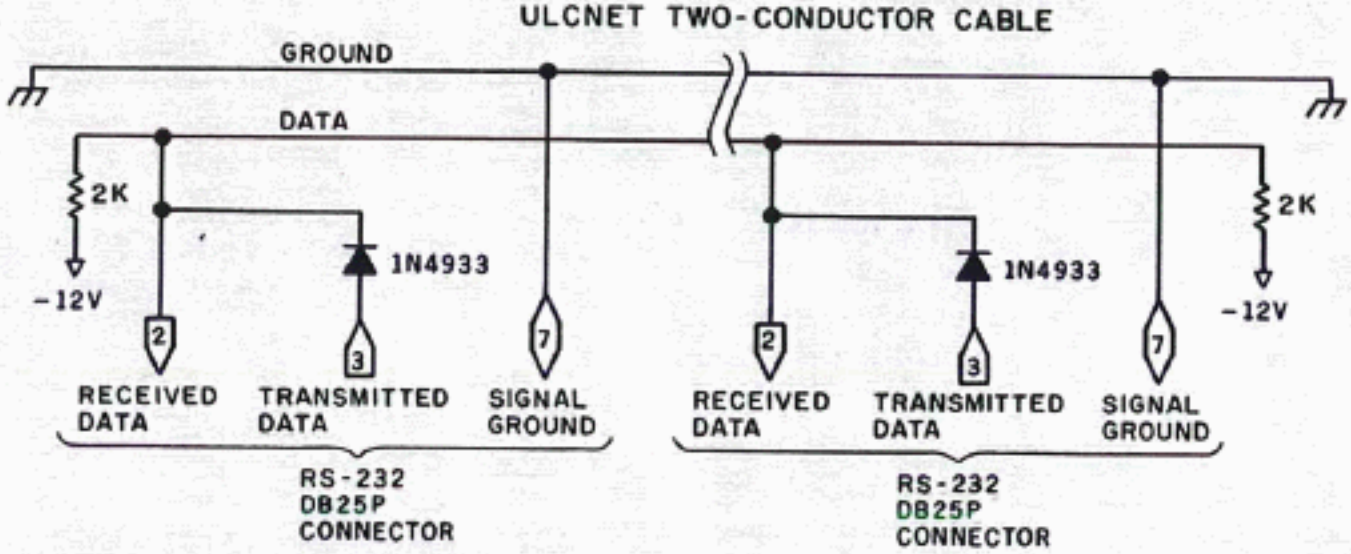

Figure 1: *Simplest version of ULCNET. The addition of a diode, cable, and terminating resistors converts RS-232 ports into a basic network for personal computers.*

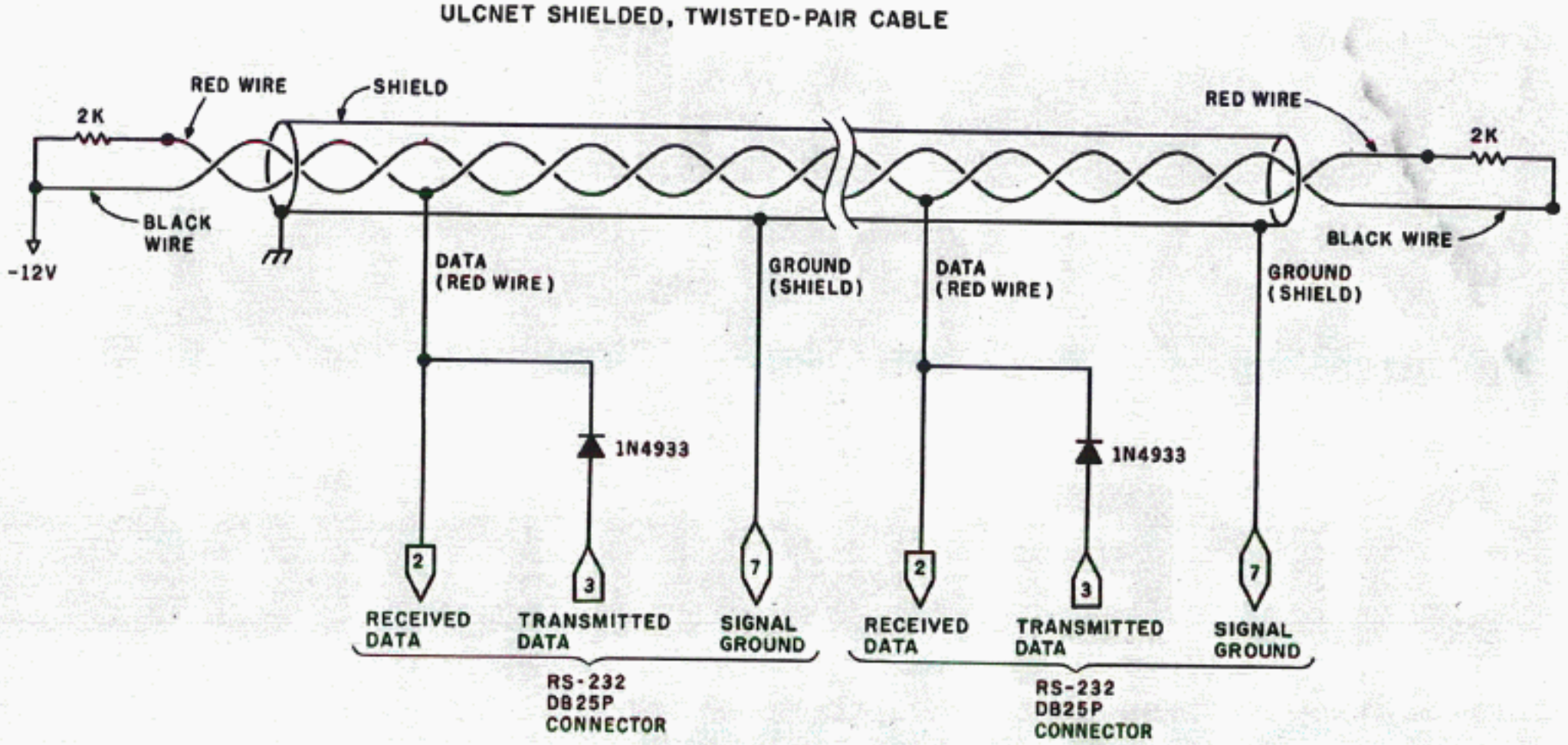

Figure 2: *ULCNET for two-wire shielded cable. In this version, a single −12 V power supply is specified, and power is transmitted to the pull-up resistors via one conductor of the cable.*

