Nut/OS Software Manual

Manual Revision: 2.8

Issue date: July 2009

Table of Contents

About Nut/OS and Nut/Net

Nut/OS Features

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

- Open Source
- Modular design
- Highly portable (AVR and ARM available, more to come)
- Cooperative multithreading
- Event queues
- Dynamic memory management
- Timer support
- Stream I/O functions
- Expandable device driver interface
- File system support

Nut/Net Features

Nut/Net is a TCP/IP stack providing

- Open Source
- ARP, IP, UDP, ICMP and TCP protocol over Ethernet and PPP
- Automatic configuration via DHCP
- HTTP API with file system access, CGI, SSI and ASP functions
- SNTP, DNS, Syslog, TFTP, FTP and more
- TCP and UDP socket API for other protocols

What to do first?

Installing the Nut/OS development environment is done in three steps.

We will first install the cross compiler and related tools, also known as the toolchain. This cross toolchain allows us to create program binaries for our target CPU. Windows users developing for AVR targets may choose between the commercially supported ImageCraft IDE or the free GNU Compiler Collection. For all other hosts and targets the GNU Compiler is the only choice.

In a second step we will install the Nut/OS distribution. For Linux we use the source code package, which requires a few more tools. This should work as well for Mac OS X. Windows users will directly install the binaries.

In a last step we will configure and build the Nut/OS libraries, using the tools we installed in the first two steps.

After all three steps have been successfully passed, we are ready to create Nut/OS applications.

Installing a Cross Toolchain

A cross compiler allows to create executable code on one computer platform (host), which is different from the computer (target) that will finally run the code. In our case, we will use a desktop computer under Linux, Mac OS X or Windows to create binaries for running on an embedded system under Nut/OS.

Although the compiler is the main tool, more is actually required, like a preprocessor, an assembler, a linker, a librarian, and several other utilities. Such a collection of tools is also called a toolchain or, when including a cross compiler, a cross toolchain.

Nut/OS applications may be built with almost any cross toolchain that offers ISO C compatibility. The free GNU Compiler Collection (GCC) and the commercial ImageCraft for AVR (ICC AVR) are officially supported. The following table provides an overview about which toolchain can be used on a specific host operating system for a specific target platform.

	Windows	Linux	OS X
AVR	WinAVR (GCC) ImageCraft AVR	AVR GCC	AVR GCC
ARM	YAGARTO (GCC)	ARM-ELF GCC	YAGARTO (GCC)
AVR32	AVR32 GCC	AVR32 GCC	Not available

Installing WinAVR

WinAVR (pronounced "whenever") is a suite of executable, open source software development tools for the AVR, hosted on the Windows platform. They are mainly based on the GCC for AVR toolchain and are quite similar to the Linux tools.

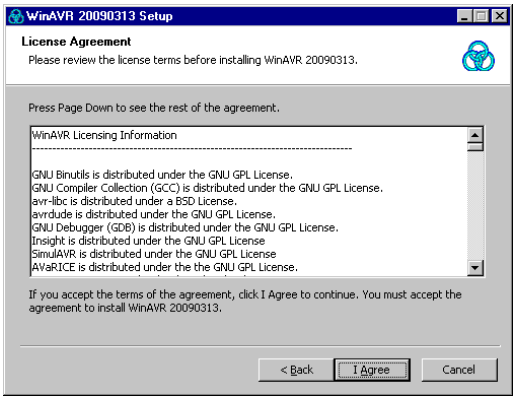
Get the latest WinAVR installer executable from the Ethernut development DVD or download it from

```
http://winavr.sourceforge.net
```

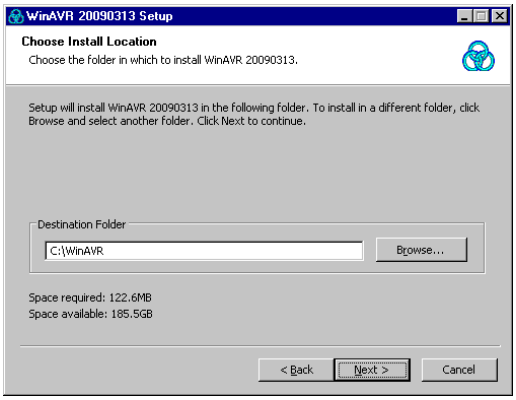
After starting the installer you will be guided through the installation steps. First select the language of the installer.

The next dialog displays some general information. Click *Next* to continue.

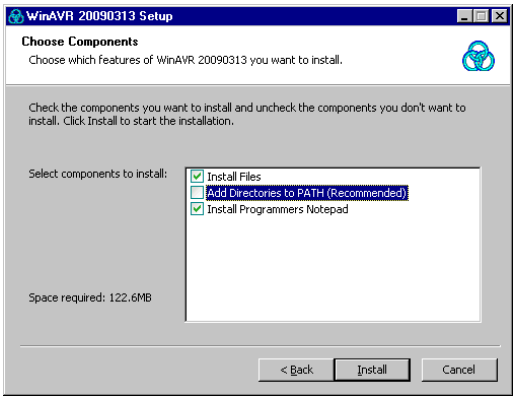
In the following window click *I Agree*, if you accept the displayed licenses.



