CIRCUIT CELLAR INK®
THE COMPUTER APPLICATIONS JOURNAL

Jim Brady

Networking with DeviceNet

Part 1: How DeviceNet Stacks Up

With all the debate on networks these days, it's easy to get confused about the differences between networks, buses, and field buses, particularly when a new technology comes along. Join Jim for the lowdown on DeviceNet.

Reading articles on networks raises a lot of questions—like whether to call DeviceNet a network, a bus, a fieldbus, or what?

The term "bus" is used for industrial networks that control things, as opposed to general office networks that move data. I decided to forget about impressive terms and stick to basics. I want to cover the meat and potatoes of what a developer needs to know to implement a DeviceNet network.

Why are there so many network protocols? Figure 1 shows 15 of them. Why not just use Ethernet?

In the beginning, I didn't ask. I developed DeviceNet interfaces because customers requested them. Then Profibus, then others. However, with time, I wondered: is there an ideal network for a given application?

This month, I show how DeviceNet compares to other device networks and explain how it works. In Part 2, I'll look at a real DeviceNet device, code and all.

Better dust off your C++ books because DeviceNet is object oriented all the way. C works, but C++ fits like a glove.

SORTING THEM OUT

Where does DeviceNet fit into the network menagerie? At first glance, all the networks look quite similar. However, each one has a unique set of features and is designed to move a specific type of data between certain kinds of equipment.

Here, "device" refers to small to mid-size equipment with multiple integrated sensors and/or actuators. Device networks are designed to interconnect these devices.

As devices get smarter and processing power cheaper, even simple devices may have high-level network ports, producing a consolidation toward the high end of Figure 1. But there will always be a need for more than one level of network. Plant engineers don't want to share their Ethernet office network with assembly robots!

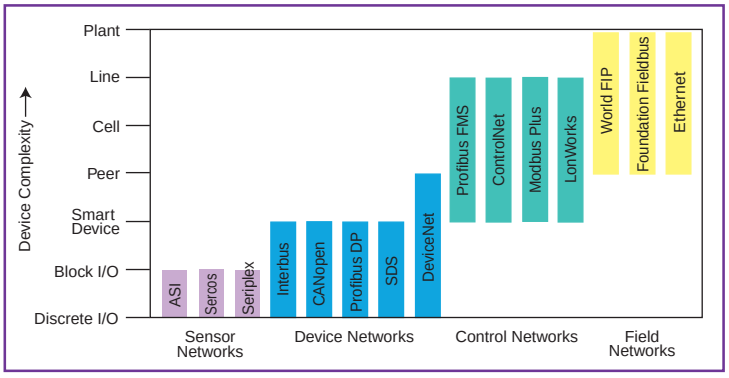

Figure 1—Automation networks at the low end move on/off messages between simple sensors and actuators. At the high end, complex equipment transfers large blocks of data plant wide. DeviceNet handles the middle ground.

NEW BREED

DeviceNet represents a new breed of device networks that offer nifty features, such as hot-plug capability, network-

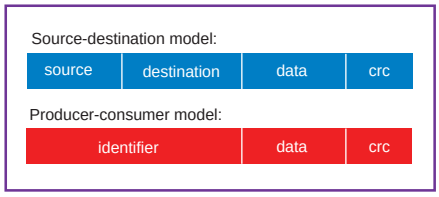

Figure 2—The new producer-consumer model identifies the data rather than the source and destination.

powered devices, peer-to-peer, fiber optics, and fault containment. And if you do it right, your device will be interchangeable with your competitor's. This may not seem good to you, but your users will like it.

Most device networks, including DeviceNet, are deterministic. In this context, "deterministic" simply means that the network can guarantee a drop-dead maximum delivery time for a critical message. Many peer-to-peer networks can't claim this because of the possibility of multiple, destructive collisions.

Ethernet using standard hubs, for example, is collision-based and therefore not deterministic. But it can be made so, thereby becoming a contender for real-time control automation.

If you have a fast processor, combined with Java, and perhaps Windows CE, most of the network code is done for you. I'm more of an 8-bit man myself, but with '386EXs at $16, it's worth considering.

Most new networks, again including DeviceNet, use a producer-consumer (also called data-centric) model as opposed to the older source-destination model. The data is considered central to the message and is what is identified, rather than the source and destination.

