Volume

2

egnite Software GmbH

optiCompo electronics

Embedded Ethernet

Ethernut Software Manual

EMBEDDED ETHERNET

Ethernut Software Manual

---

egnite Software GmbH
Westring 303 • D-44629 Herne
Fon +49 (0)2323-925 375 • Fax +49 (0)2323-925 374
http://www.egnite.de

Contents

Chapter

1

About Nut/OS and Nut/Net

Connects embedded applications to a local Ethernet and the global Internet.

Nut/OS Features

Nut/OS is a very simple Realtime Operating System (RTOS) providing the following features:

- Modular design
- Cooperative multithreading
- Event queues
- Dynamic memory management
- Timer support
- Stream I/O functions
- Expandable device driver interface
- Simple Flash ROM filesystem
- Open source to be used with GNU Compiler Collection

Nut/Net Features

Nut/Net is a TCP/IP stack providing:

- ARP, IP, UDP, ICMP and TCP protocol over Ethernet
- Automatic configuration via DHCP
- HTTP API with filesystem access and CGI functions
- TCP and UDP Socket API for other protocols
- Open source to be used with GNU Compiler Collection

Chapter

2

Quick Start

This chapter will help you quickly set up and start using Nut/OS and Nut/Net.

Directory Layout

app	Nut/OS and Nut/Net application samples
app/basemon	Source of the preloaded BaseMon program to perform basic hardware tests
app/httpd	Embedded Webserver sample
app/portdio	TCP server to control port D
app/simple	The most simple Nut/OS application
app/tcpc	Simple TCP client
app/tcps	Simple TCP server
app/threads	Multithreading sample
app/timers	Nut/OS timer support sample
app/uart	Serial communication sample
bin	Compiled binaries, ready to burn into the Ethernut board
dev	Nut/OS device driver source code
doc	Ethernut hardware and software

	manual
doc/api/html	Ethernut software reference
fs	Micro-ROM filesystem sources
include	Source code header files
lib	Precompiled Nut/OS and Nut/Net library files
net	Nut/Net source code
os	Nut/OS source code
pro	User protocol source code
tools	Additional tools to build Ethernut applications
tools/crurom	Source code of crurom utility

Prerequisites for Operation

The following software is needed to build a Nut/OS application:

- GNU Compiler Collection for AVR. Nut/OS has been tested with AVR GCC version 3.0.
- In-System programming software like AVR ISP

For additional hardware requirements, please consult the Ethernut Hardware Manual.

Windows Installation

Building and installing AVR GCC in a Windows environment can become a quite complicated task. Fortunately, many helpful people have prepared an easy to use installation program, which is available on the Ethernut CD.

When started, the installation program unpacks itself, copies the files into the requested directory and automatically recompiles the library files.

The compiler needs several environment variables to be set and will not work from the standard DOS window unless you run the batch file RUN.BAT, which is found in the installation directory.

Linux Installation

Will be added soon.

Compiling and Linking

All directories containing source code modules provide a Makefile to build the binaries. Open a Linux console or DOS prompt window with the proper GCC environment and enter

```
make
```

This will compile any source code file in the current directory, that has changed since the last build. Calling

```
make install
```

will additionally copy the resulting hex file to the bin directory or the resulting library file to the lib directory.

```
make clean
```

will delete all compiled object files and binaries in the current directory. This is useful to force a complete re-build if, for example, you changed the compiler environment.

All these commands may also be invoked from the top directory, where you installed Nut/OS. The command will be re-invoked automatically for all source code subdirectories.

Programming the Ethernut Board

In order to run your application on the Ethernut board, the resulting hex file of the compiled binary has to be moved from the PC's harddisk into the Ethernut's flash memory. Depending on the programming adapter used, you need to connect the parallel or serial port of your PC with the ISP socket of the Ethernut board. Refer to the Ethernut Hardware Manual for further information.

In the next step you need to start your programmer software.