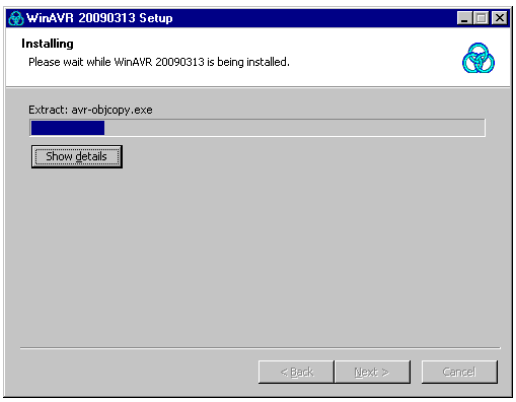
Now choose the installation folder. You can either enter the path manually or press the *Browse* button to use a folder selection dialog. Make sure that the directory name doesn't contain spaces. If possible, you should stick with the default *C:\WinAVR.* When done, click *Next*.



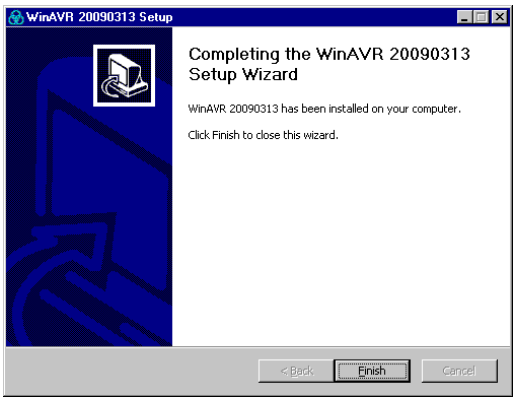
In the following dialog you can deselect various options. Although marked *Recommended* you may prefer not to have WinAVR directories added to your PATH environment. Nut/OS allows to add them to the settings later. This provides more control, especially when switching platforms. Click on *Install* to start the installation.



The status of the installation progress is displayed. When completed, click on the *Next* button.



Clicking the *Finish* button in the final dialog will start your default web browser to display a short manual.

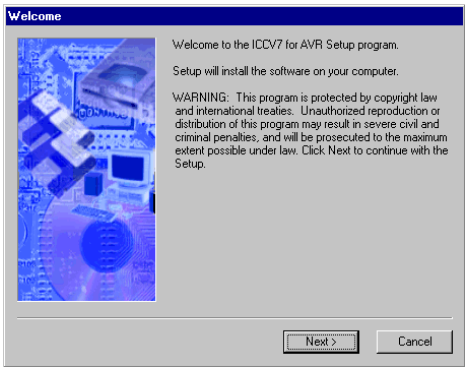


Installing ImageCraft AVR

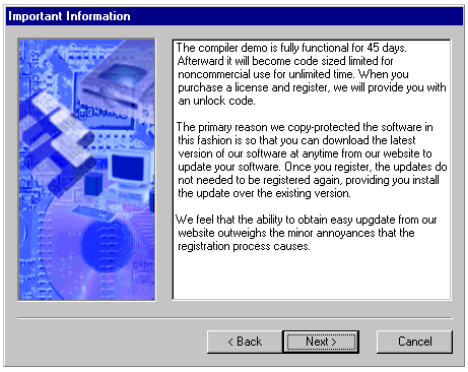
ImageCraft Inc. offers an ANSI C cross compiler IDE with strong commercial support. They sell three different product editions, standard, advanced and professional. All three editions will work for Nut/OS. A free demo is available too, which is mainly limited in code size and may not work with all Nut/OS applications.

This guide refers to ICCAVR version 7. Version 6 may still work with Nut/OS, but its support will likely be abandoned in the near future.

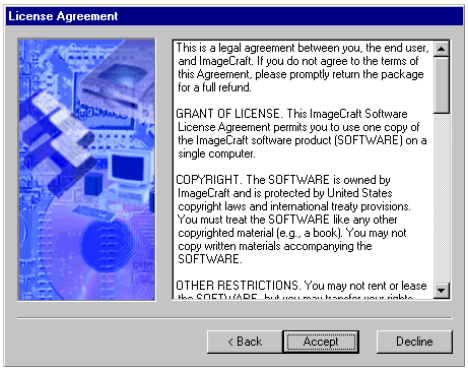
The first dialog presents you with some general information. Click *Next* to continue.



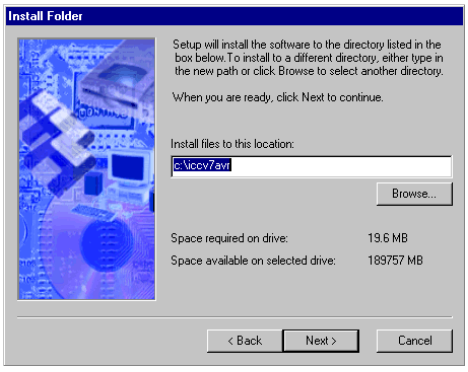
If you are installing a demo version, carefully read the text in this dialog. Click *Next* to continue.



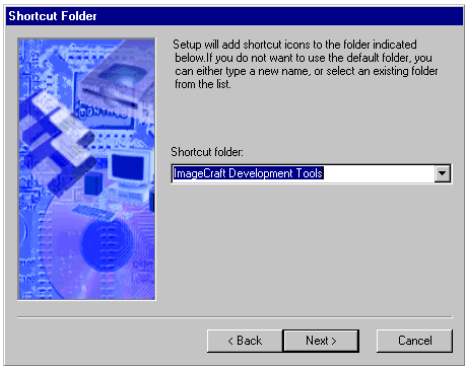
The following window displays the license. Click *Accept* to agree.



