Embedded Netsock™

An Introduction

MICRO/SYS, INC.
3730 Park Place
Montrose, CA 91020
Phone (818) 244-4600
FAX: (818) 244-4246
www.embeddedsys.com

DOC 1238

1/1/99

Micro/sys Technical Support

Micro/sys offers the best technical support in the business – and it's free!

Our application engineers are ready to assist you in getting your Embedded Netsock project up and running as quickly as possible. You can contact us as follows:

Micro/sys Technical Support
Phone: (818) 244-4600
FAX: (818) 244-4246
Email: techsupport@embeddedsys.com
Web: www.embeddedsys.com

We can also upload and download programs by modem whenever that will assist you.

Thanks for specifying Micro/sys products. We'll be glad to be a part of your team as you use our products.

DOC1238

Embedded Netsock™ - An Introduction

Introduction

As networking becomes an important requirement for more and more embedded systems, a basic understanding of network technologies is becoming more and more important during the definition phase of embedded system design.

To that end, this document provides an overview of networking. However, the overview is slanted towards those aspects of networking that most affect embedded system design. The traditional 'file and printer sharing' provided by the network that you probably *use* in your design work is not necessarily applicable to the network requirements of the embedded systems that you *design*. There are networks and there are networks.

The Micro/sys Embedded Netsock™ technology specifically addresses and implements the philosophy embodied in the following description of networking.

Issues such as routing and fragmentation may be trivialized, understated, or ignored in this overview. This is done with the goal of simplifying the networking world.

If your system will need more rigorous features and capabilities of some of the networking protocols discussed here, please refer to one of the many excellent books on the subject.

Networking basics - with an embedded systems slant

Computer networking has become as important as the computer itself. Computers cannot be islands unto themselves, and networking has become the primary avenue through which data is shared between computers.

Traditional computer networks concentrate on a workstation using network resources, such as a network disk drive or printer. The network is mostly transparent to the application running on the workstation. A word processor does not know if the file is stored on its local hard drive, or on a network server's hard drive. It doesn't matter. Similarly, the application does not care if the selected printer is connected to a local port, or is on the network.

This transparency to the application software is so complete that the network driver on the workstation is often called a 'redirector.' The redirector driver merely redirects disk or printer access to the network server. The application has no idea that this is going on.

However, things are different (as always!) with embedded systems. When a network is involved, it often means that applications on more than one computer need to work in tight coordination. They cannot be unaware of each other.

For instance, if an embedded computer is inside a conveyor belt controller, and it must be monitored and controlled by a supervisory Windows NT computer, both application programs will need to know about the other. The data to be transferred is not merely a disk file or a printer file, it is customized information that only has meaning within the context of conveyor belt monitoring and control.

> *Consideration #1: In embedded networks, the typical case is a custom program on both ends of the network cable. Shrink-wrapped software is probably not applicable to either side.*

There are so many technical issues involved with interprocessor communication over a network cable that the probable loss of ready-to-run software as an option can be daunting. Programming to the lowest level network adapter software driver requires extensive expertise in computer networks, and is very difficult to debug.

> *Consideration #2: Embedded systems should try to 'borrow' as much networking as possible from traditional computers in order to reduce development time. Careful decisions at this point can significantly reduce development time.*

One networking technology that demands to be considered is the TCP/IP protocol suite that defines hundreds of solutions to hundreds of networking issues. Originally developed with government funds, these TCP/IP solutions are in the public domain. Their full specifications are openly available and can be used free-of-charge. And because TCP/IP protocols are the foundation upon which the Internet is built, their continued growth and support are assured.

The TCP/IP protocols, however, are fairly low-level. Programming directly to them requires large amounts of programming to fill in headers for transmit packets, and to decode packet headers when they are received. More levels of technology need to be 'borrowed' from somewhere else for use in embedded systems.

To address the challenges associated with custom programs on multiple computers transferring data with TCP/IP protocols, the *sockets* model was created. Again, because this model was created with government funds at the University of California, Berkeley, it is well documented and available for use at no charge.

> *Consideration #3: Public domain, well documented, well supported networking technologies are available. Their use provides extensive benefits to embedded systems.*

TCP/IP and sockets standards are extremely general and extremely flexible. This is both a curse and a blessing. On the plus side, their flexibility and extensibility have fueled an explosive, global growth of networking - the Internet today vs. the Internet 5 years ago.

But the down side is that, being all things to all people, there is a bewildering alphabet soup of acronyms and concepts you must assimilate. It takes a long time to learn 250 different protocols, procedures, acronyms, and architectures - just so you can pick the 25 that make the most sense to your embedded system needs. 90% of what you learn is just to be able to eliminate that same 90% from your consideration.

> *Consideration #4: Simplify. Look for a pre-selected subset of TCP/IP and sockets technologies that targets embedded systems specifically.*

Embedded Netsock from Micro/sys, based on careful analysis of just what an embedded system typically needs, creates a limited view of networks. But it is a view that is relevant to the task at hand - transferring data between embedded systems and other computers.

Issues of 'academic purity' or 'universality' are purposely downplayed with Embedded Netsock. We have let the needs of the embedded system world override the needs of general, global computing.

TCP/IP basics

Some Terms Defined

Transport Control Protocol/Internet Protocol (TCP/IP) is a catchall nickname given to a large set of protocol specifications. More accurately, it might be called

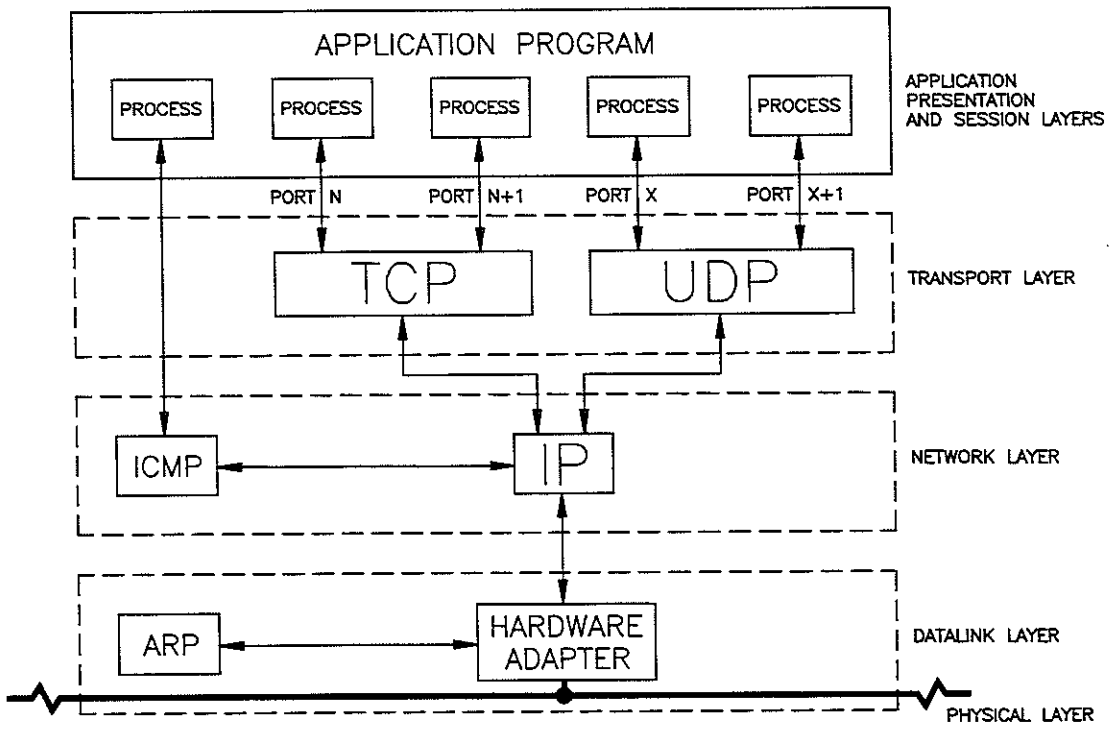
TCP/IP/ARP/RARP/ICMP/IGMP/UDP/BOOTP/DHCP/FTP/HTTP

The point here is that TCP and IP are only two of the many protocols that exist within this smorgasbord of specifications. When you say 'TCP/IP', you are really saying what has become the accepted nickname for 'the protocol suite that includes TCP and IP, among many others'.

TCP/IP is based on the concept of layers - each layer performing a specific, limited part of the networking job. The international standard model (ISO) defines seven layers, whereas TCP/IP has a slightly different break arrangement.

Refer to Figure 1 - ISO Layers and TCP/IP, throughout the following discussion.

Figure 1 – ISO Layers and TCP/IP



The *physical* layer, for example, is electrical pulses on copper wire, or light pulses on fiber optics. It is concerned with bits in motion. How do you define a 1 and a 0?

The *datalink* layer worries about what sequences of bits mean at the lowest level. Both sender and receiver must agree, for example, what the first 48 bits stand for. Otherwise, there is no basis for meaningful communication. How is a data packet defined? Ethernet is an example of a datalink layer protocol.

The *network* layer worries about how you direct a message to one computer versus another. How do you assign names or numbers to computers so that each one is unique, and each one can be the intended target of a message? IP is an example of a network layer protocol.

The *transport* layer worries about the likely occurrence that a message will be received by a specific computer (thank you, network layer!), but there may be a number of possible final destinations for it within that machine. A good example is a multitasking system where 10 applications are running at the same time on the same computer. When a packet arrives off of the network, which application should it be given to? TCP and UDP are examples of transport layer protocols.

The *session* layer worries about establishing a presence on the network for a period of time. Starting up, resolving assigned addresses, and running network-based programs.

Computers, Hosts, and Addressing

In TCP/IP networks, each attached computer is called a *host*. While this term may not be terribly descriptive, it is well entrenched, and will continue to be seen in many places. For example, in Visual Basic, the "RemoteHost" property is where you specify the number of the computer you want to send a message to.

To send a network message to a specific host, some form of network address is needed to uniquely identify the host. At the lowest level - in the hardware network adapter circuitry - there needs to be a unique address.

In the case of Ethernet, a 48-bit binary number is assigned to each Ethernet adapter manufactured. The first 24 bits indicate the manufacturer, and the second 24 bits indicate the sequence number of the specific adapter. Standard notation for an Ethernet address is six two-digit hexadecimal numbers separated with dashes. An Ethernet adapter manufactured by Micro/sys, for instance, might be

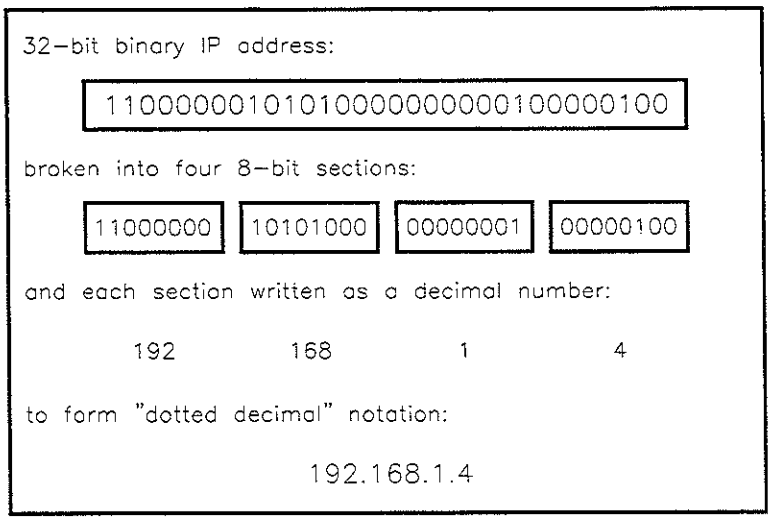
00-60-92-00-C2-9A

The problem with physical addresses is that they are difficult to administer at the network level. If a person's desktop computer is only known by its Ethernet address, every other computer that wants to communicate with it will have to be informed when its Ethernet adapter card is replaced.

This situation is overcome with the concept of a *logical* address - a way of assigning a unique number to a host without the need for it to be in hardware. Then a modifiable cross-reference list can be used to convert logical addresses into physical addresses.

The Internet Protocol (IP) is a network layer specification that provides a unique logical numbering system used for all hosts. IP addresses are a 32-bit binary number. Since nobody in their right mind would propose memorizing 32-bit binary numbers as 1's and 0's, a shorthand notation was developed. This shorthand is the *dotted decimal* notation. The 32-bit binary IP address is broken into four 8-bit values. The four values are written as decimal numbers, and are separated with decimal points.

Figure 2 – Dotted Decimal Notation



Ideally, every host (computer) in the world would have one of the 2^{32} possible IP addresses. For this reason, any host that is to be connected to the Internet must obtain an IP address that is, ultimately, coordinated by InterNIC, the world-wide administrator of all IP addresses that will be connected to the Internet. There cannot be two hosts anywhere in the world with the same IP address if they are to be connected to the Internet.

Because an IP address is a logical address, an IP address is simply assigned to a particular machine. There is no hardware in the host that is built to respond to this address. An IP address can be moved from one machine to another. In fact, many servers hold a 'pool' of IP addresses that they can 'lease' temporarily to

any host that requests it. For instance, dialing into an Internet Service Provider (ISP) will often cause the ISP to assign an IP address to the caller. This IP address is only valid during this specific connection. When the modem hangs up, the IP address will be available for assignment to the next caller.

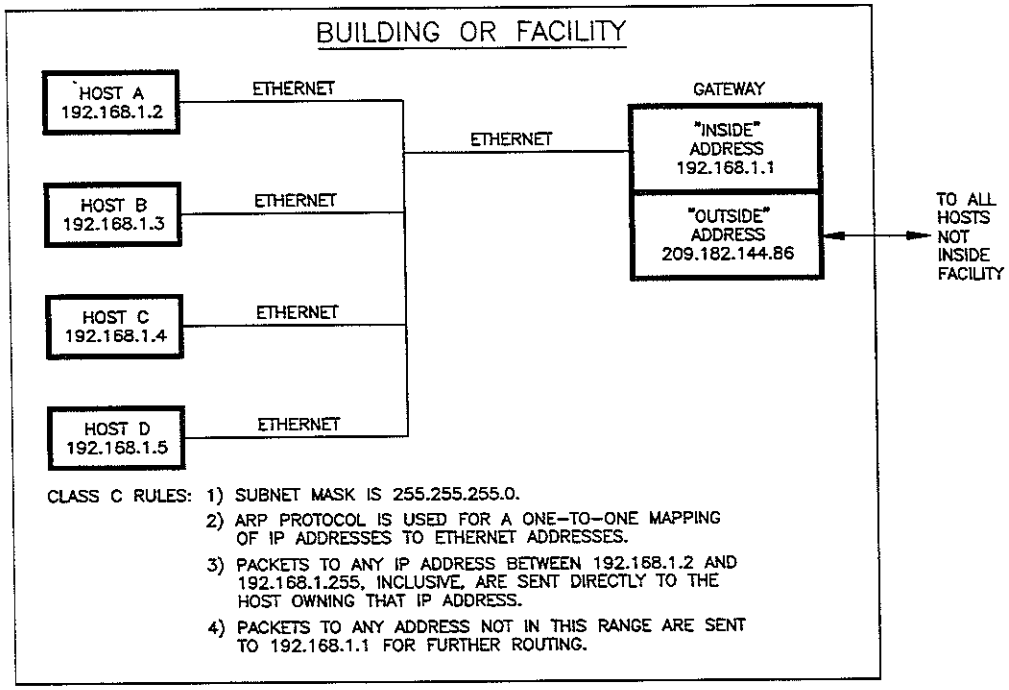
When an Ethernet adapter is replaced in a desktop computer, the physical address changes, but the logical IP address can remain the same. Therefore, no administrative changes need to be made concerning other hosts trying to reach this host. It is left up to the TCP/IP protocol stack to update the cross-reference listings at run-time (using the ARP protocol that will be discussed shortly).

Routing, Subnets, and Subnet Masks

The limited case of a simple TCP/IP network using Ethernet can be used to illustrate routing and subnets. Imagine a facility where all hosts inside the facility are interconnected on Ethernet wiring. In addition, a single *gateway* to the outside networking world is attached to the same Ethernet wiring.

The concept of *routing* involves, in addition to other issues, determining where to send a particular Ethernet packet. If the destination host is inside the facility, it is on the same Ethernet cable as the sender. If the destination host is not inside the facility, it must be sent to the gateway, which will then retransmit it to some external system for relaying to its final destination.

Figure 3 – Class Subnet Example



Defining *subnets*, based on ranges of IP addresses, provides an easy way of determining whether an intended IP address is on the same wiring or not. For example, if all internal network hosts are assigned IP address from 192.168.1.1 through 192.168.1.254 (i.e. only the last number changes), network software can easily determine where to send a packet. If the first three numbers of the destination IP address are 192.168.1, the destination host is inside the facility, and the packet can be sent directly over the Ethernet wiring. If the first three numbers are not 192.168.1, the destination host is not inside the facility, and the packet must be sent to the gateway for forwarding to its ultimate destination.

In the example above, hosts are determined to be on the same subnet if their first three IP numbers are the same. This is technically called a Class C subnet, and it provides IP addresses for 254 hosts on one subnet (0 and 255 are special cases, and are not available).

A *subnet mask* is a value that tells the networking software how many of the four IP numbers are needed to determine if two hosts are on the same subnet. For a Class C subnet, the subnet mask, in dotted decimal notation, is 255.255.255.0. As can be seen, this tells the networking software that the first three IP numbers hold significance.

Class B subnets have a subnet mask of 255.255.0.0, indicating that only the first two IP numbers must match for two hosts to be on the same subnet. A Class B subnet can obviously have many more hosts than a Class C subnet. Finally, Class A subnets have a subnet mask of 255.0.0.0, which provides the largest number of hosts on the same subnet.

Most embedded networks will probably use Class C subnets. If not connected to the Internet, the IP range that we have been using so far, 192.168.1.1 through 192.168.1.254, is the preferred IP range to use for a Class C network. This special "testing" IP subnet is known to be localized, and is safe to use.

If an outbound message is to a host whose IP address, according to the subnet mask, is not on the same subnet as the sending host, that message will need to be sent to a gateway. The gateway will have one IP address that is on the same subnet, and another IP address out the other end that will attach to other systems for forwarding packets to their ultimate destinations. What happens on the other side of a gateway is beyond the scope of this introduction.

Logical-to-Physical Address Mapping - the ARP Protocol

Once a packet is determined to be on the same subnet as its intended destination, it is merely sent to the Ethernet (physical) address that corresponds to the IP (logical) address that the packet is destined for. This requires a logical-to-physical lookup table.

Such a table is built with the Address Resolution Protocol (ARP). An ARP table is dynamic, and is kept in RAM within a host.