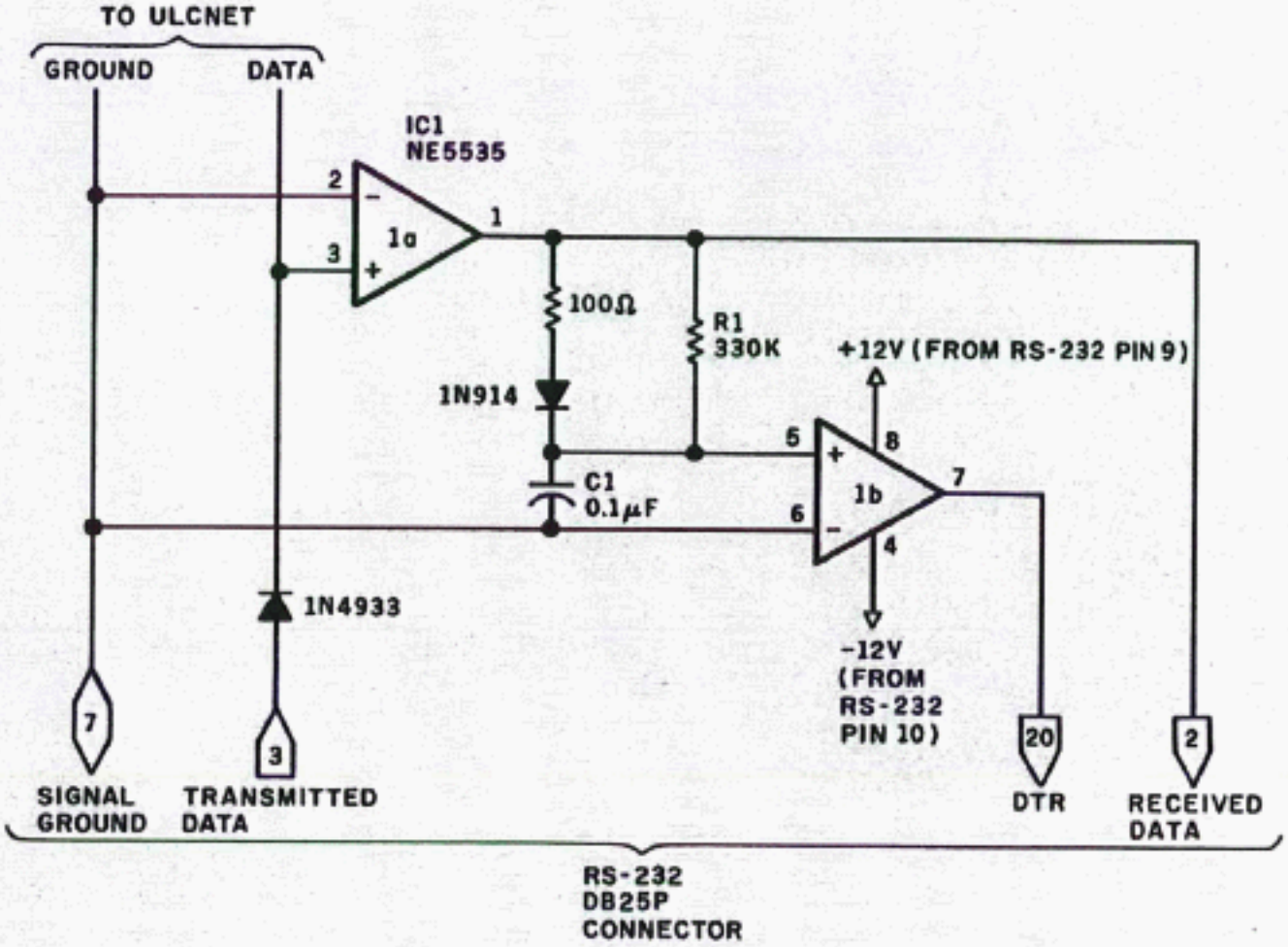

Figure 3: *Simple modifications expand network capacity. An operational amplifier reduces the load placed on the net by each node, so that a virtually unlimited number of nodes can be used. Resistor R1 and capacitor C1 control the op-amp comparator to signal that the network is busy. The components shown can be used with speeds as low as 1200 bps; see table 1 for alternate selections.*