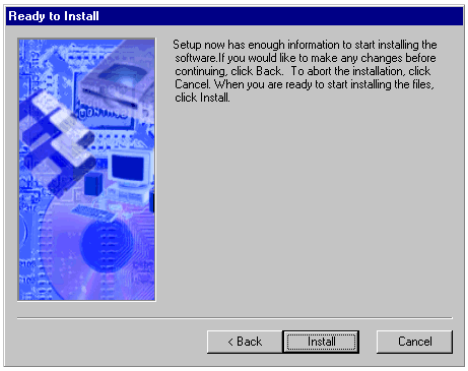
Now choose the installation folder. You can either enter the path manually or press the *Browse* button to use a folder selection dialog. Make sure that the directory name doesn't contain spaces. If possible, you should stick with the default *C:\icc7avr.* When done, click *Next*.



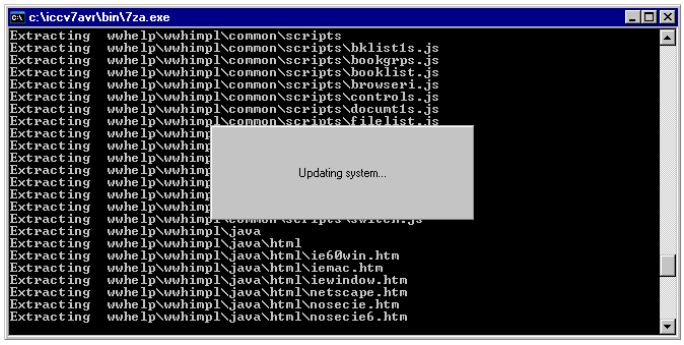
On the next dialog you are able to change the start menu entry. Simply click *Next* to keep the default.



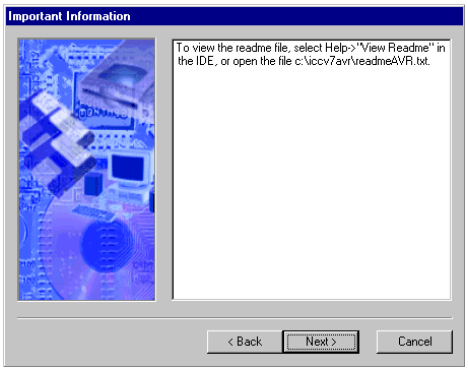
Clicking the *Install* button will start the installation.



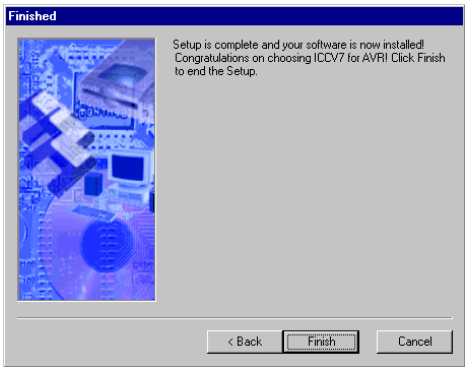
The status of the installation progress is displayed. A command line window may temporarily appear.



When the installation process is completed, another dialog displays additional information about the current version.



On the final dialog simply click the *Finish* button.



Installing AVR GCC on Debian Linux

Binary packages are available on Debian Linux, including Ubuntu. To install, simply enter the following command in a shell window:

```
sudo apt-get install binutils-avr gcc-avr avr-libc avrdude
```

The following packages are required to build and install an AVR cross compiler toolchain on Linux:

- GNU AVR Binutils
- GNU AVR Compiler Collection
- AVR Libc
- avrdude

Detailed information is available at

```
www.nongnu.org/avr-libc/user-manual/install_tools.html
```

Installing AVR GCC on other Linux distributions

If no binary package is available for your specific Linux distribution, you can build the toolchain from the source code.

In a first step we need to make sure that all required build tools are available. The following shell command will install any missing parts.

```
sudo apt-get install build-essential autoconf automake
```

Next create a directory where we will build the toolchain and change to this directory.

```
mkdir ~/toolchain-avr
cd ~/toolchain-avr
```

Get the source packages:

```
wget ftp://gcc.gnu.org/pub/binutils/releases/binutils-2.19.1.tar.bz2
wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-core-4.3.3.tar.bz2
wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-g++-4.3.3.tar.bz2
wget http://download.savannah.gnu.org/releases-noredirect/avr-libc/avr-libc-1.6.7.tar.bz2
wget http://download.savannah.gnu.org/releases-noredirect/avrdude/avrdude-5.7.tar.gz
```

Unpack the packages:

```
tar -xjf binutils-2.19.1.tar.bz2
tar -xjf gcc-core-4.3.3.tar.bz2
tar -xjf gcc-g++-4.3.3.tar.bz2
tar -xjf avr-libc-1.6.7.tar.bz2
tar -xzf avrdude-5.7.tar.gz
```

Now we set two useful environment variables and create a bin/ folder in advance, which we add to the PATH variable:

```
TARGET=avr
export TARGET
PREFIX=/usr/local/$TARGET
export PREFIX
sudo mkdir -p $PREFIX/bin
PATH=$PATH:$PREFIX/bin
export PATH
```

Start with building and installing the binutils:

```
cd binutils-2.19.1
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-languages=c,c++ \
    --disable-nls --disable-libssp --with-dwarf2
make
sudo make install
cd ../..
```

Next, build and install the compiler:

```
cd gcc-4.3.3
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-languages=c,c++ \
    --disable-nls -disable-libssp --with-dwarf2
make
sudo make install
cd ../..
```

