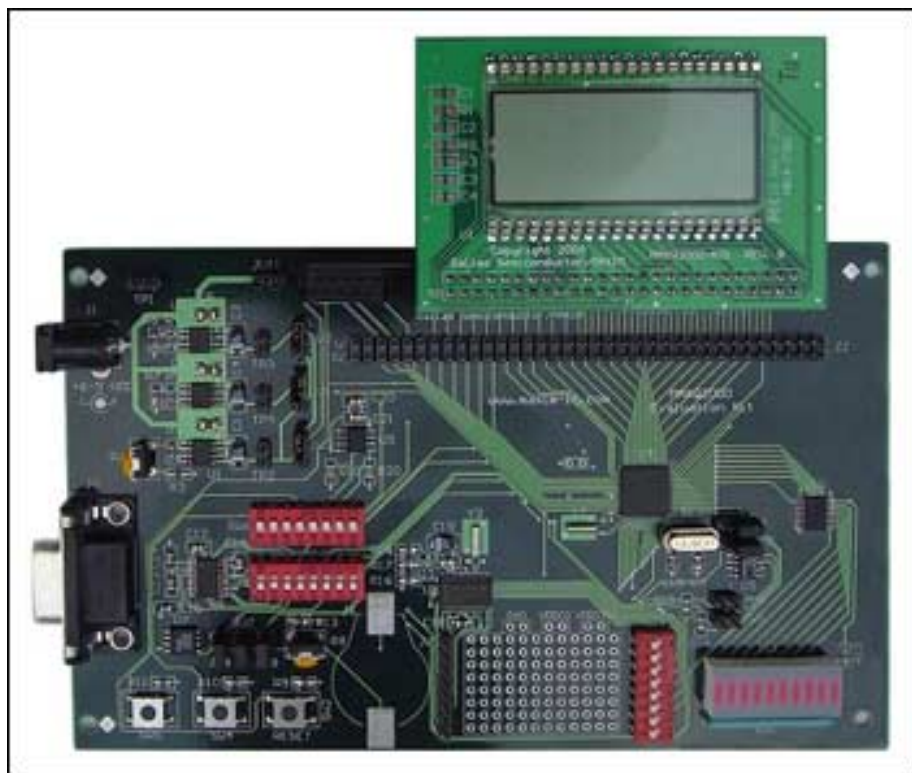


Figure 1. The MAXQ2000 Evaluation Kit has all the tools needed to prototype my bike monitor application.

Since this application is battery-powered, I needed to write my application with an eye for efficiency. Wherever possible, I used the MAXQ2000's low-power stop mode to conserve battery life, then used pushbuttons (external interrupts) or a real-time clock alarm to wake up the microcontroller.

I wanted my application to be menu driven. One push button was used to select the current application mode (i.e., speed, temperature, clock), and the other buttons were used for further control, like starting and stopping the stopwatch.

The application mode was stored in a global variable, which made the menu portion of the application simple. It only needed to wake up, change the application mode, display new data, and go back to sleep. This all happened quickly, so it was very "power-friendly."

The challenge for battery life was the sensor readings. This application had many sensors to monitor: one temperature and humidity sensor, one speed sensor, and one heart-rate sensor. Performing a temperature or humidity conversion takes 750ms, an expensive operation I avoided whenever possible. I managed with a temperature and humidity reading once every 5 seconds. (I probably will not be riding through weather changes more severe than that, even in the French Alps.)

The speed-sensor reading took much less time, but it raised some interesting design issues. We are not really measuring speed; we are measuring complete wheel revolutions in a given period of time. Since we do not know exactly when the magnet passes by the DS2423, we cannot simply use the amount of time between wheel revolutions. Instead, our algorithm needs to track a few seconds worth of entries and average them out. Consequently, our bike monitor would not react instantly to changes in speed, but should instead measure our average speed.

Practice Runs—Planning and Coding the Menu

Before I started writing any code, I defined my menu system—what would be the functions of each button. The first button would cycle among the different modes (temperature, speed, etc). The remaining buttons would be used for options within each mode. This menu (**Table 1**) not only helped me list all the features that needed coding, but it also identified how many pushbuttons my custom board needed.