Programming under Windows

In a Windows environment you may use AVR ISP. The latest version is freely available from Atmel's Website. If started for the first time you need to create a new project, selecting Project/New Project on the menubar. To program the Ethernut board you need to select the device ATmega103L and press OK. The next step is to activate the Program Memory Window and select File/Load from the menubar. A file selection box appears, allowing you to locate the hex file to be programmed into the Ethernut board.

Note, that by default Nut/OS applications are stored in Motorola S-Record format and file extension rom. After the hex file has been loaded into the Memory Window, press F5 on the keyboard to erase, program and verify the device. During programming the red Prog-LED on the Ethernut board will be lit.

There's a problem with some versions of the AVR ISP software, which sometimes refuses to correctly detect the ATmega's chip signature. In this case disable the signature check by selecting Options/Advanced on the menu bar and disable the signature check.

There are several alternatives to the above mentioned AVR ISP software, like GoISP from Bernd Mueller or UISP from Uros Platise, which are both freeware.

Chapter 3

Nut/OS

Overview.

This chapter is quite incomplete. Please check the Nut/OS Software Reference in directory doc/api/html for a more detailed description of the provided functions.

System Initialization

By default, C programs are started with a routine called main. This isn't much different in Nut/OS, however, the **main** routine is already build into the kernel and added to the application program by linking init.o.

It will initialize memory management and the thread system and start an idle thread, which in turn initializes the timer functions. Finally **NutMain** is called, which must be defined by the application program as its main routine. Because there's nothing to return to, this routine should never do so.

A sample application called **simple** demonstrates the most simple application, that could be build with Nut/OS.

Timer Management

Nut/OS provides time related services, allowing application to delay itself for an integral number of system clock ticks. A clock tick occurs every 62.5 ms.

Another useful routine is **NutGetCpuClock**, which returns the CPU clock in Herz.

Note, that Nut/OS uses on-chip hardware timer 0 of the ATmega CPU. Applications should use timer 1, if they need an independant hardware timer or a higher resolution.

Heap Management

Dynamic memory allocations are made from the heap. The heap is a global resource containing all of the free memory in the system. The heap is handled as a linked list of unused blocks of memory, the so called free-list.

The heap manager uses best fit, address ordered algorithm to keep the free-list as unfragmented as possible. This strategy is intended to ensure that more useful allocations can be made. We end up with relatively few large free blocks rather than lots of small ones.

Thread Management

Typically Nut/OS is at its most useful where there are several concurrent tasks that need to be undertaken at the same time. To support this requirement, Nut/OS offers some kind of light processes called threads. In this context a thread is a sequence of executing software that can be considered to be logically independent from other software that is running on the same CPU.

All threads are executing in the same address space using the same hardware resources, which significantly reduces task switching overhead. Therefore it is important to stop them from causing each other problems. This is particularly an issue where two or more threads need to share a resources like memory locations or peripheral devices.

The system works on the principle that the most urgent thread always runs. One exception to this is if a CPU interrupt arrives and the interrupt has not been disabled. Each thread has a priority which is used to determine how urgent it is. This priority ranges from 0 to 255, with the lowest value indicating the most urgent.

Nut/OS implements cooperative multithreading. That means, that threads are not bound to a fixed timeslice. Unless they are waiting for specific event or explicitely yielding the CPU, they can rely on not being stopped unexpectedly. However, they may be interrupted by hardware interrupt signals. In opposite to pre-emptive multithreading, coorperative multithreading simplifies resource sharing and results in faster and smaller code.

Event Management

Threads may wait for events from other threads or interrupts or may post or broadcast events to other threads.

Waiting threads line up in priority ordered queues, so more than one thread may wait for the same event.

Events are posted to a waiting queue, moving the thread from waiting (sleeping) state to ready-to-run state. A running thread may also broadcast an event to a specified queue, waking up all threads on that queue.

Usually a woken up thread takes over the CPU, if it's priority is equal or higher than the currently running thread. However, events can be posted asynchronously, in which case the posting thread continues to run. Interrupt routines must always post events asynchronously.

A waiting queue is a simple linked list of waiting threads.

Stream I/O

Most C applications make use of the printf library function, which is not available in the AVR GCC environment. Therefore Nut/OS provides its own procedure called NutPrintFormat.