To build and install the runtime library, use:

```
cd avr-libc-1.6.7
mkdir build-$TARGET
cd build-$TARGET/
../configure --prefix=$PREFIX -build=`../config.guess` --host=$TARGET
make
sudo make install
cd ../..
```

Finally AVRDUDE can be installed with the following commands:

```
cd avrdude-5.7.tar.gz
mkdir build-$TARGET
cd build-$TARGET
../configure --prefix=$PREFIX
make
sudo make install
```

In an additional step we may strip the installed binaries to save disk space and decrease load times:

```
sudo strip $prefix/bin/*
sudo strip $prefix/$target/bin/*
sudo strip $prefix/libexec/gcc/$target/4.3.3/*
```

Installing AVR GCC on OS X

Apple's Xcode package contains all the tools we need to build the cross toolchain. Open Terminal and enter

```
gcc -version
```

If the response is similar to

```
powerpc-apple-darwin9-gcc-4.0.1 (GCC) 4.0.1 (Apple Inc. build 5465)
Copyright (C) 2005 Free Software Foundation, Inc.
```

then GCC is already available and probably there is no need to install it again. However, if the response is

```
-bash: gcc: command not found
```

then you need to download Xcode from

developer.apple.com/tools/xcode/

A registration is required, but this is free of charge.

Leave the Terminal window open while installing Xcode.

After having installed Xcode, go back to the terminal to download the source packages:

```
curl -O ftp://gcc.gnu.org/pub/binutils/releases/binutils-2.19.1.tar.bz2
curl -O ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-core-4.3.3.tar.bz2
curl -O ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-g++-4.3.3.tar.bz2
curl -O http://download.savannah.gnu.org/releases-noredirect/avr-libc/avr-libc-1.6.7.tar.bz2
curl -O http://download.savannah.gnu.org/releases-noredirect/avrdude/avrdude-5.7.tar.gz
```

Unpack the packages:

```
tar -xjf binutils-2.19.1.tar.bz2
tar -xjf gcc-core-4.3.3.tar.bz2
tar -xjf gcc-g++-4.3.3.tar.bz2
tar -xjf avr-libc-1.6.7.tar.bz2
tar -xzf avrdude-5.7.tar.gz
```

Now we set two useful environment variables and create a bin/ folder in advance, which we add to the PATH variable:

```
TARGET=avr
export TARGET
PREFIX=/usr/local/$TARGET
export PREFIX
sudo mkdir -p $PREFIX/bin
PATH=$PATH:$PREFIX/bin
export PATH
```

Start with building and installing the binutils:

```
cd binutils-2.19.1
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-languages=c,c++ \
    --disable-nls --disable-libssp --with-dwarf2
make
sudo make install
cd ../..
```

Next, build and install the compiler:

```
cd gcc-4.3.3
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-languages=c,c++ \
    --disable-nls -disable-libssp --with-dwarf2
make
sudo make install
cd ../..
```

To build and install the runtime library, use:

```
cd avr-libc-1.6.7
mkdir build-$TARGET
cd build-$TARGET/
../configure --prefix=$PREFIX -build=`../config.guess` --host=$TARGET
make
sudo make install
cd ../..
```

Finally AVRDUDE can be installed with the following commands:

```
cd avrdude-5.7.tar.gz
mkdir build-$TARGET
cd build-$TARGET
../configure --prefix=$PREFIX
make
sudo make install
```

In an additional step we may strip the installed binaries to save disk space and decrease load times:

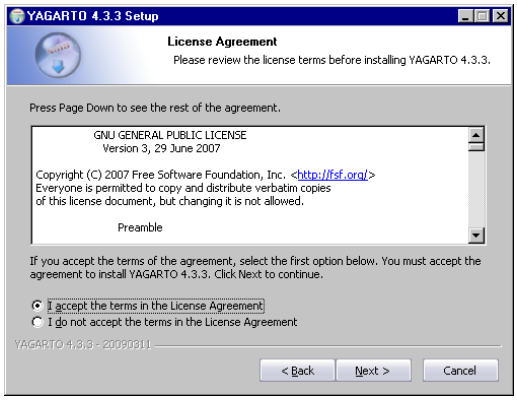
```
sudo strip $PREFIX/bin/*
sudo strip $PREFIX/$TARGET/bin/*
sudo strip $PREFIX/libexec/gcc/$TARGET/4.3.3/*
```

Installing YAGARTO on Windows

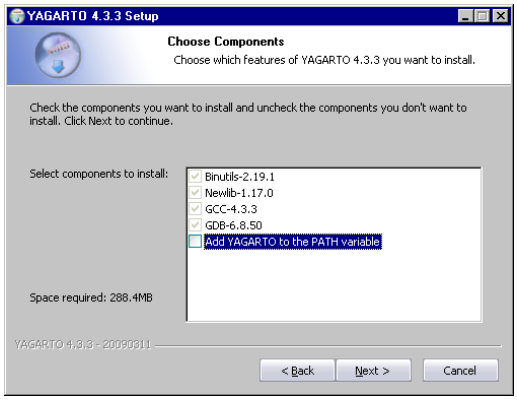
YAGARTO (Yet Another Gnu ARm TOolchain) is a GNU compiler toolchain for ARM CPUs. Being based on MinGW (Minimalist GNU for Windows), it runs efficiently and without hassle on a Windows PC.

The first dialog displays some general information. Click *Next* to continue.

