CIRCUIT CELLAR INK®
THE COMPUTER APPLICATIONS JOURNAL

Jim Brady

Networking with DeviceNet

Part 2: A Weather-Station Application

Think programming a DeviceNet interface in C++ is tough? Jim disagrees. With its excellent response times and adequate program size, Jim gets the same excitement writing code for a fast 32-bit CPU as he got from his '67 Camaro.

If you like programming as much as I do, you're in for a real treat with DeviceNet. The DeviceNet specification is fully object-oriented, with each object described in terms of attributes and services.

These items correspond to C++ class data and member functions, so if you use C++, all you have to do is understand the specification and translate it to code. Just make sure you have some strong coffee on hand when tackling the tricky parts. I had the most trouble—err…fun—with connection states and fragmented messaging.

Let's cover the PC/104 hardware first. After surveying the many processor boards available for PC/104, I went with the Micro/sys SBC1386, a 25-MHz '386EX board, shown in Photo 1. It comes with BIOS and a DOS run-time environment that runs the application out of RAM. That way, you don't need a special library and linker to generate ROMable code.

The board also includes the Borland remote debugger in flash memory. It's nice to be able to send the program to the board at 115 kbps, set some breakpoints, and let 'er rip. My program is written entirely in C++ using the Borland compiler (large memory model).

A lot's been said about the poor suitability of C++ for real-time embedded development. But, it's more than adequate for a fast-response DeviceNet interface.

The program weighs in at 45 KB of code space, including the weather-station application code. This size is comparable to DeviceNet interfaces I've done using standard C with small CPUs. I'll show you some performance measurements later on.

CAN CHIPS

The next order of business is picking a CAN controller. Table 1 compares peripheral-type CAN controllers. I went with the Intel 82527 because I like having individual mailboxes for each message type rather than one big FIFO for all of them. It's more modular.

The Siemens parts also work this way. They have 15 or 16 mailboxes—plenty for the DeviceNet predefined connection set, which has 10 connections.

A FIFO is good if you're concerned with the master beating your door down with high message rates. But at some point, your code will run out of steam anyway.

The 82527 has five operating modes. Only mode 3 (nonmultiplexed asynchronous) makes sense for a PC/104 interface. I'd prefer faster 16-bit transfers, but the 82527 in mode 3 is limited to 8 bits.

The PC/104 bus has the same timing as the ISA bus, and it takes a whopping 720 ns for an 8-bit read or write. This glacial pace is actually good because it doesn't exceed the rather long cycle and

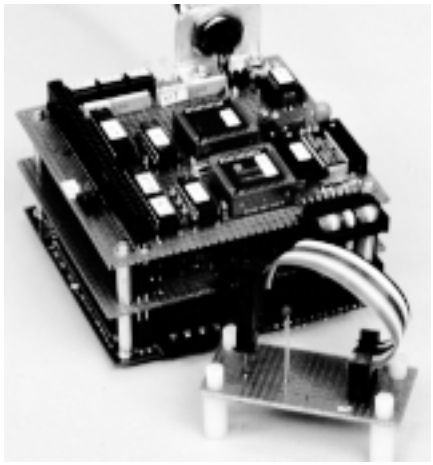

Photo 1—The PC/104 weather station is entirely powered from the DeviceNet bus. The weather station board is sandwiched between the '386EX CPU board on bottom and the DeviceNet interface on top. The humidity transducer and thermistor are on the small board in front.

access times of the 82527.

In mode 3, at maximum clock rate, the 82527 has a 288-ns access time. If you use a fast bus, you need to accommodate this slow interface. Intel's web site has app notes for interfacing the 82527 to a various processors.

I use a PAL to generate the R/W select line and the chip selects. To make sure the R/W line remains stable at the end of a bus cycle, the line is latched by an RS latch in the PAL.

MEMR sets the latch and MEMW resets it. The PAL design source file is available via the Circuit Cellar web site. The only glue logic is a couple of inverters to delay the 82527 chip select to make sure it doesn't go low until after R/W is valid.

With no video board in my PC/104 stack, there's plenty of memory space for the CAN controller's 256 bytes. I went with A0000.