This situation increases the effective bandwidth of the network by permitting one-to-many broadcast messaging and time synchronization. Figure 2 illustrates the difference between the two message models.

MOTIVATION

Automation-equipment designers are eliminating old-style point-to-point wiring. They want devices from various suppliers to coexist on the same network. Ultimately, they want interchangeability between same-type devices made by different suppliers.

They're also asking suppliers to make their devices smarter, with better diagnostics. Idiot lights are no longer enough. Diagnostic sensors should provide both alarm and warning levels.

The hope is that the device warns of abnormal levels before it's too late. If a device does fail, it can easily be swapped out for another, possibly one from a different manufacturer, without powering down the network. When the new device comes up on the network, it tells you what it is, what it can do, and lets you know if it's OK.

Table 1 compares features of some popular device networks. At this level of comparison, many differences emerge. CAN-based networks have limited range because they are sensitive to time delay on the line. Most other networks use repeaters to extend their range.

With a 500-m range and a 64-node limit, you wouldn't use DeviceNet to network a large hotel. But, it is an excellent fit in a wide range of applications. The CAN protocol that DeviceNet was built on was originally designed by Bosch for use in autos and trucks. This harsh environment isn't so different from semiconductor fab tools and other automation equipment.

With a short message length, DeviceNet is well suited for time-sensitive messaging. At 500 kbps, a node doesn't have to wait more than 0.26 ms to send.

CAN

There is much literature on CAN [1], including articles by Brad Hunting ("The Solution's in the CAN," *INK* 58) and Willard Dickerson ("Vehicular Control Multiplexing with CAN and J1850," *INK* 69). Here, I'm only going to cover the most important points.

Network protocols such as DeviceNet, SDS, and CANopen—all built on top of CAN—inherit a well-established foundation for reliable real-time communication. At least five IC vendors make CAN controllers that handle all the details of the CAN protocol for you.

What makes CAN so good for real-time messaging? Short message length and priority-based collision resolution.

The latter feature is a major reason for CAN's popularity. If a collision occurs, the higher priority message still gets through intact! In fact, the lower priority colliding node ends up receiving the higher priority colliding node's message.

With most other protocols, if a collision occurs, no data gets through, and this can happen over and over. Not cool for a message to an automobile brake!

CAN requires nodes to listen before sending. If two or more nodes decide to send at the same time, it's the same time to within a small fraction of a bit time. That means the messages line up bit-for-bit.

Because the network line is essentially wire-ORed (see Figure 3), a low bit overrides a high bit. Nodes listen to what they send, and the node sending the high bit will realize that it's receiving a low bit.

At that instant, it knows there is a collision and switches from sending to receiving. The most significant bit of the Connection ID is sent first, so the message with the lower-numbered ID wins the bitwise arbitration and gets through intact.

For this to work, nodes at opposite ends of the cable must have their bits line up to within about half a bit-time. Thus, the round-trip delay of the cable is limited to 1.0 µs at the maximum DeviceNet speed of 500 kbps. The corresponding cable

	DeviceNet	SDS	Profibus DP	LonWorks
Max. range	500 m at 125 kbps	500 m at 125 kbps	9600 m at 94 kbps	2700 m at 78 kbps
Max. speed	500 kbps	1 MBps	12 MBps	1.25 MBps
Max. nodes	64	128	126	32,385
Max. message length	8 bytes	8 bytes	244 bytes	228 bytes
Bus access	Peer, M/S	Peer, M/S	M/S	Peer, M/S
Error resistance	15-bit CRC	15-bit CRC	16-bit CRC	16-bit CRC
Deterministic	yes	yes	yes	in M/S mode
Hot-plug capability	yes	yes	yes	yes
Fault confinement	yes	yes	yes	yes
Line-powered devices	24 VDC, 8 A	yes	optional	optional
Media	twisted pair	twisted pair	twisted pair, fiber, RF	twisted pair, fiber, RF, power line

Table 1—Repeaters are required for Profibus and LonWorks to achieve this range and node count. DeviceNet and SDS are two popular CAN-based networks. Others exist, too, such as CANopen and CAN Kingdom. Profibus is a master-slave protocol that is popular in Europe and gaining support in the U.S. LonWorks, widely used in building automation, is now seeing use in the equipment automation field.

length can be determined by:

$$0.5\ (1.0\ \mu s \times 300\ m/\mu s \times 0.72) = 108\ m$$

where 300 m/µs is velocity of light and 0.72 is the velocity constant of the DeviceNet cable. That's why you have the 100-m limit on cable length at 500 kbps.