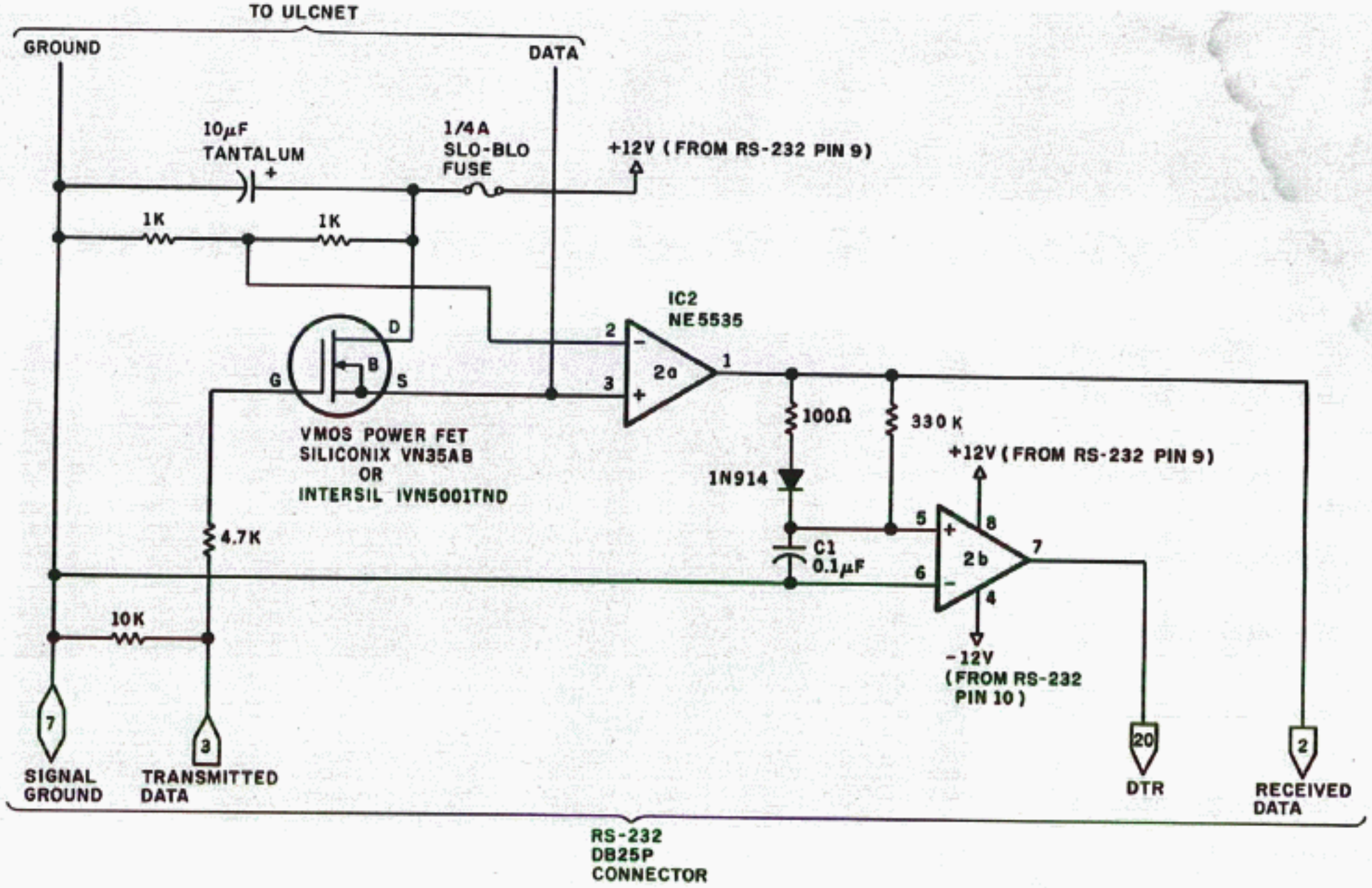

Figure 4: *Fast version of ULCNET. The primary limitation of driving power is overcome by installing an output transistor at each port. The transmitter shown may draw as much as 1 A from the −12 V supply, for short periods.*

ULCnetPC.WS4

- "Ultra-Low-Cost Network for Personal Computers"
  Ken Clements & Dave Daugherty
  BYTE, October 1981, p.50

(Retyped by Emmanuel ROCHE.)

Ten years ago, computer "hackers" listened with glee to predictions that technological advances would soon allow them to buy their very own computers. Indeed, the seers predicted, the computers of the future would fit into a spare bedroom or basement, and wouldn't even require air conditioning. The word went out: start saving $100,000 to be ready when that great time came.

The time came with a vengeance. Today, you can hardly take twenty paces around a technical organization, school, or office without bumping into, or being addressed by, yet another computer.

One of the sad outcomes of this exponential growth was creation of the computer junkie, the unfortunate soul who went out and bought each of the newest computers he or she could afford. The junkie ended up with a basement full of equipment, and a computer habit that could be satisfied only by more spending.

Just when the future was looking grim for these computer junkies, salvation took form and appeared on college campuses. Perhaps the best explanation came from a recruiter from the giant Xmumblex Corp, who took a young graduate aside and whispered: "I have just one word for you: ***networks***".

The big-computer companies and an army of computer scientists apparently will be going network crazy for the next ten years. This development thrills the computer junkies, because it provides more computer "stuff" to get excited about. And, the junkies calculate, if they could get their own personal networks going, they might be able to string together all the "coldware" collecting dust in their basements.

What stops most people from going ahead with their own networks is complexity, both in terms of cost and technical considerations. A typical coaxial network "box" may be as difficult to build and interface as was the computer you wanted to network. This stumbling block is particularly large for the computer junkie who owns no two pieces of hardware that are the same. He must come up with a new interface for each one.

But almost all those pieces of hardware have at least one RS-232C serial port. RS-232C was designed to provide point-to-point communication, and it requires some central manager "box" to produce a network. But, with as little as one diode per port, two resistors for the ends, and a -12 Volt source, you can turn RS-232C into ULCnet, the Ultra-Low-Cost Network.

Simple technique

The primary technique for this transformation is shown in Figure 1. It is amazingly simple: just connect a diode in series with the transmit line, then connect the receive line and the diode to your cable. At the ends of the cable, you will need resistors to "pull down" the line to -12 Volt, and to help soak up reflections. Serial communications via RS-232C are usually not too fast, so the type of cable and exact terminations are not critical.

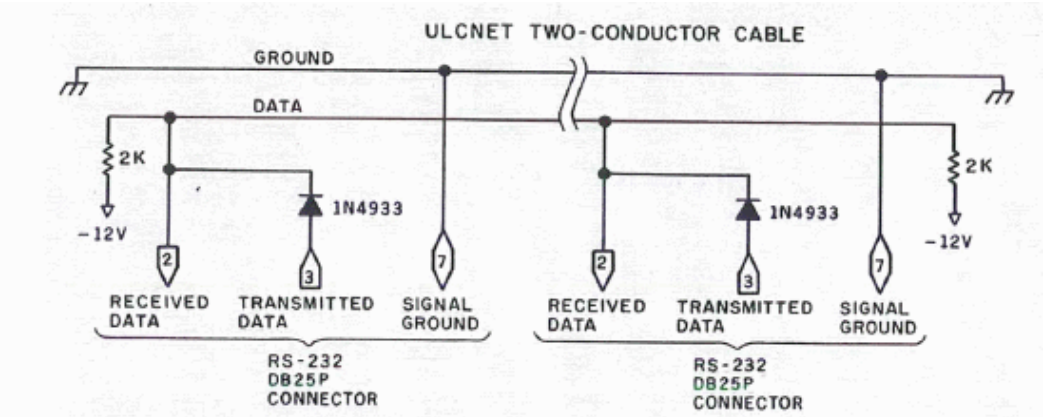

Figure 1: *Simplest version of ULCNET. The addition of a diode, cable, and terminating resistors converts RS-232 ports into a basic network for personal computers.*

Figure 1. Simplest version of ULCnet

For most applications, it is easy to use shielded twisted-pair cable for the network. This allows one of the wires in the pair to carry the -12 Volt needed by the termination resistors at the end of the cable. An example of wiring the termination is shown in Figure 2. This technique assumes that, somewhere along the line, the black wire in the pair is connected to -12 Volt, and the shield is grounded.

