My friend used to say that he loved the sound of a babbling brook—unless it was in his basement. Buildings with basements are usually constructed with a system of weeping tiles or drainage pipes for routing water from around the foundation to a place where it can drain. In many municipalities, however, building owners aren’t allowed to drain water into the local sewer system. In such cases, a sump pit is used to collect the water, and a sump pump is implemented to pump the water away from a building. But if something were to go wrong, which Murphy’s law ensures will happen, a building’s owner would indeed hear the sound of a babbling brook where it shouldn’t be heard.

I know from experience that sump pumps don’t have long life spans, and they definitely don’t give much warning when they’re about to fail. There are other points of failure as well. For instance, most pumps are installed with a check valve to prevent the water from draining back into the sump pit. These valves can fail and block the pump outlet. In cold climates, ice can block the drainpipe the system uses as an outlet. Then there are electrical failures. A bad float switch, a tripped circuit breaker, an open thermal cutout, and a power failure are a few common examples.

Thinking about such problems motivated me to develop my Pump-Eye wireless pump monitoring system, which keeps an eye on my pump so I don’t have to. Most people would be satisfied with a simple system that would sound a buzzer if the water level in their sump pit were to rise too high. But, as you’ll soon see, my idea of what a monitoring system should be is a little different.

SYSTEM BASICS

The Pump-Eye is a flexible system comprised of three electronic units: a sensor unit, a base unit, and an Ethernet unit (see Figure 1). Let’s take a look at each unit.

The sensor unit monitors the sump pit’s water level (see Photo 1). Data is displayed on a 10-segment LED bar graph so you don’t have to remove the sump pit’s lid to determine the water level. An alarm sounds when the water level exceeds the height you program into the system. A switch enables you to cancel the alarm at any time. LEDs illuminate when the AC power is off and when the sensor unit’s 9-V back-up battery needs to be replaced.

The base unit features the same indicators as the sensor unit (see Photo 2). It sounds the same alarm signal as the sensor unit. I chose the SOS Morse code sound (an old sound that’s recognizable to some of us) because it’s notably different than the sounds generated by my appliances. Canceling the alarm on the base unit cancels the alarm on the sensor unit and vice versa. Because the units are connected wirelessly, I can place the base unit anywhere in my house. Therefore, I don’t have to go to my basement to read the sensor unit’s front panel.

The Ethernet unit can connect to either the sensor unit or the base unit via an RS-232 connection. I can place the Ethernet unit in the most convenient location for connecting to an uninterruptible power supply (UPS) and network. The Ethernet unit receives commands from the unit to which it’s attached. It then sends syslog messages to a syslog server so that pump cycles can be time stamped and counted. A record is kept of the pump’s run times. The Ethernet unit can also send me an e-mail or text mes-

---

**Figure 1**—The base and Ethernet units are optional. There are five ways to set up the Pump-Eye system without having to make any software changes.
sage regarding conditions that require immediate attention (e.g., high water levels and a loss of AC power).

HARDWARE ESSENTIALS

The sensor and base units feature Freescale MC9S08GT60 microcontrollers. They communicate with each other via 2.4-GHz ZigBee transceivers based on a Freescale MC13192 SARD board using IEEE 802.15.4 MAC software. The sensor unit monitors the sump pump’s AC power and its 9-V back-up battery.

The front panel electronics on the two units are similar, but there are a few differences. The sensor unit is larger. It also has an extra connector that’s used for passing signals to the rear panel’s electronics for the sensors. Because the base unit simply acts as a remote display to show what’s happening on the sensor unit, it doesn’t need sensing electronics on the rear panel.

When you cover the top of a straw with your finger and place it in a glass of water, the air in the straw becomes pressurized. The amount of pressure depends on the height of the water in the straw, and this depends on factors such as ambient air pressure, the mass density of the water, gravity, and the height of the water outside the straw:  

\[ P = Pa + \rho g \Delta h. \]

In this formula, \( P \) is the pressure, \( Pa \) is the ambient pressure, \( \rho \) is the mass density of fluid, \( g \) is 9.8066 m/s\(^2\), and \( \Delta h \) is the height of fluid.

You can nullify the effect of a change in ambient pressure if you use a gauge pressure sensor to measure pressure relative to ambient pressure. The formula then becomes \( P = \rho g \Delta h \). You can assume that the mass density of water and gravity are constants, so the pressure will be proportional to a change in the water’s height. The sensor unit measures this pressure with a Freescale MPXM2010GS temperature-compensated gauge pressure sensor. The pressure is then converted to a percentage of normal water levels observed in the sump pit.