Table 1. Menu for the Bicycle Monitor*

Mode	Button 2 Function	Button 3 Function
Clock	Hours++	Minutes++
Speed	Change MPH-KPH	None
Temperature	Change F/C	None
Humidity	None	None
Stopwatch	Start/Stop	Reset
Heart Rate	None	None

**This table was generated to determine how the main menu for the application would work, plus what functions the other buttons would perform.*

Since the MAXQ2000 Evaluation Kit has two pushbuttons, I was able to prototype most of this functionality before moving to my custom hardware.

As a starting place for the code, I implemented the main menu loop (**Listing 1**). The most important parts of this code are changing the **application_mode** variable and the function call to **Display()**, which displays the time, temperature, or other data on the LCD screen. The other code in the menu handler implements some power optimizations; the **dotemperature**, **dohumidity**, and **docounter** flags indicate whether or not the program should check those sensors when it wakes up. Otherwise, the program will skip those operations so the microcontroller can go back to sleep faster. The sleep time is also configured from the interrupt. If the device is in stopwatch mode, it needs to refresh the display often enough for the tenths-of-a-second place to update realistically.

[Listing 1. Main menu code for the bike monitor application. To save power, sensor readings are only enabled when needed.](#)

The **Display()** function's behavior depends on the current application mode. It will call the appropriate function to display data to the LCD screen. If the application is in humidity-sensing mode, for example, it will call the **showHumidity()** function. Functions like **showHumidity()** that are called by **Display()** are small and fast; they simply take some input data (a temperature or the current time) and format it into something meaningful for the LCD screen.

Writing data to the LCD screen which we chose is simple. Each character on the LCD screen is controlled by one LCD register in the MAXQ2000's register space. The LCD controller generates common and segment signals behind the scenes, so the software only needs to load a value into an LCD register to display a digit. The function **lcd_getlccdigit** looks up the correct constant to load into the LCD register based on the input number.

```
// display the number 5 in the second LCD digit
LCD2 = lcd_getlccdigit(5);
```

With a functional menu system in place, I started looking at the pieces of the application. The program flow for the main application loop is shown in **Listing 2**. This is where the MAXQ2000 monitors its environment, measuring the temperature, humidity, and wheel revolutions.

[Listing 2. Pseudocode for the main application loop. The key sensors are read and the application returns to sleep. The display functions called during the wakeup interrupt display the sensor data.](#)

The main loop is also responsible for debouncing on the pushbuttons; it disables the external interrupts, goes to sleep for a couple hundred milliseconds, then wakes up and reenables the pushbuttons. After the debounce routine, the code determines which sensors it needs to read. Results of those readings are stored in global data locations so that the **Display()** function can use them. After the sensor readings, the MAXQ2000 goes back into low-power sleep mode, only to be awakened by a pushbutton interrupt or an RTC interrupt.

The RTC interrupt is the pulse of the bike-monitor application: it is the normal way that the **Display()** routine is called to update the LCD screen. Once the RTC interrupt routine completes, the MAXQ2000 goes back to the main loop, starts any sensor readings, and returns to sleep, and waits for the next clock interrupt.

With the menu engine, clock alarm, and main loop coded, the core portions of the application were in place. Still working with the MAXQ2000 Evaluation Kit, we started to fill in the holes, to code the remaining functions of our application.

Temperature and Humidity Sensors

One of the first functions to be coded was the temperature and humidity measurements. The DS1923 can measure both; its rugged iButton form factor contains both a temperature sensor and a humidity sensor. To interface the DS1923, we used a port of the 1-Wire public domain kit, an open-source collection of routines for communicating with iButtons and other 1-Wire devices.^[5] Since the DS1923's measurements take 750ms and we are power conscious, we only measured occasionally: once when we entered the temperature (or humidity) mode, and then once every 5 seconds while we stayed in that mode.

Current Time Display

The MAXQ2000 also makes the clock display easy to implement. The integrated RTC accumulates the number of seconds in a 32-bit counter with 8-bits of sub-second resolution. On power-up, the RTC counter initializes to 0. To calculate the time of day, the application simply divides the number of total seconds by 60 to determine the number of minutes that have passed, and then divides the number of minutes by 60 to determine the number of hours that have passed. The values are then reduced modulo 60 and 24, respectively, to get the time of day.

Pushbuttons were used to set the current time—also a simple operation since the MAXQ2000's RTC counter can be set in software. To add an hour or a minute, the appropriate number of seconds is added to the RTC counter.