Figure 4 - Address Resolution Protocol

1. *An ARP cache is created in RAM, and initialized to an empty state.*

2. *Each time an outbound IP packet is ready to be sent, the ARP cache is checked to see if the target IP address has a corresponding Ethernet address entry.*

3a. *If so, the Ethernet address is added to the packet, and it is sent directly to the host, which is on the same subnet.*

3b. *If there is no entry for this IP address, a broadcast message is transmitted that will be received by all hosts on this subnet. In effect, this ARP request says "does anybody know the Ethernet address associated with this IP address?"*

4. *Most often, the host who owns the IP address in question will respond to this ARP request by examining the received packet, and sending it back to the originator after entering its Ethernet address in the proper location in the reply packet.*

5. *The requesting host receives the ARP reply, extracts the new IP-to-Ethernet mapping, creates a new ARP cache entry, and sends the outbound packet to the newly discovered Ethernet address.*

6. *Future packets to this IP address will use the ARP cache entry to quickly determine the Ethernet address of the destination.*

Because ARP is a dynamic protocol, the ARP cache is built at run time. If an Ethernet adapter in a host is changed, a new IP-to-Ethernet mapping is needed. To allow for this, most ARP cache implementations cause cache entries to be purged after a reasonable amount of time has passed. This will automatically cause new mappings to be picked up as needed.

Transport Protocols - TCP and UDP

While the IP protocol handles logical addressing - getting packets to the right host - other protocols carry the actual messages. These protocols have their

own formats. A completed packet is passed to the IP layer, which adds IP addressing in an IP header, and sends the packet out through lower level layers. The process is reversed for receive.

The two major transport protocols are Transport Control Protocol (TCP) and User Datagram Protocol (UDP). Both of these protocols add another level of addressing - called a *port* - to the logical addressing provided by the IP protocol. Therefore, the destination of either a TCP or a UDP packet requires that two elements be specified: the IP address of the host, and the port within that host that the packet should be delivered to.

Remember that our definition of the transport layer involved the issue of determining which application a particular packet should be delivered to once it has arrived at the correct host. The port part of the address accomplishes this.

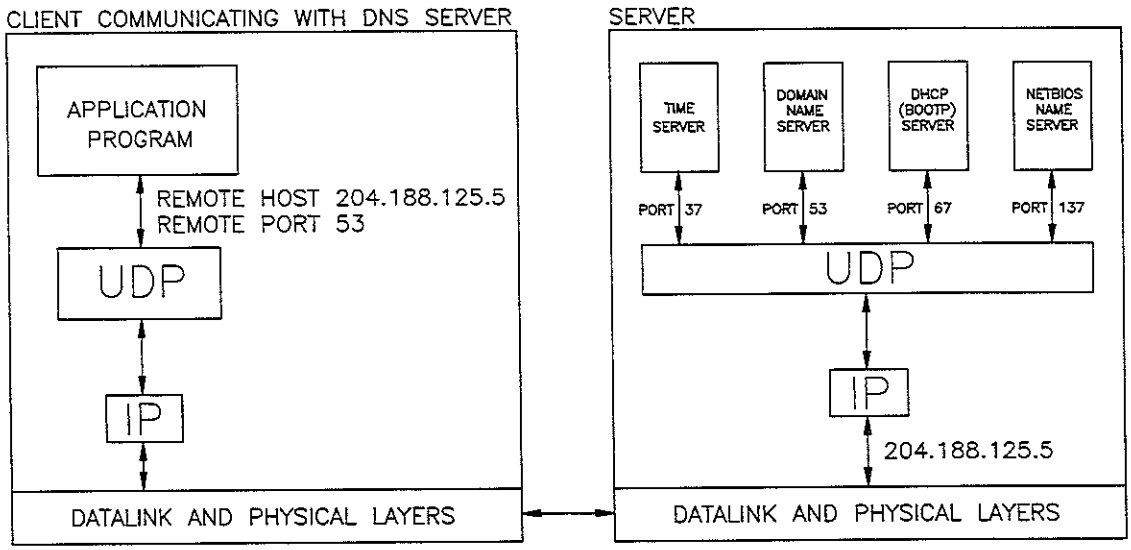
UDP is the simpler of the two transport protocols. It is open-ended: a packet is sent from one host to the other, and that's it. There is no acknowledgement back to the sender that the packet ever arrived at its destination. UDP packets are called *datagrams*. They are like postcards you drop into the mail. They are one-way, and you must take further action if you want to know if they arrived. Technically, UDP is classified as a *connectionless* protocol.

By riding inside IP packets, UDP packets inherit all of the logical addressing, subnet issues, and logical-to-physical mapping features previously discussed. UDP is great for embedded systems sending informational packets easily and quickly from one host to another.

The TCP protocol is another animal. It is classified as a *connection-oriented* or *stream* protocol. TCP is more analogous to a telephone conversation between two people. It is bi-directional, real-time, and it allows a stream of data to be passed from one location to the other. In embedded systems thinking, it is more like an RS232 cable between two stations. However, it is a *virtual* circuit, using existing Ethernet cabling to simulate a direct, private conversation between the two hosts.

In addition, TCP is classified as a *reliable* protocol, while UDP is not. When you send a TCP packet, the protocol makes sure that it is received. Acknowledgements are required, and lost packets are resent. TCP is far more complex, and takes larger protocol drivers and more configurations than UDP.

Figure 5 – Transport Layer: UDP Ports



Interestingly, TCP, a reliable protocol, is built on top of IP, which is an unreliable protocol. Thus, you can add reliability to an unreliable protocol by implementing additional actions. This can be important in embedded systems.

Naming Hosts, and Finding IP Addresses

Under TCP/IP, the IP address of a host must be known in order to contact it. But 192.168.1.38 is tough to remember. So a number of "user-friendly" host naming systems have been developed over the years. The two that remain important are the Domain Name Server (DNS) scheme of the Internet, and Microsoft's Windows Internet Naming Service (WINS).

DNS implements a hierarchical naming system. It is the form of Internet names, such as www.embeddedsys.com (Micro/sys' web site). Every host on the Internet has a unique *fully qualified domain name* (FQDN) under DNS.

DNS is a worldwide, distributed, FQDN-to-IP-address lookup system. A hierarchical set of DNS servers breaks down all possible domain names into *zones*, and DNS servers share their databases with each other. DNS is very demanding in terms of administration. There are many complex record types, and detailed pointers are needed to keep everything working.

From earlier desktop networking systems, there is a legacy of NetBIOS names. NetBIOS is a flat naming system - a name can only be used once. It does not scale to global networks. NetBIOS names for a host are 15 ASCII characters long.

Because of the number of NetBIOS hosts installed, Microsoft devised the WINS system for NetBIOS-name-to-IP-address lookup system. It works well for limited networks, much like those of embedded systems. WINS requires less administration than DNS.

Dynamic Host Configuration Protocol (DHCP)

To centralize network administration, Windows NT Server, and some other systems, include Dynamic Host Configuration Protocol (DHCP) servers. DHCP accomplishes just what its name implies - it automatically configures various network hosts (computers) as they come online.

With DHCP protocols, a host can power up, and download virtually all necessary network configuration from a DHCP server. For instance, the hosts' assigned IP address, the associated subnet mask, the IP address of a name server (either DNS or WINS), and other network configuration items can be requested. The DHCP server will manage many network configurations that would normally require the attention of an administrator.

Configuring embedded systems with the services of a DHCP server significantly simplifies the setup and configuration requirements of the network system.

Other Protocols - ICMP, FTP, HTTP, etc.

The TCP/IP protocol suite includes many additional protocols. Internet Control Message Protocol (ICMP) deals with reporting and addressing various network errors and reconfiguration needs. Most people know of it for the *echo* command, also known as *ping*. This is one of the most important diagnostic utilities within the TCP/IP suite.

File Transfer Protocol (FTP) deals with moving files from one host to another. Hypertext Transfer Protocol (HTTP) is the basis for the World Wide Web. These protocols are extremely operator intensive. While they have some applicability to embedded systems, these complex protocols may be overkill in most situations. Less complex protocols may be better choices.

The Winsock API

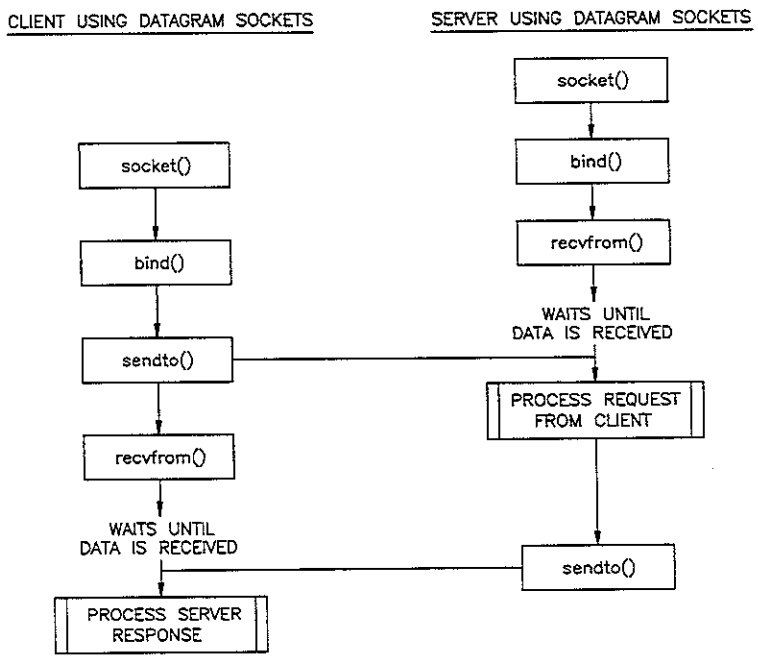
Originally, UC Berkeley researchers tried to fit network-based inter-processor communication into the UNIX file I/O model. It just wouldn't fit. So they developed the sockets model.

Under sockets, you create an *endpoint*, which is the IP address and port associated with a particular network attachment point for a process (like an Ethernet adapter and the application assigned a specific processing task behind the adapter). With a socket created on two different hosts, packets can be sent from one socket to the other, and therefore, from one process to the other. Underlying network details are mercifully hidden from the programmer.

The sockets model was widely embraced by Windows programmers - with one major problem. Windows and UNIX are very, very different in internal operation. Therefore, industry-wide participation in creating a Windows version of sockets was aggressively undertaken. The result is the Winsock Application Programming Interface (API).

Winsock is fairly simple to use. You *startup* the network with one call, create a *socket* with another call, then *bind* the local IP address and a specific port number to that socket. Thereafter, connection-oriented protocols like TCP *send* and *recv* packets, while connectionless protocols like UDP *sendto* and *recvfrom*.

Figure 6 – Datagram Sockets (SOCK_DGRAM) Operation



At the time of creation, connection-oriented sockets are given the type SOCK_STREAM and use TCP, while connectionless sockets are given the type SOCK_DGRAM and use UDP.

A Winsock Quirk - Network Order vs. Host Order

One significant difference between computers in the UNIX world, and computers derived from the PC architecture is byte ordering. Byte ordering involves the order in which multi-byte values are stored. Most computers that UNIX was originally developed on store bytes in memory from high byte to low byte. PC architectures store bytes in memory from low byte to high byte. IP addresses are 4-byte values; port numbers and protocol specifiers are 2-byte values.

Where this causes trouble is when a multi-byte value is sent out on the network. TCP/IP protocols were developed on UNIX machines, and therefore, high bytes were naturally sent first by the hardware. Therefore, multi-byte values on the network are sent high byte first. This is now called *network order.*

Inside the host, the storage order is call *host order.* On many UNIX machines, network order and host order are identical. But on PC architecture, network order and host order are opposite.

Take the example of a port number to send a UDP packet to. Say the desired port is 5001 decimal. This is equivalent to 1389 hex. In PC architecture, this is stored, by the hardware, with 89 in the first byte and 13 in the second byte. Sending this to network hardware will cause the 89 to be transmitted first, followed by the 13. But the destination host is looking for port 5001 in network order, which is 13 followed by 89. Without conversion, the packet will not arrive at its intended destination.

Under Winsock programming, great care must be taken to remember which order a value you are using is currently in. A number of byte-order conversion functions are always provided with Winsock implementations.

Limiting the TCP/IP - Winsock Universe

With the foregoing, abbreviated discussion of TCP/IP and sockets, it's time to start limiting the networking universe to the core of what's needed for embedded systems.

For instance, in an academic, purist world, absolute separation between the TCP/IP layers can be argued. Any datalink protocol should be able to work on top of any physical protocol; any transport control should be able to work on top of any network protocol.

On the surface this all sounds nice; But this means that almost nothing can be 'known' about the adjacent layer. There must be a continual passing of information about, for instance, how many bytes are needed for a network layer address. You may never change it, but, theoretically, it must be able to change. So you continually pass around the unchanging information about the size of the addressing protocol you have chosen.

And when a packet is received from the network, it must be passed from one layer to the next, to the next, to the next, until it is finally delivered to its intended destination. This could result in a message being copied from one memory buffer to another five times before anyone looks at the data to take action on it. This can steal precious processing bandwidth from an embedded system's processor.

So Embedded Netsock is more concerned with efficiency than with theory. Some of the normal networking choices have been limited. Some of the implementation takes liberties. This keeps the Embedded Netsock system smaller and faster.

Here is the basic networking mindset that Embedded Netsock is based on, and some of the rationale:

1. **The embedded systems will be connected to the same Ethernet subnet.** *Without intervening gateways, modems, phone lines, routers, etc., expected reliability is greatly increased. A 'unreliable' protocol such as UDP is all but guaranteed to be successful, as Ethernet packets on the same subnet are extremely reliable. There are no issues of packet fragmentation, re-assembly, etc. Therefore, Embedded Netsock is targeted for embedded networks within a single facility.*

2. **A Windows 95/98/NT computer will be on the same Ethernet subnet.** *While not mandatory for Embedded Netsock operation, the addition of such a network master provides an excellent means of controlling, monitoring, and administrating the embedded network. Specifically, Windows NT Server provides such a wealth of network capabilities - right out of the box - that it is highly recommended as the supervisory computer in a network of embedded systems. Built-in NT Server services, like the DHCP, WINS, and DNS servers, allow centralized, automatic management of embedded network issues such as IP addresses and name servers.*

3. **Network communication between facilities and between networks, if needed, is better performed between Windows 95/98/NT computers than between embedded systems.** *With a Windows machine collecting and managing all data within a single facility over the Ethernet link, there is an easy way to expand to include other facilities. Merely use the built-in, larger*

scale networking of the Windows machines to network between the facilities. And if a corporate Management Information System (MIS) network wants to have access to a summary (or even gory details) of what is going on in the embedded network, a Windows machine with two network adapters - one for each network - is the ideal link. It can collect, organize, and summarize embedded network data for the MIS network's use.

4. **Connectionless sockets using the UDP protocol provide excellent embedded network capabilities.** *The speed, simplicity, and small code size of UDP, coupled with the reliability due to the assumption of a common Ethernet subnet, make this the ideal embedded network protocol. UDP datagrams can be sent from any embedded system to any other, or to a supervisory Windows 95/98/NT computer. If desired, a small amount of acknowledgement code can be added to the top level application to ensure data reception. This is no different than embedded systems communicating over RS232 or RS485 serial links - just faster and easier. And more reliable due to checksums performed on packets at lower protocol levels.*

5. **A client-server network model, more than a peer-to-peer model, is applicable to most embedded networks.** *While Embedded Netsock supports peer-to-peer networking (you can send a UDP packet to any host whose IP address you know), more than likely it is less important that one piece of machinery talk to another. Very often, though, a centrally located supervisory computer will want to query, or poll, the various embedded systems to receive current status, or to send updated operating parameters. In the client-server model, this makes each embedded system a server: the embedded servers continuously operate the machine in which they are embedded, and respond to queries and updates they receive from the supervisory computer, which is acting as a client. Application programs operating on such embedded servers are straightforward and easy to implement.*

6. **Embedded web servers are not necessarily the answer to embedded networking.** *We worry about a high school student in Des Moines pointing his browser to a kidney dialysis machine at County/USC Hospital, or to a high-speed injection-molding machine in Houston. With the ease of development of client-server applications, thanks to TCP/IP and sockets, custom coding on both ends of the network application is quite feasible, and even preferred. The risks of unauthorized browser access to embedded systems should not be ignored. In addition, when an application requires the use of an off the shelf web browser for control, you never know how or when periodic browser software updates may adversely affect the application. These issues can be addressed by using these other TCP/IP technologies.*

Embedded Netsock Application Development

Embedded Netsock is pre-installed in flash memory on selected Micro/sys embedded PCs. Included are all levels of drivers, from the Winsock API through the hardware Ethernet adapter driver.

Network-based application programs can be written, and then compiled with 16-bit Microsoft or Borland C/C++ compilers, with DOS .EXE files as the target file type. Verified compilers include:

Borland C++ 3.1	Microsoft C/C++ 5.1
Borland C++ 4.5	Microsoft C/C++ 6.0
Borland C++ 5.0	Microsoft C/C++ 7.0
	Microsoft Visual C++ 1.5

By merely including the NETSOCK.H include file in your program, all Embedded Netsock functions are available to the application program. No libraries need to be linked into the application.

Using the standard Micro/sys development process, an Embedded Netsock application can be downloaded into RAM on the embedded PC through a COM port. Borland's Turbo Debugger can then be used to debug the program. (Some Microsoft compilers generate less than complete debug info, and may not offer all debug capabilities available from Borland compilers.)

Alternatively, the application can be downloaded to flash memory, and executed automatically when the embedded PC is powered up or reset.

The Embedded Netsock APIs

Embedded Netsock offers two distinct APIs.

The first is a subset of the Winsock API that is very close to the syntax and operation of standard Winsock calls. This API is ideal for programmers that have experience in Winsock development. This API includes:

bind()
closesocket()
ENgetnetconfig()
getsockopt()
htonl()
htons()
inet_addr()
inet_ntoa()
ioctlsocket()
ntohl()
ntohs()
recvfrom()
sendto()
setsockopt()
socket()
WSACleanup()
WSAGetLastError()
WSASetLastError()
WSAStartup()

Because the Winsock API includes a number of unchanging parameters, and requires care in handling network byte order versus host byte order, Embedded Netsock offers a second API. This alternative API eliminates redundant parameters, and uses only host byte order values. Functions are similar, but simpler than standard Winsock functions.

This second Embedded Netsock API is ideal for first time network programmers who will not be writing standard Winsock programs in addition to Embedded Netsock programs. These programs are easier to understand when source code is read. This alternative API includes:

ENclosesocket()
ENgetnetconfig()
ENgetsockopt()
IPaddress()
IPstring()
ENgetsockrxavail()
ENrecvfrom()
ENsendto()
ENsetsockopt()
ENsocket()
ENCleanup()
ENStartup()

The structure of a typical Embedded Netsock application

A typical Embedded Netsock application program starts up the network then enters a "do forever" loop wherein it handles local process or machine control functions. At the end of the processing loop is a check to see if there are any network messages to be handled. If so, it handles them prior to restarting the main control loop.