MESSAGE RELIABILITY

The CAN controller calculates a 15-bit CRC value for the received data and compares it against the CRC it received. If an error is detected, the node originating the message is notified so it can resend the message.

If the originating node sends too many messages for which it gets an error back, it goes offline. That way, a bad device won't crash the entire network.

Your CPU detects this event by reading the CAN chip's status register. You have the option of staying offline or initiating an error-recovery sequence.

If the CAN controller originating a message doesn't hear back from at least one other device that the message was correctly received, it will resend. Thus, a lonely node will just sit there and send over and over.

When you do get a message from your CAN controller, you know it's correct. And when you tell the CAN controller to send a message, it keeps trying until the message gets through. Pretty good, considering all this is handled by one $8 chip.

To make CAN into a usable network, you need a way to string messages together, establish connections, and handle errors. That's where DeviceNet comes in.

DeviceNet CONNECTIONS

DeviceNet provides a structure for establishing logical connections between devices, releasing connections if they go unused for too long, and stringing messages together if you need more than 8 bytes. It also provides an object-oriented framework to tell you how to structure your network code.

If your device-type is in the DeviceNet library, it even tells you how your device should behave. That part is necessary to make devices completely interchangeable.

Central to DeviceNet is a concept called a connection. Think of it as a telephone connection. When you call someone, you establish a connection. That connection is yours, and other people talking on the same fiber have different connections. The connection breaks when you hang up or in some cases if you stop talking for a while.

In DeviceNet, each connection is identified by an 11-bit number called a message identifier or connection ID. This number includes your device's Media Access Control Identifier (MAC ID), which is a number from 0 to 63, usually set by a switch on your device.

DeviceNet provides a set of 11 predefined connections, called the predefined master/slave connection set (see Table 2). Wait a minute…master/slave?

Yes, that's a letdown after expecting peer-to-peer, but most DeviceNet products on the market today are slave-only devices. The implementation is much simpler and less memory consuming.

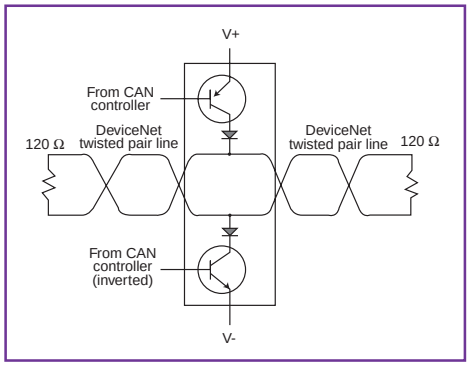

Figure 3—A DeviceNet line with a one-line driver uses a low output from the CAN controller to turn both transistors on while a high output turns both off. It works like a wire-OR, where any low output dominates all other nodes' high outputs.

A peer device must include a lot of code to dynamically establish and configure connections. If you really want to include peer capability, the standard enables you to put it in a device along with the predefined connection set.

In fact, the DeviceNet standard distinguishes between a slave device that also has peer capability, and a slave-only device. In this article, I'm sticking to the simpler slave-only device, which uses only predefined connections.

DeviceNet MESSAGES

DeviceNet has two basic message types: explicit and I/O. The predefined connection set includes one explicit connection as well as four I/O connections of different kinds.

Explicit messages include the path to locate the data of interest. This consists of the class ID, instance number, and attribute ID. They also specify an action to be taken (e.g., set or get). Finally, they include the master's MAC ID because a slave must respond only to its master.

With of all this baggage, explicit messages aren't efficient. They are used mainly for initial configuration, although in theory they could be used for everything. Your device must support this connection, but others are optional.

In an I/O poll connection, the master periodically sends a request saying in effect, "Hey! Send me your data." It is an efficient exchange because the master doesn't need to send any baggage.

When the slave device sees a message with this connection ID, it returns a prearranged set of data. On more complex devices, the master can usually select from various data sets. If only one I/O connection is supported, it's usually this one because it is general purpose.