Stopwatch

Implementing the stopwatch was a bit more challenging. Using the RTC for this function was appealing since it was already keeping time. We needed to be able to start and stop the stopwatch at will, but we could not start or stop the RTC. A possible solution was to use one of the MAXQ2000's three timers, tuned to generate interrupts every 100ms that would then update the LCD screen. Rather than introduce another interrupt to the system, however, I implemented the stopwatch through a more software-oriented approach.

A stopwatch is nothing more than an accumulator of time. A simple stopwatch just remembers when it started and continually displays the elapsed time. For our application, however, the problem gets more complex as we needed a stopwatch that paused. (In our application the time elapsed since the start is no longer valid.) Instead, we considered *accumulated* time to solve this problem.

When the stopwatch starts from 0, we set the 'start time' variable to the time reported by the RTC, and we set the 'accumulated time' to 0. When the RTC interrupt hits, we update our 'current time' variable, only if the stopwatch is running; we display the difference between our current time variable and the recorded start time. If we stop the stopwatch, we set the accumulated time to the difference between the start and current times, and then set both the start and current time to 0. When the stopwatch restarts, we again set the start time to the current RTC time, so the stopwatch runs the same as before. The difference is that when we display the elapsed time, we also add the accumulated time. The code in **Listing 3** shows the logic when the timer is started or stopped.

[Listing 3. This code is the software glue to hold the stopwatch together. The challenging task is handled here. If the user pauses the clock, we need to accumulate the time, since we do not want to stop the RTC.](#)

Speed Measurement

To measure the speed, we needed some special hardware. A small circuit with a DS2423 and a reed switch needed to be mounted somewhere close to the front wheel, so the magnet on the wheel would pass our sensor on every revolution. The circuit was based on the same principle used by the old Dallas Semiconductor weather station for wind-speed measurements: a magnet passes over a reed switch and closes a contact, causing the DS2423 to increment an internal counter. **Figure 2** shows this circuit.

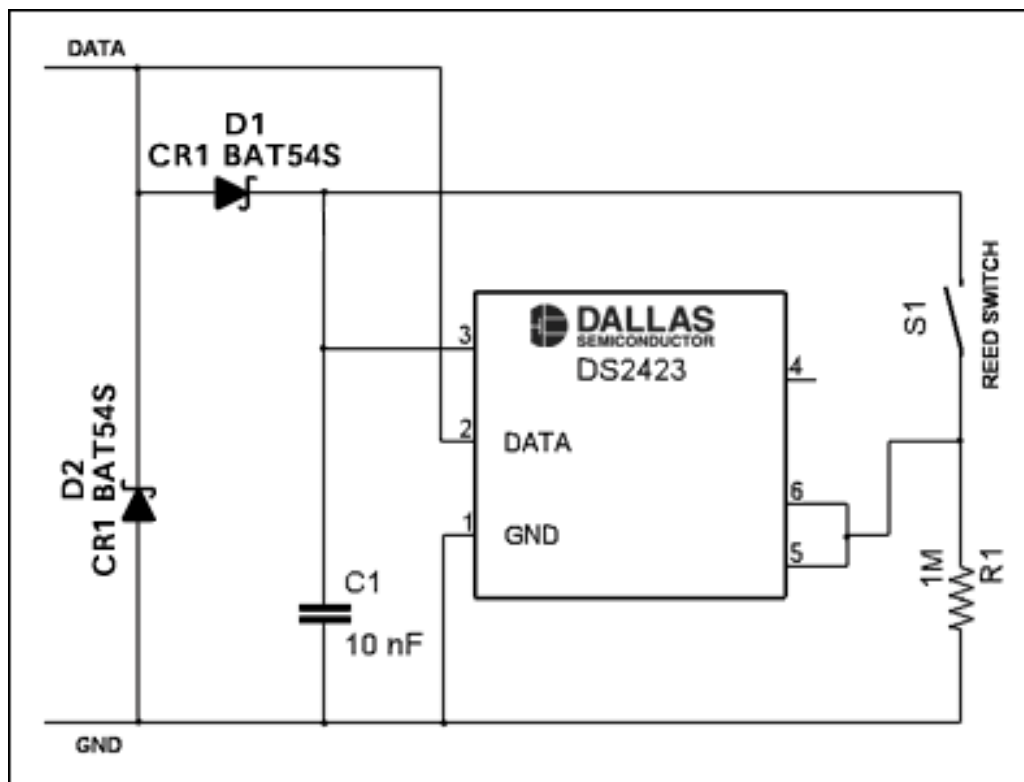


Figure 2. A 1-Wire counter, the DS2423, tied to a reed switch. When a magnet passes by, the switch momentarily closes, causing the DS2423's count to increment.