File System

Neither Nut/OS nor Nut/Net require a file system, but Webservers are designed with a file system in mind. To make things easier for the programmer, Nut/OS provides a very simple file system, where files are located in ROM.

Hardware Interrupts

All hardware interrupt vectors of the ATmega CPU point to Nut/OS internal interrupt entries. Device drivers, wether written as Nut/OS extensions or as part of an application must register callback routines by calling **NutRegisterInterrupt**, if they want to handle interrupts.

Chapter 4

Nut/Net

Overview.

Most TCP/IP implementations came from desktop PCs, requirying large code and buffer space. Available memory of embedded systems like the Ethernut board is much smaller. Nut/Net has been specifically designed for small systems.

Although this chapter tries to explain some basics, it makes no attempt to describe all aspects of TCP/IP in full detail. It is assumed that you have a working knowledge of the protocol.

Network Device Initialization

Before using any Nut/Net function, the application must register the network device driver by calling NutRegisterDevice and configure the network interface by calling NutNetAutoConfig. This routine will try to retrieve the local IP address, network mask and default gateway from a DHCP server. If no DHCP server responds within 10 seconds, NutNetAutoConfig uses the previously stored configuration from the on-chip EEPROM of the ATmega CPU. If the EEPROM doesn't contain any address, NutNetAutoConfig will wait for an ICMP packet and use the IP address contained in its header. In this case the netmask will be 255.255.255.0 and no default gateway will be configured. Refer to the hardware manual on how to send this initial ICMP (ping) packet to an Ethernut board.

Applications may also choose to configure a fixed IP address and network mask by calling NutNetIfConfig.

Socket API

On top of the protocol stack Nut/Net provides an easy to use Application Programming Interface (API) based on sockets. A socket can be thought of as a plug socket, where applications can be attached to in order to transfer data between them. Two items are used to establish a connection between applications, the IP address to determine the host to connect to and a port number to determine the specific application on that host.

Because Nut/Net is specifically designed for low end embedded systems, its socket API is a subset of what is typically available on desktop computers and differs in many aspects from the standard Berkely interface. However, programmers used to develop TCP/IP applications for desktop system will soon become familiar with the Nut/Net socket API.

TCP/IP applications take over one of two possible roles, the server or the client role. Servers use a specific port number, on which they listen for connection requests. The port number of clients are automatically selected by Nut/Net.

Nut/Net provides a socket API for the TCP protocol as well as the UDP protocol. The first step to be done is to create a socket by calling NutTcpCreateSocket or NutUdpCreateSocket.

TCP server applications will then call NutTcpAccept with a specific port number. This call will block until a TCP client application tries to connect that port number by calling NutTcpConnect. After a connection has been established, both partners exchange data by calling NutTcpSend and NutTcpReceive.

UDP server applications will provide their port number when calling NutUdpCreateSocket, while UDP client applications pass a zero to this call, in which case Nut/Net selects a port number currently not in use. Data is transfered by calling NutUdpSendTo and NutUdpReceiveFrom. Finally NutUdpDestroySocket may be called to release all memory occupied by the UDP socket structure.

DHCP Protocol

The Dynamic Host Configuration Protocol (DHCP) is based on the UDP protocol and permits to dynamically assign IP addresses when the network is started.

Nut/Net provides a DHCP client, which is automatically invoked when NutNetIfConfig is called with IP address 0.0.0.0. The DHCP client broadcasts requests to the network until an IP address offer is received from a DHCP server. Then the client sends a response telegram to accept the offer and after the DHCP server acknowledges this acceptance, Nut/Net configures the interface with the offered IP address, netmask and optional default gateway.

HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is based on TCP.

TCP Protocol

The Transmission Control Protocol (TCP) is a connection oriented protocol for reliable data transmission. Nut/Net takes care, that all data is transmitted reliable and in correct order. On the other hand this protocol requires more code and buffer space than any other part of Nut/Net.

Applications should use the socket API to make use of the TCP protocol.