I tried placing a hose in the sump pit to sense the water level, but I quickly discovered that it wasn’t too reliable. This was probably because of the surface tension of the water clinging to the inside of the thin hose (5/64” in diameter). Therefore, I decided to use a 0.5” in diameter copper pipe as a sensor interface. The ratio of the area affected by surface tension to the total area is less significant with the larger diameter. I bought a length 0.125” in diameter brass pipe to use as a nipple for the hose that connects the MPXM2010GS to the copper pipe. I soldered the brass pipe to a standard 0.5” copper cap in which I had drilled a hole. The cap is soldered to the top of the copper pipe.

The copper pipe solved the problem of holding the open end of the hose at a fixed height, and it also alleviated my concerns about dirt clogging the thin hose. A plastic clamp screwed to the side of the plastic sump pit holds the pipe in place. I had originally placed the pipe to the bottom of the sump pit, but I found a negative pressure developing in it after numerous pump cycles. I concluded that this was the result of scavenging around the bottom of the pipe because of water currents caused by the pump. Keeping the end of the pipe at the height of the low water level prevented this from happening.

CONSTRUCTION DETAILS

The sensor unit prototype features a modified MC13192 SARD board along

![Photo 1](https://www.circuitcellar.com/)

**Photo 1**—A hose connects the Pump-Eye sensor unit to a copper pipe. The pipe gets fastened to the side of the sump pit.

![Photo 2](https://www.circuitcellar.com/)

**Photo 2**—Normally I keep the base unit upstairs and the sensor unit near the sump pit in the basement. I can use the Silence Alarm button on either unit to cancel the SOS alarm. If I’m in the basement when the alarm sounds, I can cancel the alarm upstairs by holding the sensor unit’s button for 1 s.
with two wire-wrapped boards and an MPXM2010GS sensor on a breakaway sensor board, which was included in my Freescale Wireless Design Challenge 2004 kit. The sensor board needed modification because the trace for pin 1 on the board was missing. A 26-conductor ribbon cable connects the SARD board to the front panel board. A 10-conductor ribbon cable connects the front panel board to the rear panel board where the signal conditioning circuitry resides.

The base unit prototype is similar to the sensor unit. It also features a modified SARD board and a front panel. However, the only connection from the front panel to the rear panel is a pair of wires for the piezo buzzer.

I used Hammond Manufacturing plastic cases with aluminum end panels for the prototypes. For the sensor unit, I used a 1598CSGY case. The base unit has a 1598BSGY case, which is slightly smaller. I chose these cases because they have internal slots for holding the circuit boards. The spacing is just right for the 10-segment LED package, a 20-pin solder tail DIP socket, and a 20-pin wire-wrap socket. This places the display against red 3M tape applied to the inside of the front panel.

I cut the slot for the 10-segment LED package on a milling machine with a 0.125” bit. I made all of the other holes with a hand punch after I applied the labels, which I made with the Micrografx Designer program and printed on Avery permanent white I.D. labels (part no. 06573). The top of the label was covered with a glossy transparent film.

Each front panel features a Lumex SSA-LXB10SRW 10-segment LED in a DIP package. I chose this LED because it can produce adequate brightness with less than 0.5 mA of current. I needed this type of efficiency because the regulator on the SARD board can supply only 100 mA of current. The circuitry on the SARD board was already drawing about 45 mA, so I couldn’t afford to use a lot of current for the LEDs.

I used low-current HLMP-4700 LEDs from Fairchild Semiconductor for the Check AC power and Check Battery indicators because they’re bright. I used 2.21-kΩ resistors for the 10-segment LEDs and 680-Ω resistors for the other two LEDs. At first glance, these may seem high for driving LEDs with a 3-V supply, but they produce good results. The last item on each panel is a momentary normally open push button switch that’s used to silence the alarm. It’s connected to a GPIO pin on the processor.

The rear panels have a piezoelectric ceramic buzzer (Digi-Key part no. 9948) that can be driven directly with DC. The buzzer sounds when the water level breaches a predetermined level. There is also a 2.1-mm barrel jack for the power supply, an on/off toggle switch for power, and a DE9S RS-232 connector. The DE9S is wired 1:1 on pins 2, 3, 5, 7, and 8 to a DE9P that plugs into the SARD board. The power switch is in series with the supply line that terminates in a 2.1-mm barrel plug supplying power to the SARD board. The sensor unit also has point-to-point wiring for the diodes that isolate the 9-V battery from the 9-V regulated power supply.