In mode 3, the 82527 provides one I/O port. To get enough I/O for all my switches and LEDs, I added an 82C55A at memory address A1000. That gave me plenty of I/O lines, including enough for a four-wire serial interface to the ADC on my weather-station board.

CHIP SETUP

The 82527 has 15 mailboxes for CAN messages, each with 15 registers. Setting up a mailbox requires telling it what its message identifier is and if it is send or receive. Done properly, your program only gets an interrupt for a message directed to your device.

The 82527 also has a group of registers that control message filtering, interrupt masking, data rate, and sample timing. There are some tricky ones that set the sample point within a bit time as well as the limit on how much that sample point can jump around.

There is a tradeoff—you want to let it jump as much as possible to accommodate oscillator tolerance, and you also want the sample point to be close to the end of the bit time to accommodate long cables. But you can't allow it to jump so much that it goes past the end of a bit time.

After a lot of calculation, I ended up sampling at 87% of the way through a bit time, with the jump limit (SJW) equal to 12% of a bit time. That accommodated the worst-case cable length, with a jump width still large enough to handle crystal errors of about ±0.2%, which is plenty for any crystal. The *Intel 82527 Architectural Overview* provides information for this calculation.

REAL TIME

I can't help but be excited about writing code for a fast 32-bit CPU after designing 8-bit systems for years. The feeling of power is like the feeling I got from my first car, a '67 Camaro with a 327 engine.

To make sure the 18.2-Hz BIOS clock interrupt wouldn't hurt me, I measured its duration by looking at how big a chunk it took out of a tight loop that pulsed an I/O pin. According to my scope, it is just 56 µs, including the time it takes to run my own timer interrupt at INT 1C, which is chained to the BIOS clock interrupt. I use this timer to update my DeviceNet connection timers.

When a DeviceNet message arrives, the 82527 pulls IRQ5 high. According to Intel, the '386EX has a worst-case interrupt latency of 63 clock cycles, or ~2.5 µs at 25 MHz, neglecting wait states.

So, my DeviceNet interrupt handler has to wait for a maximum of 58.5 µs (i.e., 56 + 2.5) before it runs. This situation happens when a DeviceNet message comes in just after a BIOS clock tick.

The DeviceNet interrupt handler in Listing 1 reads the message-length byte to find out how long the message is and then reads only the data bytes it needs to. Most DeviceNet messages are well under 8 bytes. The most frequent message, the I/O Poll Request, has no data bytes at all!

By the way, check the disassembled machine instructions with your debugger to make sure functions like `peekb()` are getting expanded inline. Depending on compiler settings, they may not be. For an 8-byte message, the duration of my DeviceNet interrupt handler is 100 µs with `peekb()` inline or 160 µs otherwise.

These timing measurements show there's still plenty of time left for process-

	Intel 82527	Philips SJA1000	Siemens SAE 81C90	Siemens SAE 81C91
Package	PLCC 44 QFP 44	DIP 28 SO 28	PLCC 44	PLCC 28
Parallel CPU Interface	8-bit multiplexed 8-bit nonmultiplexed 16-bit multiplexed	8-bit multiplexed	8-bit multiplexed	8-bit multiplexed
Access time	288 ns	45 ns	120 ns	120 ns
Serial Interface	SPI, 8 MHz	None	4 wire, 5 MHz	4 wire, 5 MHz
I/O Ports	1 or 2 eight-bit ports	None	2 eight-bit ports	None
Organization	15 mailboxes; 1 is double-buffered	64-byte FIFO	16 mailboxes	16 mailboxes
Identifier mask registers	1 global for mailboxes 1–14, and 1 special for mailbox 15	1 global	None	None
Identifier match registers	1 per mailbox	1 global	1 per mailbox	1 per mailbox
Message timestamp	No	No	Yes	Yes
Max. DC current	50 mA	15 mA	30 mA	30 mA
Approx. price	$7.50	$7.30	$6	$5.30

Table 1—Now you can compare various peripheral-type CAN controllers. The Philips device stores all messages in a FIFO, while other devices store messages in mailboxes based on their identifier.

ing messages. DeviceNet recommends a response time of 1 ms for I/O Poll messages and 50 ms for Explicit messages. These measurements also show that a faster PC/104 bus wouldn't help much. Out of the 100-µs total time for the interrupt handler, only about 6 µs (eight bytes at 720 ns per bus cycle) is spent doing bus transfers.