UDP Protocol

The advantage of the User Datagram Protocol (UDP) is its reduced overhead. User data is encapsulated in only eight additional header bytes and needs not to be buffered for retransmission. However, if telegrams get lost during transmission, the application itself is responsible for recovery. Note also, that in complex networks like the Internet, packets may not arrive in the same order as they have been sent.

Applications should use the socket API to make use of the UDP protocol.

ICMP Protocol

The Internet Control Message Protocol (ICMP).

Nut/Net automatically responds to an ICMP echo request with an ICMP echo reply, which is useful when testing network connections with a Packet InterNet Groper (PING) program, which is available on nearly all TCP/IP implementations for desktop computers.

IP Protocol

The Internet Protocol (IP).

ARP Protocol

The Address Resolution Protocol (ARP).

Ethernet Protocol

Conversion Functions

If multi-byte values are to be transfered over the network, the most significant byte must always appear first, followed by less sgnificant bytes. This is called the network byte order, which differs from the host byte order, the order how multi-byte values are stored in memory. The compiler (AVR GCC) used to compile Nut/OS stores the least significant bytes first. Several functions are provided to swap bytes from network byte order to host byte order or vice versa.

`htonl` and `htons` convert 4-byte resp. 2-byte values from host to network byte order, while ntohl and ntohs convert 4-byte resp. 2-byte values from network to host byte order.

Another useful conversion is provided by inet_addr and inet_ntoa. While the first converts an IP address from the decimal dotted ASCII representation to a 32-bit numeric value in network byte order, the second procedure offers the reverse function.

Network Buffers

Nut/Net uses a special internal representation of TCP/IP packets, which is designed for minimal memory allocation and copying when packets are passed between layers.

A network buffer structure contains four equal substructures, each of which contains a pointer to a data buffer and the length of that buffer. Each substructure is assiociated to a specific protocol layer, datalink, network, transport and application layer. An additional flag field in the network buffer structure indicates, if the associated buffer has been dynamically allocated.

Network buffers are created and extended by calling NutNetBufAlloc and destroyed by calling NutNetBufFree. When a new packet arrives at the network interface, the driver creates a network buffer with all data stored in the datalink substructure. The Ethernet layer will then split this buffer by simply setting the pointer of the network buffer substructure beyond the Ethernet header and adjusting the buffer lengths before passing the packet to the IP or ARP layer. This procedure is repeated by each layer and avoids copying data between buffers by simple pointer arithmetic.

When application data is passed from the top layer down to the driver, each layer allocates and fills only its specific part of the network buffer, leaving buffers of upper layers untouched. There is no need to move a single data byte of an upper layer to put a lower level header in front of it.

Chapter 5

Creating a simple TCP server

This chapter explains how to use Nut/OS and Nut/Net to create a simple TCP server program.

Initializing the Ethernet Device

As with other Nut/OS applications you need to declare a function named NutMain as a thread, containing an endless loop.

```
#include <sys/thread.h>

THREAD(NutMain, arg)
{
    for(;;) {
    }
}
```

To communicate via Ethernet, you have to initialize the Ethernet hardware. This takes two steps. The first is to register the device. A call to NutRegisterDevice will add all hardware dependant routines and data structures to your final code and inform Nut/OS about the I/O port address and interrupt number to be used.

The Ethernut board is equipped with a Realtek 8019AS Ethernet controller using a base port address of 8300 hex and interrupt 5:

```
NutRegisterDevice(&devEth0, 0x8300, 5);
```

devEth0 is the device information structure of the Realtek device driver. It contains the device name (eth0), the type of this interface (IFTYP_NET) and, among other things, the address of the hardware initialization routine. However, NutRegisterDevice will only set up some data structures but not touch the controller hardware in any way. With network devices, this is done by calling

```
NutNetIfConfig("eth0", mac, 0, 0);
```

The first parameter is the name of the registered device. The second parameter needs some additional attention. It's an array of 6 bytes, containing the MAC address of the Ethernet controller. A MAC address, also referred to as the hardware or Ethernet address is a unique number assigned to every Ethernet node. The upper 24 bits are the manufacturer's ID, assigned by the IEEE Standards Office. The ID of Ethernut boards manufactured by egnite Software GmbH is 000698 hex. The lower 24 bits are the board's unique ID assigned by the manufacturer of the board.