### SARD MODIFICATIONS

The MC13192 SARD board has a 26-pin header labeled J3. This connector brings some of the analog inputs and GPIO pins to the odd pins on J3. The even pins connect to acceleration sensors that aren’t used in this project.

I needed to supply 5 V to the pressure sensor and 3 V to the quad op-amp I used for signal conditioning in the sensor unit, so I added the supply rails to J3. I also added four more GPIO pins and a couple of ground pins because there weren’t any assigned to J3.

The changes involved simply adding wires to spare pins on the even side of J3. The modified SARD board’s schematic is posted on the Circuit Cellar FTP site.

---

**Listing 1—The base unit sends these data packets to the sensor unit.**

```c
uint8_t base_app_data[10];

// Prepare the packet to be sent
switch (msg_num)
{
    case ACK_MSG :
        base_app_data[0] = 'S'; // Set token
        base_app_data[1] = sensor_app_data[1]; // Lower case command char
        base_app_data[2] = sensor_app_data[2]; // Value token
        base_app_data[3] = NVbuf[0]; // MSB
        base_app_data[4] = NVbuf[1]; // LSB
        base_app_data[5] = 0x0;
        base_app_data[6] = 0x0;
        base_app_data[7] = 0x0;
        base_app_data[8] = 0x0;
        base_app_data[9] = 0x0;
        break;

    case CLEAR_STATS_MSG :
        base_app_data[0] = 'C';
        base_app_data[1] = 'L';
        base_app_data[2] = 'E';
        base_app_data[3] = 'A';
        base_app_data[4] = 'A';
        base_app_data[5] = 0x0;
        base_app_data[6] = 0x0;
        base_app_data[7] = 0x0;
        base_app_data[8] = 0x0;
        base_app_data[9] = 0x0;
        break;

    case SET_MSG :
        base_app_data[0] = 'S'; // Set token
        base_app_data[1] = msg_char; // Lower case command char
        base_app_data[2] = 'V'; // Value token
        base_app_data[3] = NVbuf[0]; // MSB
        base_app_data[4] = NVbuf[1]; // LSB
        base_app_data[5] = 0x0;
        base_app_data[6] = 0x0;
        base_app_data[7] = 0x0;
        base_app_data[8] = 0x0;
        base_app_data[9] = 0x0;
        break;
}
```
**SIGNAL CONDITIONING**

AC power is acceptable when the 9-V regulated supply is present, so it must be plugged into the same outlet as the sump pump. If the circuit breaker trips or if there’s a power failure, the 9-V supply will drop to zero. Resistors R23 and R24 form a voltage divider for the 9-V supply. When 9 V is present from the supply, the voltage at the junction of R23 and R24 is approximately 2.5 V. This is read as a logic-high level by port A bit 3 on the processor.

R21 and R22 divide the 9-V back-up battery voltage down before being buffered. These resistors present a total load of 1.33 MΩ to keep the current consumption to less than 7 μA. At this rate, the battery should last almost six years, assuming a capacity of 500 mAh and a final voltage of 7 V. The voltage at the junction of R21 and R22 is buffered by one section of an LM6134A op-amp and fed to analog-to-digital input AD1 on the processor.

The MPXM2010GS pressure sensor output is ratio-metric to the supply voltage, so I powered it with 5 V because I wanted the output to be as sensitive as possible. The output is differential centered on half the supply voltage. The full-scale output signal for the range of the water’s height was only about 2 mV, which is less than the resolution of the least significant bit of the 10-bit ADC. Therefore, the signal must be amplified before being fed into the ADC.

The amplifier design is based on information from Michelle Clifford’s application note entitled “Water Level Monitoring.” The application note contains an incomplete schematic, a few of the resistor values are wrong by an order of magnitude, and the formula for gain has an error. The rest of the application note, however, is quite helpful. I used op-amps that I had on hand and added a potentiometer to adjust the offset.

I set the offset so the amplifier’s output produced about 500 mV without a hose connected to the sensor. Overall, the gain of the amplifier is close to 500. I observed an increase in output voltage of 1.2 V with a change in water height of about 10”. This means the design can handle a delta of around 20” without saturation, which is plenty for this application. The amplifier’s gain is set for the most part with the ratio of R17 and R19 to R18 and R20. U2A and U2B have a gain of only about 0.001. The amplifier’s output is fed to the analog-to-digital input AD0 on the processor.