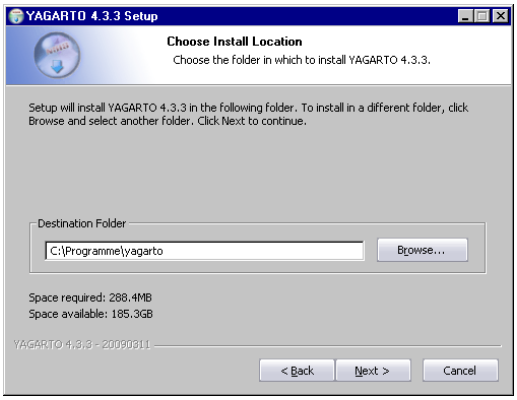
In the following window you need to accept the license. Click the *Next* button when done.



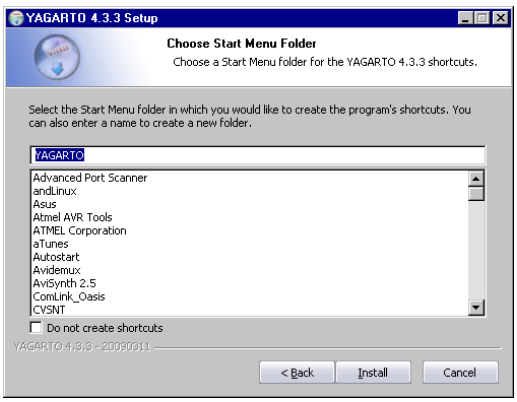
Although checked by default you may prefer not to have YAGARTO directories added to your PATH environment. Nut/OS allows to add them to the settings later. This provides more control, especially when switching platforms.



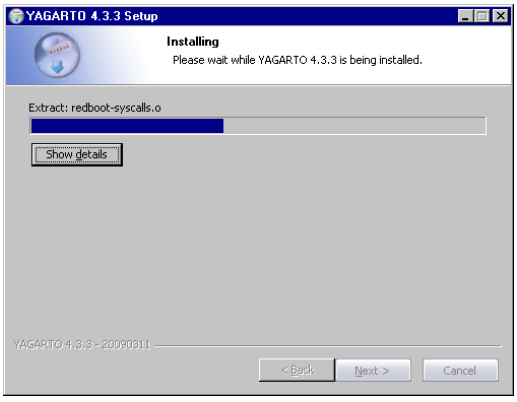
Now choose the installation folder. You can either enter the path manually or press the *Browse* button to use a folder selection dialog. Make sure that the directory name doesn't contain spaces. On English language systems you should always avoid *Program Files* when using tools that have been derived from Unix-like systems. When done, click *Next*.



Click *Install* to start the installation.



The status of the installation progress is displayed. When completed, click the *Next* button.



Clicking the *Finish* button will close the installer.

Installing ARM-ELF GCC on Linux

If no binary package is available for your specific Linux distribution, you can build the toolchain from the source code.

In a first step we need to make sure that all required build tools are available. The following shell command will install any missing parts.

```
sudo apt-get install build-essential autoconf automake
```

Next create a directory where we will build the toolchain and change to this directory.

```
mkdir ~/toolchain-avr
cd ~/toolchain-avr
```

Get the source packages:

```
wget ftp://gcc.gnu.org/pub/binutils/releases/binutils-2.19.1.tar.bz2
wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-core-4.3.3.tar.bz2
wget ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-g++-4.3.3.tar.bz2
wget ftp://sources.redhat.com/pub/newlib/newlib-1.16.0.tar.gz
```

Unpack the packages:

```
tar -xjf binutils-2.19.1.tar.bz2
tar -xjf gcc-core-4.3.3.tar.bz2
tar -xjf gcc-g++-4.3.3.tar.bz2
tar -xzf newlib-1.16.0.tar.gz
```

Now we set two useful environment variables and create a bin/ folder in advance, which we add to the PATH variable:

```
TARGET=arm-elf
export TARGET
PREFIX=/usr/local/$TARGET
export PREFIX
sudo mkdir -p $PREFIX/bin
PATH=$PATH:$PREFIX/bin
export PATH
```

Start with building and installing the binutils:

```
cd binutils-2.19.1
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-interwork \
    --enable-multilib --disable-nls --disable-shared -disable-threads \
    --with-gcc --with-gnu-as --with-gnu-ld
make
sudo make install
cd ../..
```

Now an intermediate bootstrap compiler is required, which enables us to create the runtime library:

```
cd gcc-4.3.3
mkdir build-$TARGET
cd build-$TARGET/
sudo ../configure --target=$TARGET --prefix=$PREFIX \
    --disable-nls --disable-shared --disable-threads \
    --with-gcc --with-gnu-ld --with-gnu-as --with-dwarf2 \
    --enable-languages="c,c++" --enable-interwork \
```

```
    --enable-multilib --with-newlib \
    --with-headers=../../newlib-1.16.0/newlib/libc/include \
    --disable-libssp --disable-libstdcxx-pch \
    --disable-libmudflap --disable-libgomp -v
mkdir -p libiberty libcpp fixincludes
make all-gcc
sudo make install-gcc
cd ../..
```

With this compiler you can create the runtime library. We need to tell sudo to use our PATH because make install requires our fresh installed arm-elf-ranlib:

```
$ cd newlib-1.16.0
$ mkdir build-$target
$ cd build-$target/
$ ../configure --target=$target --prefix=$prefix --enable-interwork \
> --enable-multilib
$ make
$ sudo PATH=$PATH make install
$ cd ../..
```

Finally, build and install the cross compiler:

```
cd gcc-4.3.3/build-$TARGET
make
sudo make install
cd ../..
```

In an additional step we may strip the installed binaries to save disk space and decrease load times:

```
sudo strip $PREFIX/bin/*
sudo strip $PREFIX/$TARGET/bin/*
sudo strip $PREFIX/libexec/gcc/$TARGET/4.3.3/*
```

Installing YAGARTO on OS X

Ready built binaries of the GCC ARM-ELF toolchain are available at

```
http://www.yagarto.de/
```

Installing ARM-ELF GCC on OS X

If YAGARTO fails to install or if you want to try a different version of any of the utilities included in the YAGARTO distribution, you can install the toolchain from the source code.