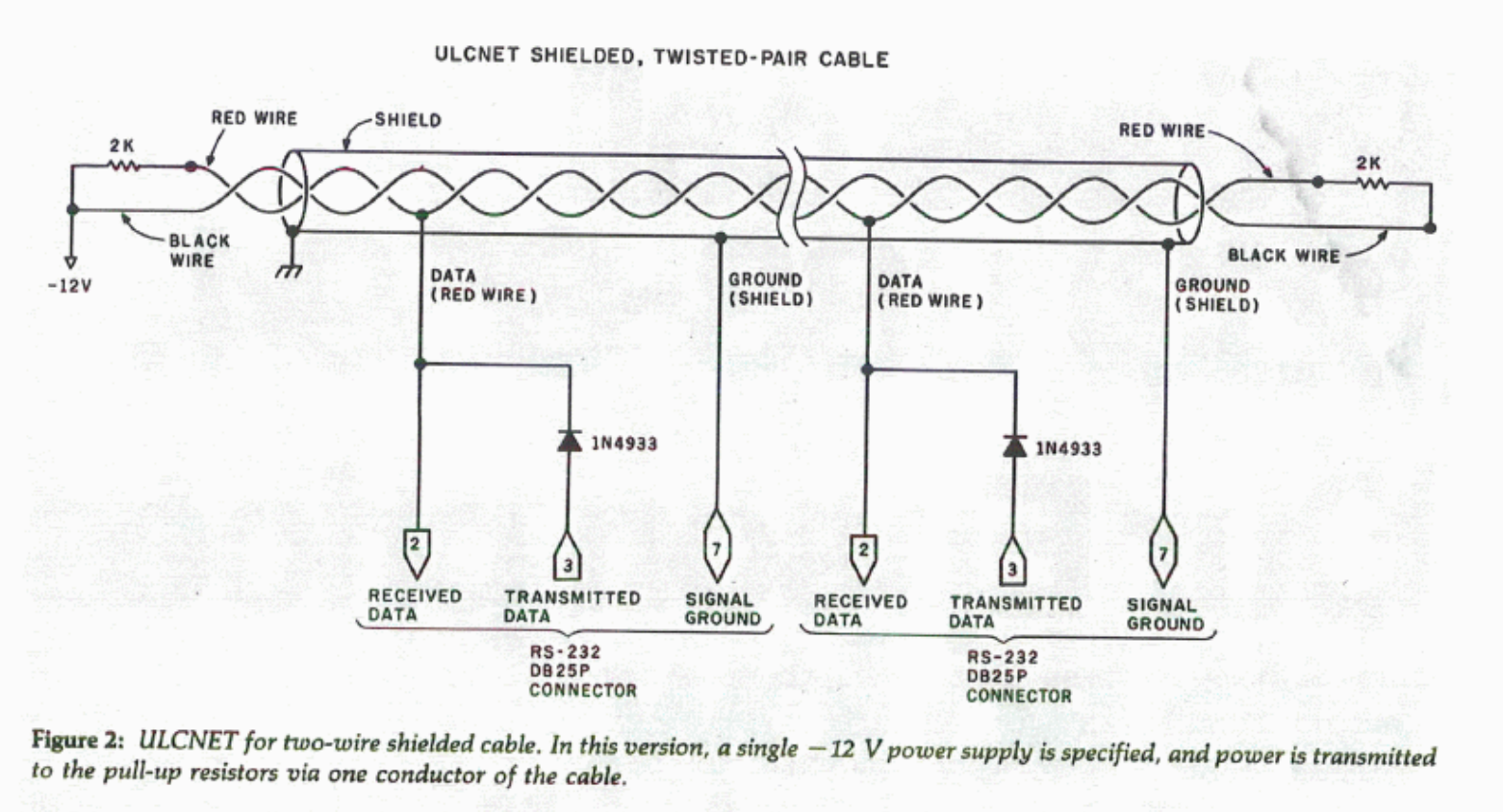

Figure 2: *ULCNET for two-wire shielded cable. In this version, a single −12 V power supply is specified, and power is transmitted to the pull-up resistors via one conductor of the cable.*

Figure 2. ULCnet for two-wire shielded cable

When characters are sent through an ULCnet port, they are received at all the ports on the network, including the port that did the sending. However, if two or more ports send different messages at the same time, the transmitting ports will each receive something other than what they sent: the logical OR of the two messages. This allows an extremely important property, namely ***collision detection*** (a property also used in Xerox's Ethernet).

The ULCnet uses the fact than an RS-232C port holds its transmit line at negative voltage when not transmitting, and then pulses the transmit line positive at the start of a character. The RS-232C standard defines a positive

level as a transmitted 0, and a negative level as a binary 1. In other words, a character starts with a 0, followed by a byte of code transmitted low-order bit first. At least one binary 1 is inserted after each byte-long character, and it is called the stop bit.

The termination resistors on the ULCnet provide the negative level, and each port may "pull" the line to a positive level by the start pulse of a character. In terms of bits, the resistors supply the 1s, and the ports supply the 0s.

The speed and distance limits of the ULCnet come from a combination of the drive-current limitations of an RS-232C port and the load each receiver puts on the net. The limits lead to a three-way trade-off of distance, speed, and number of receivers. For example, you might use the ULCnet at 19,200 bps (bits per second) for six devices separated by 20 feet, or you might connect three devices with two miles of wire and run at 300 bps.

Improvements

Some simple modifications can be made to expand the network capability. The first modification gets the number of receivers out of the trade-off equation. Figure 3 shows an alternate ULCnet connection in which an op amp (operational amplifier) is used to buffer the incoming signal. This reduces to almost nothing the load each node places on the network, thereby allowing as many connections as desired on the network.