**SOFTWARE**

One of the challenges with a wireless system is making it robust. You don’t know when a packet will be corrupted or lost altogether, so effort must be made to verify packet contents and start over when communication breaks down. With that goal in mind, I designed a simple protocol to verify the data packets exchanged between the sensor and base units.

The data packet sent to the base unit from the sensor unit contains specific characters for identification purposes. It contains specific characters for identification purposes. The packet contains all of the data necessary for the base unit to display the same information that the sensor unit displays on the front panel (as well as to a dumb terminal I had connected to the serial port during development).

Listing 1 (p. 51) shows the data packets the base unit sends. Listing 2 shows how the sensor unit identifies the data packet before taking appropriate action. A short code snippet showing the data packet the sensor unit sends to the base unit is

```c
void process_recv_data(void)
{
    // Only process the packet if the command structure is valid:
    // Is it an ACK to our request?
    if ((base_app_data[0] == 'A') &&
        (base_app_data[3] == 'S') &&
        (base_app_data[9] == 0x0))
    {
        // Grab the packet number
        packet_num_recv = (base_app_data[1] << 8) + base_app_data[2];
        // Grab the status byte
        base_status_byte = base_app_data[4];
        // Grab the Base Unit Radio Scans
        BU_radio_scans = (uint32_t)base_app_data[5] << 24;
        BU_radio_scans += (uint32_t)base_app_data[6] << 16;
        BU_radio_scans += (uint32_t)base_app_data[7] << 8;
        BU_radio_scans += (uint32_t)base_app_data[8];
        return;
    }
    // Is it a Command to clear the Statistics?
    if ((base_app_data[0] == 'C') &&
        (base_app_data[1] == 'L') &&
        (base_app_data[2] == 'E') &&
        (base_app_data[3] == 'A') &&
        (base_app_data[4] == 'R') &&
        (base_app_data[5] == 0x0))
    {
        // Handle the command
        Handle_Command('c');
        return;
    }
    // Is it a Command to Change a parameter?
    if ((base_app_data[0] == 'S') &&
        (base_app_data[2] == 'V') &&
        (base_app_data[5] == 0x0))
    {
        // Grab the value bytes
        NVbuf[0] = base_app_data[3];
        NVbuf[1] = base_app_data[4];
        // Handle the command
        Uart_Print("\nThe Base Unit sent a new Threshold / Level value...");
        Handle_Command(base_app_data[1]);
        return;
    }
}
```

---

Listing 2—the sensor unit verifies the data packets from the base unit before taking appropriate action.
posted on the Circuit Cellar FTP site [sensor_to_base_unit.txt]. Malformed packets are simply ignored at the application level. Timers keep track of how long it has been since the arrival of the last good packet.

I developed the software for the sensor and base units with Metrowerks CodeWarrior for the HC08. I started by studying the programs in the wireless application demonstration, which I downloaded from Freescale’s web site. The programs use the Freescale 802.15.4 MAC and PHY libraries for the MC13192. Unfortunately, the Freescale Wireless Design Challenge kit didn’t come with a Background Debug mode (BDM) interface, so I couldn’t immediately compile and load the programs. I figured out how to add the correct bootloader interface to the project and link it properly so it could be downloaded using the embedded bootloader already programmed in the SARD boards’ flash memory.

I used code from MyApp_Ex06A.c as the basis for the radio interface in the base unit. I used code from MyApp_Ex06B.c as the basis for the radio interface in the sensor unit. These files were renamed Base_Radio.c and Sensor_Radio.c, respectively. I added C++ style comments to identify my changes. You may download the files from the Circuit Cellar FTP site.

Both units implement the radio functions as a state machine in an endless loop in main in these files. At the end of the loop, the MAC Mlme_Main function is called, and then the sensor unit’s SensorProcess function or the base unit’s BaseProcess function is called. The software is set up to capitalize on the fact that the front panels are common between the two units. Both units use the Common, Power, and Timer C files and their respective header files.

Figure 2 is a high-level software flow diagram for the sensor unit. When the sensor unit powers up, it checks to see if the Mode/Silence Alarm switch is being held. This action indicates that you want to enter Calibration mode. After the sensor unit initializes, it runs an LED show for 30 s or until it finds and associates with the base unit (whichever happens first). If the sensor unit can’t find and associate with a base unit, it tries again 10 min. later. When it associates with a base unit and runs uninterrupted for 45 s without a response to a poll for status, it will try to find the base unit again and associate with it. The LEDs show reruns during these retries.