Since we already used code from the 1-Wire Public Domain Kit to talk to our DS1923 Hygrochron, we also used some of the same kit's code to read counts from the DS2423 1-Wire Counter.

Heart-Rate Monitoring

This portion of the application was the most difficult to create for me, a software guy. I needed to overcome my ignorance about op-amps. My Polar Heart Rate transmitter outputs a short 5kHz magnetic induction signal every time it sees a voltage across its electrodes (i.e., a heart beat). To pick this signal up, I used an inductor-capacitor pair tuned to resonate at 5kHz. However, the resulting signal's magnitude was only about 10 mV. I wanted to tie this signal to an external interrupt, so I would not need to worry about sampling the external heart-rate circuit. To get from 10mV to 3.6V, I passed the signal through several filtering and amplifying op-amp stages, ending with a simple comparator. Now when the circuit detects a heartbeat, I get several short pulses from 0 to 3.6V over about 5ms. Since this detection is tied to an external interrupt, in software I just needed to record the time that the heart beat happened. Since this takes less time than the 5ms pulse, I also needed to temporarily disable the heart-rate interrupt so I would not detect any of the follow-on pulses.[6] [7]

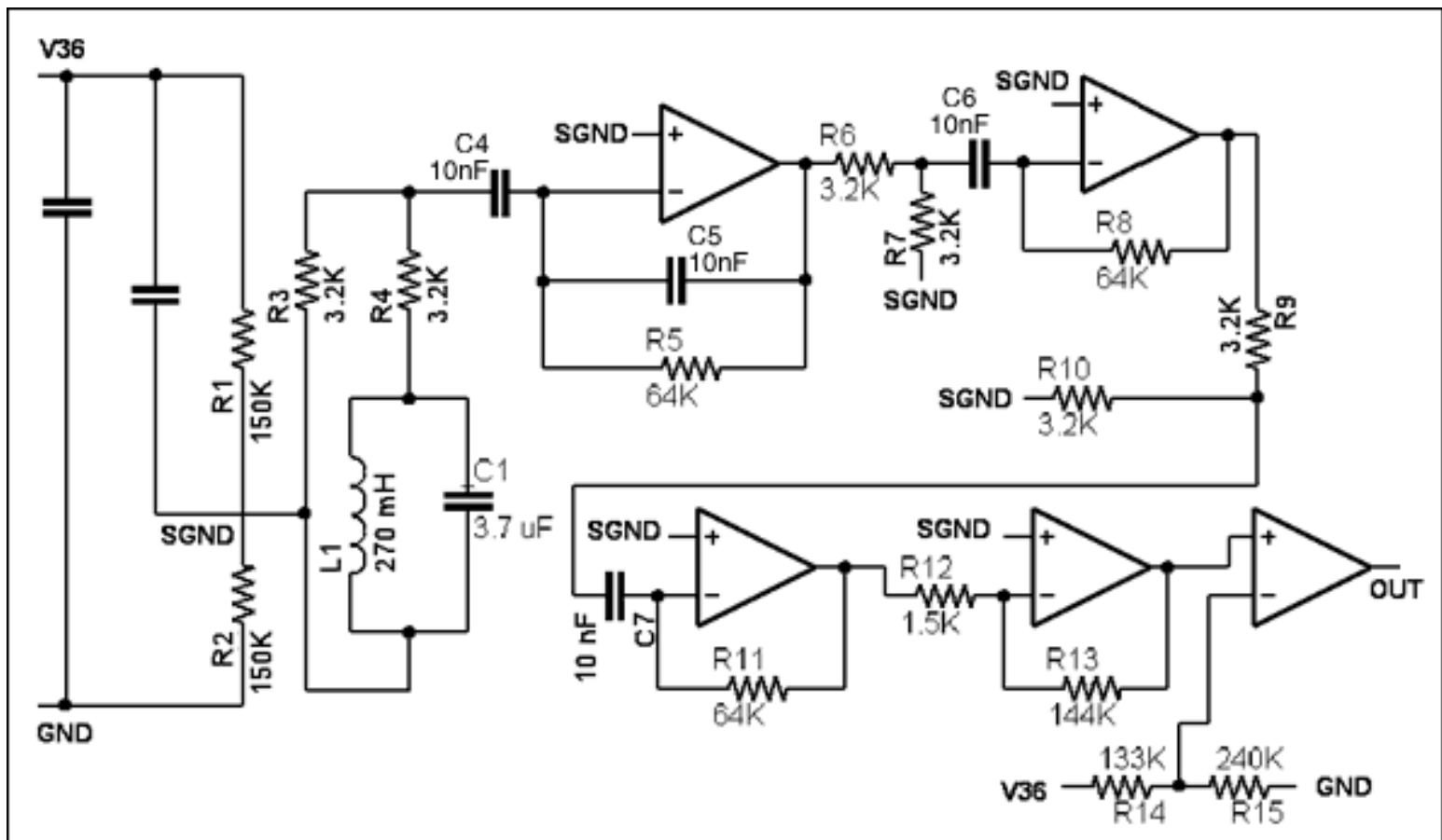


Figure 3. Schematic for the heart-rate monitor sensor. The inductor-capacitor pair is tuned to pick up 5kHz signals, which the op amps amplify and filter. I used MAX4483's in my prototype circuit.

In Need of New Hardware

At this point I had not done anything where the MAXQ2000 Evaluation Kit really limited me. I was now ready, however, to put the system together for my journey to France, so I needed more mobile hardware.

My board design was basically a reduced version of the MAXQ2000 Evaluation Kit; it used the same LCD screen and voltage regulators. Many unused EV kit parts were removed from the design, but a few items were added: screw holes for a battery pack, two extra pushbuttons, and some extra connection headers.

There were a few things more which I needed to remember while designing the board. One was the size of the board. I needed to keep it small so it would easily mount on my bicycle's handlebar. Another was the location and positioning of the 1-Wire interfaces. The ideal location for the iButton holder (for the Hygrochron) was the back of