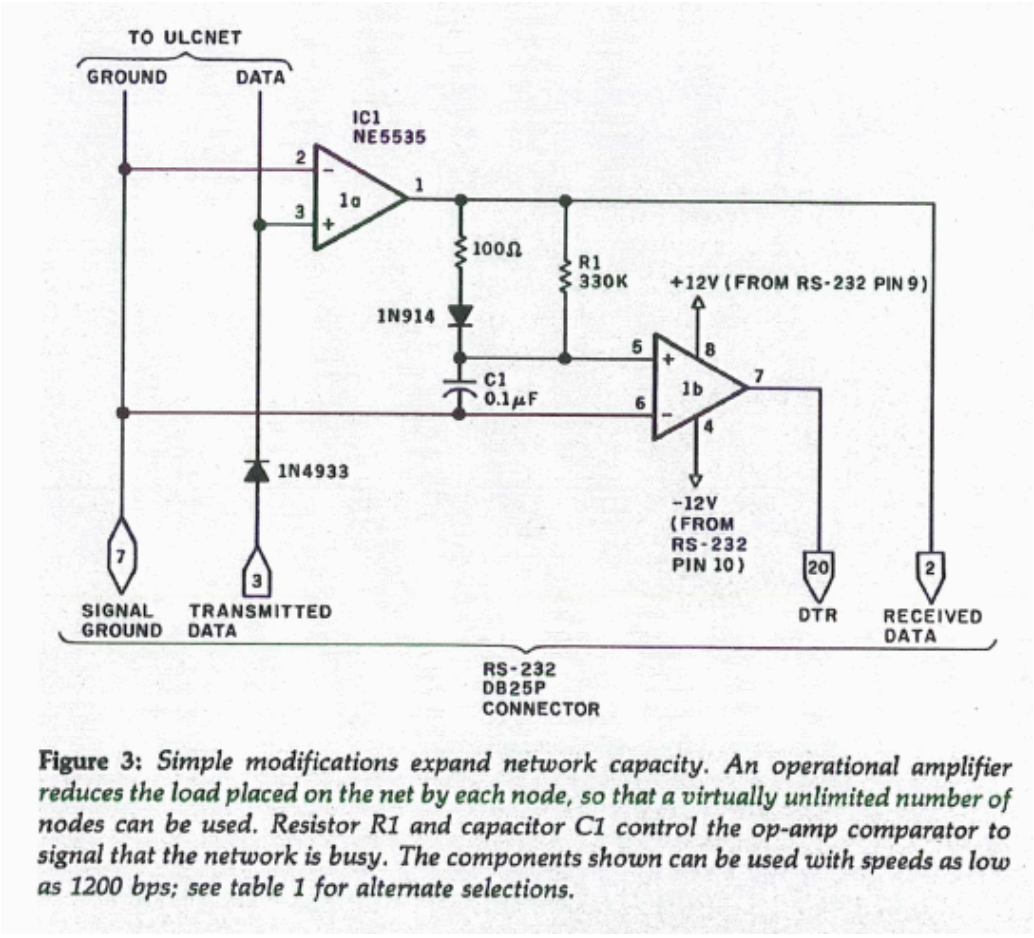

Figure 3: *Simple modifications expand network capacity. An operational amplifier reduces the load placed on the net by each node, so that a virtually unlimited number of nodes can be used. Resistor R1 and capacitor C1 control the op-amp comparator to signal that the network is busy. The components shown can be used with speeds as low as 1200 bps; see table 1 for alternate selections.*

Figure 3. Simple modifications expand network capacity

Some RS-232C ports have +12 Volt and -12 Volt supplied on pins 9 and 10 of their DB-25 connector (these can be used to power the op amp). Most, however, do not, so the user will need to run a pair of wires to the power supply of the computer. If some other power source is used, the user must be sure its ground reference is the same as pin 7 of the RS-232C port.

Figure 3 also shows a circuit that drives the DTR (Data Terminal Ready) input of the RS-232C port. This circuit is used to detect activity on the network, and it will assert (pull high) DTR if the network is busy. The circuit works by charging C1, a 0.1 uF capacitor during the start bit of a character. The capacitor will then discharge through the 330-kilohm resistor R1 when characters are no longer being transmitted. The choice of values for these two components is set by the slowest data rate to be used on the network. The

choice shown was picked for 1200 bps operation. If 2400 bps is desired as the lowest rate, then halve R1's value. The resistor can be scaled in this manner for the lowest transfer rate desired. Table 1 suggests resistor values for various data rates, but plan to experiment.

Table 1. Suggested resistor values

Data rate (bps)	Size of R1 (kilohm)
1200	330
2400	160
4800	82
9600	39
19.2 K	22
38.4 K	10
76.8 K	5.1
153.6 K	2.2

The purpose of the ***busy flag*** circuit shown in Figure 3 is to relieve the software of checking the condition of the network, and to provide a signal that can be used with an interrupt-driven system. (These techniques are discussed later.)

Aiming for speed

Figure 4 is included for those who crave speed. Here, the drive limitation is overcome by using a power FET (Field-Effect Transistor) to drive coaxial cable. The cable can be either standard 50 Ohm coax, or the 75 Ohm coax commonly used in cable TV operations. Whichever you choose, you ***must*** use a matching resistor (50 Ohm or 75 Ohm) on ***each*** end of the cable.