Figure 2—In the software for the sensor unit, calibration is performed if the mode switch is held at power-up. Timers are used to recover from communication problems with the base unit.
The sensor unit continually reads the inputs for water level, back-up battery voltage, AC power status, and switch state. The ADC readings for the water level and battery voltage are both filtered using a fast integer math filter. A piecewise linear look-up technique calculates the percentage of water level based on the calibration values recorded for the low and high water marks, which are dynamically populated into the look-up table. It works on the rise over run principle. Each segment of the 10-segment bar graph represents 10% of the water height in the sump pit.

If the sensor unit has associated with a base unit, packets are sent and received each second. After a data packet is transmitted to the base unit, the sensor unit waits 500 ms before a request is sent for the base unit's status packet to check if the Alarm Silence button on the base unit has been pressed. The cycle then repeats after another 500 ms.

A high-level software flow diagram for the base unit is shown in Figure 3. The base unit's software is simpler than the sensor unit's software. At power-up, it runs the LED show continually until a sensor unit associates and starts sending data packets. When a data packet arrives, it's checked for valid contents and then the data is used to update the outputs on the base unit so that they mirror the sensor unit. A status packet is immediately prepared so that it can be sent when the sensor unit requests it.

If a sensor unit is associated with the base unit and isn't heard from for 30 s, the base unit restarts the PAN coordinator in an attempt to find a quieter channel. As I already mentioned, the sensor unit also has retry timeout periods that will cause it to periodically search for the base unit.

**SERIAL PORTS**

In the process of developing the system and experimenting with it, I found it helpful to output messages to a terminal to verify program operation. This grew into a simple command parser that allowed me to display status, statistics, and thresholds. It also allowed me to set the low water (0%) ADC value, the high water (100%) ADC value, the alarm threshold percentage, and the battery OK threshold for testing purposes.

An interesting feature that's built into the communication scheme is the ability to set these thresholds on the sensor unit from a terminal connected to either the sensor unit or the base unit. The base unit sends commands via the wireless link to the sensor unit to be interpreted by the command parser. It's the same as a direct connection to the terminal. This proved to be useful when I was fine-tuning the system during the developmental stages of the project.

**ETHERNET UNIT**

The Ethernet unit is a Freescale DEMO9S12NE64 board that's controlled by the serial port from either the sensor unit or base unit. How do the terminal commands and Ethernet unit commands coexist on the same 19,200 data rate, 8 data bits, no parity, and 1 stop bit serial port without confusing each other? The secret lies in the fact that the Ethernet unit is controlled solely with nonprintable control characters! The Ethernet unit conveniently ignores all characters output from the base and sensor units that are intended for the terminal. This enabled me to test the entire Ethernet unit interface simply by using control characters from a dumb terminal.

Listing 3 shows the control characters issued to the Ethernet unit and the name associated with the event that triggers them. All of the control characters received by the Ethernet unit (except those associated with the pump's starting and stopping) cause a TCP/IP UDP packet to be sent to a server over the Ethernet LAN in my house. The packet is formatted according to the BSD Syslog Protocol (RFC 3164) except for the time and date fields because the Ethernet unit doesn't have a real-time clock.

**Listing 3**—The control characters, sent via the RS-232 port, are interpreted by the Ethernet unit and trigger it to send syslog messages and e-mails depending on the severity of the situation.

<table>
<thead>
<tr>
<th>Control Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>SOH or ^A</td>
</tr>
<tr>
<td>0x02</td>
<td>STX or ^B</td>
</tr>
<tr>
<td>0x03</td>
<td>ETX or ^C</td>
</tr>
<tr>
<td>0x04</td>
<td>EOT or ^D</td>
</tr>
<tr>
<td>0x05</td>
<td>ENQ or ^E</td>
</tr>
<tr>
<td>0x06</td>
<td>ACK or ^F</td>
</tr>
<tr>
<td>0x07</td>
<td>DLE or ^P</td>
</tr>
<tr>
<td>0x08</td>
<td>DC1 or ^Q</td>
</tr>
<tr>
<td>0x09</td>
<td>DC2 or ^R</td>
</tr>
<tr>
<td>0x0A</td>
<td>DC3 or ^S</td>
</tr>
<tr>
<td>0x0B</td>
<td>NAK or ^U</td>
</tr>
<tr>
<td>0x0C</td>
<td>SYN or ^V</td>
</tr>
<tr>
<td>0x0D</td>
<td>ETB or ^W</td>
</tr>
<tr>
<td>0x0E</td>
<td>CAN or ^X</td>
</tr>
</tbody>
</table>
The server receiving the UDP packets runs on a PC powered by a UPS. I tried a couple of syslog servers that were free on the 'Net. Weird Solutions’s Syslog Turbo and Kiwi Enterprises’s Kiwi Syslog Daemon both worked well. They both time stamped the message sent by the Ethernet unit, but I found Syslog Daemon to be easier to use. The syslog server enabled me to log the date and time of every pump cycle.