If your CPU is slow, an easy way to get the 1-ms response time for I/O Poll messages is to put the data you want to send in the CAN chip's mailbox ahead of time, ready to go. When an I/O Poll request comes in, immediately tell the CAN controller to send it. With this strategy, I measured the weather station's I/O Poll response time at a worst case of 140 µs.

I later changed the code to be consistent with my priority-based event handler, which runs in the main loop. My DeviceNet interrupt handler puts the message into a buffer, sets a bit in a 16-bit event-word to indicate a message is in, and exits.

The bit's position within the event-word determines its priority. When the main loop detects this bit and no higher priority bits exist, it calls the link consumer to consume the message, gets the data from the Assembly Object, and calls the link producer to produce the message. This orderly approach lengthens the I/O Poll response time to 340 µs, which is still plenty good.

Listing 1—The interrupt handler for DeviceNet messages copies the message from the 82527 into a buffer, saves the length, and frees the 82527 for the next message. The run time is 100 µs.

```
#define CAN_BASE 0xA000
UCHAR global_CAN_buf[10];
UINT global_event;

// Handles receipt of incoming DeviceNet messages
// The three dots are required in C++ mode
void interrupt far can_isr(...)
{
    UCHAR i, int_source, addr, mailbox, length;

    int_source = peekb(CAN_BASE, 0x5F);  // read interrupt source
    if ((int_source < 3) || (int_source > 7)) return;

    mailbox = int_source - 2;
    for (i=0; i < 10; i++) global_CAN_buf[i] = 0;

    // compute address of config register in mailbox of interest
    addr = 6 + (mailbox << 4);        // multiply by 16
    length = peekb(CAN_BASE, addr);   // read message length
    length = length >> 4;
    global_CAN_buf[9] = length;       // save message length
    for (i=0; i < length; i++){       // move message from 82527
      addr++;
      global_CAN_buf[i] = peekb(CAN_BASE, addr);
    }
    addr = 1 + (mailbox << 4);        // point to control 1 reg.
    pokeb(CAN_BASE, addr, 0x55);      // clear INT_PENDING bit
    addr--;                           // point to control 0 reg.
    pokeb(CAN_BASE, addr, 0xFD);      // clear NEWDAT
    global_event |= 0x0001 << mailbox; // set bit in global_event
    outp(0x20,0x20);                  // nonspecific EOI
}
```

MESSAGE FLOW

Figure 1 shows message routes in the system. Explicit and I/O Poll messages come in through their respective mailboxes. Explicit messages are routed via the path specified in the message and can access almost any object in the device. I/O Poll messages grab preselected data from a buffer in the Assembly object and quickly send it.

The weather-station sends three bytes—device status, temperature, and humidity. I can send more data by adding it to the existing assembly or creating a second assembly. The device manufacturer determines which data goes into the assemblies.

At the top of Figure 1 is the unconnected port, which the master uses to allocate the connections it wants to use. Technically, connections don't exist prior to allocation.

This situation implies using C++ dynamic allocation. Although you can do this, I chose to create static objects at the beginning of `main()` and use the constructor to set the initial connection state to nonexistent.

CONNECTIONS

Connections have states other than nonexistent and established, and some are unique to one or the other connection. This setup is so confusing, I made a state transition diagram. Figure 2 combines the behavior of both types of connections, using colors to tell them apart.

When the master allocates the Explicit connection, the connection simply transitions to the established state and it's ready to use. The connection timer starts at 10 s.

If it times out, the connection goes to one of two possible states depending on whether the connection is in autodelete or deferred-delete mode. In autodelete mode, if it times out, it's gone. In deferred-delete mode, it stays around and goes back to the established state if a message comes in.

The I/O connection, when allocated, goes to the configuring state. In this state, it cannot process I/O messages and must wait for the master to set its expected packet rate via the Explicit connection. Then it is in the established state and can begin handling I/O Poll requests.

TIMERS

For each connection, you need a time-out timer. You also need a timer for sending fragments. The BIOS clock is handy, but who wants to deal with 18.2 Hz?