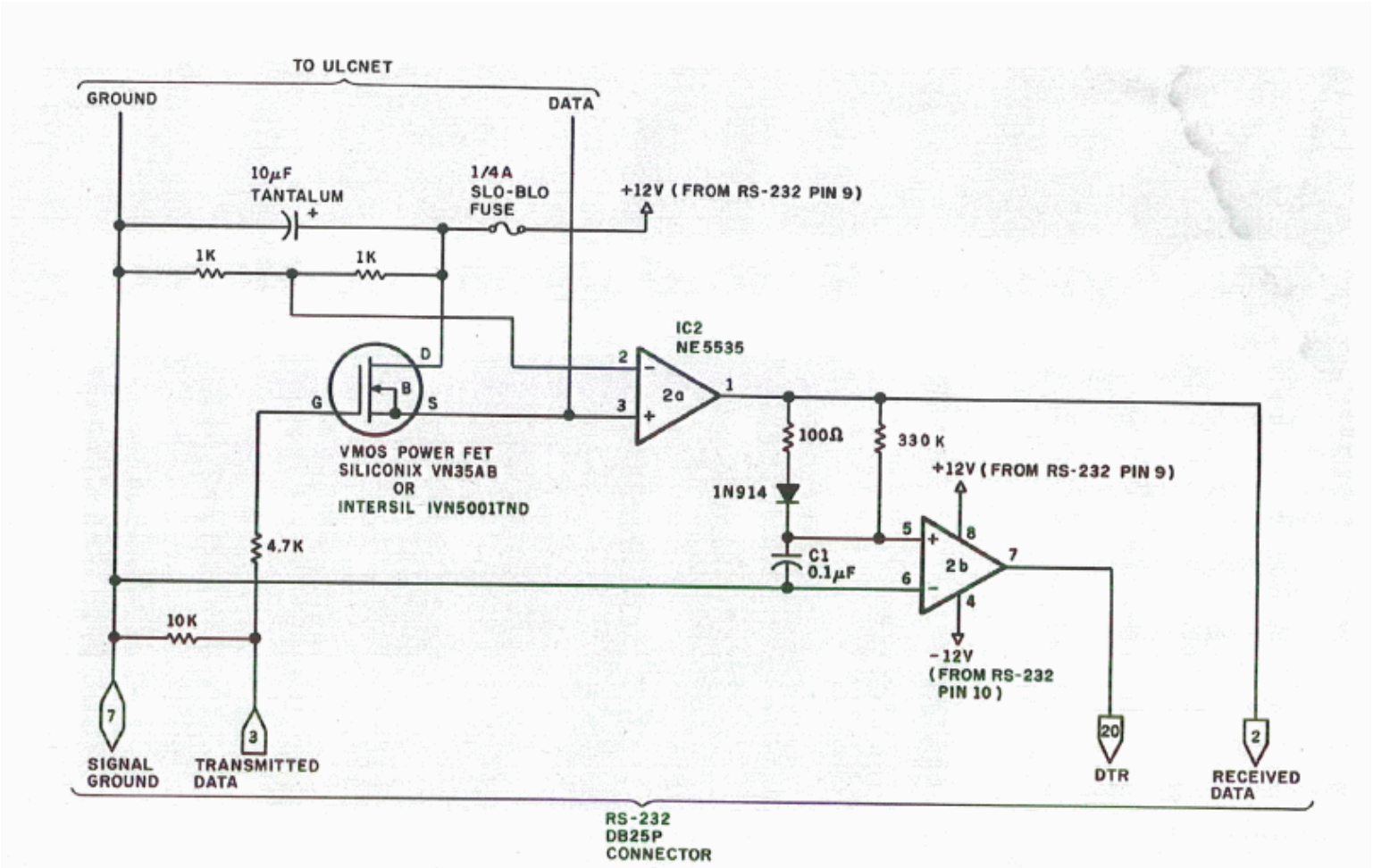

Figure 4: *Fast version of ULCNET. The primary limitation of driving power is overcome by installing an output transistor at each port. The transmitter shown may draw as much as 1 A from the −12 V supply, for short periods.*

Figure 4. Fast version of ULCnet

In this form of the ULCnet, the logical 0 is represented by a +12 Volt level, and the logical 1 is at 0 Volt. The same busy-detect circuit is used, and all of the network techniques will remain the same. This version of ULCnet is included for those who have very fast controller devices on their ports, and want to operate in the 50 Kbps to 1 Mbps range.

To make this fast version work, it is important to have a very solid source of +12 Volt that can put out about one amp for a very short time. The fuse included in Figure 4 is meant to shut down the connection if the computer turns on the power FET and leaves it on. If not corrected, this error condition would cause the entire network to halt.

One way to set up a network is shown in Figure 5. This setup would allow all the computers to share the hard disk and the printer. The computer directly connected to the hard disk and printer would be partially dedicated to servicing the requests for these resources.

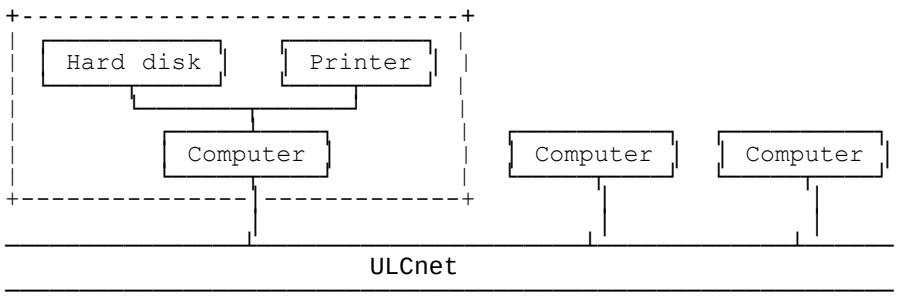

Figure 5. One possible ULCnet configuration

Design issues

Now that we have discussed the hardware for the ULCnet, let us look at some of the issues involved in designing software for the network. These issues are: node-addressing concepts, message formats, task layering, low-level transmission and reception, communication protocols and error recovery, dialogue pipes, special types of networking communications, and networking under multi-tasking operating systems.

First, let us define a node as any device connected to the ULCnet that has the ability to transmit information, receive information, or both.

If there are more than two nodes on a network, some mechanism is needed to uniquely specify the destination of transmitted information. This need is fulfilled by assigning to each node a unique numeric address. A single digit may be sufficient to specify the node for which a message is intended.

Many mechanisms can be used to inform the node's software of its particular address. The possibilities include establishing a switch setting on an input port, including the information in the software for each node (but each node would then need a unique version of the network software), or having the software query the user for an address during initialization.

An address does not necessarily have to be a number, as long as it can be uniquely recognized. It could be a character string such as EVA or SHIRLEY, but you must be willing to pay the cost of pattern matching in order to adopt this scheme.

A ***nameserver*** mechanism allows the nicety of character strings for addresses without sacrificing the advantage of number matching for decoding addresses. The nameserver consists of a file and a program on a node with mass storage that associates an ASCII (American Standard Code for Information Interchange) string with an address number. The nameserver accepts requests for registration, de-registration, and name queries.

Special ***generic addresses*** also can be set aside for special purposes. For instance, the nameserver could be assigned a generic address to be used by all nameserver-related messages, making it un-necessary to know which node the nameserver is actually on.

Another generic address could be set aside to represent a ***broadcast*** message -- one that all nodes on the network would want to receive. A typical use of a broadcast message is sending a company-wide memo to all employees on the network. The generic address eliminates the need to address the same memo to each person on the network.

Special types of nodes, such as mass-storage nodes or printers, can have their own addresses. For example, the address M might be reserved for the printer node. If there is only one printer on your network, M would mean that printer. If there is more than one printer on the network, an additional field called

the logical printer number could be used to specify the printer for which the message is destined.

Message formats

A message is a pre-determined sequence of fields by which two nodes communicate. A message normally consists of several parts: the header, the body, and some kind of error-checking mechanism, such as checksum, at the end.