The following program skeleton shows this structure, using the standard Winsock API calls.

```
/*  SRVR_PRO.C
 *
 *  Process control system using Embedded Netsock to allow supervisory
 *     computer to set setpoint and monitor conditions over TCP/IP link.
 *
 *  8/5/98
 */

#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <ctype.h>

#define NETSOCK_MASTER
#include "netsock.h"

#define PROCCNTRL_PORT  5002      // the UDP port this server will listen on
#define FLAGS_ZERO 0              // flags parameter for recvfrom() and sendto()


// local prototypes

void DelayUntilNextLoopUpdate(void);
void UpdateLoop(void);
int  StartNetwork(void);
void HandleNetwork(void);
int  MessageReceived(SOCKET s);
int  analogrd(int chan);
void analogwr(int chan, int dta);

// global variables

int setpoint, processvar;
int debug;
SOCKET msgsock;
struct sockaddr_in local, from;
```

```
int main(void)
  {
      int err;

      setpoint = 0;
      processvar = 0;


      printf("\n\r\n\rMicro/sys Embedded Netsock Process Control\r\n\r\n");

      // Initialize Embedded Netsock and start the network

      err = WSAStartup();
      if (err == SOCKET_ERROR)
        {
            printf("Error %d from WSAStartup()\n\r",err);
            WSACleanup();
            return(-1);
        }

      // Create a socket to use for network communications
      msgsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
      if (msgsock == SOCKET_ERROR)
        {
            printf("Error %d from socket()\r\n",msgsock);
            WSACleanup();
            return(-1);
        }

      local.sin_family = AF_INET;
      local.sin_port = htons(PROCCNTRL_PORT);

      // Associate a local address to the socket
      err = bind(msgsock, &local, sizeof(local));
      if (err == SOCKET_ERROR)
            {
                  printf("Error %d from bind()\r\n",err);
                  WSACleanup();
                  return(-1);
            }
      printf("Starting server mode, UDP port %d...\r\n\r\n", PROCCNTRL_PORT);

      for (;;)
       {
            DelayUntilNextLoopUpdate();
            UpdateLoop();
            HandleNetwork();
       }
  }
```

```
//---------------------------------------------------------------
//
//     MessageReceived()
//
//     check for any recevied messages from the supervisory computer,
//     and return 1 if so, 0 if not.
//
//---------------------------------------------------------------

int MessageReceived(SOCKET s)
  {
       unsigned long RXAvail;

       // Check to see if Netsock has received a datagram
       ioctlsocket(s, FIONREAD,(unsigned long *)&RXAvail);
       if (RXAvail)
         return(1);
       else
         return(0);
  }



//---------------------------------------------------------------
//
//     HandleNetwork
//
//     check for any received messages from the supervisory computer,
//     perform any requested action, and return an acknowledgement
//     message.
//
//     It is here that a command set is defined and implemented.
//     In this case, three inbound commands are handled:
//
//           sp     to set new setpoint value
//           pv     to cause current process variable to be returned
//           ^C     to cause the program to exit
//
//---------------------------------------------------------------

void HandleNetwork(void)
  {
       int fromlen,numbytes,commanddone,err;
       char datagram[80];
       char command[10];
       char parameters[10];

       if (MessageReceived(msgsock))
         {

              /* Get the datagram and process it */

              fromlen = sizeof(from);
              numbytes = recvfrom(msgsock, datagram, sizeof(datagram),
                          FLAGS_ZERO, &from,(int far *) &fromlen);

              memcpy(command, datagram, 2);
              command[2] = 0;
              memcpy(parameters, datagram+2, 10);
              commanddone = 0;
```

```
            /*-------------------------------------------------------------*/
            if (stricmp(command, "sp") == 0)
              {
                   sscanf(parameters, "=%d", &setpoint);
                   sprintf(parameters,"%s","");
                   commanddone = 1;
               }

            /*-------------------------------------------------------------*/
            if (stricmp(command, "pv") == 0)
              {
                   sprintf(parameters,"%d",processvar);
                   commanddone = 1;
               }

            /*-------------------------------------------------------------*/
            // return response to host in rcvfrom()'from" structure

            strcpy(datagram, command);        // start with command

            if (commanddone)
              strcat(datagram, parameters);   // append valid command results
            else
              strcat(datagram, "??");         // append invalid cmd response

            // Send the datagram through the socket
            // to the sender of the last message
            err = sendto(msgsock, datagram, strlen(datagram), FLAGS_ZERO,
                                 &from, sizeof(from));

            if (err == SOCKET_ERROR)
              printf("sendto() error.\r\n");

            /*-------------------------------------------------------------*/
            if (stricmp(command, "^C") == 0)
              {
                   exit(0);
              }
            /*-------------------------------------------------------------*/
        }
}
```

Embedded Netsock™

Reference Manual

Release 1.10

MICRO/SYS, INC.
3730 Park Place
Glendale, CA 91020
Phone: (818) 244-4600
FAX: (818) 244-4246
www.embeddedsys.com

DOC 1239

5/1/00

Micro/sys Technical Support

Micro/sys offers the best technical support in the business – and it's free!

Our application engineers are ready to assist you in getting your Embedded Netsock project up and running as quickly as possible. You can contact us as follows:

Micro/sys Technical Support
Phone: (818) 244-4600
FAX: (818) 244-4246
Email: techsupport@embeddedsys.com
Web: www.embeddedsys.com

We can also upload and download programs by modem whenever that will assist you.

Thanks for specifying Micro/sys products. We'll be glad to be a part of your team as you use our products.

DOC1239

❑ Table of Contents

About Embedded Netsock 1

Configuration of Embedded PC Networking 3
- Using Flash Setup 3
- Using Command Line 6

Programming Overview 7

The Embedded Netsock Header File 9

Programming Under the Winsock API 11
- Structures 12
- Global Variables 16
- Error Returns 17
- Standard Winsock API Functions 19

Programming Under the Alternate API 49
- Structures 50
- Global Variables 51
- Error Returns 52
- Alternate API Functions 53

❑ About Embedded Netsock

The Embedded Netsock™ firmware system from Micro/sys provides a turnkey, built-in TCP/IP network driver system for use with Ethernet networks.

Embedded Netsock is carefully specified to include the subset of TCP/IP that is most applicable to embedded systems. Please refer to *Embedded Netsock™ - an Overview* (Micro/sys part number DOC1238) for general information on the design of Embedded Netsock.

The Embedded Netsock firmware system is pre-installed on a Micro/sys Embedded PC OEM computer to create a particular product order number. For example, the Netsock/100 product is a Micro/sys SBC1190 Embedded PC with the Embedded Netsock firmware installed.

This document provides full documentation covering the usage of all user-callable Embedded Netsock functions. It is intended for software programmers who are creating programs to run on one of the Micro/sys Netsock products.

❑ Configuration of Embedded PC Networking

Depending upon the version of the Netsock computer you are using, there is one of two ways to pass parameters such as IP address network mask to the Embedded Netsock stack:

Method	Netsock Versions	See Page
Flash Setup Utility	All Netsock versions *except* Netsock/410	3
Command Line Parameters	Netsock/410 *only*	6

Flash Setup Utility (all versions *except* Netsock/410)

Micro/sys embedded PCs include on-board Flash Setup™ utilities that can be used to configure the embedded PC, including some aspects of network operation. These Flash Setup utilities are part of the BIOS Boss™ firmware that is factory-installed.

Access to BIOS Boss, and therefore Flash Setup, is accomplished by attaching the Micro/sys-supplied Download Cable to the COM2 serial port (or COM B on 80C188 computers). Resetting or powering up the embedded PC will cause the BIOS Boss firmware to run.

The Flash Setup utility can be used to configure the following networking parameters:

Local computer IP address
Local computer subnet mask
Local computer name
Single other important IP address

Alternatively, Flash Setup can indicate that the first two or three items above are to be obtained at startup from a Dynamic Host Configuration Protocol (DHCP) server somewhere on the same Ethernet subnet.

Each time the Embedded Netsock firmware is started, it accesses the parameters set previously in the Flash Setup screens. If any of the settings require that a remote DHCP server be accessed, Embedded Netsock will perform DHCP initialization. Any values requested and received from the DHCP server will override values previously entered into the Flash Setup screens. The Flash Setup values, however, will not be changed by received DHCP values.

When the CPU card is powered up in the 'LOAD' mode, the firmware prompts you as to whether or not you wish to start the BIOS Boss. Details on the initial BIOS Boss screens are documented in the CPU card's User's Manual. However, if the Embedded Netsock is installed, an additional menu selection (N<e>twork setup) is added to the Flash setup menu.

The "N<e>twork setup" selection is used to enter the *Network setup menu* screen. Note that items from the menu are selected by hitting the key corresponding to the letter in the brackets (e.g. To select the *Ethernet IR<Q>* menu selection, the letter 'Q' should be typed).

```
Network setup menu ——— Micro/sys, Inc.
IP address <S>ource: Specify an IP address
  <I>P address:       192.168.  1. 50
  S<u>bnet mask:      255.255.255.  0


<L>ocal name source:   Specify a local name
Local <N>ame:          NETSOCK100

Ethernet Address:      00-60-92-00-10-00
Ethernet base address: 300h
Ethernet IR<Q>:        IRQ4

<A>dvanced network menu
```

The usage of each of the menu selections is as follows:

IP address <S>ource

This selection allows you to change the source of the IP address between one of two locations. When *Obtain IP address from DHCP server* is selected, there must be a server on the network which can supply an IP address from a given pool of addresses. DHCP (Dynamic Host Configuration Protocol) ensures that unique addresses are supplied to each remote host on the network (assuming they all have unique Ethernet addresses).

When *Specify an IP address* is selected, two additional menu selections (*<I>P address* and *S<u>bnet mask*) are then made visible and must be filled in. The IP address is then static rather than allocated dynamically. The network administrator is then responsible for making sure that the IP address of each host is unique.

<I>P address

This menu selection is only made visible when *IP address <S>ource* is selected and changed to *Specify an IP address*. An IP address that is unique to this particular remote host must then be entered. The four-octet address is entered in the standard IP address format (i.e. four decimal numbers from 0 to 255, separated by periods).

S<u>bnet mask

When *IP address <S>ource* is set to *Specify an IP address*, this field is made visible along with the *<I>P address* field. The subnet mask is used to determine whether a packet should be routed to a host on the same physical network, or whether it should be routed through a gateway to a different physical network. For example, a subnet mask of 255.255.255.0 allows for 254 hosts on a single physical network (254 instead of 256 due to the fact that the octets of 0 and 256 are reserved).

<L>ocal name source

This field can be changed to one of two different selections. It can be set to either *Specify a local name* or to *Obtain a local name from DHCP server.* When it is set to *Obtain a local name from DHCP server*, a server on the network must be configured to supply a unique name to each remote host.

When *Specify a local name is selected*, an additional field (*Local <N>ame*) is made visible and must be set. The network administrator must then fill the *Local <N>ame* field with a name that is unique to that remote host if a DNS or WINS type of server is being used.

Local <N>ame

This item is only made visible when *<L>ocal name source* is selected and changed to *Specify a local name.* It is used to descriptively identify the remote host. This name can be retrieved by the user application.

Ethernet Address

This field is supplied for information only and cannot be changed. The Ethernet address is of the form:

00-60-92-xx-yy-zz

where xx, yy, and zz are hexadecimal numbers assigned by Micro/sys. The first three numbers are managed by the IEEE (Institute of Electrical and Electronics Engineers) in an effort to ensure that all Ethernet addresses are unique. This allows any mixture of single-board computers and desktop computers to coexist on a network without conflicts in network access.

Ethernet base address

This field displays the base I/O port address of the Ethernet interface chip. The base address is for information only and cannot be changed. If any other boards are plugged into the expansion bus of the CPU card, they should be configured for addresses that will not conflict with this address.

Ethernet IR<Q>

This menu selection allows you to rotate the IRQ line used by the Ethernet chip through one of four different settings. The IRQ line selected should not be shared with any other on-board or off-board peripherals.

<A>dvanced network menu

Selecting this item takes you to a different screen that shows the advanced network options.

```
Advanced network menu ── Micro/sys, Inc.

<S>ignificant other IP:      0.  0.  0.  0
```

<S>ignificant other IP

Selecting this item from the menu allows you to enter another IP address that may be retrieved and used within your program. Embedded Netsock makes no use of this address.

Command Line Parameters (Netsock/410 *only*)

The Netsock/410 version has a different BIOS and uses a different load method than other Netsock models.

After DOS is booted on the Netsock/410, you can load and run the ZIP.COM file transfer utility on both the Netsock/410 and on the host development PC. First connect the 'RUN' end of the CA4038 cable to the COM2 port of the Netsock/410, and the far end of the CA4038 cable to your development PC. Use the ZIP.COM menus to transfer the specific sample application you are interested in to the C: drive on the Netsock/410.

To launch the application, reboot the Netsock/410. then log onto C: and make three entries at the Netsock/410 command prompt. The first is to load the Intel 82559 device packet driver with a single parameter, which is the software interrupt number to be used. This is traditionally in the range 0x70 to 0x7F. The second is to load the Embedded Netsock Protocol TSR. The third is to launch the sample program. For example:

```
C:> e100bpkt  0x7e <CR>
C:> nets110  IP=192.168.1.41  Mask=255.255.255.0 <CR>
C:> machine <CR>
```

The Embedded Netsock Protocol TSR has a number of command line options to set IP address and subnet mask. All command line entries are case insensitive.

If a DHCP server is to be used to assign an IP and mask to the Netsock computer at startup, use the following command line:

```
C:> nets110 IP=DHCP <CR>
```

If there is no command line IP specified, the IP address defaults to 192.168.1.50. If there is no command line mask specified, the mask defaults to 255.255.255.0.

❑ Programming Overview

Embedded Netsock offers a subset of a standard TCP/IP driver stack. The programming model offered, is that of the "sockets" model created by UC Berkeley researchers.

This document assumes that you have a basic understanding of TCP/IP and sockets. Micro/sys document DOC1238 can be consulted for an overview of these concepts.

Each network endpoint creates a "socket" through which messages can be sent. When created, a socket is given an access number, which is then used for all further accesses through this socket.

A number of sockets can be active at any time.

Embedded Netsock offers two different Application Programming Interfaces (APIs):

> **Standard Winsock API**: a subset of the standard Winsock API used for network programming under Windows. Programmers familiar with Winsock can immediately write programs under Embedded Netsock.
>
> **Alternate API**: a simplified set of functions that mirror the Winsock functions, but that require fewer housekeeping chores, and fewer parameters to be passed. This API is much easier to learn, as it does not carry any historical baggage with it.

The goals of the Alternate API are:

1) To eliminate the cumbersome **sockaddr_in** and **in_addr** structures,
2) To eliminate host vs. network byte-ordering issues, and
3) To directly return error values instead of requiring a second call.

The API you intend to program under needs to be indicated to the Embedded Netsock system with a single definition prior to including the NETSOCK.H file, as shown in the following section.

❑ The Embedded Netsock Header File

To use Embedded Netsock on all versions except Netsock/410, you merely need to include the file NETSOCK.H in your program.

NETSOCK.H defines all necessary structures, creates global variables, defines error codes, and resolves function calls.

For applications consisting of multiple source files, NETSOCK.H requires ***one and only one*** source file to have the line:

```
#define NETSOCK_MASTER
```

Therefore, the following lines are needed, according to the number of source files needing to access Embedded Netsock functions:

Applications with a Single Source File:

```
#define NETSOCK_MASTER
#include "netsock.h"
```

Applications with Multiple Source Files:

One selected source file:

```
#define NETSOCK_MASTER
#include "netsock.h"
```

All other source files:

```
#include "netsock.h"
```

In addition, you can tell NETSOCK.H which of the two programming APIs you intend to use. For the standard Winsock API (the default), you do not have to do anything.

To use the Alternate API, you must define the constant ALTERNATE_API before including the NETSOCK.H file, as follows:

```
#define NETSOCK_MASTER
#define ALTERNATE_API
#include "netsock.h"
```

The two APIs should not be mixed in the same program.

❑ Programming Under the Winsock API

The Winsock API uses a well-known model to create networked applications.

First, a call to **socket()** creates a socket of a particular type.

Next, a call to **bind()** associates a local port number to the newly created socket.

Calls to **setsockopt()** can be made to modify the attributes of the socket.

To send a message to a remote computer, a **sockaddr_in** structure is built. This structure holds the IP address and destination port number that the message will be sent to. Then a call to **sendto()** causes the message to be sent.

A receive buffer can be established and passed in a call to **recvfrom()**. This function will then return any messages received by this computer that indicates a destination port number matching the local port number specified in the previous **bind()** call. In this way, a number of processes can be "waiting" or "listening" on a number of different port numbers, and respond accordingly. A **sockaddr_in** structure will be returned by **recvfrom()** indicating the sender of the message. This structure can be turned around and used in a **sendto()** call to reply.

The receive status of a socket can be checked with a call to **ioctlsocket()**. This will return the number of bytes currently received, but unread, in the socket's receive buffer.

A number of other Winsock API functions exist to assist in conversions and other functions.

Structures

IP Addresses (in_addr)

An IP address is a 32-bit unsigned integer. For historical and flexibility reasons, this 32-bit integer is held in a structure, the **in_addr** (or internet address) structure. As complicated as this structure looks, it is only a 32-bit unsigned integer.

This classic definition of an IP address overlays three definitions of the 32-bit unsigned integer - four bytes, two words, or a single long. The IP address is always a 32-bit unsigned integer, but the **in_addr** structure allows any component of it to be accessed randomly.

The **in_addr** structure goes a long way in making TCP/IP programs difficult to read. Remember, it's just a 32-bit unsigned integer.

The IP address stored in an **in_addr** structure must be in network order, that is, high-order bytes stored before low-order bytes. If the value to assign to the **in_addr** structure is a value returned from a call to **inet_addr()**, it will already be in network byte order. If it is to be assigned to a 32-bit value that is in host byte order, a call to **htonl()** will be required to reverse the byte order.

```
// Standardized 32-bit IP address structure
// Confusing because of ability to access as bytes, words, or long
struct in_addr {
        union {
           struct { unsigned char s_b1, s_b2, s_b3, s_b4; } S_un_b;      // byte access
           struct { unsigned short s_w1, s_w2; } S_un_w;                 // word access
           unsigned long S_addr;                                         // long access
        } S_un;
};

#define s_addr  S_un.S_addr                    // nickname for accessing in_addr as long
```

Socket Addresses or Names (sockaddr_in)

Under the Winsock model, each endpoint of a network connection is a socket. There are two components to a TCP/IP endpoint - the IP address of a computer, and the transport layer "port" number that routes inbound messages to the correct application or subfunction within that computer.

The **sockaddr_in** structure is a 16-byte structure for compatibility with other protocol families. The generic structure is called **sockaddr**, with **sockaddr_in** denoting an Internet, or TCP/IP, socket address. Much Winsock literature calls the **sockaddr_in** structure the "name" of the socket.