Nut/Net will store this address in EEPROM, but we may also define a static variable to keep it in the application:

```
static u_char mac[] = { 0x00,0x06,0x98,0x09,0x09,0x09 };
```

The two remaining parameters of NutNetIfConfig are used to specify the minimum IP (Internet Protocol) information, the IP address of our node and the network mask. In our example we simply set both to zero, which will invoke a DHCP client to query this information from a DHCP server in the local network. Therefore it may take some seconds until the call returns, depending on the response time of the DHCP server.

If there's no DHCP server available in your network, you must specify these two 4-byte values in network byte order. In this byte order the most significant byte is stored at the first address of a multibyte value, which differs from the byte order used by AVR processors (our host byte order). Fortunately Nut/Net provides a routine named inet_addr, which takes a string containing the dotted decimal form of an IP address and returns the required 4-byte value in host byte order.

If you want to assign IP address 192.168.171.2 with a network mask of 255.255.255.0, call:

```
NutNetIfConfig("eth0", mac, inet_addr("192.168.171.2"),
inet_addr("255.255.255.0"));
```

NutNetIfConfig will initialize the Ethernet controller hardware and should lit the link LED on your board, if it is properly connected to an Ethernet hub or switch. At this point Ethernut will already respond to ARP and ping requests:

```
#include <dev/nicrtl.h>
#include <sys/thread.h>
#include <arpa/inet.h>

static u_char mac[] = { 0x00,0x06,0x98,0x09,0x09,0x09 };

THREAD(NutMain, arg)
{
    NutRegisterDevice(&devEth0, 0x8300, 5);
    NutNetIfConfig("eth0", mac, 0, 0);
    for(;;) {
    }
}
```

Conneting a Client With a Server

Upto now we followed the standard path, common to all TCP/IP applications created for Ethernut. The next task is to create the application specific part.

In order to communicate via TCP, we need a TCP socket, which is actually a data structure containing all information about the state of a connection between two applications.

```
TCPSOCKET *sock;
```

```
sock = NutTcpCreateSocket();
```

This call allocates the data structure from heap memory, initializes it and returns a pointer to the structure. The next step specifies, wether we take over the client or the server part of the communication. As a client we would try to connect to a specified host on a specified port number. Here's an example to connect to port 12345 of the host with the IP address of 192.168.171.1:

```
NutTcpConnect(sock, inet_addr("192.168.171.1"), 12345);
```

In our sample application we decided to take over the server part, which is done by calling

```
NutTcpAccept(sock, 12345);
```

This call will block until a client tries to connect us. As soon as that happens, we can send data to the client by calling NutTcpSend or receive data from the client through NutTcpReceive.

Communicating with the Client

Most application protocols in the Internet environment exchange information by transmitting ASCII text lines terminated by a carriage return followed by a linefeed character. This might not be a big problem while sending data, but it requires some extra effort for incoming data, as arriving segments may contain either a fraction of a line or multiple lines or both. And even sending data becomes more complicated with numeric values, because we need to transfer them to there ASCII representation first.

For stream devices Nut/OS provides a bunch of functions for ASCII data I/O, like NutPrintFormat and NutDeviceGetLine. In order to use them, Nut/Net offers the ability to create a virtual stream device from a TCP socket.

```
NUTDEVICE *sostream;
sostream = NutSoStreamCreate(sock);
```

This device can be used like any other Nut/OS stream device and simplifies line oriented data I/O.

The first thing servers usually do after a client connected them is to send a welcome message:

```
NutPrintString(sostream, "200 Welcome to Ethernut\r\n");
```

Note, that it's a good idea to prepend a numeric code in front of server responses. This way both, a human user as well as a client program can easily interpret the message. On the other hand, the message above occupies 26 bytes of SRAM space. Alternatives are:

```
NutPrintString(sostream, "200 OK\r\n");
```

or

```
NutPrintString_P(sostream, PSTR("200 Welcome to Ethernut\r\n"));
```

In the second example the message is stored in program space and only temporarely copied to SRAM when needed.