In an I/O change-of-state (COS) connection, the device sends its data when it changes more than a selected amount. This choice is good for slowly changing data.

The I/O cyclic connection uses the same connection ID as the COS connec-

Connection ID Bits											Description
10	9	8	7	6	5	4	3	2	1	0	
	Message ID				Slave's MAC ID						Group 1 Message
0	1	1	0	1							Slave's I/O Change-of-State/Cyclic Message
0	1	1	1	0							Slave's I/O Bit-Strobe Response
0	1	1	1	1							Slave's I/O Poll Response
		Slave's MAC ID						Message ID			Group 2 Message
1	0							0	0	0	Master's I/O Bit-Strobe Request
1	0							0	0	1	Reserved
1	0							0	1	0	Master's Change-of-State/Cyclic Ack.
1	0							0	1	1	Slave's Explicit Response
1	0							1	0	0	Master's Explicit Request
1	0							1	0	1	Master's I/O Poll Request
1	0							1	1	0	Unconnected Port
1	0							1	1	1	Duplicate MAC ID Check

Table 2—These 11 connections come from two DeviceNet message groups, with the Connection ID made up differently in each case. Group 1 messages are higher priority, used for the slave's I/O messages. All messages originate from the master, except for the Duplicate MAC ID Check and the slave's COS/cyclic message.

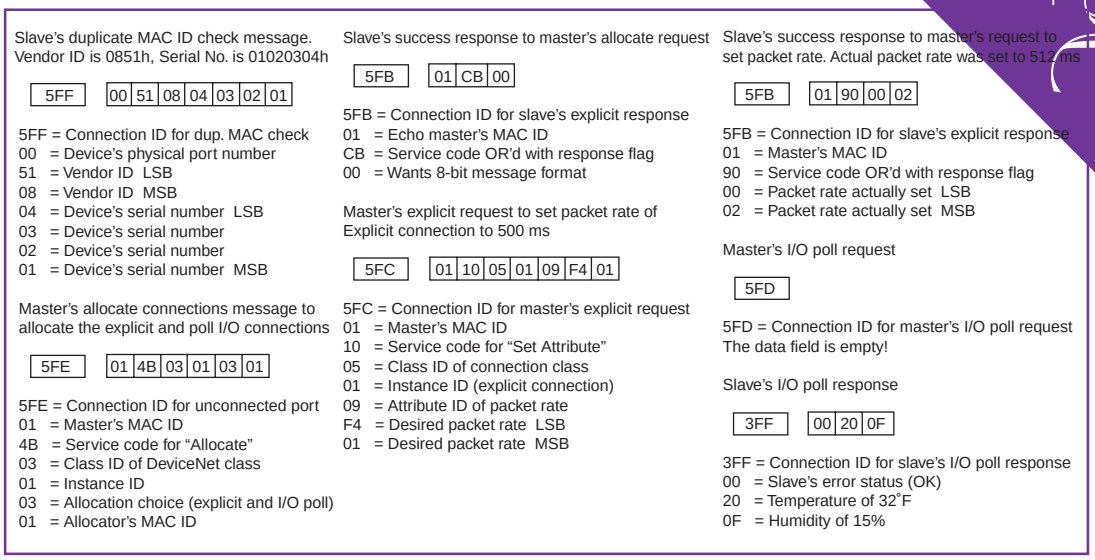

Figure 4—These are messages you see on a DeviceNet analyzer during a typical start-up sequence. The Connection IDs are based on a slave MAC ID of 03Fh. Note the variable-length data field. All message values are in hex, and the least significant bit is always sent first.

tion. The master chooses between the two when it allocates the connection. With the cyclic connection, the device sends its data at a selected rate. This choice works well for rapidly changing data.

In an I/O bit-strobe connection, the device sends only a few bits of data in response to the master's bit-strobe request. This is a good choice for simple on/off sensors.

With DeviceNet, no connections exist until they are allocated. How do you allocate a connection in the first place? I think the designers of DeviceNet must have struggled with this.

It turns out that a special connection always exists—the unconnected port. The master sends an allocate message to the unconnected port to allocate connections and specify which of the predefined connection set it wants to use. Once you have connections, you can send messages over them.