the board since it rarely needed to be replaced. However, I also needed an additional pair of pins to connect to the DS2423 circuit, which was several inches below the main bike board. These pins were on the back of the board so the wires extended directly down from the bike monitor board to the DS2423.



Figure 4. The bike monitor board. The backside has a 1-Wire can holder for the DS1923, a battery holder for three AA batteries, and connection pins to run to the DS2423 circuit.

The Ride to France

With all the hardware and software in place, I was ready to test it out. I mounted the bike board to my handlebars with some plastic ties. This was, admittedly, not an ideal long-term solution, but I knew my debugging was not over yet. I could cut the plastic ties easily enough to return the bike board to the office for reprogramming.

Before even getting on the bike, I was fairly sure that a few functions would not require any rework; the clock, stopwatch, temperature, and humidity displays were low-risk items. Nothing about getting on the bike was going to change their behavior. I was not worried about the heart-rate monitor either. If it worked fine in the office, how different could it be on the bike? My major concern was the speed measurement. In the office, I passed a magnet by the sensor and saw the measured speed increase and decrease, but I could not know if my calculations to compute miles-per-hour were wrong.

Therefore, I got on the bike and activated the speed measurement. Before I even left the driveway, I was going 7 miles per hour (mph) according to the bike board, which seemed somewhat high, but not outrageous. As I was biking around the neighborhood, my top recorded speed was around 50mph. Now, I might be in decent shape, but I suspected that my mountain bike would not even hold together at that speed. I was generally confident that I miscalculated by a factor of 2 somewhere, so I headed back to the office to peek at my equations in the code. Everything looked fine there (the diameter of a wheel is $2\pi R$, correct?), so I changed the code to display the count reported from the DS2423. As I was almost expecting, the count incremented by two every time the wheel went around. I had placed the magnet so that it passed too close to the sensor, making the whole DS2423 circuit bounce around a lot each time the magnet passed. Backing the magnet off a few millimeters fixed the problem, and slowed me down to 20mph to -25mph, right where I should be.