The structure allows for much variation. The basic component for constructing a message usually is a byte. A field is defined as one or more bytes that designate a particular section of a message. Typical fields in a message are shown in Figure 6, and explained below.

Figure 6. Proposed message format

SOT: Start Of Transmission
This byte is useful for informing all receivers that the beginning of a message is now on the network, and that the next byte will be the address byte. Obviously, the byte must not be confused with bytes in the middle of a message.

To Address:
The address of the intended receiver.

From Address:
The address of the node that transmitted the message. As will be shown later, this field is important for sending acknowledgments back to the transmitter.

Message Number:
A unique number that distinguishes one message from the next. The usefulness of this field will be illustrated in the sections of this article dealing with duplicate messages.

Bytecount:
Tells a receiver how many bytes to expect in the message body. It can be used as a receive loop counter, to be decremented each time a byte is received. When the counter equals zero, the user knows the checksum byte will follow immediately.

Message ID:
Distinguishes three types of messages within a network system. The ***data message*** contains the essential information to be transmitted from one node to another. The ***message acknowledgment*** acknowledges a data message, and the third type of message, ACKACK, acknowledges a message acknowledgment.

Data:
Zero or more bytes of information that follow the Message ID.

Checksum:
The error-checking byte, computed as the n-bit sum of all the bytes in the message (except the SOT byte and the checksum itself). The transmitter sums up all the bytes in its transmitted message and "ships out" the lower n bits of that sum as the last byte of the message. Meanwhile, the receiver does the analogous operation on the message it receives. If all the characters were received correctly, the receiver's lower n-bit sum should match the transmitter's checksum.

Layering the tasks

The network software can be broken up into three separate layers for implementation (see Figure 7). These layers are the basic transmitter and receiver subroutines, the protocol layer, and the application program. Breaking up the network software in this manner is useful, because it allows the implementer to concentrate on a subset of network functions, without having to give much consideration to the rest of the functions. As an added benefit, the layered structure limits the software modifications needed in order to bring up networking capability for particular network tasks and particular machines.

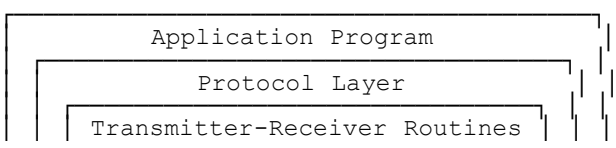

Figure 7. Network protocol is based on the layer concept

As an example, let's say network software is to be brought up on two of the same type of microcomputers, each having a different serial interface. Subroutines in the transmitter/receiver layer that specifically deal with the serial interface are the only parts of the network software that need changing. On the other end, a printer-application program and a disk-write program should be able to use the same protocol layer and transmitter/receiver layer.

The transmitter

A buffer and a byte count are the necessary parameters this routine needs from the protocol layer. The transmitter should neither know nor care what type of message is in the buffer. First, the transmitter will need to know if anyone else is currently using the network. In an interrupt environment, this can be determined by a flag set when a character is received, and reset when a carrier-detect interrupt occurs. If the flag is reset, therefore, it shows that the network is not in use.

If the transmitter is to be implemented without the aid of interrupts, it will be necessary to wait the length of time needed to receive one character (based on the data-transfer rate). If no characters are received in this time, it is assumed no one is in the middle of transmission.

Once it has been determined the network is not busy, the transmitter must send out the SOT field. A potential "race" problem resulting in a collision could occur at this point, since two transmitters could conceivably start this transmission simultaneously.

Because the network is set up so that transmitters receive what they transmit, the received character should always be compared to the character that was just transmitted. If the two characters do not match, a collision has occurred. Later, we will decide how to recover from such a collision.

Assuming the transmitter received what it transmitted, it continues to send out bytes until all, including the checksum, have been sent. If the transmitter is interrupt-driven, it may want to set a flag to inform the protocol layer that transmission was successful. For a transmitter running without interrupts, this information could be returned as a parameter to the routine that called the transmitter.

The receiver

A receiver activated by interrupts will be able to synchronize with the beginning of a message by the carrier-detect interrupt that occurs after the end of any message. Receivers without interrupts or latched carrier-detect pulses must repeatedly wait until a whole character time has gone by without receiving anything. The next field to be received should be the SOT field. If it is not, it will be necessary to go back to the previous step until an SOT is detected.

Once the SOT is detected, the next field should be the Destination Address. When this field is received, it should be compared with the receiver's own address to determine whether the message is intended for this receiver. If your network supports broadcast messages, all receivers must check to see if the message is a broadcast message. Additionally, printer and disk storage nodes must also check to see if the destination address is their generic address. If no address match exists, the receiver should go back to hunting for an SOT field (unless this receiver is a gossip monger).

If the message is addressed to a particular receiver, the address and all subsequent bytes should be received and summed together for comparison with the checksum byte at the end of the message. If your particular network uses parity, the message should also be checked for each character received. The receiver should not care what type of message was received; it should simply inform the protocol layer of receipt. With an interrupt-driven receiver, a flag can be set at completion to inform the protocol layer. Additional information, such as whether any errors occurred during the message, could also be communicated to the protocol layer via common memory. If the receiver is not interrupt-driven, this information can be passed back as parameters to the protocol layer.

The protocol layer

For the following discussion, ***source*** will be defined as the node that transmitted the original message, and ***destination*** as the node to which the

message was addressed.

When computer A sends a message to computer B, there is no guarantee that computer B will receive it. Many things could go wrong. There might be a loose connection somewhere. Computer B might not be running, or it might not be listening to the network. Computer C could start transmitting at the same time as computer A.

Protocol schemes detect and correct such situations. Protocol is basically a conversation between a source and a destination, trying to ensure that what the source transmitted was actually received by the destination.

The simplest protocol is one in which the source sends a message to a specific destination and assumes the message arrived. If your network is in good working order and you know that a particular destination is running properly, this protocol will be sufficient most of the time. You probably would want to use this protocol, for example, when you are sending messages to your friend Carol, who is using computer B. If she is there, she will probably send a message back, thereby acknowledging that she received your message. You would also use this protocol for broadcast messages, to prevent the network from getting jammed by everyone trying to send acknowledgments at the same time.

When you are doing things on your network, such as writing a file to a disk, assuming the file got there is not enough. You need some real acknowledgment that the file got to the disk. If no acknowledgment comes back from the destination, or if the destination returns to the source an acknowledgment stating that the disk is full, the source will have to take some error-recovery measures. These are discussed later.