With the Micro/sys board, the 25-MHz system clock is divided by 21 to drive timer 0 in the '386EX. Timer 0 further divides by 65,536, producing 18.2 Hz. Loading 0xE884 into the timer 0 count register resulted in BIOS clock interrupts at a more friendly rate of 20.0 Hz.

The connection time-out time depends on the expected packet rate, which is set by the master. When a DeviceNet message comes in, I reload the timer for that connection, and my timer interrupt handler then decrements it at a 20.0-Hz rate. If it reaches zero before another message comes in, the connection times out.

ANALOG INPUT POINT

The Analog Input Point in the DeviceNet

object library models an analog sensor. Listing 2 shows some code for this class.

The specification defines eight attributes, many of which are optional. I implemented the ones for sensor value, sensor status, and data type. The data type tells the master whether the value is an integer, float, or what. For the weather station, I use an unsigned char that corresponds to a data type of 2.

My Analog Input Point class implements the DeviceNet Get Attribute Single service using a member function. Thus, the master can read any of the three attributes using an Explicit message.

These attributes aren't settable, so my class doesn't have the Set Attribute Single service. In the future, I may allow the master to set the data type to a float, switching my sensor value to floating point.

IDENTITY OBJECT

Every device must be able to give its name, rank, and serial number. The Identity object holds this information. It also keeps track of device state and does device resets.

There are two types of resets. Type zero simulates an off/on power cycle. To do this, I send a response back to the master and suspend writes to my watchdog timer.

A type-one reset changes settable configuration parameters back to their factory default values and does a type-zero reset. The weather station has no configurable settings, so both resets are identical.

FRAGMENTED MESSAGES

The weather station's I/O Poll response is just three bytes—one byte each for device status, temperature, and humidity.

If I used floating point or added more sensors, the CAN message limit of 8 bytes would quickly be exceeded. I'd need to send the data in two or more fragments.

I/O message fragments are like normal messages except the first byte provides a fragment flag and a fragment count. That leaves seven bytes for data.

For maximum speed, I/O message fragments are sent back-to-back with no acknowledge (ack) message from the master other than the CAN level acknowledge bit.

Fragmenting an Explicit message is more complex. You send a fragment, wait for an ack message, and send the next fragment. If the ack takes too long, resend the fragment. If you time out again, give up trying to send the message.

Many error cases can arise, like getting an ack from the master with a fragment number different from what you sent, getting a message that's not an ack while you're sending fragments, and so on.

The weather station is capable of sending and receiving fragmented Explicit messages. Its serial number and product name are long enough to require it.

Fragmented messages are a big part of DeviceNet conformance testing. My program managed to pass a self-inflicted test using the ODVA conformance software. This software generates every conceivable bogus response and breaks all but the best code. You like challenges, right?

GETTING PHYSICAL

DeviceNet requires you to keep the network data and power isolated from green-wire ground by 1 MΩ or greater. If anything can reference your circuit to green-wire ground (e.g., an RS-232 port), you must optoisolate the network.

My PC/104 DeviceNet interface is shown in Figure 3. The weather station is isolated from ground and has no ports other than DeviceNet, so I didn't need optoisolators.

Power consumption is 5 W, so I powered the whole thing from DeviceNet power. The voltage varies between 11 and 25 VDC, so use a wide input-range DC-to-DC converter.

DeviceNet also needs a miswiring-protection circuit, which lets you mix up the network connections in any possible way without frying your device or the network. The DeviceNet specification includes a circuit for this. The Philips 82C251 CAN transceiver has ESD protection and line protection up to 40 V continuous.

DeviceNet is fairly specific in its interface guidelines. I used two BCD rotary switches to set MAC ID and one more for data rate.