Apple's Xcode package contains all the tools we need to build the cross toolchain. Open Terminal and enter

```
gcc –version
```

If the response is similar to

```
powerpc-apple-darwin9-gcc-4.0.1 (GCC) 4.0.1 (Apple Inc. build 5465)
Copyright (C) 2005 Free Software Foundation, Inc.
```

then GCC is already available and probably there is no need to install it again. However, if the response is

```
-bash: gcc: command not found
```

then you need to download Xcode from

```
http://developer.apple.com/tools/xcode/
```

A registration is required, but this is free of charge.

Leave the Terminal window open while installing Xcode.

After having installed Xcode, go back to the terminal to download the source packages:

```
curl -O ftp://gcc.gnu.org/pub/binutils/releases/binutils-2.19.1.tar.bz2
curl -O ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-core-4.3.3.tar.bz2
curl -O ftp://gcc.gnu.org/pub/gcc/releases/gcc-4.3.3/gcc-g++-4.3.3.tar.bz2
curl -O ftp://sources.redhat.com/pub/newlib/newlib-1.16.0.tar.gz
```

Unpack the packages:

```
tar -xjf binutils-2.19.1.tar.bz2
tar -xjf gcc-core-4.3.3.tar.bz2
tar -xjf gcc-g++-4.3.3.tar.bz2
tar -xzf newlib-1.16.0.tar.gz
```

Now we set two useful environment variables and create a bin/ folder in advance, which we add to the PATH variable:

```
TARGET=arm-elf
export TARGET
PREFIX=/usr/local/$TARGET
export PREFIX
mkdir -p $PREFIX/bin
PATH=$PATH:$PREFIX/bin
export PATH
```

Start with building and installing the binutils:

```
cd binutils-2.19.1
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX --enable-interwork \
    --enable-multilib --disable-nls --disable-shared -disable-threads \
    --with-gcc --with-gnu-as --with-gnu-ld
make
make install
cd ../..
```

Now an intermediate bootstrap compiler is required, which enables us to create the runtime library:

```
cd gcc-4.3.3
mkdir build-$TARGET
cd build-$TARGET/
../configure --target=$TARGET --prefix=$PREFIX \
    --disable-nls --disable-shared --disable-threads \
    --with-gcc --with-gnu-ld --with-gnu-as --with-dwarf2 \
    --enable-languages="c,c++" --enable-interwork \
    --enable-multilib --with-newlib \
    --with-headers=../../newlib-1.16.0/newlib/libc/include \
    --disable-libssp --disable-libstdcxx-pch \
    --disable-libmudflap --disable-libgomp -v
mkdir -p libiberty libcpp fixincludes
make all-gcc
make install-gcc
cd ../..
```

With this compiler you can create the runtime library:

```
cd newlib-1.16.0
mkdir build-$target
cd build-$target/
../configure --target=$target --prefix=$prefix --enable-interwork \
  --enable-multilib
make
make install
cd ../..
```

Finally, build and install the cross compiler:

```
cd gcc-4.3.3/build-$TARGET
make
make install
cd ../..
```

In an additional step we may strip the installed binaries to save disk space and decrease load times:

```
strip $PREFIX/bin/*
strip $PREFIX/$TARGET/bin/*
strip $PREFIX/libexec/gcc/$TARGET/4.3.3/*
```

Installing AVR32 GCC on Windows and Linux

Support for the AVR32 platform is available since Nut/OS 4.9 and still experimental.

Readily built binaries of the AVR32 GNU toolchain are available for download at

```
http://www.atmel.com
```

Installing Nut/OS

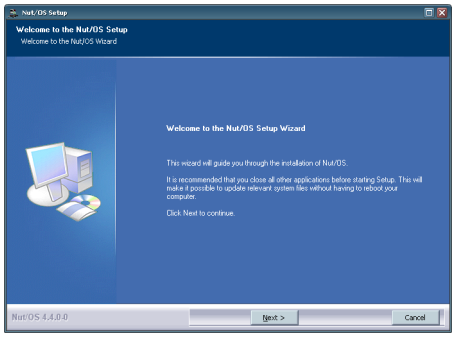
This chapter explains how to install the Nut/OS distribution, either from the source code or the binary package.

Installing Nut/OS on Windows

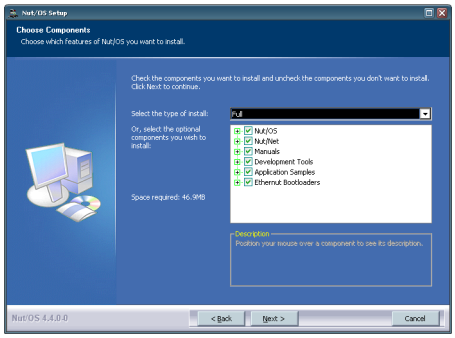
The installation for Windows is packed into a self-extracting executable installer named ethernut-X.Y.Z.exe, where X.Y.Z has to be replaced with the version number. The file contains the complete code, some Nut/OS tools and the API reference.