As expected, the clock, stopwatch, temperature, and humidity functions all worked fine. However, I ran into a problem when I tried the heart-rate monitor. It reported a heart rate of 230 beats per minute, *before I even got on the bike*. I may not be in as good shape as that famous Texas biker, but that did not seem quite right. I took it back in the office to debug, hooked it up to the external debugger, and it ran fine again (about 60 beats per minute). It seemed to me that the difference in power supplies might be causing some noise in the sensor. The debug board provided a nice, clean 5V to the bike monitor, compared to a more unstable 4.8V from the three AA batteries. I first thought to verify that this was not a quantity issue; maybe the regulator's behavior was marginal at 4.8V. I hooked up a 9V battery to supply power, and the heart-rate sensor still acted wildly. The Maxim power regulators were no longer suspect.

My next suspicion was that the batteries were probably not giving a quiet 4.8V supply. Originally, I had 1 μ F capacitors on the heart-rate circuit to settle the 3.6V input line. After scrounging for some fatter capacitors, I put an extra 22 μ F between V36 and ground, and an extra 1 μ F between V36 and the signal ground. That seemed to calm the circuit. The reception range on the heart-rate monitor was somewhat marginal, but I got it working by leaning down a little closer to the board.

Next Stop: France...or the Next MAXQ Project

With the bike monitor application working, it was time to start training for the next Tour. While chances are remote that you will hear about me placing in the race, chances are good that you will hear more about the MAXQ microcontroller. Its peripherals and performance made this application easy to architect, develop, and debug. With plenty more MAXQ microcontroller versions on the horizon, you will see them find their way into many applications.

Pick up an MAXQ2000 Evaluation Kit and see what a MAXQ microcontroller can do for you.

References

- [1] Maxim Integrated Products, [MAXQ2000 data sheet](#).
- [2] Maxim Integrated Products, [DS1923 data sheet](#).
- [3] Maxim Integrated Products, [DS2423 data sheet](#).
- [4] Data Harvest Group, Inc. [The Polar Heart Rate Exercise Sensor](#).
- [5] [1-Wire Public Domain Kit](#)
- [6] Nilsson, James W., and Susan A. Riedel, Electric Circuits, *Fifth Edition*. Addison-Wesley, Reading, MA. (1996).
- [7] Bowden, Bill, [Operational Amplifier Basics](#).

Resources

- [MAXQ2000 Home Page](#)
- [MAXQ2000 Evaluation Kit](#)

Appendix 1

The DS1923 Hygrochron iButton is a rugged environmental sensor that can be used to measure and log temperature and humidity conditions. Hygrochrons can be given a mission—a set of instructions telling the device how often and what to sample. Up to 8k samples of temperature and humidity can be logged in the iButton's internal memory, or an unlimited number returned in a single-shot fashion to the host. The Hygrochron iButton, like all iButtons, has a unique 64-bit identification number and resides in a stainless-steel can that is highly resistant to dirt, moisture, dropping, and laundry cycles.

Appendix 2

The MAXQ is a unique, new 16-bit RISC microcontroller architecture from Dallas Semiconductor. Its highly orthogonal design allows nearly all instructions to execute in a single cycle. The architecture is designed to be high performance, low power, and electrically quiet. It is also uniquely modular: MAXQ peripherals such as timers, I/O interfaces, and analog components are designed for fast, easy integration in new MAXQ microcontrollers. Hardware debug support makes source code debugging on real hardware possible, without the need for expensive emulators. Best of all—you do not even need to learn a new instruction set to use it (although where is the fun in that?). The MAXQ is supported by tools with which you are probably already familiar, such as IAR's Embedded Workbench,

Phyton's Project MQ, and Rowley's CrossWorks.

A similar article was published in the November 2005 issue of *Circuit Cellar* (issue #184).

The DS2423 is no longer recommended for new designs.

Application Note 3845: www.maxim-ic.com/an3845

More Information

For technical questions and support: www.maxim-ic.com/support

For samples: www.maxim-ic.com/samples

Other questions and comments: www.maxim-ic.com/contact

Keep Me Informed

Preview new application notes in your areas of interest as soon as they are published. Subscribe to [EE-Mail - Application Notes](#) for weekly updates.

Related Parts

DS1923: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

DS2423: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

MAXQ2000: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

AN3845, AN 3845, APP3845, Appnote3845, Appnote 3845

Copyright © by Maxim Integrated Products

Additional legal notices: www.maxim-ic.com/legal