In the next step TCP servers typically await a command from the client, perform the associated activity, return a response to the client and await the next command.

Disconnecting

Finally the client will disconnect or, as preferred, send a special command to force the server to disconnect. The server will then call

```
NutSoStreamDestroy(sostream);
```

to release all memory occupied by the virtual stream device and then call

```
NutTcpCloseSocket(sock);
```

to terminate the socket connection. Next, the server may create a new socket and wait for a new client connection.

Trying the Sample Code

The complete source code of a TCP server example can be found in subdirectory app/tcps of your installation directory. If there's no DHCP server in your local network, you need to modify the call to NutNetIfConfig in the C source file named tcps.c as explained above.

Note, that your PC and the Ethernut board should use the same network mask but different IP addresses. And they must reside in the same network, unless you add specific routes to the Nut/Net routing table. However, IP routing might get rather complex and is beyond the scope of this manual. You might refer to a good book explaining that matter.

Most local networks are configured as class C, which means, that a maximum of 254 different IP addresses are available and the IP network mask is specified as 255.255.255.0. All hosts in this network must have equal numbers in the first three parts of their IP addresses. In this case 192.168.171.1 and 192.168.171.5 belong to the same network, but 192.168.171.1 and 192.168.181.5 don't.

After you have done the changes, open a Linux console or DOS prompt with the GCC environment and enter

```
make install
```

This will create an updated binary file named tcps.rom, located in subdirectory bin.

If your local network supports DHCP, you may use the precompiled binary for a first try, but may later modify the default MAC address.

After programming your Ethernut board with this binary, open a Linux console or DOS prompt window and type

```
telnet x.x.x.x 12345
```

replacing x.x.x.x with the IP address of your Ethernut board. The telnet window should display

```
200 Welcome to tcps. Type help to get help.
```

You may now enter any of the following commands using lower case letters:

memory	Displays the number of available SRAM bytes.
threads	Lists all created threads.
timers	Lists all running timers.

Chapter

6

Creating a Webserver

This chapter explains additional utility functions to help you creating an embedded webserver application.

In fact the previous chapter provides everything to create any type of TCP server, which includes a server talking the HTTP protocol. Generally speaking, HTTP defines the ASCII text line to be exchanged between a webserver and a client, the so called Webbrowser.

Nut/Net provides a bunch of routines, which reduces the development effort of creating an embedded webserver to a few API calls.

Initializing and Connecting

The task of initializing the Ethernet hardware and creating a TCP socket remains the same with every TCP server and is not explained any further. If not familiar with this part, please refer to the previous chapter.

Two specific initializations may be done for a webserver application, registering CGI functions and registering authentication, which are explained later.

Serving Clients

After a connection has been established by NutTcpAccept on port 80, the well known port for HTTP servers, a virtual stream device must be created from the TCP socket by calling NutSoStreamCreate.

Finally a simple call to NutHttpProcessRequest does it all. It receives HTTP requests, parses them, checks client authorizations, sends requested documents, calls registered CGI functions or sends error responses. Here's all the code needed to build an Ethernut webserver.

```
#include <dev/nicrtl.h>
#include <sys/thread.h>
#include <sys/timer.h>
#include <sys/print.h>
#include <netinet/sostream.h>
```

```
#include <arpa/inet.h>
#include <pro/httpd.h>

static u_char mac[] = { 0x00,0x06,0x98,0x09,0x09,0x09 };

THREAD(NutMain, arg)
{
    TCPSOCKET *sock;
    NUTDEVICE *sostream;

    NutRegisterDevice(&devEth0, 0x8300, 5);
    NutNetIfConfig("eth0", mac, 0, 0);
    for(;;) {
        sock = NutTcpCreateSocket();
        NutTcpAccept(sock, 80);
        sostream = NutSoStreamCreate(sock);
        NutHttpProcessRequest(sostream);
        NutSoStreamDestroy(sostream);
        NutTcpCloseSocket(sock);
    }
}
```

However, as long as no documents are available and no CGI functions are registered, the webserver will respond to any request with

```
404 Not found
```

Creating Documents

The communication between a webserver and its clients may be reduced to two simple functions:

1. Client requests a document

2. Server sends the document

Normal webservers running on a computer with mass storage devices will use the file system to retrieve the requested document. To offer similar functionality, Nut/OS contains a very simple file system based on data structures stored in flash memory. A command line utility named crurom is provided to convert files from a subdirectory on your PC harddisk to a C source code file. This file contains the file structure used by Nut/OS can be linked to application.