Note that the IP address field *sin_addr* is the standard 32-bit **in_addr** structure described above, and is therefore in network byte order. The *sin_port* field must be set by the programmer to the desired port number. If set from a literal value, the **htons()** function must be used to convert the port number from host byte order to network byte order.

```
// standardized structure for specifying socket address (name) under TCP/IP
struct sockaddr_in {
        short                   sin_family;     // address family (i.e. AF_INET)
        unsigned short          sin_port;       // transport layer port in network order
        struct in_addr          sin_addr;       // 32-bit IP address structure (above)
        char                    sin_zero[8];    // filler
};
```

WSAData returned from WSAStartup()

The **WSAStartup()** function returns a structure describing the Winsock system currently running.

```
// standard structure for detailed information regarding the Winsock implementation
typedef struct {
   unsigned int         wVersion;
   unsigned int         wHighVersion;
   char                 szDescription[WSADESCRIPTION_LEN+1];
   char                 szSystemStatus[WSASYSSTATUS_LEN+1];
   unsigned short       iMaxSockets;
   unsigned short       iMaxUdpDg;
   char far *           lpVendorInfo;
}  WSAData;
```

wVersion
The version of Winsock API that the caller is expected to use. This will be 0x0101, indicating Winsock API version 1.1.

wHighVersion
This will be the same as *wVersion*, 0x0101.

szDescription
A null-terminated ASCII string into which Embedded Netsock copies a description of the Embedded Netsock implementation. The text may contain any characters except control and formatting characters, and will indicate "Embedded Netsock" and the running version of Embedded Netsock.

szSystemStatus
A null-terminated ASCII string into which Embedded Netsock copies relevant status or configuration information. Typically, the string "Running" is returned.

iMaxSockets
The maximum number of sockets that can be supported by Embedded Netsock.

iMaxUdpDg
Maximum size that can be sent as a UDP datagram. **getsockopt()** can also be used to retrieve this value, as option SO_MAX_MSG_SIZE, after a socket has been created.

lpVendorInfo
Pointer to an byte value representing the Embedded Netsock version. This is the same version number expressed in ASCII in the szDescription field. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

Network Configuration

The **NetsockConfig** structure is filled in by Embedded Netsock in response to a call to the **ENgetnetconfig()** function. This structure defines the operating parameters that Embedded Netsock is currently operating with.

The structure is initially set to the values entered into the Flash Setup system of the embedded PC. If DHCP is not used at run-time, these values will be unchanged. However, if DHCP is used to obtain any network parameters at run-time, this structure will be updated with the parameters obtained from the DHCP server, whose IP address will be placed in the *DHCPServerIP* field.

Note that many of the fields are reserved for future use, and are not applicable to the current release of Embedded Netsock.

Also note that on Netsock/410 *only,* the only valid fields returned are IPaddr and Netmask.

```
struct NetsockConfig {
        unsigned char DHCPGetIP_mask_gate;
        unsigned char IPAddr[4];
        unsigned char NetMask[4];
        unsigned char reserved1[4];
        char LocalName[40];
        char reserved2[40];

        unsigned int  AdaptorIOPort;
        unsigned char AdaptorIRQnum;
        unsigned char AdaptorEtherAddr[6];

        unsigned char NumSockets;
        unsigned char reserved3;
        unsigned char SignificantOtherIP[4];
        unsigned char DHCPGetLocalName;
        unsigned char reserved4;
        unsigned char reserved5;
        unsigned char DHCPServerIP[4];
        unsigned char reserved6[4];
        unsigned char reserved7[4];
        unsigned char reserved8;
        unsigned char reserved9;
        };
```

Global Variables

Embedded Netsock uses approximately 96k of system RAM for code, working variables, and network buffers. This memory is obtained from the run-time system through the use of a **_dos_allocmem()** call. This memory is outside of the memory being used by your .EXE application program. Note that this places restrictions on the size of .EXE files that will run under Embedded Netsock. You must have at least 96k more system RAM than the size of the .EXE application program.

The Embedded Netsock system is initialized in RAM by either the Winsock API **WSAStartup()** function, or by the Alternate API **ENStartup()** function. During this initialization process, four global variables are set. These variables are as follows:

```
unsigned int EmbeddedNetsockBaseSegment;
unsigned int EmbeddedNetsockSize;
unsigned int EmbeddedNetsockConfigSize;
unsigned int EmbeddedNetsockVersion;
unsigned int EmbeddedNetsockLoadError;
```

EmbeddedNetsockBaseSegment is the segment address in system memory where the Embedded Netsock software resides. Embedded Netsock uses approximately 96k of RAM from this point. Any call from the application .EXE file to **_dos_allocmem()** will return system memory above Embedded Netsock, up to the maximum physical RAM installed in the lower 640K address space.

EmbeddedNetsockSize is the size in bytes of the loaded Embedded Netsock system.

EmbeddedNetsockConfigSize is the size in bytes of the internal Flash Setup network configuration data area. This is not normally accessible to application programs. Use the **ENgetnetconfig()** function in application programs to get the current configuration of the Embedded Netsock system.

EmbeddedNetsockVersion is the version number of the Embedded Netsock system currently loaded. If version-dependent code is included in application programs, this global variable can be checked for verification that the required version level is in memory and initialized. The lower byte is the MAJOR version number; the higher byte is the MINOR version number. This value will change with each release of Embedded Netsock.

EmbeddedNetsockLoadError is the error value that is set during a call to **WSAStartup()**. This variable should be checked after **WSAStartup()** to ensure that Embedded Netsock is available and running.

Error Returns

Socket creation and use errors

INVALID_SOCKET	(SOCKET)(~0)
SOCKET_ERROR	(-1)

Error codes returned by WSAGetLastError()

WSAEACCES	10013
WSAEFAULT	10014
WSAEINVAL	10022
WSAEMFILE	10024
WSAENOTSOCK	10038
WSAEDESTADDRREQ	10039
WSAEMSGSIZE	10040
WSAEPROTOTYPE	10041
WSAENOPROTOOPT	10042
WSAEPROTONOSUPPORT	10043
WSAEAFNOSUPPORT	10047
WSAENOBUFS	10055
WSAETIMEDOUT	10060
WSAEHOSTUNREACH	10065
WSASYSNOTREADY	10091
WSAVERNOTSUPPORTED	10092
WSANOTINITIALISED	10093

Standard Winsock API Functions

The following pages list full descriptions for the functions that may be used when programming under the Standard Winsock API. If programming under the Alternate API, please refer to the section on Alternate API functions.

bind()

The **bind()** function associates a local address with a socket. By convention, the local address is referred to as a *name*.

int bind (
SOCKET *s*,
const struct sockaddr_in FAR* *name*,
int *namelen*
);

Parameters

s
[in] A descriptor identifying an unbound socket.

name
[in] The address to assign to the socket, stored in a **sockaddr_in** structure.

namelen
[in] The length of the *name*.

Remarks

Each socket must have four items defined in order to be used for end-to-end communication: 1) the local IP address, 2) a local port number to be used by the application using this socket, 3) the remote host's IP address, and 4) the remote host application's port number.

bind() is used to associate a local port number to the specified socket (the local IP address is already known by the TCP/IP system).

The **bind()** function is used on a socket before subsequent calls to the **sendto()** or **recvfrom()** functions. When a socket is created with a call to the **socket()** function, it has no local port number assigned to it. Use **bind()** to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts under TCP/IP: the address family, a host IP address, and a port number that identifies the application. The first two bytes in this block (corresponding to the *sa_family* member of the **sockaddr_in** structure) *must* contain the address family that was used to create the socket, in the case of Embedded Netsock, AF_INET. Otherwise, the error WSAEFAULT will occur.

Return Values

If no error occurs, **bind()** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEFAULT	The *namelen* parameter is too small or the *sin_family* field of the specified *name* is not AF_INET.
WSAENOTSOCK	The descriptor is not a socket.

See Also

socket(), WSAGetLastError()

closesocket()

The **closesocket()** function closes an existing socket.

int closesocket (
SOCKET *s*
);

Parameters

s
[in] A descriptor identifying a socket to close.

Remarks

The **closesocket()** function closes a socket. Use it to release the socket descriptor *s* so further references to *s* will fail with the error WSAENOTSOCK. The associated naming information and queued data are discarded.

An application should always have a matching call to **closesocket()** for each successful call to **socket()** to return any socket resources to the system.

Return Values

If no error occurs, **closesocket()** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAENOTSOCK	The descriptor is not a socket.

See Also

ioctlsocket(), **setsockopt()**, **socket()**

ENgetnetconfig()

The **ENgetnetconfig()** function retrieves the structure detailing the current configuration of Embedded Netsock.

int ENgetnetconfig (
struct NetsockConfig * *cfg*,
int length
);

Parameters

cfg
[in] A far pointer to an area of user memory to hold the configuration data.

length
[in] The size in bytes of the user memory area.

Remarks

The **ENgetnetconfig()** function causes Embedded Netsock to write current configuration data to the user's buffer. If the *length* of the user buffer is smaller than the size of the **NetsockConfig** structure being returned, *length* bytes will be copied to the user area, and an error will be returned.

Return Values

If no error occurs, **ENgetnetconfig()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **WSAStartup()** must occur before using this function.
ENE_BUFFERSOOSMALL	The *length* was not large enough to hold the entire structure.

See Also

WSAStartup()

getsockopt()

The **getsockopt()** function retrieves the specified socket option.

int getsockopt (
SOCKET *s*,
int *level*,
int *optname*,
char FAR* *optval*,
int FAR* *optlen*
);

Parameters

s
[in] A descriptor identifying a socket.

level
[in] The level at which the option is defined. Under Embedded Netsock, the only supported level is SOL_SOCKET.

optname
[in] The socket option for which the value is to be retrieved.

optval
[out] A pointer to the buffer in which the value for the requested option is to be returned.

optlen
[in/out] A pointer to the size of the *optval* buffer.

Remarks

The **getsockopt()** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options affect socket operations, such as the timeouts and debug status.

Under Embedded Netsock, options are only present at the uppermost "socket" level, requiring the *level* parameter to be SOL_SOCKET.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For most options, it will be the size of an integer.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

If the option was never set with **setsockopt()**, then **getsockopt()** returns the default value for the option.

Embedded Netsock supports the following options for **getsockopt()**. The Type column identifies the type of data addressed by *optval*.

Value	Type	Meaning
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.
SO_DEBUG	BOOL	Debugging is enabled.
SO_MAX_MSG_SIZE	unsigned int	Maximum size of a message for SOCK_DGRAM sockets.
SO_RCVTIMEO	int	Receive time-out
SO_SNDTIMEO	int	Send time-out

Calling **getsockopt()** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned from **WSAGetLastError()**.

SO_BROADCAST
A broadcast message can be sent through this socket. Default is off (0).

SO_DEBUG
Embedded Netsock will supply output debug information if the SO_DEBUG option is set by an application. Default is off (0).

SO_RCVTIMEO
The number of milliseconds the **recvfrom()** function will wait for a message from the specified remote host. A value of 0 indicates wait forever; this is the default value.

SO_SNDTIMEO
The number of milliseconds the **sendto()** function will attempt to send a message to the specified remote host. A value of 0 indicates try forever; this is the default value. Timeouts may be caused by a number of underlying network issues, such as IP-to-Ethernet address resolution failure, etc. It is important to note that with SOCK_DGRAM type sockets using UDP protocol, there is no *expected* response from the remote host. A timeout on a **sendto()** call indicates that the message could not be sent out from the local system.

SO_MAX_MSG_SIZE
The maximum size (in bytes) of a datagram message that is allowed by Embedded Netsock.

Return Values

If no error occurs, **getsockopt()** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEFAULT	One of the *optval* or the *optlen* parameters is not valid, or the *optlen* parameter is too small.
WSAEINVAL	The *level* parameter is invalid.
WSAENOTSOCK	The descriptor is not a socket.

See Also

setsockopt(), **socket()**, **WSAGetLastError()**

htonl()

The **htonl()** function converts a **u_long** (32-bit unsigned integer) from host to TCP/IP network byte order.
u_long htonl (
u_long *hostlong*
);

Parameters

hostlong
[in] A 32-bit number in host byte order, that is, low byte to high byte storage in memory.

Remarks

The **htonl()** function takes a 32-bit number in host byte order and returns a 32-bit number in the network byte order used in TCP/IP networks.

Return Values

The **htonl()** function returns the value in TCP/IP's network byte order.

The host order value 0xC0A80121 will be returned as network order 0x2101A8C0.

See Also

htons(), ntohl(), ntohs()

htons()

The **htons()** function converts a **u_short** (16-bit unsigned integer) from host to TCP/IP network byte order.

u_short htons (
u_short *hostshort*
);

Parameters

hostshort
[in] A 16-bit number in host byte order, that is, low byte to high byte storage in memory

Remarks

The **htons()** function takes a 16-bit number in host byte order and returns a 16-bit number in network byte order used in TCP/IP networks.

Return Values

The **htons()** function returns the value in TCP/IP network byte order.

The host order value 0x5001 will be returned as network order 0x0150.

See Also

htonl(), ntohl(), ntohs()

inet_addr()

The **inet_addr()** function converts a string containing an IP "dotted decimal" address into a proper address for an **in_addr** structure, which is network byte ordering.

unsigned long inet_addr (
const char FAR * *cp*
);

Parameters

cp
[in] A null-terminated character string representing a number expressed in the IP standard "dotted decimal" notation (i.e. "192.168.1.39").

Remarks

The **inet_addr()** function interprets the character string specified by the *cp* parameter. This string represents a numeric IP address expressed in the standard "dotted decimal" notation. The value returned is in TCP/IP's network order, and is therefore suitable for use as an IP address to be assigned to a socket address structure.

Values specified using the "dotted decimal" notation take the form:

a.b.c.d

Each part is interpreted as a byte of data and assigned, from left to right, to the four bytes of an IP address. When an IP address is viewed as a 32-bit integer quantity on the PC architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left because low-order bytes are stored in memory before high-order bytes.

Return Values

If no error occurs, **inet_addr()** returns an unsigned long value containing a suitable binary representation of the IP address given. If the string in the *cp* parameter does not contain a legitimate IP address (e.g. if a portion of an "a.b.c.d" address exceeds 255), **inet_addr()** returns the value INADDR_NONE.

The **inet_addr()** function returns an unsigned long value in TCP/IP's network order. For display within a PC architecture, a call to **ntohl()** will be needed to reverse the byte order.

See Also

inet_ntoa(), ntohl()

inet_ntoa()

The **inet_ntoa()** function converts a network address into a string in IP standard dotted decimal format.

char FAR * inet_ntoa (
struct in_addr *in*
);

Parameters

in
[in] A **in_addr** structure that represents a TCP/IP host address.

Remarks

The **inet_ntoa()** function takes a TCP/IP address structure specified by the *in* parameter and returns an ASCII string representing the address in "dotted decimal" notation as in "a.b.c.d".

The string returned by **inet_ntoa()** resides in memory that is allocated by Embedded Netsock. The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next Embedded Netsock function call, but no longer. Therefore, the data should be copied to local application memory before another Embedded Netsock call is made.

Return Values

If no error occurs, **inet_ntoa()** returns a char pointer to a static buffer containing the IP address in standard "dotted decimal" notation. Otherwise, it returns NULL.

See Also

inet_addr()

ioctlsocket()

The **ioctlsocket()** function provides additional information about a socket.

int ioctlsocket (
SOCKET *s*,
long *cmd*,
u_long FAR* *argp*
);

Parameters

s
[in] A descriptor identifying a socket.

cmd
[in] The command to perform on the socket *s*. Must be FIONREAD.

argp
[in/out] A pointer to a parameter for *cmd*.

Remarks

The **ioctlsocket()** function can be used to determine how many bytes of received data are available at a specific socket. Use to determine the amount of data pending in the network's input buffer that can be read from socket *s*. The *argp* parameter points to an unsigned long value in which **ioctlsocket()** stores the result. For type SOCK_DGRAM sockets, FIONREAD returns the size of the first datagram queued on the socket.

Return Values

Upon successful completion, the **ioctlsocket()** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEINVAL	The *cmd* parameter is not valid.

See Also

getsockopt(), setsockopt(), socket()

ntohl()

The **ntohl()** function converts a **u_long** (32-bit unsigned integer) from TCP/IP network order to host byte order.

u_long ntohl (
u_long *netlong*
);

Parameters

netlong
[in] A 32-bit number in TCP/IP network byte order, that is, high byte to low byte storage in memory.

Remarks

The **ntohl()** function takes a 32-bit number in TCP/IP network byte order and returns a 32-bit number in host byte order.

Return Values

The **ntohl()** function returns a 32-bit value in host byte order.

The network order value 0x2101A8C0 will be returned as host order 0xC0A80121.

See Also

htonl(), htons(), ntohs()

ntohs()

The **ntohs()** function converts a **u_short** (16-bit unsigned integer) from TCP/IP network byte order to host byte order.

u_short ntohs (
u_short *netshort*
);

Parameters

netshort
[in] A 16-bit number in TCP/IP network byte order, that is, high byte to low byte storage in memory.

Remarks

The **ntohs()** function takes a 16-bit number in TCP/IP network byte order and returns a 16-bit number in host byte order.

Return Values

The **ntohs()** function returns a 16-bit value in host byte order.

The network order value 0x0150 will be returned as host order 0x5001.

See Also

htonl(), htons(), ntohl()

recvfrom()

The **recvfrom()** function receives a datagram and stores the source IP address and port from which the datagram was sent.

int recvfrom (
SOCKET *s*,
char FAR* *buf*,
int *len*,
int *flags*,
struct sockaddr_in FAR* *from*,
int FAR* *fromlen*
);

Parameters

s
[in] A descriptor identifying a bound socket.

buf
[out] A buffer for the incoming data.

len
[in] The length of *buf*.

flags
[in] Included for completeness, must be 0.

from
[out] A pointer to a buffer that will hold the source address upon return.

fromlen
[in/out] A pointer to the size of the *from* buffer.

Remarks

The **recvfrom()** function is used to read incoming data on a socket, and to capture the address from which the data was sent.

For SOCK_DGRAM sockets, data is extracted from the first enqueued message, up to the size of the buffer supplied. If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **recvfrom()** generates the error WSAEMSGSIZE. With the UDP protocol (SOCK_DGRAM sockets), the excess data is lost.

The *from* parameter will be set to the network address of the remote host that sent the data. The *from* parameter should point to a **sockaddr_in** structure. The value pointed

to by *fromlen* is initialized by the caller to the size of this structure and is modified, on return, to indicate the actual size of the address stored in the **sockaddr_in** structure.

If no incoming data is available at the socket, the **recvfrom()** function waits for data to arrive. If the SO_RCVTIMEO option has been set with **setsockopt()**, the **recvfrom()** function will return with an error condition if no message is received within the timeout specified.

The *flags* parameter is included only for compatibility. It must be set to 0, as Embedded Netsock does not support any flags. NETSOCK.H defines the FLAGS_ZERO constant that can be used, if desired.

Return Values

If no error occurs, **recvfrom()** returns the number of bytes received. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEFAULT	The *fromlen* parameter is too small to accommodate the *from* address or the *from* parameter was not specified.
WSAEINVAL	The socket has not been bound with **bind()**, or a non-zero *flag* was specified.
WSAENOTSOCK	The descriptor is not a socket.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAETIMEDOUT	A message was not received within the time set by an earlier call to **setsockopt()** with the SO_RCVTIMEO option name.

See Also

socket(), sendto(), setsockopt()

sendto()

The **sendto()** function sends data to a specific destination host IP address and port.

int sendto (
SOCKET *s*,
const char FAR * *buf*,
int *len*,
int *flags*,
const struct sockaddr_in FAR * *to*,
int *tolen*
);

Parameters

s
[in] A descriptor identifying a socket.

buf
[in] A buffer containing the data to be transmitted.

len
[in] The length of the data in *buf*.

flags
[in] Included for completeness, must be 0.

to
[in] A pointer to the address of the target socket.

tolen
[in] The size of the address in *to*.