The Ethernet unit can determine the pump’s run time by measuring the time between the pump_time_start and pump_time_stop control characters. These characters are sent at the 90% and 10% water level points as the pump is running. Therefore, 20% more is added to the time, which is acceptable because the pump rate should be constant. I verified this with a stopwatch. The Ethernet unit also sends e-mail messages when it receives any of the control characters associated with a name prefixed by alarm in Listing 3.

INSTALL AND CALIBRATE

Allow the sump pump to run and shut off by itself at the normal low water level. Next, install the copper pipe so the bottom of the pipe is just above the water level. Connect the flexible hose between the top of the copper pipe and the sensor tube fitting on the rear of the sensor unit.

Plug the sensor unit power supply into the same AC outlet as the sump pump. Power the sensor unit while holding the Mode button. The bottom bar graph LED should start blinking. Release the button. The sensor unit will capture the lowest reading while the bottom LED is blinking. After 5 or 10 s, press and release the Mode button. The top bar graph LED should start blinking. The sensor unit will capture the highest reading while the top LED is blinking.

Slowly fill the sump pit with water until the sump pump starts pumping. When the water level has dropped, press the Mode button one more time. This will cause the sensor unit to store the lowest and highest readings and enter the normal Run mode that it operates in when you power up the unit without holding the Mode button.

You can install the base unit where wherever it’s convenient. If possible, power up the base unit before the sensor unit. If you decide to use the Ethernet unit, power it from a UPS and connect it to a LAN that is powered from a UPS as well. Connect the Ethernet unit to the RS-232 connector on the sensor or base unit using a male-to-male null serial cable.

KEEP AN EYE OUT

The Pump-Eye system is a welcome addition to my home. It works well and is more flexible than I originally intended. I’m especially pleased with how the enclosures turned out. The base unit has become a real conversation piece in my dining room.

Having the ability to measure my pump’s run time enables me to detect impending pump failures and outlet blockages. It also enables me to calculate the pump’s performance. For instance, I know my pump takes 15 s to pump approximately 36 liters of water into a test tank. That’s about 38 gallons per minute. Pumping the same quantity of
water outside takes 19 s because it has to pump the water through a pipe that’s much higher and longer. This kind of information is helpful when I’m evaluating the other pumps I have been keeping as back-up units.

The process of time stamping events is helpful for postmortem analysis and when identifying trends. I’ve noticed that the pump has run as often as every couple of minutes during rain storms. Under such conditions, it takes much longer to empty the sump pit. One stormy day last summer, the pump ran 792 times during a 24-h period. That day it ran as long as 40 s per pump cycle and was close to a 50% duty cycle at times. This information is documented in the syslog.txt file posted on the Circuit Cellar FTP site.

It’s commonly suggested that building owners should water around their foundations during long dry spells. This is supposed to prevent foundation problems. The Pump-Eye will give me an idea of just how dry it really is around my foundation.

It’s surprising how many people have tales to tell about sump pump failure and how they wish they had known earlier that their pump wasn’t working. The Pump-Eye might be just what they need!

David Kanceruk (david.kanceruk@gmail.com) is a senior software developer at Vansco Electronics LP in Winnipeg, Canada. A graduate of the industrial electronics program at Red River College, he has been developing hardware and software for embedded systems for more than 25 years. He has also served as an external examiner on thesis day at the University of Manitoba. He is interested in home automation, automobiles, music, and the truth.


**PROJECT FILES**
To download the code and additional files, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2006/189.

**RESOURCES**

**SOURCES**

| HLMP-4700 LED | www.fairchildsemi.com |
| DEMO9S12NE64, MC13192 SARD board, and C9S08GT60 microcontroller | www.freescale.com |
| 1598BSGY and 1598CSGY Enclosures | www.hammondmfg.com |
| Kiwi Syslog Daemon | www.kiwisyslog.com |
| SSA-LXB10SRW LED | www.lumex.com |
| Syslog Turbo | www.weird-solutions.com |