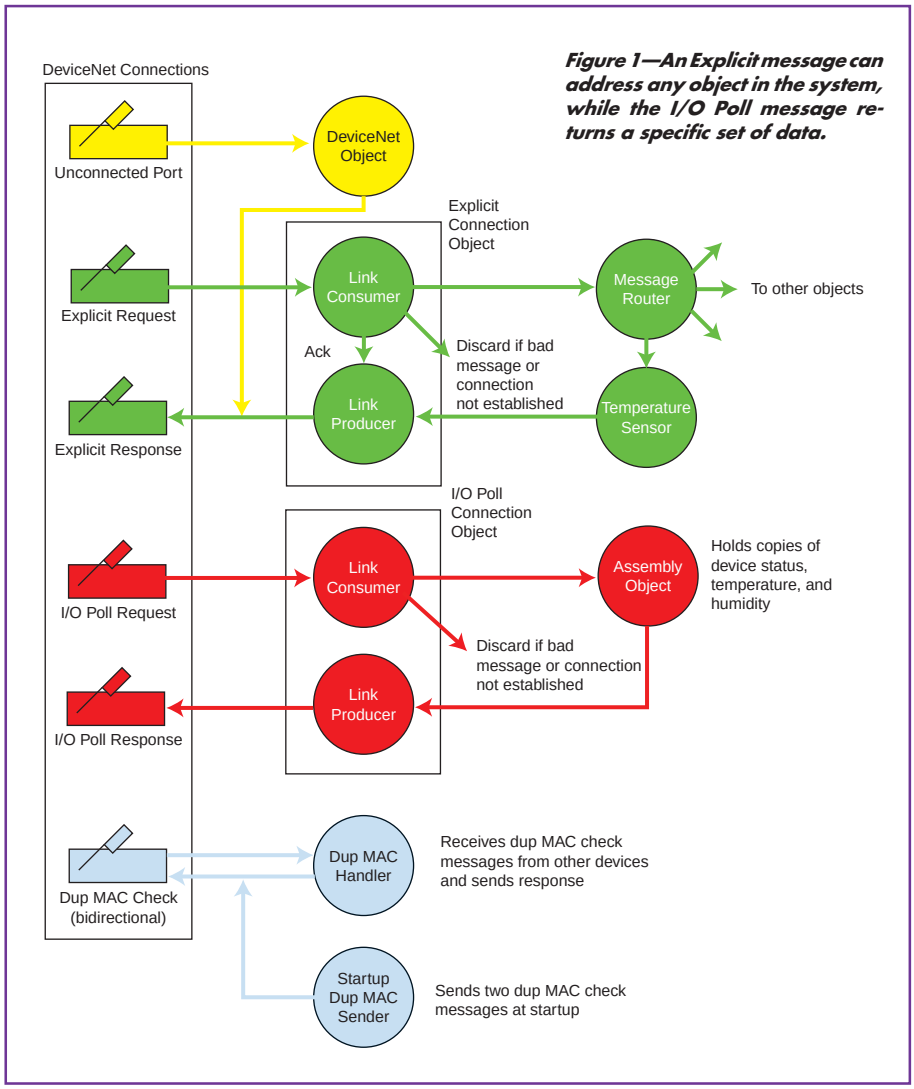

Figure 1—An Explicit message can address any object in the system, while the I/O Poll message returns a specific set of data.

I also went with the recommended bi-color LED for module and network status.

Of the three network-connector choices, I used the circular micro style. It has five pins—two for differential data, two for power, and one for the drain wire.

The data lines are referenced to power V–, so your CAN transceiver must also be referenced to this to prevent exceeding its common-mode voltage range.

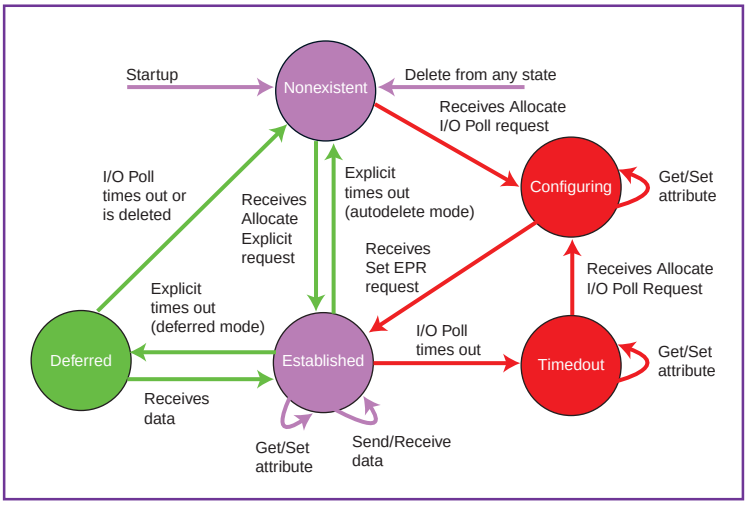

Figure 2—This state transition diagram for connection objects shows the events that cause the connection object to change state. Explicit connection states and events are shown in green, I/O Poll in red, and shared in violet.