Remarks

The **sendto()** function is used to write outgoing data through a socket. For SOCK_DGRAM sockets, care must be taken not to exceed the maximum packet size of the underlying network, which can be obtained by using to retrieve the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol, the error WSAEMSGSIZE is returned and no data is transmitted.

The *to* parameter can be any valid IP address and port, including a broadcast address. To send to a broadcast address, an application must have used **setsockopt()** with SO_BROADCAST enabled. Otherwise, **sendto()** will fail with the error code WSAEACCES.

The successful completion of a **sendto()** does not indicate that the data was successfully delivered. It merely indicates that the message was sent out on the physical medium.

The **sendto()** function is used to send a datagram to a specific peer socket on a remote computer identified by the *to* parameter. The *to* parameter is required.

To send a broadcast on a SOCK_DGRAM type socket, the address in the *to* parameter should be constructed using the special IP address INADDR_BROADCAST, together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram should not exceed 512 bytes.

Calling **sendto()** with a *len* of zero is permissible and will return zero as a valid value. For SOCK_DGRAM type sockets, a zero-length UDP datagram is sent.

The *flags* parameter is only included for compatibility. It must be set to 0, as Embedded Netsock does not support any flags.

Return Values

If no error occurs, **sendto()** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEACCES	The requested address is a broadcast address, but the SO_BROADCAST option was not set in an earlier call to **setsockopt()** .
WSAEINVAL	A non-zero *flag* was specified.
WSAEFAULT	The *tolen* parameter is too small.
WSAENOTSOCK	The descriptor is not a socket.
WSAEMSGSIZE	The socket is SOCK_DGRAM type, and the message is larger than the maximum supported by Embedded Netsock.
WSAEDESTADDRREQ	A destination address is required in a *to* parameter.
WSAETIMEDOUT	The message could not be sent within the time set by an earlier call to **setsockopt()** with the SO_SNDTIMEO option name.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.

See Also

recvfrom(), socket(), setsockopt()

setsockopt()

The **setsockopt()** function sets a socket option.

int setsockopt (
SOCKET *s*,
int *level*,
int *optname*,
const char FAR * *optval*,
int *optlen*
);

Parameters

s
[in] A descriptor identifying a socket.

level
[in] The level at which the option is defined. Must be SOL_SOCKET for Embedded Netsock.

optname
[in] The socket option for which the value is to be set.

optval
[in] A pointer to the buffer in which the value for the requested option is supplied.

optlen
[in] The size of the *optval* buffer in bytes.

Remarks

The **setsockopt()** function sets the current value for a socket option associated with a socket. Options affect socket operations, such as whether broadcast messages can be sent on the socket, or how long to wait before timing out on a **sendto()** or **recvfrom()**.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options that require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero.

The *optlen* parameter should be equal to **sizeof(int)** for Boolean options. For other options, *optval* points to an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

The following options are supported by Embedded Netsock for **setsockopt()**. The Type column identifies the type of data addressed by *optval*.

Value	Type	Meaning
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_RCVTIMEO	int	**recvfrom()** time-out
SO_SNDTIMEO	int	**sendto()** time-out

SO_BROADCAST
A broadcast message can be sent through this socket. Default is off (0).

SO_DEBUG
Embedded Netsock will supply output debug information if the SO_DEBUG option is set by an application. Default is off (0).

SO_RCVTIMEO
The number of milliseconds the **recvfrom()** function will wait for a message from the specified remote host. A value of 0 indicates wait forever; this is the default value. If a value greater than 0 but less then 500 is specified, the value will be saved but the effective timeout will be 500 ms.

SO_SNDTIMEO
The number of milliseconds the **sendto()** function will attempt to send a message to the specified remote host. A value of 0 indicates try forever; this is the default value. If a value greater than 0 but less then 500 is specified, the value will be saved but the effective timeout will be 500 ms.

Timeouts may be caused by a number of underlying network issues, such as IP-to-Ethernet address resolution failure, etc. It is important to note that with SOCK_DGRAM type sockets using UDP protocol, there is no *expected* response from the remote host. A timeout on a **sendto()** call indicates that the message could not be sent out from the local system.

Return Values

If no error occurs, **setsockopt()** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEFAULT	The *optlen* parameter is incorrect.
WSAEINVAL	*level* is not SO_SOCKET, or the information in *optval* is not valid.
WSAENOPROTOOPT	The option is unknown or unsupported.
WSAENOTSOCK	The descriptor is not a socket.

See Also

bind(), getsockopt(), socket()

socket()

The **socket()** function creates a socket that is set to a specific address family and protocol.

SOCKET socket (
int *af,*
int *type,*
int *protocol*
);

Parameters

af
[in] An address family specification.

type
[in] A type specification for the new socket.

protocol
[in] A particular protocol to be used with the socket that is specific to the indicated address family.

Remarks

The **socket()** function causes a socket descriptor and any related resources to be allocated and bound to a specific low-level protocol.

The *af* parameter is the addressing family to use. Embedded Netsock only supports the TCP/IP address family, so the *af* parameter must be AF_INET.

The *type* parameter is the type of socket to create. Embedded Netsock supports two types:

Type	**Explanation**
SOCK_DGRAM	Supports datagrams, which are connectionless buffers of a fixed maximum length. Embedded Netsock uses UDP as the protocol for this type of socket.
SOCK_RAW	Supports low-level protocols. Embedded Netsock uses ICMP as the protocol for this type of socket.

SOCK_DGRAM sockets allow sending and receiving of datagrams to and from arbitrary remote hosts using **sendto()** and **recvfrom()**.

SOCK_RAW sockets allow the programmer to use ICMP protocols, which can be used for troubleshooting (i.e. echo/ping), and various network routing and error reporting systems.

Return Values

If no error occurs, **socket()** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code can be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.

See Also

bind(), **getsockopt()**, **recvfrom()**, **sendto()**, **setsockopt()**

WSACleanup()

The **WSACleanup()** function terminates use of the Embedded Netsock TCP/IP stack.

int WSACleanup (void);

Remarks

An application is required to perform a successful **WSAStartup()** call before it can use Embedded Netsock services. When it has completed the use of Embedded Netsock, the application may call **WSACleanup()** to free any resources allocated on behalf of the application.

Any sockets open when **WSACleanup()** is called are reset and automatically deallocated.

Return Values

The return value is zero if the operation was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this function.

See Also

WSAStartup()

WSAGetLastError()

The **WSAGetLastError()** function gets the error status for the last operation that failed.

int WSAGetLastError (void);

Remarks

This function returns the last network error that occurred. When a particular Embedded Netsock function indicates that an error has occurred, this function should be called to retrieve the appropriate error code.

A successful function call, or a call to **WSAGetLastError()**, does not reset the error code. To reset the error code, use the **WSASetLastError()** function call with *iError* set to zero.

Return Values

The return value indicates the error code for the last Embedded Netsock operation that failed. See the Error Code section for a description of the errors returned by **WSAGetLastError()**.

See Also

WSASetLastError()

WSASetLastError()

The **WSASetLastError()** function sets the error code that can be retrieved through the **WSAGetLastError()** function.

void WSASetLastError (
int *iError*
);

Parameters

iError
[in] Specifies the error code to be returned by a subsequent **WSAGetLastError()** call.

Remarks

This function allows an application to set the error code to be returned by a subsequent **WSAGetLastError()** call. Note that any subsequent Embedded Netsock routine called by the application will override the error code as set by this routine.

Return Values

None

Error Codes

None

See Also

WSAGetLastError()

WSAStartup()

The **WSAStartup()** function initiates use of the Embedded Netsock TCP/IP stack, and starts all underlying network layers.

int WSAStartup (
WORD *wVersionRequested,*
LPWSADATA *lpWSAData*
);

Parameters

wVersionRequested
[in] The version of the Winsock API that the Embedded Netsock programmer wants to program to.

lpWSAData
[out] A pointer to the **WSAData** data structure that is to receive details of the Embedded Netsock support available.

Remarks

This function *must* be the first Embedded Netsock function called by an application. It allows an application to start up Embedded Netsock, and to retrieve details of the Embedded Netsock support available. The application may only call other Embedded Netsock functions after a successful **WSAStartup()** call.

Embedded Netsock supports Winsock API version 1.1. Therefore, the *wVersionRequested* parameter must be set to 0x0101.

The Embedded Netsock revision level, which is different from the Winsock API version, is available in global variable *EmbeddedNetsockVersion.* This variable is valid after a call to **WSAStartup().**

The following code fragment demonstrates how an application makes a **WSAStartup()** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = 0x0101;

err = WSAStartup()( wVersionRequested, &wsaData );
if ( err != 0 )
  { handle error }
```

Once an application has made a successful **WSAStartup()** call, it may proceed to make other Embedded Netsock calls as needed.

When it has finished using the services of Embedded Netsock, the application can call **WSACleanup()** in order to allow Embedded Netsock to free any resources for the application.

Details of the actual Embedded Netsock capabilities are described in the returned WSAData structure, defined as follows:

```
struct WSAData {
    WORD            wVersion;
    WORD            wHighVersion;
    char            szDescription[WSADESCRIPTION_LEN+1];
    char            szSystemStatus[WSASYSSTATUS_LEN+1];
    unsigned short  iMaxSockets;
    unsigned short  iMaxUdpDg;
    char FAR *      lpVendorInfo;
};
```

The members of this structure are:

wVersion
The version of Winsock API that the caller is expected to use. This will be 0x0101, indicating Winsock API version 1.1.

wHighVersion
This will be the same as *wVersion*, 0x0101.

szDescription
A null-terminated ASCII string into which Embedded Netsock copies a description of the Embedded Netsock implementation. The text may contain any characters except control and formatting characters, and will indicate "Embedded Netsock" and the running version of Embedded Netsock.

szSystemStatus
A null-terminated ASCII string into which Embedded Netsock copies relevant status or configuration information. Typically, the string "Running" is returned.

iMaxSockets
The maximum number of sockets that can be supported by Embedded Netsock.

iMaxUdpDg
Maximum size that can be sent as a UDP datagram. **getsockopt()** can also be used to retrieve this value, as option SO_MAX_MSG_SIZE, after a socket has been created.

lpVendorInfo
Pointer to an integer representing the Embedded Netsock version. This is the same version number expressed in ASCII in the szDescription field. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

Return Values

WSAStartup() returns zero if successful. Otherwise, it returns one of the error codes listed below. Note that the normal error mechanism whereby the application calls **WSAGetLastError()** to determine the error code cannot be used.

Error Codes

WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of the Winsock API requested is not provided by Embedded Netsock.

See Also

WSACleanup()

❑ Programming Under the Alternate API

The Alternate API also uses the concept of sockets to create networked applications. However, the functions are simplified.

First, a call to **ENsocket()** creates a socket of a particular type, and associates a local port number to the newly created socket.

Calls to **ENsetsockopt()** can be made to modify the attributes of the socket.

To send a message to a remote computer, a call is made to **ENsendto()**, with the IP address and destination port passed as parameters..

A receive buffer can be established and passed in a call to **ENrecvfrom()**. This function will then return any messages received by this computer that indicates a destination port number matching the local port number specified in the **ENsocket()** call. In this way, a number of processes can be "waiting" or "listening" on a number of different port numbers, and respond accordingly. The IP address and sending port number will be returned by **ENrecvfrom()** indicating the sender of the message. These values can be turned around and used in an **ENsendto()** call to reply.

The receive status of a socket can be checked with a call to **ENgetsockrxavail()**. This will return the number of bytes currently received, but unread, in the socket's receive buffer.

A number of other alternate API functions exist to assist in conversions and other functions.

Structures

Network Configuration

The **NetsockConfig** structure is filled in by Embedded Netsock in response to a call to the **ENgetnetconfig()** function. This structure defines the operating parameters that Embedded Netsock is currently operating with.

The structure is initially set to the values entered into the Flash Setup system of the embedded PC. If DHCP is not used at run-time, these values will be unchanged. However, if DHCP is used to obtain any network parameters at run-time, this structure will be updated with the parameters obtained from the DHCP server, whose IP address will be placed in the *DHCPServerIP* field.

Note that many of the fields are reserved for future use, and are not applicable to the current release of Embedded Netsock.

```
struct NetsockConfig {
       unsigned char DHCPGetIP_mask_gate;
       unsigned char IPAddr[4];
       unsigned char NetMask[4];
       unsigned char reserved1[4];
       char LocalName[40];
       char reserved2[40];

       unsigned int  AdaptorIOPort;
       unsigned char AdaptorIRQnum;
       unsigned char AdaptorEtherAddr[6];

       unsigned char NumSockets;
       unsigned char reserved3;
       unsigned char SignificantOtherIP[4];
       unsigned char DHCPGetLocalName;
       unsigned char reserved4;
       unsigned char reserved5;
       unsigned char DHCPServerIP[4];
       unsigned char reserved6[4];
       unsigned char reserved7[4];
       unsigned char reserved8;
       unsigned char reserved9;
       };
```

Global Variables

Embedded Netsock uses approximately 96K of system RAM for code, working variables, and network buffers. This memory is obtained from the run-time system through the use of a **_dos_allocmem()** call. This memory is outside of the memory being used by your .EXE application program. Note that this places restrictions on the size of .EXE files that will run under Embedded Netsock. You must have at least 96K more system RAM than the size of the .EXE application program.

The Embedded Netsock system is initialized in RAM by either the Winsock API **WSAStartup()** function, or by the Alternate API **ENStartup()** function. During this initialization process, four global variables are set. These variables are as follows:

```
unsigned int EmbeddedNetsockBaseSegment;
unsigned int EmbeddedNetsockSize;
unsigned int EmbeddedNetsockConfigSize;
unsigned int EmbeddedNetsockVersion;
unsigned int EmbeddedNetsockLoadError;
```

EmbeddedNetsockBaseSegment is the segment address in system memory where the Embedded Netsock software resides. Embedded Netsock uses approximately 96K of RAM from this point. Any call from the application .EXE file to **_dos_allocmem()** will return system memory above Embedded Netsock, up to the maximum physical RAM installed in the lower 640K address space.

EmbeddedNetsockSize is the size in bytes of the loaded Embedded Netsock system.

EmbeddedNetsockConfigSize is the size in bytes of the internal Flash Setup network configuration data area. This is not normally accessible to application programs. Use the **ENgetnetconfig()** function in application programs to get the current configuration of the Embedded Netsock system.

EmbeddedNetsockVersion is the version number of the Embedded Netsock system currently loaded. If version dependent code is included in application programs, this global variable can be checked for verification that the required version level is in memory and initialized. The lower byte is the MAJOR version number; the higher byte is the MINOR version number. This value will change with each release of Embedded Netsock.

EmbeddedNetsockLoadError is the error value that is set during a call to **WSAStartup()**. This variable should be checked after **WSAStartup()** to ensure that Embedded Netsock is available and running.

Error Returns

Socket creation and use errors

INVALID_SOCKET	(SOCKET)(~0)
SOCKET_ERROR	(-1)

Embedded Netsock Alternate API errors

ENE_LDERR_BIOS	(-1)
ENE_LDERR_ADAPTER	(-2)
ENE_LDERR_MEM	(-3)
ENE_LDERR_NETSOCK	(-4)
ENE_NETWORKSTART	(-5)
ENE_NOTRUNNING	(-6)
ENE_SOCKTYPE	(-7)
ENE_INVALIDSOCKET	(-8)
ENE_MAXSOCKETS	(-9)
ENE_MEMORY	(-10)
ENE_BUFFERTOOSMALL	(-11)
ENE_MSGTOOBIG	(-12)
ENE_TIMEOUT	(-13)
ENE_PARAM	(-14)
ENE_NOBROADCAST	(-15)
ENE_HOSTUNREACH	(-16)

Alternate API Functions

The following pages list full descriptions for the functions that may be used when programming under the Alternate Winsock API. If programming under the Standard API, please refer to the section on Standard API functions.

ENCleanup()

The **ENCleanup()** function terminates use of the Embedded Netsock TCP/IP stack.

int ENCleanup (void);

Remarks

An application is required to perform a successful **ENStartup()** call before it can use Embedded Netsock services. When it has completed the use of Embedded Netsock, the application may call **ENCleanup()** to free any resources allocated on behalf of the application.

Any sockets open when **ENCleanup()** is called are reset and automatically deallocated.

Return Values

The return value is zero if the operation was successful. Otherwise a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.

See Also

ENStartup()

ENclosesocket()

The **ENclosesocket()** function closes an existing socket.

int ENclosesocket (
int *s*
);

Parameters

s
[in] A descriptor identifying a socket to close.

Remarks

The **ENclosesocket()** function closes a socket. Use it to release the socket descriptor *s* so further references to *s* will fail with the error ENE_NOTSOCK. The associated port and queued data are discarded.

An application should always have a matching call to **ENclosesocket()** for each successful call to **ENsocket()** to return any socket resources to the system.

Return Values

If no error occurs, **ENclosesocket()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_INVALIDSOCKET	The descriptor is not a socket.

See Also

ENgetrxavail, **ENsetsockopt()**, **ENsocket()**

ENgetnetconfig()

The **ENgetnetconfig()** function retrieves the structure detailing the current configuration of Embedded Netsock.

int ENgetnetconfig (
struct NetsockConfig * *cfg*,
int length
);

Parameters

cfg
[in] A far pointer to an area of user memory to hold the configuration data.

length
[in] The size in bytes of the user memory area.

Remarks

The **ENgetnetconfig()** function causes Embedded Netsock to write current configuration data to the user's buffer. If the *length* of the user buffer is smaller than the size of the **NetsockConfig** structure being returned, *length* bytes will be copied to the user area, and an error will be returned.

Return Values

If no error occurs, **ENgetnetconfig()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_BUFFERSOOSMALL	The *length* was not large enough to hold the entire structure.

See Also

ENStartup()

ENgetsockopt()

The **ENgetsockopt()** function retrieves the specified socket option.

int ENgetsockopt (
int *s,*
int *optname,*
);

Parameters

s
[in] A descriptor identifying a socket.

optname
[in] The socket option for which the value is to be retrieved.

Remarks

The **ENgetsockopt()** function retrieves the current value for a socket option associated with a socket of any type, in any state, and returns the result. Option values must be positive integers (0 - 32767). Options affect socket operations, such as the timeouts and debug status.

If the option was never set with **ENsetsockopt()**, then **ENgetsockopt()** returns the default value for the option.

Embedded Netsock supports the following options for **ENgetsockopt()** .

Value	**Meaning**
SO_BROADCAST	Socket is configured for the transmission of broadcast IP messages.
SO_DEBUG	Debugging is enabled. (Not implemented.)
SO_MAX_MSG_SIZE	Maximum size in bytes of a message for SOCK_DGRAM sockets.
SO_RCVTIMEO	Receive time-out in milliseconds
SO_SNDTIMEO	Send time-out in milliseconds

Calling **ENgetsockopt()** with an unsupported option will result in an error code of ENE_PARAM being returned.

SO_BROADCAST
A broadcast message can be sent through this socket. Default is off (0).

SO_DEBUG
Embedded Netsock will supply output debug information if the SO_DEBUG option is set by an application. Default is off (0). Currently unimplemented.

SO_RCVTIMEO
The number of milliseconds the **ENrevcfrom** function will wait for a message from a remote host. A value of 0 indicates wait forever; this is the default value. Values less than 500 milliseconds can be set and read back, but in these cases the system will use 500 milliseconds as the actual timeout value.

SO_SNDTIMEO
The number of milliseconds the **ENsendto()** function will attempt to send a message to the specified remote host. A value of 0 indicates try forever; this is the default value. It is important to note that with SOCK_DGRAM type sockets (using UDP protocol), there is no *expected* response from the remote host. A timeout on a **ENsendto()** call indicates that the message could not be sent out from the local system. In most cases, this will be due to the inability to resolve a destination IP address to an Ethernet address. Values less than 500 milliseconds can be set and read back, but in these cases the system will use 500 milliseconds as the actual timeout value.

SO_MAX_MSG_SIZE
The maximum size in bytes that can be sent through a SOCK_DGRAM socket.

Return Values

If no error occurs, **ENgetsockopt()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_INVALIDSOCKET	The descriptor is not a socket.
ENE_PARAM	The *optname* is invalid.

See Also

ENsetsockopt(), **ENsocket()**

ENgetsockrxavail()

The **ENgetsockrxavail()** function provides the number of receive bytes available at a socket.

int ENgetsockrxavail (
int *s*,
);

Parameters

s
[in] A descriptor identifying a socket.

Remarks

The **ENgetsockrxavail()** function can be used to determine how many bytes of received data are available at a specific socket. Use to determine the amount of data pending in the network's input buffer that can be read from socket *s*.

Return Values

Upon successful completion, the **ENgetsockrxavail()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_INVALIDSOCKET	The descriptor *s* is not a socket.

See Also

ENgetsockopt(), **ENsetsockopt()**, **ENsocket()**

ENrecvfrom()

The **ENrecvfrom()** function receives a datagram and stores the source IP address and port from which the datagram was sent.

int ENrecvfrom (
int *s*,
char * *buf*,
int *len*,
unsigned long* *fromIP*,
unsigned int * *fromport*
);

Parameters

s
[in] A descriptor identifying a bound socket.

buf
[out] A buffer for the incoming data.

len
[in] The length of *buf*.

*from*IP
[out] A pointer to a long variable that will hold the source IP address upon return.

fromport
[out] A pointer to an integer variable that will hold the source port upon return.

Remarks

The **ENrecvfrom()** function is used to read incoming data on a socket, and to capture the address from which the data was sent.

For SOCK_DGRAM sockets, data is extracted from the first enqueued message, up to the size of the buffer supplied. If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **ENrecvfrom()** generates the error ENE_BUFFERTOOSMALL. With SOCK_DGRAM sockets (UDP protocol), the excess data is lost.

The *fromIP* parameter will be set to the IP address of the remote host that sent the data. The *formIP* parameter should point to a unsigned long. The value pointed to by *fromport* will be set to the port number the remote host used when sending the data.

If no incoming data is available at the socket, the **ENrecvfrom()** function waits forever for data to arrive. If the SO_RCVTIMEO option has been set with **ENsetsockopt()**, the **ENrecvfrom()** function will return with an error condition if no message is received within the timeout specified.

Return Values

If no error occurs, **ENrecvfrom()** returns the number of bytes received. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_INVALIDSOCKET	The descriptor is not a socket.
ENE_BUFFERTOOSMALL	The message was too large to fit into the specified buffer and was truncated.
ENE_TIMEOUT	A message was not be received within the time set by an earlier call to **ENsetsockopt()** with the SO_RCVTIMEO option name.

See Also

ENsocket(), ENsendto(), ENsetsockopt()

ENsendto()

The **ENsendto()** function sends data to a specific destination host IP address and port.

int ENsendto (
int *s*,
const char FAR * *buf*,
int *len*,
unsigned long destIP,
unsigned int *destport*
);

Parameters

s
[in] A descriptor identifying a socket.

buf
[in] A buffer containing the data to be transmitted.

len
[in] The length of the data in *buf*.

destIP
[in] The IP address of the remote host to send the message to.

destport
[in] The port number on the remote host to send the message to.

Remarks

The **ENsendto()** function is used to write outgoing data through a socket. For SOCK_DGRAM sockets, care must be taken not to exceed the maximum packet size of the underlying network, which can be obtained by using **ENgetsockopt()** to retrieve the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol, the error ENE_MSGTOOBIG is returned and no data is transmitted.

The *destIP* parameter can be any valid IP address, including a broadcast address. To send to a broadcast address, an application must have used **ENsetsockopt()** with SO_BROADCAST enabled. Otherwise, **ENsendto()** will fail with the error code ENE_NOBROADCAST.

It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram should not exceed 512 bytes.

The successful completion of a **ENsendto()** does not indicate that the data was successfully delivered. It merely indicates that the message was sent out on the physical meduim.

The **ENsendto()** function is used to send a datagram to a specific peer socket on a remote computer identified by the *destIP* parameter.

Calling **ENsendto()** with a *len* of zero is permissible and will return zero as a valid value. For SOCK_DGRAM type sockets, a zero-length UDP datagram is sent.

Return Values

If no error occurs, **ENsendto()** returns the total number of bytes sent, which can be less than the number indicated by *len*. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_NOBROADCAST	The requested address is a broadcast address, but the SO_BROADCAST option was not set in an earlier call to **ENsetsockopt()** .
ENE_INVALIDSOCKET	The descriptor is not a socket.
ENE_MSGTOOBIG	The socket is SOCK_DGRAM type, and the message is larger than the maximum supported by Embedded Netsock.
ENE_TIMEOUT	The message could not be sent within the time set by an earlier call to **ENsetsockopt()** with the SO_SNDTIMEO option name.
ENE_HOSTUNREACH	The remote host cannot be reached from this host at this time.

See Also

ENrecvfrom(), ENsocket(), ENsetsockopt()

ENsetsockopt()

The **ENsetsockopt()** function sets a socket option.

int ENsetsockopt (
int *s*,
int *optname*,
int *optval*,
);

Parameters

s
[in] A descriptor identifying a socket.

optname
[in] The socket option for which the value is to be set.

optval
[in] The value for the requested option.

Remarks

The **ENsetsockopt()** function sets the current value for a socket option associated with a socket. Options affect socket operations, such as whether broadcast messages can be sent on the socket, or how long to wait before timing out on a **ENsendto()** or **ENrecvfrom()**.

The following options are supported by Embedded Netsock for **ENsetsockopt()**.

Value	**Meaning**
SO_BROADCAST	Allow transmission of IP broadcast messages on the socket.
SO_DEBUG	Output debugging information. (Not implemented.)
SO_RCVTIMEO	**ENrecvfrom()** time-out
SO_SNDTIMEO	**ENsendto()** time-out

SO_BROADCAST
A broadcast message can be sent through this socket. Default is off (0).
SO_DEBUG
Embedded Netsock will supply output debug information if the SO_DEBUG option is set by an application. Default is off (0). Not implemented.

SO_RCVTIMEO
The number of milliseconds the **ENrecvfrom()** function will wait for a message from the specified remote host. A value of 0 indicates wait forever; this is the default value. Values less than 500 milliseconds can be set and read back, but in these cases the system will use 500 milliseconds as the actual timeout value.

SO_SNDTIMEO
The number of milliseconds the **ENsendto()** function will attempt to send a message to the specified remote host. A value of 0 indicates try forever; this is the default value. It is important to note that with SOCK_DGRAM type sockets (using UDP protocol), there is no *expected* response from the remote host. A timeout on a **ENsendto()** call indicates that the message could not be sent out from the local system. In most cases, this will be due to the inability to resolve a destination IP address to an Ethernet address. Values less than 500 milliseconds can be set and read back, but in these cases the system will use 500 milliseconds as the actual timeout value.

Return Values

If no error occurs, **ENsetsockopt()** returns zero. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENE_PARAM	*optval* is invalid or negative.
ENE_INVALIDSOCKET	The descriptor is not a socket.

See Also

ENgetsockopt() , **ENsocket()**

ENsocket()

The **ENsocket()** function creates a socket that is set to a specific address family and protocol.

int ENsocket (
int *type*,
unsigned int *localport*
);

Parameters

type
[in] A type specification for the new socket..

localport
[in] The port number to use when sending or receiving data on this socket.

Remarks

The **ENsocket()** function causes a socket descriptor and any related resources to be allocated and bound to a specific lower level protocol.

The *type* parameter is the type of socket to create. Embedded Netsock supports two types:

Type	**Explanation**
SOCK_DGRAM	Supports datagrams, which are connectionless buffers of a fixed maximum length. Embedded Netsock uses UDP as the protocol for this type of socket.
SOCK_RAW	Supports low-level protocols. Embedded Netsock uses ICMP as the protocol for this type of socket.

SOCK_DGRAM sockets allow sending and receiving of datagrams to and from arbitrary remote hosts using **ENsendto()** and **ENrecvfrom()**.

SOCK_RAW sockets allow the programmer to use ICMP protocols, which can be used for troubleshooting (i.e. echo/ping), and various network routing and error reporting systems.

Return Values

If no error occurs, **ENsocket()** returns a descriptor referencing the new socket. Otherwise, a specific error code is returned. All error codes are negative.

Error Codes

ENE_NOTRUNNING	A successful **ENStartup()** must occur before using this function.
ENEL_MAXSOCKETS	No more socket descriptors are available.
ENE_MEMORY	No buffer space is available. The socket cannot be created.
ENE_SOCKTYPE	The specified type is not supported.

See Also

ENgetsockopt() , ENrecvfrom(), ENsendto(), ENsetsockopt()

ENStartup()

The **ENStartup()** function initiates use of the Embedded Netsock TCP/IP stack, and starts all underlying network layers.

int ENStartup (void);

Remarks

This function *must* be the first Embedded Netsock function called by an application. It allows an application to start up Embedded Netsock, and to retrieve details of the Embedded Netsock support available. The application may only call other Embedded Netsock functions after a successful **ENStartup()** call.

The Embedded Netsock revision level is available in global variable *EmbeddedNetsockVersion*. This variable is valid after a call to **ENStartup()**.

Once an application has made a successful **ENStartup()** call, it may proceed to make other Embedded Netsock calls as needed.

When it has finished using the services of Embedded Netsock, the application can call **ENCleanup()** in order to allow Embedded Netsock to free any resources for the application.

Return Values

ENStartup() returns zero if successful, or the ENE_NETWORKSTART error code if not.

Error Codes

ENE_NETWORKSTART	Indicates that the underlying network subsystem is not ready for network communication.

See Also

ENCleanup()

Embedded Netsock™

Sample Programs

Release 1.10a

MICRO/SYS, INC.
3730 Park Place
Glendale, CA 91020
Phone (818) 244-4600
FAX: (818) 244-4246
www.embeddedsys.com

DOC 1240

10/25/00

Micro/sys Technical Support

Micro/sys offers the best technical support in the business – and it's free!

Our application engineers are ready to assist you in getting your Embedded Netsock project up and running as quickly as possible. You can contact us as follows:

Micro/sys Technical Support
Phone: (818) 244-4600
FAX: (818) 244-4246
Email: techsupport@embeddedsys.com

We can also upload and download programs by modem whenever that will assist you.

Thanks for specifying Micro/sys products. We'll be glad to be a part of your team as you use our products.

DOC1240

1.0 Introduction

This document details the installation, operation, and theory of operation of the sample programs that accompany the Micro/sys Netsock™ computer with it's built-in Embedded Netsock™ TCP/IP software system.

We hope that one of the sample programs will somewhat similar to the application that you have to create. If so, you may be able to modify one of these programs to suit your needs.

If not, these sample applications are a good tutorial on using the Micro/sys Embedded·Netsock system.

The sample applications are:

Machine Control: An Netsock computer manages a "machine" it is installed inside, and accepts commands from Windows computer on the same TCP/IP network. The Windows computer can start, stop, speed up, and slow down the "machine" (blinking LEDs).

Remote Data Acquisition: A Netsock computer waits for requests from a Windows computer on the same TCP/IP network, and responds when queried. The Windows computer can set digital outputs and D/A converter outputs, and read digital inputs and D/A inputs.

Remote Data Acquisition from Multiple Sites: Similar to the Remote Data Acquisition example except that the Windows computer can monitor up to five controllers.

Process Control: A Netsock computer manages the process of heating and cooling an environmental chamber to maintain the desired set point. The Windows computer monitors the state of the system as reported by the Netsock computer and can also change the set point if desired.

Tank Status/Control A Netsock computer, embedded inside a remote control panel with VGA and a touchscreen, controls the parameters of a local system controller for a storage tank.

Each sample application is composed of two parts: an application that runs on a desktop Windows system and an application that runs on Micro/sys Netsock embedded PC hardware. These two applications communicate over an Ethernet link.

The Windows application, written in Visual Basic, uses the Winsock DLL to perform Ethernet transfers.

The Netsock application, written in C, uses the Micro/sys Embedded Netsock firmware to perform Ethernet transfers.

In each sample application, a small set of commands and responses are created. These commands and responses are related to the task at hand - in one case they implement data acquisition functions, and in another they implement machine control functions.

We chose ASCII commands for each sample application, hoping that they would help us create more readable programs. You can use any type of commands and responses appropriate to your task at hand.

In network terms, these applications are client/server. The Netsock computer is a server, waiting for requests from a remote computer, and then satisfying them. In one case, the Machine Control application, the Netsock application has other things to do while waiting for requests. In the other case, the Remote Data Acquisition application, the Netsock computer does nothing during the wait for network requests.

The Netsock computer ships with a specified IP address of 192.168.1.50, and a subnet mask of 255.255.255.0. In addition, the Machine Control application is downloaded into the Netsock computer. Therefore, if you have a Windows computer that you can setup to be on the 192.168.1.n subnet, you can test the system without any further configuration by running the Windows Machine Control application.

If you have an installed TCP/IP network that you want to use with the Netsock system, you will have to run the Flash Setup™ configuration software built into the Netsock computer. See Micro/sys document DOC1139, Embedded Netsock Reference Manual, for details on using Flash Setup.

If you have a DHCP server on a Windows NT server on the same subnet as the Netsock computer, you can use Flash Setup to enable DHCP on the Netsock embedded PC. Embedded Netsock™ can then use DHCP to acquire an IP address on the proper subnet.

2.0 Installing Sample Applications on Windows System

The Windows sample applications are written in Visual Basic version 5.0. Windows 95 or above is required to run them. This installation program does not currently work with Windows NT 4.0.

The Netsock computer sample applications are written in Borland C++ version 5.0. The supplied executables can be downloaded into the Netsock computer without the need for compiling.

To modify the sample applications, it is advisable to use the development tools listed above, as no porting is necessary.

2.1 Network Setup on Windows System

The Windows workstation that is to be communicating with the Netsock computer must have a network adapter installed, and must have the TCP/IP protocol installed and configured.

If network is not configured for TCP/IP protocol, do the following:

Go into Control Panel. Select the **Network** icon, **Configuration** tab. Click **Add|Protocol|Add.** Select **Microsoft** as the **Manufacturer** and **TCP/IP** under **Network Protocols**. Click **OK**.

Windows will now install the needed files and ask you to reboot.

If your system does not already have an IP address assigned to it, simply click **ControlPanel|Network|Configuration**, select **TCP/IP|Properties,** select **Specify an IP address** and enter the IP address and subnet mask you want to use. Click **OK**.

If you do not already have an IP address set up, here are some recommendations. The standard "test network" for Class C TCP/IP networks is **192.168.1.n**. We recommend that you use this unless you are trying to run on an existing TCP/IP installation. Use **192.168.1.1** for the IP address and **255.255.255.0** for the subnet mask on the Windows computer.

In fact, Micro/sys ships the Netsock computer with a specified IP address of **192.168.1.50**. The subnet mask is set to **255.255.255.0**. This allows you to immediately test the Netsock computer on a TCP/IP network without having to do any configuration, as long as the Windows computer is also given an IP address from **192.168.1.1** through **192.168.1.254**.

2.2 Installation of Sample Executables

Close any open applications.

Insert the "SAMPLE APPLICATIONS – Disk 1" (Programs & DLLs) floppy disk into drive A:. **Start|Run**, then enter **A:\Setup.exe**, click **OK**. Follow the instructions. Insert Disk #2 and Disk #3 when prompted. The setup program will create a **C:\Program Files\Ensamples** directory by default and place the sample application files there. (To install the files in another directory, enter a new path when prompted for the directory location.)

The setup program places the required support files (.DLL and .OCX) in the Windows\System directory, and the Windows registry is updated. Although the setup program places **Ensamples** on the Windows **Start Menu**, it only calls the Machine Control program. You will need to go to the sample program directory (C:\Program Files\Ensamples by default) to run the other sample programs.

The **Netsock** directory contains the C++ sample programs to be downloaded to the Netsock embedded PC.

2.3 Installation of Sample Sources

An empty folder named VBprojects is included that can be used to hold the source files, which are located on the "Source files" (Disk 4). You can copy the source files there, or anywhere you like. They are not needed in order to run the installed applications (Disks 1-3).

3.0 Downloading and Launching Applications

There are two download methods used for Netsock products, as follows:

Method	Used On
BIOS Boss XMODEM download	All Netsock versions *except* Netsock/410
ZIP.COM file transfer	Netsock/410 *only*

3.1 BIOS Boss XMODEM Download

When you power up the Netsock computer, the onboard firmware checks for the `LOAD' configuration of the CA4035 cable connected to COM 2 (COM B) of the Netsock computer. When the `LOAD' end of the cable is used on COM 2 (COM B), powering up the system invokes the BIOS Boss. The BIOS Boss is a built in utility for altering system information and programming. By using the BIOS Boss, stand-alone executable files may quickly be downloaded onto the Netsock computer.

Applications are launched automatically when the "load" cable is removed and the Netsock board is reset.

3.2 ZIP.COM File Transfer (Netsock/410 only)

After DOS is booted on the Netsock/410, you can load and run the ZIP.COM file transfer utility on both the Netsock/410 and on the host development PC. First connect the 'RUN' end of the CA4038 cable to the COM2 port of the Netsock/410, and the far end of the CA4038 cable to your development PC. Use the ZIP.COM menus to transfer the specific sample application you are interested in to the C: drive on the Netsock/410.

To launch the application, reboot the Netsock/410. then log onto C: and make three entries at the Netsock/410 command prompt. The first is to load the Intel 82559 device packet driver with a single parameter, which is the software interrupt number to be used. This is traditionally in the range 0x70 to 0x7F. The second is to load the Embedded Netsock Protocol TSR. The third is to launch the sample program. For example:

```
C:> e100bpkt  0x7e <CR>
C:> nets110  IP=192.168.1.41  Mask=255.255.255.0 <CR>
C:> machine <CR>
```

The Embedded Netsock Protocol TSR has a number of command line options to set IP address and subnet mask. All command line entries are case insensitive.

If a DHCP server is to be used to assign an IP and mask to the Netsock computer at startup, use the following command line:

```
C:> nets110 IP=DHCP <CR>
```

If there is no command line IP specified, the IP address defaults to 192.168.1.50. If there is no command line mask specified, the mask defaults to 255.255.255.0.

4.0 Sample Application: Remote Machine Control

This sample application is composed of a server program, **machine.exe**, which runs on the Netsock embedded PC, and a client program, **MachCtrl.exe** that runs on a Windows computer.

machine.exe is downloaded into the Netsock computer prior to shipment from Micro/sys, and does not need to be reloaded unless a different application has been downloaded since receipt of the computer. If required, download **machine.exe** file from the Embedded Netsock Sample Programs diskette according to Section 3.0.

On the Windows computer,

1. **Start|Run** and then enter the path and **MachCtrl.exe**. Click **OK**.

   Enter the IP Address of the server in the dialog box that comes up and press **Enter.**

2. Click on the **Start** button to send a "start" message to the server and begin polling for speed. This enables the other buttons you increase/decrease speed and stop. Each click of the **Faster** or **Slower** button sends a message to increment or decrement the speed by 1. The speed display text box is updated when a response with the new speed is received. Clicking the **Stop** button sends a message to stop the machine and all buttons except for the **Start** button are disabled when a message is received indicating that the speed is 0 (machine stopped).

3. Clicking the **Exit** button in the upper right hand corner ends the program.

4.1 Theory of Operation

For this sample application, a set of command strings, from the Windows client and a set of matching response strings from the Netsock server, were created. The following table shows the command and response set created for this application:

Command	Response	Comment
1	1	Start
+	+	Faster
-	-	Slower
0	0	Stop
S	S=##	Read current speed
	# = 0 to 12	

4.1.1 Visual Basic Application Implementation Details

This application has a single form, *frmMachineControl*, and a single code module, Module1, which contains the global variables needed by the program.

When **MachCtrl.EXE** is launched, the *form frmMachineControl* is loaded. This form includes a single Winsock control, which is given the name *Winsock*. All Ethernet communication is done through this single Winsock control. The Protocol property is set to UDP, the *RemotePort* and Bind properties are set to 5001. You can load the Winsock control with the *RemoteHost* property set to a generic IP address (e.g. "1.0.0.0") and then change that property later to connect to an actual server IP address.

An *InputBox* is used to enter the IP address of the Netsock embedded PC. The Winsock control RemoteHost property is then set to this address.

A timer control named *tmrSpeedUpdate* is enabled to begin polling the server for speed values. The *BackColor* property of the form is set when the form loads to prevent Windows 95 from changing the form color to black, which makes the black labels on the form unreadable.

There are command buttons for Start, Faster, Slower, Stop and Exit. If the machine is stopped (Speed Value = 0) the Faster, Slower and Stop buttons are disabled. Clicking the Start button causes a message of "1" to be sent, which tells the server to start the machine.

Any button clicked causes the polling to stop while the application waits for a response from the server. This is accomplished by disabling the timer control *tmrSpeedUpdate,* which does the speed polling. The polling is begun again by enabling *tmrSpeedUpdate*, once a response is received.

When data is received the Winsock Data Arrival event occurs. If the message received is "1", the Faster, Slower and Stop buttons are enabled and the Start button is disabled. If the message received starts with "S", the Speed Value is extracted from the string and the Text property of the textbox named *txtSpeed* is updated with the number. If the Speed Value is 0, command buttons are enabled/disabled as mentioned earlier.

Each click of the Faster or Slower button sends a message to the server telling it to increment/decrement the Speed Value of the machine by 1. Pressing the Stop button causes a message of "0" to be sent, which is a request to stop the machine.

There is a command button named *cmdCancel* with the caption "Exit". Clicking this ends the program.

4.1.2 Embedded PC application implementation details

The program that runs on the Netsock computer has a simple task. It controls three different outputs on the Netsock computer but does so at a rate controlled by the remote Machine Control program.

When the program begins, Embedded Netsock is loaded and initialized:

```
err = WSAStartup(0x101, &SocketData);
.
.
.
msgsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

local.sin_family = AF_INET;
local.sin_port = htons(DACQ_PORT);
err = bind(msgsock, &local, sizeof(local));
.
.
```

After Netsock is up and running, the application simply waits for direction from the remote Machine Control program. In the beginning, the outputs of the Netsock computer are frozen because the default speed of the machine is 0. Until the speed is changed by the remote system, the Netsock computer outputs will remain still.

Once the machine has been started, it does the following functions:

- Blinks the LED
- Rotates a signal across Port A of the Digital I/O connector
- Periodically sends a '.' out the COM port.

While it is performing these functions, the program is constantly watching for messages coming in over the network from the remote Machine Control program. This is done by making frequent calls to **ioctlsocket()** to determine if a message has come in:

```
err = ioctlsocket(msgsock, FIONREAD, &messagesize);
```

Once a message has been received, the program views the message and, if the message is a valid command, the appropriate action is performed:

If a '**+**' is received, then it speeds up the rate of it's functions.
If a '**-**' is received, then it slows the rate of it's functions.
(This is done by changing the delay value used by **vdelay().**
If a '**0**' is received, then it stops all functions and waits for a '**1**'
In this example, the slowest speed setting is **0,** which is off.
The fastest speed is 10.

The application will continue to perform the above mentioned functions indefinitely, until it is directed to stop.

5.0 Sample Application: Remote Data Acquisition

This sample application allows a desktop PC to acquire and display both analog and digital inputs from a Netsock embedded PC that is running the **dataserv.exe** program.

If required, download **dataserv.exe** file from the Embedded Netsock Sample Programs diskette according to Section 3.0.

On the Windows computer,

1. **Start|Run** and enter the path and **RemDatAcq.exe**. Click **OK**.

2. Enter the IP Address of the Netsock computer server in the dialog box that comes up and press **Enter.** The default is 192.168.1.50.

   The Main screen comes up with buttons that open the other screens when clicked.

3. Click **View All Analog In** button to view the display for the Analog In channels. To view a channel in detail, click the channel button on the left. This opens up the **View All Analog In Detail** screen which does a display for a single channel. There is a digital display of voltage, as well as a gauge display. Clicking the close button (with the door icon) returns to the previous display. Clicking the close button on the **View All Analog In** screen returns to the Main screen.

4. Click **Control Analog Out** button to go to that screen. There are four slider controls (one for each analog output channel) set to be moved horizontally. Click on one of these controls to activate it. Clicking to the right or left of the slider increases/decreases the voltage by 1.00 V. Clicking the right or left arrow keys on the keyboard increases the voltage by .01V. When the slider is moved, a message is sent to the server requesting the new voltage for that channel and the new voltage value is displayed in the text box to the right of the slider control. Clicking the close button returns to the main screen.

5. Clicking the **Digital I/O** button opens that screen. Ports A and C are for output while Port B is only for input. Opening this form causes the application to begin sending messages requesting update information for the three ports. Clicking the mouse on one of the buttons for Port A or C causes a message to be sent to turn that bit on or off and the application waits for a response telling it that this has occurred. If this is successful, the color of the "light" changes, indicating that it has been turned on or off.

6. Clicking the **Exit** button ends the program.

5.1 Theory of Operation

For this sample application, a set of command strings from the Windows client and a set of matching response strings, from the Netsock embedded PC were created. The following table shows the command and response set created for this application:

Command	Response	Comment
WR_AOT_#=HHH # = 0 TO 3 (channel) HHH = 12-bit output value, hexadecimal	WR_AOT	Outputs to DAC channel
RD_AIN_# # = 0 to 7	RD_AIN_#=HHH HHH = 12-bit input value, hexadecimal	Reads ADC channel
RD_RNG	RD_RNG=# #=0 0 to +5v range 1 -5 to +5v range 2 0 to +10v range 3 -10 to +10v range	Reads output range of DAC
WR_DIO_a# = # a = A, B or C (82C55 port) # = 0 to 7 (port bit number) # = 0 or 1 (clear or set)	WR_DIO	Output to 82C55 output bit
RD_DIO_a a = A, B or C (82C55 port)	RD_DIO_a=HH a = A, B or C HH = 8-bit input value, hexadecimal	Reads 82C55 input port

The 82C55 digital I/O device on the Netsock embedded PC is initialized so that ports A and C are output ports, and port B is an input port.

5.1.1 Visual Basic Application Implementation Details

This application has five forms, *frmRemDataAcq*, *frmAOUTControl*, *frmAINDisplay*, *frmAINDetailDisplay* and *frmDIO*. It has a single code module, Module1, which contains the global variables needed by the program.

When **RemDataAcq.EXE** is launched, the form *frmRemDataAcq* is loaded. This form includes a single Winsock control, which is given the name *Winsock*. All Ethernet communication is done through this single Winsock control. The Protocol property is set to UDP, the *RemotePort* and Bind properties are set to 5001. It was found that you can load the Winsock control with the *RemoteHost* property set to a generic IP address (e.g. "1.0.0.0") and then change that property later to connect to an actual server IP address. The *BackColor* property of the form is set when the form loads to prevent Windows 95 from changing the form color to black. This form is the main form, and contains four command buttons. The first three open other forms. The Winsock control *DataArrival* Event occurs when new data arrives. The *GetData* method is used to retrieve the data as a string. The string is then parsed and important data is assigned to variables and used to update control values.

View All Analog In

The top command button, *cmdAINDisplay*, has the caption "View All Analog In". When it is clicked, form *frmAINDisplay* is loaded and shown, and the timer control *tmrAINUpdate* is enabled. This starts the polling for the Analog Input values. This is done by sending the message "RD_AIN_#" where # is the Analog In Channel number. There are 8 Analog In Channels, 0 to 7, and controls to display the voltage value of each one.

When data is received, the Winsock Data Arrival event occurs. The data is assigned to the variable *gstrData*. If the data received begins with RD_AIN, the hex string that represents the voltage is extracted and converted to an integer between 0 and 4095. The channel number is also extracted and used to update the correct controls on the form. This integer is divided by a constant and the result is used as the Value property of the *AINVoltage* progress bar control, which displays the voltage. A textbox control named *txtAIN* is used for a digital display of the voltage. This is done by using the Value property of AINVoltage.

There is a command button with the name of *cmdCancel* with the icon of a door closing on it. When this is clicked, the form *frmAINDisplay* is hidden, the timer control *tmrAINUpdate* is disabled to stop the polling for Analog In display data, and the application returns to the main form.

View Analog In Detail

On form *frmAINDisplay* there are 8 command buttons with channel names. When one of these is pressed, the form *frmAINDisplay* is hidden and controls on the form *frmAINDetailDisplay* are set up to display the channel selected in detail. The caption property of a panel control named *PanelAINDisplay* is assigned the voltage value from the textbox on the form *frmAINDisplay* for that channel.

The caption property of a label control named *lblChannel* is assigned the channel number to display. The "needle" of a "gauge control" is set to display the correct voltage. This is accomplished by using a line control named *LineNeedle* and setting the correct X1, X2, Y1 and Y2 coordinates. The form *frmAINDetailDisplay* is then shown. When the form is loaded, *a dial face,* for the *gauge control,* is set up by using line controls on a rectangle shape control.

There is a command button with the name of *cmdCancel* with the icon of a door closing on it. When this is clicked, the form *frmAINDetailDisplay* is hidden and the application returns to the previous form *frmAINDisplay.*

Control Analog Out

When the main form button with the caption "Control Analog Out" is clicked, the form *frmAOUTControl* is shown. There is a slider control named *SliderAOT* for each of the Analog Output channels 0 to 3. If a slider control is moved, the slider control value is converted to a voltage decimal. The number is multiplied by a constant so that it can be represented as hex, and if the hex number is too short in length, it is padded on the left with zeroes to make it 3 characters long.

The text property of a textbox control named *txtAOUTDisplay* is used to display the voltage setting of the slider control. This is derived from the value property of the slider control. A message starting with "WR_AOT" and containing the channel number and voltage setting is sent to the server. Both Change and Click events are used for the slider controls. The Change event allows changes to be made by using the keyboard right and left arrow keys, while the Click event allows changes to be made with the mouse.

There is a command button with the name of *cmdCancel* with the icon of a door closing on it. When this is clicked, the form *frmAOUTControl* is hidden and the application returns to the main form.

Digital I/O

When the main form button with the caption "Digital I/O" is clicked, the form *frmDIO* is loaded into memory, the timer control *tmrDIO* is enabled and the form is shown. Enabling the timer control *tmrDIO* starts the polling for Digital I/O information.

There are 3 Digital I/O ports A, B and C. The output ports A and C are each represented by 8 command buttons with the name *cmdLight*. The picture property of these command buttons is set to a bitmap image file of either a black light (off) or a red light (on). When the form loads, the picture property is initially set to the black light image (off). The input port B is represented by 8 image controls with the name *imgINLight*. The picture property of these image controls uses the same bitmap image files as the *cmdLight* command buttons, and is initially set to the black light image (off).

When a *cmdLight* button is clicked, the color of the light is changed by changing the picture property of the control, and a message is sent to server to turn the light on or off there. When the timer control *tmrDIO* is enabled, messages are sent requesting the settings for the individual ports.

When data is received, the Winsock Data Arrival event occurs. The data is assigned to the variable *gstrData*. If the data received begins with RD_DIO, the hex string that represents the Digital I/O port settings is extracted and each of the two characters is converted to a binary number string by using the user-defined H2Bin (Hex to Binary) function. This function simply takes a single character hex number that is passed to it and returns the equivalent binary string. The two binary number strings that are returned are then combined into one 8 character binary string that represents the port line settings (On or Off). The picture property of the image controls or command buttons is used to display this.

There is a command button with the name of *cmdCancel* with the icon of a door closing on it. When this is clicked, the form *frmDIO* is hidden and the application returns to the main form. The timer control *tmrDIO* is disabled to stop the polling for Digital I/O settings.

Exit

Clicking this button ends the program.

Message Handling

When the application sends a message to the server, it waits for a response. When the message is sent by code in a timer control Timer event, this is accomplished by disabling the timer control until the response is received, so that a second message is not sent before an answer is received. On messages starting with "WR_AOT" or "WR_DIO", the code in the *SendMsg* function of *frmRemDatAcq* set a global variable *gWaiting* to TRUE. No other messages are sent until this *gWaiting* is set to FALSE again. This occurs when the correct response is received and is handled by the code in the Winsock control *DataArrival* event.

5.1.2 Embedded PC application implementation details

The program that runs on the Netsock computer has two primary functions. First it supplies information to the Remote Data Acquisition program regarding the input ports of the Netsock computer. It also performs sets the output ports of the Netsock computer as requested by the Remote Data Acquisition program.

Once the system has been started and the network connections initialized properly, the program simply waits for commands from the remote system. This program does not perform any function until it is requested to do so by the remote system.
Initially, the digital I/O ports of the Netsock computer are set as follows:

Port A	OUTPUT
Port B	INPUT
Port C	OUTPUT

This program uses the function **recvfrom()** to retrieve messages from the network. Unless a receive timeout option has been specified by a call to **setsockopt()**, the **recvfrom()** function will wait indefinitely for a message to arrive through the network. However, since this example has no duties to perform while it is waiting for messages, the use of **recvfrom()** is sufficient.

Once a message has been received, the program views the message and, if the message is a valid command, the appropriate action is performed:

The program first views the first six characters of the message which are saved as the *command.* The remaining portion of the message contains the *parameters* for the command. The program must parse the *parameters* portion of the message differently depending on the *command.*