It is far beyond the scope of this manual to explain the creation of HTML documents, animated images and all the other cool stuff available in The Web. If not familiar with the basics, there are far too many books covering them.

To get your webpages on the Ethernut board, create a new subdirectory containing all your web documents and possibly other sub-subdirectories, as you would with any other webserver. But keep in mind, that the available space is very limited. The good news is, that your webpages require only flash ROM space, no SRAM memory. With typical applications at least 64 kByte should be available.

When you've done your documents, simply run the crurom utility, redirecting its output into a file. Supposing your web documents are located in a directory name /nutdevel/nutdocs and your webserver source code is in /nutdevel/httpd, type the following in a Linux console window:

```
cd /utdevel/nutdocs
crurom > ../httpd/urom.c
```

You can use the same commands in a DOS prompt window, but may change to the proper drive first and replace all slashes with backslashes. If the crurom utility is not placed in a directory with your path, you must enter the complete pathname.

It's essential to redirect the output to a file outside the current subdirectory tree. Otherwise this file would be included into the file system, which is most probably not what you want.

The resulting file urom.c or whatever name you choose for output redirection will contain C source code, which is normally compiled and liniked to your application as any other source code file.

Creating CGI Funktions

The Common Gateway Interface is a way for interfacing applications with webservers. If you register a CGI function by calling

```
NutRegisterCgi("biton.cgi", SwitchBitOn);
```

Nut/Net will call a function named SwitchBitOn if the browser requests the URL cgi-bin/binon.cgi. Note, that Nut/Net will not send any response to the client if the SwitchBitOn function returns zero. However, if the function returns -1, Nut/Net will send an internal error page to the browser.

The function must be part of your application and must create the normal respone to the client. NutNet calls this function with two parameters:

```
ShowQuery(NUTDEVICE *sostream, REQUEST *req);
```

The first parameter is the virtual stream device, which your application passed to NutHttpProcessRequest. The second parameter is a pointer to a REQUEST structure containing all information about the client request.

Furthermore, Nut/Net provides two helper function to return a standard HTTP response header to the client, NutHttpSendHeaderTop and NutHttpSendHeaderBot. The content of the returned document can be send to the client using the standard NutPrint... functions. This way the document is created dynamically, which offers another advantage of CGI functions.

Restricting Access

Calling NutRegisterAuth provides a way to restrict access to certain subdirectories. To protect a directory named nonpub, you should call

```
NutRegisterAuth("nonpub", "user:pass");
```

If a browser tries to requests any document from this directory, the user is prompted to enter a username and a password. With the example above the required user name is user and the password is pass.

Trying the Sample Code

The complete source code of a sample HTTP server can be found in subdirectory app/httpd of your installation directory. If there's no DHCP server in your local network, you need to modify the call to NutNetIfConfig in the C source file named nutmain.c as explained in the previous chapter.

Please do also refer to previous chapters on how to compile the program and burn it into the Ethernut board.

The micro-ROM filesystem contains a simple Flash animation and the source code contains a very simple CGI function. They can be invoked by querying the following two URLs with your browser:

```
http://x.x.x.x/index.html
```

```
http://x.x.x.x/cgi-bin/text.cgi
```

where x.x.x.x must be replaced by the IP address of your Ethernut board.

Chapter 7

Data Structures

Nut/OS and Nut/Net data structure layouts.

EEPROM Contents

The first 64 bytes of the ATmega on-chip EEPROM is reserved by Nut/OS, the following 64 bytes by Nut/Net. The remaining 3968 bytes are available for applications.