Not much has to be done for the Windows distribution. Simply download the latest version from the Ethernut project website or run it directly from the Starterkit CD, if available.

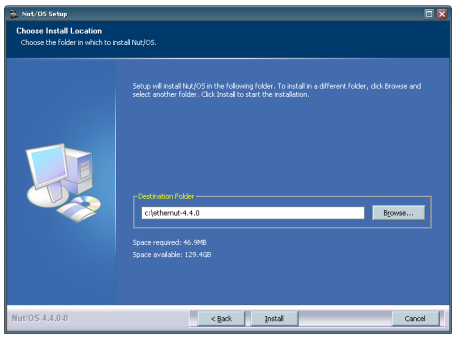
After starting the installer, you can choose the language. The selected language is used during the installation only. All other parts and the Nut/OS documentation are available in English only.



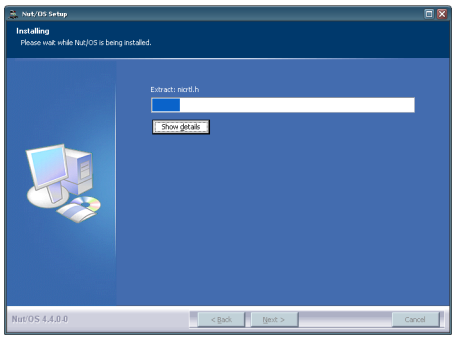
After selecting the installer language and clicking OK, a welcome screen appears. Click *Next* to continue.



The next dialog lists all Nut/OS components, which are included in the installation. If unsure, leave Full selected and click *Next* to continue.

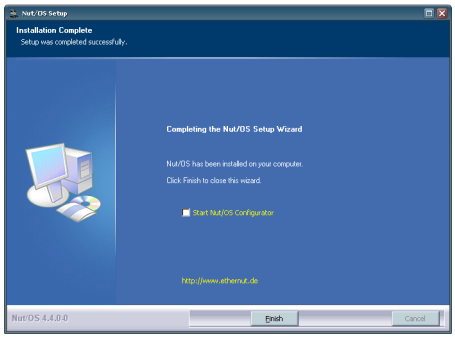


You can now select the installation directory. By default all files will be installed in *C:\ethernut-X.Y.Z*. It is recommended to leave it that way. In any case avoid directory names containing spaces. Click *Install* to start the installation.



The next dialog will show the installation progress. This will take a few seconds. You can click *Show details* to view the list of files being installed.

When all files have been copied to the installation directory, a final dialog appears. Leave the checkbox to *Start Nut/OS Configurator* unchecked. Click *Finish* to close the installation wizard.



Installing Nut/OS on Debian Linux

Unfortunately there is no binary package available yet. The installation for Linux is packed into a source code package named ethernut-X.Y.Z.tar.bz2, where X.Y.Z symbolizes the version number. The file contains the complete code, some Nut/OS tools and the API reference.

Building the full package requires a number of additional tools and libraries. It may fail at various stages and isn't trivial. However, several parts are optional and you should be able to create Nut/OS applications with a subset. The following table lists the main dependencies:

Module	Provisions	Special Requirements
Nut/OS Sources (mandatory)	✔ Target selection via shell script ✔ Configurable by editing C header files ✔ Tool for creating simple file system images	✔ None
CLI Configurator (optional)	✔ Multiple build and application trees ✔ Automated script based target configuration ✔ Batch build	✔ Lua 5.0 libraries
GUI Configurator (optional)	✔ Easy to use configuration interface ✔ Multiple build and application trees ✔ Automated script based target configuration	✔ Lua 5.0 libraries ✔ wxWidgets 2.8 libraries ✔ GNU C++ compiler
GUI Discoverer (optional)	✔ Scans local networks for Nut/OS nodes ✔ Remote configuration of Nut/OS nodes	✔ WxWidgets 2.8 build from source ✔ wxPropertyGrid extension ✔ GNU C++ compiler
API Reference (optional)	✔ HTML documents created from source code	✔ Doxygen 1.4 or later ✔ GraphViz 2.2 or later

We will present the full build here. If something goes wrong, then simply skip that step. The Nut/OS configure script will detect missing parts automatically.

Make sure that the latest build tools available:

```
sudo apt-get install build-essential automake autoconf
```

The Nut/OS Configurator's GUI is based on wxWidgets, which uses GTK+ on Linux systems. wxWidgets is a C++ class library, which allows to create GUI applications that look and feel native. If available, you can simply install the binary packages:

```
sudo apt-get install libwxgtk2.8-0 libwxgtk2.8-dev wx2.8-headers wx-common
```

If binaries are not available, you can build wxWidgets libraries from source code:

```
sudo apt-get install libgtk2.0-dev
wget http://ovh.dl.sourceforge.net/sourceforge/wxwindows/wxGTK-2.8.10.tar.gz
tar xzf wxGTK-2.8.10.tar.gz
cd wxGTK-2.8.10/
mkdir build-gtk2-ansi-static
cd build-gtk2-ansi-static/
../configure –disable-shared
make
sudo make install
```

In the next step we are going to install the Lua language, which is used later by the Nut/OS Configurator to parse the configuration files. Binaries should be available for most Linux distributions:

```
sudo apt-get install lua50 liblua50 liblua50-dev liblualib50 liblualib50-dev
```

Note, that version 5.0 is required, version 5.1 will not work.