Many CAN controllers have individual mailboxes for incoming messages. You can assign each connection ID to a different mailbox. The Intel and Siemens chips have 15 and 16 mailboxes, respectively, enough for the predefined connection set with room to spare.

This makes your program modular from the start. Different messages come in different boxes.

STRINGING MESSAGES TOGETHER

An explicit message from the master uses five bytes of the eight bytes available for the path, service, and master MAC ID. This leaves only three bytes for actual data.

If the master sends a four-byte long int, two separate messages are needed. These messages are called fragments and are just like a regular message, except the first part of the data field contains fragment information.

For an explicit-message fragment, the first byte contains a fragment flag, and the second byte specifies the fragment type (first, middle, last) and the fragment count. The fragment count is only a six-bit value but can roll over any number of times, allowing for messages of unlimited length.

For an I/O-message fragment, only one byte is used for fragment information (the one specifying fragment type and count) to maximize the space for actual data. In this case, the fragment flag is implied if the produced connection size is greater than 8.

For explicit or I/O fragments, the receiver simply concatenates the data obtained from each fragment, stopping when it sees the flag indicating the final fragment.

With fragmented messages there is the question of whether the receiving device needs to send an acknowledge for each fragment received. It does for an explicit message but not for an I/O message. I/O message fragments are sent back-to-back for maximum speed.

SOME REAL MESSAGES

Figure 4 shows some real DeviceNet messages based on a typical out-of-the-box slave MAC ID of 63 and a master ID of 1. This situation is typical because it gives the slave device the lowest priority and the master the highest.

For clarity, Figure 4 shows only the connection ID and data fields of the CAN message frame. There are two additional fields. One is a length field that tells the receiver how many bytes of data to expect.

The other field is used for acknowledgment. The node receiving the message sends an ack bit if the message was OK.

CAN messages are variable length. Depending on the number of data bytes sent, a frame can range from 44 to 108 bits.

The first message your device deals with is a duplicate MAC ID check message. Before going online, your device must make sure it has a unique MAC ID.

To do this, your device broadcasts two duplicate MAC ID check messages, 1 s apart, addressed to its own MAC ID. All devices receive this message, but none respond unless your device addresses them!

After sending the duplicate MAC ID check message twice and hearing no response, you can go online.

But since you have no connections yet, you must ignore all messages with two exceptions—a message to your unconnected port to allocate connections, or a duplicate MAC ID check message that contains your MAC ID.

Duplicate MAC ID check messages would be from another device hoping to go online but set to the same ID as yours. Your response to this message prevents the offending device from going online.

When you get a message to your unconnected port it will be the master specifying which connections out of the predefined connection set it wants to allocate. The allocation choice byte contained in this message will have bits set which correspond to the connections it wants.

If you support these connections, allocate them and return a success response. Otherwise return an error and don't allocate any connections.

Keep track of the master MAC ID that allocated these connections, because from now on this is your master. A message containing a different master MAC ID is

ignored.

For each connection allocated, start a timer that deletes the connection if it times out. For the explicit connection, the timer defaults to 10 s. For the I/O poll connection, it defaults to zero and must be set by the master before the connection is used.

The time-out value is controlled by an attribute called the expected packet rate (EPR), which the master can set. Your time-out value, in milliseconds, equals the EPR × 4. Thus, the EPR for the explicit connection defaults to 2500.

In the I/O message example shown in Figure 4, note that no data, path, or service code is sent with the master's request. The data set returned with the slave's response is already specified in the manufacturer's device profile or electronic datasheet.

More complex devices may have many sets of data the master can choose from. The Master selects the set it wants by changing the slave's produced connection path.

Because no baggage is involved in the I/O message, it's an efficient process. A device such as the one modeled in Figure 5 can send its entire sensor and status data in one I/O message. With explicit messaging, many full-length messages would be needed.

OBJECT LIBRARY

With DeviceNet, a device is modeled as a collection of objects. Each object has attributes and behaviors, and can be implemented directly as a C++ class. Figure 5 shows the object model for a DeviceNet device with two analog sensors.

Each class has a class ID, objects have an instance ID, and attributes have an attribute ID. By specifying these three ID numbers, any attribute in the device can be addressed.