What happens if the destination receives a correct message and sends back an acknowledgment that is not received by the source? In this case, the source thinks its original message did not get through, but it actually did. To avoid this situation, an acknowledgment of an acknowledgment received (ACKACK) can be added to the protocol. If, after sending an acknowledgment, the destination does not receive the ACKACK, it will have to take some kind of error-recovery action.

What happens if the source receives the acknowledgment and sends the ACKACK, but the destination does not receive the ACKACK? Somebody has got to have the last word, and there can be no guarantee that a message and all its associated protocol are transmitted and received successfully. Especially on a low-speed network, the criterion for deciding how much protocol to use is "as little as possible for a particular application". An intelligent system might provide all three types of protocol (i.e., message, message-ACK, and message-ACK-ACKACK) and allow the application program to decide which one to use.

Error recovery

What should be done when a message was sent and no acknowledgment came back? Or when an acknowledgment was sent but no ACKACK came back? Both these cases call for a timing mechanism. A source that transmitted something and is expecting a reply from the destination must wait a certain amount of time for that reply to come back. If the reply does not come back within that time, it will be assumed an error condition exists.

How long should this time be? There is no way to guarantee that a destination really did receive the message and will transmit an acknowledgment within the time the source has set. The waiting time, then, should be more than long enough to cover any reasonable situation.

Once the source has waited a set amount of time without receiving a reply, a reasonable action would be to retransmit the original message at least once more, and again wait the specified amount of time for a reply. The same strategy could be used by the destination when it sends acknowledgments and waits for an ACKACK. If you are doing your network without the aid of a hardware timer, you will need a time-counting subroutine that continually checks to see if a reply was received, and decrements the counter. If the counter reaches 0 before a reply is received, then a ***timeout error*** exists. If your software has access to a hardware timer, you can use it to set an interrupt.

If no reply is received after repeated attempts to transmit a message, there is nothing to do but give up and report the problem to the program that initiated the network call.

This retransmission scheme introduces another problem. Suppose the source sends a message that is received by the destination, but the destination sends back an acknowledgment that is never received by the source. After timing out, therefore, the source retransmits the original message, and the destination receives it a second time. The Message Number field, along with the From Address field, can be used to correct such situations.

All receivers should keep a list of the last n messages received. The list

need contain only the message number and the From Address. When a new message is received, the list should be examined for a match. If a duplicate is detected, the message should be "dumped", but the appropriate response should be sent back to the transmitter of the duplicate message. If the duplicate was an original message, an acknowledgment should be sent back or, if the duplicate message was an acknowledgment, an ACKACK should be sent back.

Collisions are another issue. Assuming that all transmitters check the state of the network before starting transmission, collisions can happen only when two or more transmitters start their transmissions within one character time of each other. When collisions happen, all transmitters involved should immediately stop transmitting and allow the network to return to the "not busy" condition.

Now, some kind of mechanism is needed to tell colliding transmitters when they can start transmitting again. If they all wait an equal amount of time, they will collide again. Therefore, they must all wait different lengths of time.

One way to ensure this setup is to establish a priority order based on node address. If a node with the address of 1 collides with a node with the address of 3, then node 1 will wait one unit of time before attempting retransmission, while node 3 will wait three units of time. One problem with this scheme is that, under heavy load conditions where collisions are more frequent, nodes with high address numbers may never be able to get a message through, because they must wait so long after each collision.

A fairer scheme would be one in which each node has a random-number generator guaranteed to create a unique sequence of random numbers. All nodes would then have equal priority in retransmissions after collisions.

A typical application program

As an example of a typical application program, let us consider a request to a filing system on a hard-disk node.

The "save" request would first want to send to the filing system a message containing the file name and the number of sectors to be saved. The request probably would ask the protocol layer to expect an acknowledgment, and allow the protocol layer to take care of retransmissions, if necessary. Along with the acknowledgment would come information from the filing system indicating whether or not the request can be accommodated. It it cannot be accommodated, the request program must report the failure to its caller.

If the request can be accommodated, the save request program must break up the file to be saved into convenient blocks (probably a disk sector). When errors occur during transmission, it is more economical to retransmit small blocks than large ones. In either case, the save request should send an ACKACK to the filing system to say it agrees to what the filing system considers the state of the request.

Once the file has been partitioned into blocks, the save request should hand them in sequence to its protocol layer for transmission to the filing system. The request should ask its protocol layer to expect an acknowledgment for each block transmitted. Each block should have a unique number that can be checked by the filing system against block numbers already received. In this manner, duplicate blocks can be dumped.

By the value of the last block number, both parties know when the file transfer is completed. If implementation is done in a straightforward manner, the last block number should equal the corresponding field in the original request message.

The save request should ask the protocol layer to send an ACKACK to the filing system when it submits the last block for transfer. Upon receipt of this ACKACK, the filing system can be sure it will not be getting a retransmission of the last block, and it can close the file and forget about the request.

When extended conversations are taking place between two nodes on the network (as in the previous file transfer examples), the network can be made to appear constantly busy by never allowing more than a character time to elapse between messages. In this way, no other user on the network can interfere with the conversation.

If the data rate is controlled by software on the two conversing nodes, you might consider increasing the rate after the initial conversational link has been established. The rate could be increased beyond what is normally acceptable to every node on the network, but it must be changed back after the conversation is completed. While the process is going on, every other node on the network should recognize it as a network-error condition. Because the nodes have not seen a transition from a busy network to a non-busy network, they will not be looking for an SOT field anyway. This scheme can get a little tricky when attempting to end a conversation, especially if the last acknowledgment or ACKACK did not get through but the data rate on one node has

already been reduced to its former value.

Multitasking environments

Networking in multitasking environments raises many issues that cannot be considered here, but a few obvious ones should be pointed out.

The protocol layer probably should be set up as a process by introducing another parameter to indicate whether the application program will "go to sleep" waiting for a reply or acknowledgment. The protocol layer would then have to give the application program a "wake up" by indicating whether the message got through to the receiving process.

Since messages could in this way be addressed to one of several processes on a node, the address fields for To and From addresses would need to be extended to include a Process ID number.

The software design presented in this article reflects only one of many possibilities.

Now that you have a taste of what networking is all about, you can experiment and enjoy implementing your own ULCnet.

EOF