APPLYING DeviceNet

In addition to all of the objects for DeviceNet, you need code for your application. The weather station is simple enough that I just extended the Analog Input Point object. It reads the ADC and computes sensor values.

If you have separate application objects, you must link them with the Identity object. It keeps track of device status and does resets. A standard object used for SEMI-compliant devices, the S-Device Supervisor, is designed to do this.

This article is mainly about DeviceNet, but the weather station's details are on the Circuit Cellar web site. I hope to add sensors for barometric pressure and wind speed and build units for other locations.

My network master is a '486 with a DeviceNet card from Softing GmbH. It comes with a library that makes it easy to use. Also check out National Instruments' card which works with LabVIEW and CVI.

Aside from being a fun combination of real-time software and hardware, this project shows how straightforward it is to

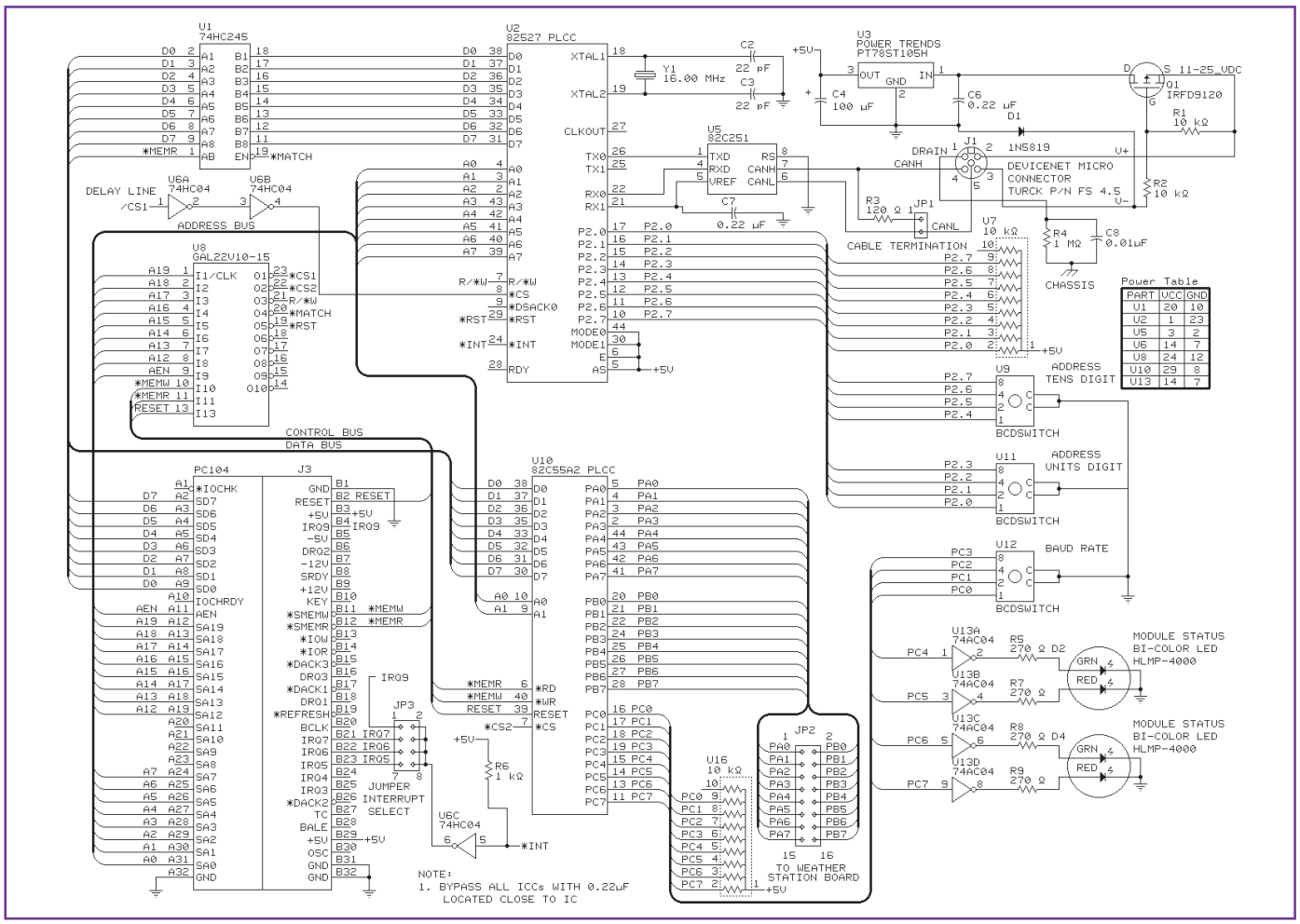

Figure 3—A simple eight-bit interface puts the Intel 82527 CAN controller on the PC/104 bus. The 24-V DeviceNet power is dropped to 5 V by U3 and then powers the entire weather station. Transistor Q1 protects against miswired network power. The PAL source code is available via the Circuit Cellar web site.

Listing 2—Here's the Analog Input Point class for the temperature and humidity sensors. The Explicit message handler allows the master to get any of the three attributes. They are read-only, so the Set Attribute service is not supported.

```
class ANALOG_INPUT_POINT{
  private:
  UCHAR value;                        // sensor value
  UCHAR data_type;                    // data type of value
  BOOL status;                        // alarm status
  static UINT class_revision;         // revision of object
  public:
  static void handle_class_inquiry(UCHAR*, UCHAR*);
  void handle_explicit(UCHAR*, UCHAR*);
  ANALOG_INPUT_POINT() {value = 0; status = 0; data_type = 2;}
};