First	**Last**	**Bytes**	**Description**
0000	0000	1	Size of the Nut/OS configuration structure
0001	0002	2	Magic cookie
0003	0039	61	Reserved
0040	0040	1	Size of the Nut/Net configuration structure
0041	0049	9	Name of the network device
004A	004F	6	Ethernet MAC address
0050	0053	4	Previously used IP address
0054	0057	4	IP netmask
0058	005B	4	Default route gateway IP address
005C	005F	4	Configured IP address
0060	0080	32	Reserved

Chapter

8

Frequently Answered Questions

The Nut/OS FAQ.

Licence

Q: Did I get this right? Can I copy Ethernut for commercial products without paying royality?

A: Yes. Schematics and board layout may be used in private or commercial products without paying any fee. Although many parts of the software had been taken from other projects, they can be used without paying royality fee and may be re-distributed in binary form with or without source code. But note, that you are not allowed to remove any copyright notices.

Socket API

Q: How to serve more than one client at the same time.

A: Create several threads listening on the same port.

In-System-Programming

Q: AVRISP doesn't work on my NT machine. What's wrong?

A: Note, that Windows NT is not supported by the AVRISP program. A beta version with Windows NT/2000 support is available at www.kanda-systems.com.

Q: Can I use the STK500 to burn Ethernuts?

A: We thought yes, but William D. Carroll wrote: "For the STK500 the 10-Pin ISP socket does not work as expected. Your circuit uses pin 3 on the ISP to control the 4053 and select either the ISP socket or the RS232 interface. Pin 3 on the STK500 is a NC pin and therefore carries no signal. To get around this here is what I did using the supplied 2-pin cable in the STk500 Kit connect ISP Pins 1-2,5-6,7-8 and 9-10 to EtherNut ISP Pins 1-2,5-6,7-8 and 9-10 and place a jumper on the ethernut ISP between pins 2-3 this will allow you to program and verify and read the ATMega103."

Chapter

9

Reference Material

Books

Comer D. Internetworking with TCP/IP, Vol I: Principles, Protocols, and Architecture, Prentice Hall

Covers many protocols, including IP, UDP, TCP, and gateway protocols. It also includes discussions of higher level protocols such as FTP, TELNET and NFS.

Comer D., Stevens D. Internetworking with TCP/IP, Vol II: Design, Implementation and Internals, Prentice Hall

Discusses the implementation of the protocols with many code examples.

Comer D., Stevens D. Internetworking with TCP/IP, Vol III: Client-Server Programming and Applications, Prentice Hall

Discusses application programming using the internet protocols. It includes examples of telnet, ftp clients and servers.

Stevens W. TCP/IP Illustrated Vol 1, Addison-Wesley

One of if not the most recommended introduction to the entire TCP/IP protocol suite, covering all the major protocols and several important applications.

Stevens W. TCP/IP Illustrated Vol 2, Addison-Wesley

Discusses the internals of TCP/IP based on the Net/2 release of the Berkely System.

Stevens W. TCP/IP Illustrated Vol 3, Addison-Wesley

Covers some special topics of TCP/IP.

RFCs

RFCs (Request For Comment) are documents that define the protocols used in the Internet. Some are standards, others are suggestions or even jokes. Many Internet sites offer them for download via http or ftp.

Postel, Jon, RFC768: User Datagram Protocol

Postel, Jon, RFC791: Internet Protocol

Postel, Jon, RFC792: Internet Control Message Protocol

Postel, Jon, RFC793: Transmission Control Protocol

Plummer, D.C, RFC826: Ethernet Address Resolution Protocol

Braden, R.T, RFC1122: Requirements for Internet Hosts - Communication Layers

T. Berners-Lee, R. Fielding, and H. Frystyk, RFC1945: Hypertext Transfer Protocol

Web Links

http://www.ethernut.de Ethernut support

http://ethernut.sourceforge.net Ethernut developer's forum

http://www.opticompo.com Ethernut Online Shop

http://www.atmel.com Manufacturers of AVR microcontrollers

http://www.avrfreaks.org Lots of useful infos about AVR microcontrollers

http://nav.webring.yahoo.com/hub?ring=avr Many links to AVR specific information

Chapter

10

Index