```
numbytes = recvfrom(msgsock, datagram, sizeof(datagram), FLAGS_ZERO,
                                          &from,(int far *) &fromlen);
.
.
memcpy(command, datagram, 6);
command[6] = 0;
memcpy(parameters, datagram+6, numbytes-6);
parameters[numbytes-6] = 0;
commanddone = 0;
```

For write commands, the program performs the appropriate write to either the D/A converters or one of the two digital I/O output ports. It then sends the *command* back to the sender as confirmation that the original message was received properly.

For read commands, the program reads the specified input port and returns the *command* with that port's value appended to the command.

See the **Theory of Operation** section for a detailed description of the valid *commands* and *parameters.*

6.0 Sample Application: Remote Data Acquisition from Multiple Sites

This sample application allows a desktop PC running **MultiStationAcq.EXE**, to acquire and display both analog and digital inputs from up to 5 Netsock embedded PCs that are running the **dataserv.exe** program.

First, download the **dataserv.exe** file from the Embedded Netsock Sample Programs diskette according to Section 3.0.

On the Windows computer,

1. **Start|Run** enter the path and **MultiStationAcq.EXE**. Click **OK**.

2. When the form opens type in the IP Address of the first board and press tab when done. You can now enter the IP address of the next board. Addresses for up to five boards are allowed by the application.

3. After entering an IP address, you can click with the mouse on the option button below that board's IP address to select it. The Analog Channel 1 and Digital Inputs Port B values are displayed. To display another board, click the option button for that board after entering the IP address. You can switch back and forth between boards by clicking the option buttons. Only one board is displayed at a time.

4. Clicking the **Exit** button in the lower right hand corner ends the program.

6.1 Theory of Operation

In this example, the Windows client and the Netsock embedded PC communicate in the same way as the in the previous sample application. In fact, although this Windows client program may be quite different than in the previous application, the program that runs on the Netsock computer is the same. Please refer to section 5.1 for a detailed description of the command/response set.

6.1.1 Visual Basic application implementation details

This application has a single form, *frmMultiRemDataAcq*, and a single code module, Module1, which contains the global variables needed by the program.

When **MultiStationAcq.EXE** is launched, the form *frmMultiRemDataAcq* is loaded. This form includes a single Winsock control, which is given the name *Winsock*. All Ethernet communication is done through this single Winsock control. The *Protocol* property is set to UDP, the *RemotePort* and *Bind* properties are set to 5001. You can load the Winsock control with the *RemoteHost* property set to a generic IP address (e.g. "1.0.0.0") and then change that property later to connect to an actual server IP address. The *BackColor* property of the form is set when the form loads to prevent Windows 95 from changing the form color to black, which makes the black labels on the form unreadable.

IP addresses of the Netsock computer boards are typed into the text boxes. There are five textboxes for up to five IP addresses. When something is typed into a textbox, the option button below it is enabled. Selecting an option button sets the Winsock control *RemoteHost* property to that IP address. Disabling the option button when the textbox is empty prevents the accidental setting of the IP address to a null value.

When an option button is selected, this enables the timer control named *tmrUpdate*, which begins polling for data from the selected Netsock computer board. In this application only one Netsock computer board is selected at a time, so only one Winsock control is needed.

The *tmrUpdate* Timer event causes messages to be sent which request the Analog Channel 1 input data and the Digital I/O data. The *gCounter* variable determines which message is sent at a given time, and *tmrUpdate* is disabled after a message is sent so that the application waits for a response before sending a new message.

When data is received, the Winsock Data Arrival event occurs. The data is assigned to the variable *gstrData*. If the data received begins with RD_AIN, the hex string that represents the voltage is extracted and converted to an integer between 0 and 4095. This integer is divided by a constant and the result is used as the Value property of the *AINVoltage* progress bar control, which displays the voltage. A panel control named *PanelAINDisplay* is used for a digital display of the voltage. This is done by using the Value property of *AINVoltage*. If the data received begins with RD_DIO, the hex string that represents the Digital I/O Port B setting is extracted and each of the two characters is converted to a binary number string by using the user-defined H2Bin (Hex to Binary) function, which simply takes a single character hex number that is passed to it and returns the equivalent binary string. The two binary number strings that are returned are then combined into one 8 character binary string that represents the Port B line settings (On or Off). A series of 8 image controls is used to display this. "Lights" are turned on or off by changing the Picture property of the image controls.

A command button named *cmdCancel* (with the Caption "Exit") is used to end the program.

6.1.2 Embedded PC application implementation details

The program **dataserv.exe** that runs with **MultiStationAcq.exe** is the same program as in the previous sample application. For details on this program, please refer to section 5.1.2

7.0 Sample Application: Process Control

This sample application allows a desktop PC running **ProcessControl.EXE** to communicate with a single Netsock computer and send/receive process control data to an Netsock embedded PC that is running the **process.exe** program. The example used in this application uses a temperature set point and monitors the current temperature of some sort of environmental chamber. The Netsock computer adjusts the output to a heating/cooling unit in order to achieve the set temperature.

Before running **ProcessControl.EXE**, download **process.exe** from the Embedded Netsock Sample Programs diskette according to Section 3.0.

On the Windows computer,

1. **Start|Run** enter the path and **ProcessControl.EXE**. Click **OK**.

2. Enter the IP Address of the Netsock computer server in the dialog box that comes up and press **Enter**. The default is 192.168.1.50.

3. The Controller form opens. Click on the π or θ arrows to increase/decrease the temperature set point. Hold down the mouse on an arrow to increase/decrease the set point more quickly. Click in the set point text box to type in a number with the keyboard. Up to 3 digits will be accepted. Press tab to exit the text box. When the set point is changed, a message is sent to the server to request a new set point. The program is consistently polling for updates on the current temperature. When a response is received, the current temperature digital display is updated.

4. Polling is done for the current Output Value. A shape control is used to display this as a bar. Cooling is represented as blue, heating as red. When the value is near 0, the bar is white.

5. Clicking the **Exit** button in the upper right hand corner ends the program.

7.1 Theory of Operation

The **process.exe** program on the Netsock computer runs independently of any monitoring programs. It continuously monitors the temperature and adjusts the output accordingly. In the event that **ProcessControl.EXE** is running on a network computer, then the Netsock computer must also respond to the many requests sent by that program.

The following command and response set was created for this application:

Command	Response	Comment
GT	GT=####	Get the current temperature
GP	GP=####	Get the current setpoint
GO	GO=####	Get the output value. The output value will range from 0 to 4095 where 2047 is the midpoint representing no output.
SP=#### #### = -100 to 300	SP	Set the desired temperature in °F

7.1.1 Visual Basic application implementation details

This application has a single form, *frmProcessControl*, and a single code module, Module1, which contains the global variables needed by the program. The application uses global variables for Current Temperature and Set Point.

When **ProcessControl.EXE** is launched, the form *frmProcessControl* is loaded. This form includes a single Winsock control, which is given the name *Winsock*. All Ethernet communication is done through this single Winsock control. The Protocol property is set to UDP, the *RemotePort* and Bind properties are set to 5001. You can load the Winsock control with the *RemoteHost* property set to a generic IP address (e.g. "1.0.0.0") and then change that property later to connect to an actual server IP address. An *InputBox* is used to enter the IP address of the Netsock embedded PC. The Winsock control *RemoteHost* property is then set to this address. A timer control names *tmrGTUpdate* is enabled to begin polling for temperature values from the server.

There are two image controls named *imgUp* and *imgDown*, and two timer controls named *tmrDown* and *tmrUp*. When the mouse is held down over the up arrow image, the "up" timer event is started, which increases the Set Point value. When the mouse is held down over the down arrow image, the "down" timer event is started, which decreases the Set Point value. When the mouse is released, the Set Point stops changing. This is accomplished by enabling the Up and Down timer controls when the mouse is held down, and disabling these timer controls when the mouse is released. A textbox named *txtSetPoint* displays the current set point as its Text property. This textbox accepts editing. Clicking on it with the mouse and typing in the new number can change the Set Point value. Pressing the keyboard Tab key exits the textbox its Change event occurs, which causes a message to be sent to the server with the new Set Point value. The timer control *tmrGTUpdate* is disabled to stop the polling and wait for a response to the message. Once the response is received, polling begins again by enabling the timer control again.

If the textbox *txtSetPoint* is edited, the length of the string that is entered is checked to make sure that it is less than 5 characters long. If it is larger than 5, then only the first 4 characters are accepted.

When the timer *tmrGTUpdate* is enabled, separate messages are sent to the server requesting update values for Current Temperature, Set Point, and Output Value. The timer is then disabled to stop the polling and wait for a response from the server. When the response is received, the timer is enabled and polling resumes.

When a message is sent, the timer control *tmrConnection* is enabled. This checks to see if there is a connection error by waiting 3 seconds for a response. If there is no response in that time, a message box is displayed indicating that there may be a connection error. When the OK button is clicked, a message is sent to see if the connection has been reestablished. If a response is received (Data Arrival event) *tmrConnection* is disabled.

When data is received from the server, the Winsock control Data Arrival event occurs.

If the string received starts with "GO", the string contains the Output Value. A shape control with the name *OutputValue* is used for a "bar graph" style graphical *respresentation* of this. When a reply to the message "GO" (get Output Value) is received, the Output Value is extracted from the string and converted to a number between 0 and 4095. For display purposes, an Output Value of 2000 or less represents "cooling" and the shape control (bar) *FillColor* property is set to blue. If the Output Value is between 2000 and 2100, it is considered to be zero for display purposes and is represented by a small white box. If the Output Value is greater than or equal to 2100, it represents "heating" and the *FillColor* property is set to red. The magnitude of the Output Value is displayed by changing the Left and Width values of the shape control *OutputValue*.

If the string received starts with "GT" (get Current Temperature), the temperature value is extracted from the string. The Caption property of a panel control named *panelCurrentTemp* is then updated with this string represented as degrees Fahrenheit.

If the string received starts with "GP" (get Current Set Point), the temperature value is extracted from the string. The Text property of a textbox control named *txtSetPoint* is then updated with this string represented as degrees Fahrenheit.

If the string received starts with "SP", this means that a message sent to the server with a Set Point value was received and responded to.

A command button with the name *cmdCancel* and the caption "Exit" ends the program when clicked.

7.1.2 Embedded PC application implementation details

The program that runs on the Netsock computer is responsible for monitoring and controlling an environmental control system. It is responsible for the following tasks:

- Check the temperature of the unit in question
- Make any necessary adjustments to the temperature control output
- Watch for and respond to messages coming in from a monitoring system on the network

Once the system has been started and the network connections initialized properly, the program begins continuously performing the above mentioned tasks. The general idea of this example is that the Netsock computer will be controlling the temperature inside an environmental chamber. Since the Netsock computer has A/D and D/A converters optional, it is well suited to this task. However, in this example, since we do not have a thermometer connected, nor is there a heating/cooling element to control, aspects of this example are simulated.

The function **ReadTemperature()** simply returns a local temperature value. In an actual application, the A/D converters would probably be employed to read the temperature from a thermometer.

Every second, the function **SimulateTemperatureChange()** is called to adjust the simulated temperature value from –5 to +5 degrees depending on the value of **outputvalue**.

outputvalue is an integer value from 0 to 4095 which is set by the function **SetOutput()**. **outputvalue** was chosen to be a 12-bit value so that it may be sent directly to a 12-bit D/A converter controlling a heating/cooling unit. Because this example can cool as well as heat the environment, the **outputvalue** must be used for both cooling and heating. Therefore, the median value of 2047 is used as a neutral/off position for the heating/cooling unit. 0 is the coldest setting and 4095 is the hottest setting.

The desired temperature set point, output value of the heating/cooling unit, and the current temperature of the system are all kept and managed by the Netsock computer. All of this information is, however, available to any monitoring system that requests it. Remote monitoring systems can also change the set point with a simple request to the Netsock computer.

The program is constantly watching for messages coming in over the network from the remote Process Control program. Once again, this is done by making frequent calls to **ioctlsocket()** to determine if a message has come in. Once a message has been received, the program views the message and, if the message is a valid command, the appropriate action is performed:

See the **Theory of Operation** section for a detailed description of the valid *commands* and *parameters*.

8.0 Sample Application: Remote Tank Controller

This sample application simulates a remote storage tank process controller. A single Netsock embedded PC running the **control.exe** program communicates with the local control system of a storage tank. The remote controller utilizes a touchscreen panel and video display. The application demonstrates communication between the remote controller and the tank status display on the desktop PC that is running the **TankStatus.exe** program.

Before running **TankStatus.exe**, download **control.exe** file from the Embedded Netsock Sample Programs diskette according to Section 3.

On the Windows computer,

1. Start|Run and enter the path and **TankStatus.exe**. Click **OK**.

2. Enter the IP address of the Netsock computer remote controller in the dialog box that comes up and press **Enter**. The default is **192.168.1.50.** The main screen comes up with a tank status display. Upon communication with the desktop PC, the remote control panel will be enabled and the red indicator will turn green. The remote controller's display will show "REMOTE CONTROLLER ENABLED".

3. The remote control panel touchscreen can now be used to change the tank parameters. There are three tank parameters that can be set from the remote controller. These are the tank inlet rate, the tank outlet rate, and the tank level. In the real world, the tank level would be a function of the input and output rates. For demonstration purposes, the tank level can be changed also. These functions demonstrate communication from the remote controller to the local tank process control system.

   The "SET" button on the remote controller initiates the command sequence. After hitting "SET", you are prompted to select one of the three tank parameters. After selecting one, the user is prompted to enter a value for the parameter. The range of values that are valid is also displayed. After a valid value is entered, the "SEND" button will send the command to the local tank control system. Any out of sequence keystrokes will cause error messages to be displayed. At any time, hitting the "CLR" button will restart the command sequence.

4. The tank status display can also send a message to the remote controller and enable or disable the remote control panel. These functions demonstrate communication from the local tank process control system to the remote controller unit.

   To display a message on the remote controller's display, type the desired message and then hit the "SEND" button on the tank status display. To disable or enable the remote control panel, hit the "DISABLE REMOTE" or "ENABLE REMOTE" button, which will have different text depending upon whether the remote controller is enabled or disabled. A message will be displayed on the remote controller display showing the state of the remote panel. The remote indicator will be green if the remote panel is enabled or red if it is disabled. If disabled, the remote panel's buttons will be inoperative.

5. Clicking the **Exit** button in the upper right hand corner ends the program.

8.1 Theory of Operation

For this sample application, a set of command strings was created for sending commands between the Netsock embedded PC and the Desktop PC. A set of matching response strings, when appropriate, was also created for responding to the command strings. The following table shows the command and response set, in both directions, that was created for this application.

Netsock computer to Desktop

Command	Response	Comment
ST_IN=### # = 0 to 9	None	Sets input rate of tank from 0 to 500 (GPM)
ST_OT=### # = 0 to 9	None	Sets output rate of tank from 0 to 400 (GPM)
ST_LV=#### # = 0 to 9	None	Sets level of tank from 1000 to 9000 (G)

Desktop to Netsock computer

Command	Response	Comment
ST_MS=###...# # = Message (42 Char MAX)	ST_MS=OK	Write message to remote control panel
ST_ST=# # = 0 (Stop) # = 1 (Start)	ST_ST=OK	Enable / disable remote control panel
ST_IP=	ST_IP=OK	Set IP address of PC upon communication

8.1.1 Visual Basic Application Implementation Details

This application has a single form, **frmTankStatus**, and a single code module, **Module1**, which contains global variables needed by the program.

When **TankStatus.EXE** is launched, the form *frmTankStatus* is loaded. This form includes a single Winsock control, which is given the name **Winsock**. All Ethernet communication is done through this single Winsock control. The *Protocol* property is set to UDP, the *RemotePort* and *Bind* properties are set to 5001. The Winsock control is loaded with the *RemoteHost* property set to a generic IP address (e.g. "1.0.0.0") and then the *RemoteHost* property is changed later to connect to an actual server IP address. An *InputBox* is used to enter the IP address of the Netsock embedded PC. The Winsock control *RemoteHost* property is then set to this address. A timer control named *tmrConnect* loads to begin polling the remote controller for the message ST_IP=OK, which indicates that the remote controller has found the IP address of the desktop PC running the *TankStatus* application and a connection has been made. Once this message is received, the timer control is disabled, which stops the polling. The *BackColor* property of the form is set when the form loads to prevent Windows 95 from changing the form color to black, which makes the black labels on the form unreadable.

There are command buttons for Send, Enable Remote, and Exit. If the Send button is clicked, a message of "ST_MS=" plus the contents of the "Send Manual Message to Remote" textbox is sent to the remote controller. If the message textbox is empty, only "ST_MS=" is sent as a message. When the Enable Remote button is clicked, the picture property of the image of a light to the right of the button changes to indicate that the remote controller is enabled (green light). The caption property of the button is changed so that it now reads "Disable Remote". A message of ST_ST=1 is sent to the remote controller to tell it to enable. When the Disable Remote button is clicked, the picture property of the image of a light to the right of the button changes to indicate that the remote controller is disabled (red light). The caption property of the button is changed so that it now reads "Enable Remote". A message of ST_ST=0 is sent to the remote controller to tell it to disable.

When data is received the Winsock Data Arrival event occurs. If the message received is "ST_IP=OK", the timer control *tmrConnect* is disabled, as mentioned above. The remote controller is enabled by calling the *cmdEnable_Click* routine (same as clicking the Enable Remote button). If the message received begins with "ST_IN=", the Inlet value is changed and the Inlet textbox is updated. If the message received begins with "ST_OT=", the Outlet value is changed and the Outlet textbox is updated. If the message received begins with "ST_LV=", the Tank Level value is changed.

The updating of the Tank Level display is done by code in the *tmrChangeLevel* timer event routine. A timer control called *tmrChangeLevel* is used. Limits of 1000 gallons (lower limit) and 9000 gallons (upper limit) have been established as a safe operating range for the tank (for demonstration purposes). When the tank level value is between these limits, the Tank Level textbox value is updated, changes in the Tank Level are calculated, and the new Tank Level is simulated in the Tank Status display. The Tank display is simulated by using "water colored" shape controls and changing their Top and Height properties to move the "water level" up and down. If the tank level drops below 1000 gallons, a warning message is displayed in the Send Message textbox and the Inlet value is made higher than the Outlet value, so that the tank begins to fill up again. A limit of 500 gallons per minute is placed on the Inlet value to keep it realistic. The warning message is also sent to the remote controller to be displayed there. A timer control *tmrWarning* is enabled to change the message in five seconds. The tank level display is updated. If the tank level goes above 9000 gallons, a warning message is displayed in the Send Message textbox and the Inlet value is set to 0 to simulate closing the Inlet valve. The tank begins to empty again. Again, a warning message is also sent to the remote controller to be displayed there, the timer control *tmrWarning* is enabled to time the warning message and the tank level display is updated.

The timer control *tmrWarning* has an interval property of five seconds. This changes the message displayed after five seconds. If the message in the textbox was "WARNING - TANK LEVEL LOW...", the message is changed to "INCREASING INLET RATE TO (value)". If the message in the textbox was "WARNING - TANK LEVEL HIGH...", the message is changed to "CLOSING INLET VALVE". Five seconds later, this second message is erased. The routine *cmdSend_Click* is called which sends the contents of the message textbox to the remote controller. The timer control is then disabled to stop the messages. The last message that is sent to the remote controller clears the message there.

A user-defined *SendMsg* routine uses the Winsock *SendData* method to send the messages.

There is a command button named *cmdCancel* with the caption "Exit". Clicking this ends the program.

8.1.2 Embedded PC Application Implementation Details

The program that runs on the Netsock embedded PC is responsible for implementing the full functionality of a remote process control panel. It is responsible for the following tasks:

- Draw the Graphical User Interface (GUI) for the remote control panel
- Control the functionality of all the GUI components
- Control communications between the remote control panel and the desktop PC

When the system is started, the following sequence of events occurs:

The network functions are initialized with **InitNetwk()**. This function initializes the Netsock computer network functions and binds the socket used for network communication. After the successful completion of this function, the communication between the host and remote can begin.

The video display and touchscreen functions are initialized with the **SetMPC204Base()**, **InstallDebounce()**, and **setuptranslation()** functions. *SetMPC204Base()* simply sets the base address of the MPC204 video card. *InstallDebounce()* installs the software needed to debounce the press of a touchscreen cell. The *setuptranslation()* function translates the touchscreen coordinates for the size of touchscreen used. The GUI is drawn on the display with the **DrawGUI()** function.

The GUI consists of a 16-button keypad with numbers zero through nine and six function keys, a 42 character text display window, an enabled / disabled indicator, a "Tank #4 Controller" title block, and a Micro/sys logo with "Micro/sys Netsock computer Demo" written under it. The six function keys are Set, Clear, Send, In Rate, Out Rate, and Tank Level. After the GUI is drawn, the program checks for a message from the desktop PC while no key is hit on the remote controller. As long as no key is pressed, the remote will check for messages. The command sequence is started when the "SET" key is pressed. If any other key is pressed, the remote will go back and check for messages from the PC. When the command sequence is begun with the press of "SET", the operator must then enter one of the three functions (In Rate, Out Rate, or Tank Level). The desired value is then entered for the function. All numeric ranges and error checking for each function are performed during the command sequence. If the wrong value is entered, the operator is sent a user prompt message in the text display window. Upon the pressing of the "SEND" key, the command is sent. At any time during the command sequence, hitting the "CLR" button clears the display and restarts the checking of messages from the PC.

This program uses the function **recvfrom()** to retrieve messages from the desktop PC. This function is called in the function **checkmessages()**. Once a message has been received, the program views the message and, if the message is valid, the appropriate action is performed. The first time *checkmessages()* is called, the ST_IP= message will be received from the desktop PC. The remote controller responds with a ST_IP=OK message and sets the "to" structure equal to the "from" structure. This tells the remote what the IP address of the desktop PC is where it will be sending all further messages. The ST_IP= command will only be sent and responded to once. The messages are processed as follows:

The program first views the first six characters of the message which are saved as the *command.* The remaining portion of the message contains the *parameters* for the command. The program must parse the *parameters* portion differently depending upon the command.

```
numbytes = recvfrom(msgsock, datagram, sizeof(indatagram), FLAGS_ZERO,
                                                &from, (int far *)  &fromlen);
memcpy(command, indatagram, 6);
command[6] = 0;
memcpy(parameters, indatagram+6, numbytes-6);
parameters[numbytes - 6] = 0;
commanddone = 0;
```

For write commands, the command is built from the command and *value[]* variables that are set during the touchscreen command sequence. From these variables, the *outdatagram* is constructed and sent to the desktop PC after the SEND button is pressed. The program uses the **sendto()** function to send messages to the desktop PC. The "to" structure has the IP address of the desktop PC from the initial call to *checkmessages().* If a socket error is received by *sendto(),* the **WSACleanup()** function is called automatically.

```
err = sendto(msgsock, datagram, strlen(datagram), FLAGS_ZERO,
                                                    &to, sizeof(to));
if (err == SOCKET_ERROR)
      WSACleanup();
```

For write commands, no response is sent back to the remote control panel. For read commands, an "OK" is appended to the command and sent back to the desktop PC as a confirmation that the original command was received properly.

See the **Theory of Operation** section for a detailed description of the valid commands and parameters.