// Handle explicit request to Analog Input Point
void ANALOG_INPUT_POINT::handle_explicit(UCHAR request[],
  UCHAR response[])
{
  UINT service, attrib, error;
  service = request[1];  attrib = request[4];  error = 0;
  memset(response, 0, BUFSIZE);       // clear response buffer
  switch(service){
    case GET_REQUEST:
    switch(attrib){                   // return requested attribute
      case 3:  // value
      response[0] = request[0] & NON_FRAGMENTED;
      response[1] = service | SUCCESS_RESPONSE;
      response[2] = value;
      response[LENGTH] = 3;
      break;

      case 4:  // status
      response[0] = request[0] & NON_FRAGMENTED;
      response[1] = service | SUCCESS_RESPONSE;
      response[2] = (UCHAR)status;
      response[LENGTH] = 3;
      break;

      case 8:  // data type
      response[0] = request[0] & NON_FRAGMENTED;
      response[1] = service | SUCCESS_RESPONSE;
      response[2] = data_type;
      response[LENGTH] = 3;
      break;
      default:  error = ATTRIB_NOT_SUPPORTED;  break;
    }
    break;
    default:  error = SERVICE_NOT_SUPPORTED;  break;
  }
  if (error)  // return error response{
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = ERROR_RESPONSE;
    response[2] = error;
    response[3] = NO_ADDITIONAL_CODE;
    response[LENGTH] = 4;
  }
}
```

program a DeviceNet interface in C++. With a little care, C++ can provide good response times and reasonable program size. EPC

Jim Brady has designed embedded systems for 15 years. You may reach him at jimbrady@ix.netcom.com.

SOFTWARE
Complete source code and schematics for this article are available via the Circuit Cellar

SOURCES

DeviceNet Information
Open DeviceNet Vendor Assn., Inc.
(954) 340-5412
Fax: (954) 340-5413
www.odva.org

SBC1386
Micro/sys, Inc.
(818) 244-4600
Fax: (818) 244-4246
www.embeddedsys.com

Digi-Key
(218) 681-6674
Fax: (218) 681-3380
www.digikey.com

DeviceNet cards
National Instruments, Inc.
(512) 794-0100
Fax: (512) 794-8411
www.natinst.com

Softing GmbH
ICT, Inc.
(978) 557-5882
Fax: (987) 557-5884
www.softing.com