The DeviceNet Object Library is contained in Volume II of the standard. In addition to network-related objects, it includes about 25 objects that model real-world switches, sensors, actuators, PID loops, position sensors, and controllers. There are more on the way, including an Analog Sensor Object that models advanced sensors, with capabilities such as calibration, auto-zero, offset, gain, and setpoints.

CONFORMANCE TESTING

When you complete your DeviceNet product, how do you know it meets the standard? The University of Michigan will test your product to see if it conforms to the rigors of the DeviceNet protocol.

If this sounds too intimidating, you can avoid embarrassment by getting the software and a DeviceNet interface card so you can test it yourself. When you do go to the test lab, bring your laptop and compiler along. The process is designed to be a fix-it-as-you-go experience.

DeviceNet STANDARDS

The DeviceNet standard keeper is the Open DeviceNet Vendor Association (ODVA). It manages the evolving standard and assists vendors in developing their products and the Device Profiles for them. Within the association are 14 active special-interest groups, organized along product lines. The ODVA Web site lists these groups.

You can get your feet wet by getting a DeviceNet catalog at no charge from ODVA. The first chapter has an excellent overview of CAN and DeviceNet. If you decide to jump in, you can purchase the full DeviceNet specification (the CD version includes a good search engine) from ODVA. Later you can become a member, join a SIG, and play a part in defining network standards for your industry.

Next month, I'll turn some DeviceNet objects into C++ classes, embed them in a '386EX, and hang a DeviceNet interface on it. EPC

Jim Brady has designed embedded systems for 15 years. You may reach him at EbaraJim@aol.com.

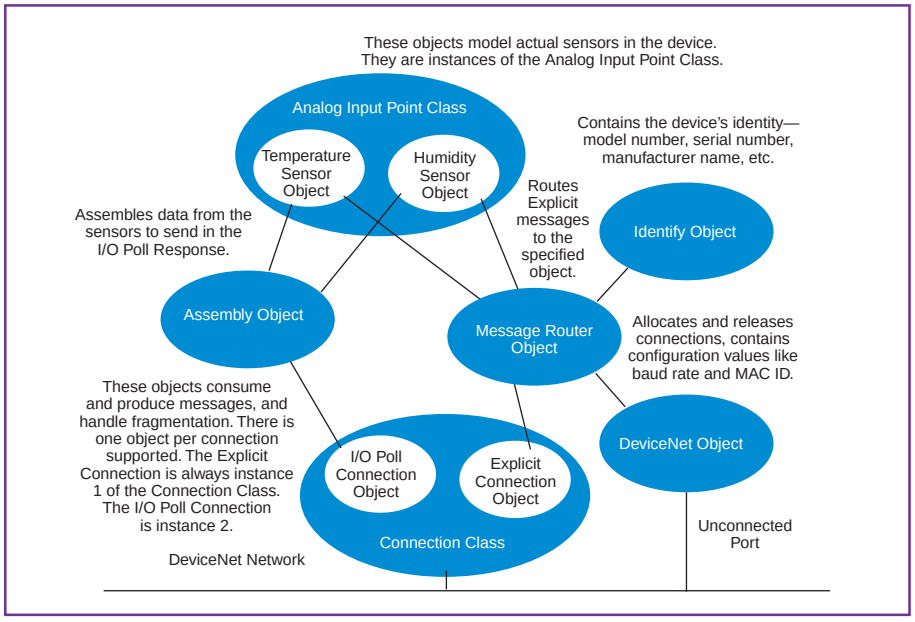

Figure 5—Here's the Object Model for a DeviceNet device with two analog sensors. These objects can be implemented as instances of C++ classes, based on the detailed models provided in the DeviceNet Object Library.

REFERENCE

[1] J. Schill, "An Overview of the CAN Protocol," *Embedded Systems Programming*, p. 46, Sept 1997.

SOURCES

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

DeviceNet Conformance Testing
University of Michigan
(734) 764-4336
Fax: (734) 936-0347
www.eecs.umich.edu/~sbus

DeviceNet Interface Cards
Huron Networks, Inc.
(313) 995-2637
Fax: (313) 995-2876
www.huronnet.com

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

SST, Inc.
(519) 725-5136
Fax: (519) 725-1515
www.sstech.on.ca