Again, this package can be built from source code:

Download lua-5.0.3.tar.gz from www.lua.org and

```
gzip -d downloads/lua-5.0.3.tar.gz
tar xf downloads/lua-5.0.3.tar
cd lua-5.0.3/
```

The Lua source archive doesn't provide a configure script. Instead it comes with a prepared Makefile, which includes a file named config. This file allows to modify several build options. On our system the original file worked fine, but you may at least have a look at the contents. To build and install the Lua tools and libraries, run

```
make
sudo make install
```

If everything worked so far, almost all requirements are fulfilled. The remaining packages are only needed to create the API documentation and can be left out. To install them, use:

```
apt-get install doxygen doxygen-doc graphviz
```

Now get the latest Nut/OS source code package from the Ethernut development CD or download it from

```
http://www.ethernut.de/en/download/
```

In this example we use ethernut-4.8.3.tar.bz2

To keep things simple and similar with Windows installations, create a folder *ethernut* in your home directory. In the following chapters it is assumed that you did so. Copy the package to this directory, then unpack, configure and install it. It's highly recommended to use a symbolic link to the source code directory, as shown below:

```
$ tar xjf ethernut-4.8.3.tar.bz2
$ ln -s ethernut-4.8.3 nut
$ cd nut/
$ ./configure
$ make
$ sudo make install
```

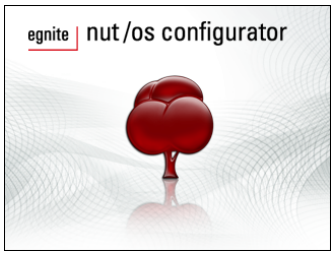
Using the GUI Configurator

On Windows launch the Configurator from the *Windows Start Menu*.

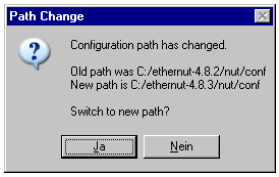
However, if you are using the ImageCraft Demo, then the Configurator must be started from the IDE. Check the accompanying documentation about how to configure an external tool (use *C:\ethernut-X.Y.Z\nutconf.exe* as the *Program* *C:\ethernut-X.Y.Z* as the *Initial Directory*).

Linux and OS X users should open a shell (OS X terminal), change to the parent directory of the Nut/OS installation and execute nutconf, e.g.:

```
cd ~/ethernut
nutconf
```

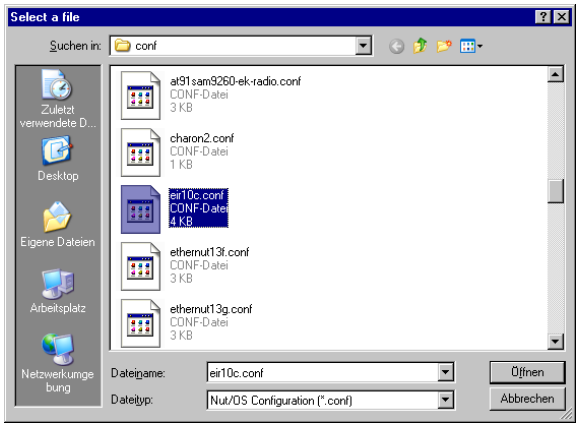


When started, the Configurator's splash window appears. Due to incompatibilities between Windows and GTK the window will not go away automatically. This is usually no big deal, unless you are upgrading from a previous Nut/OS version. In this case the splash window may cover an underlying message box, which is waiting for an input.



Click on the splash window to close it and confirm any path change message by clicking *Yes*.

In the following dialog select the configuration file that is most similar to your board. After clicking *Open*, the selected file will be loaded and the hardware related configuration for this board will be automatically set by the Configurator.

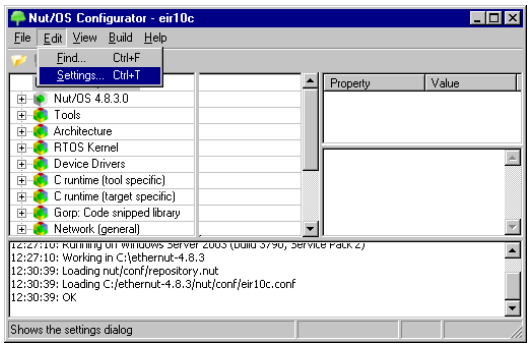


The following table lists the configuration files and the related target hardware.

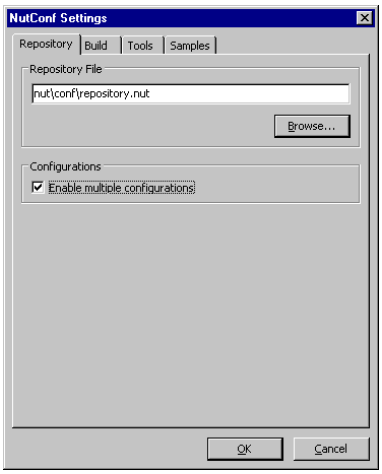
Configuration File	Target Hardware
arthernet1.conf	Arthernet AVR Ethernet Modul - open source project by Guido Fischer
at91eb40a.conf	Atmel AT91EB40A evaluation board with AT91R40008 CPU
at91sam7s.conf	Generic AT91SAM7S board
at91sam7se-ek.conf	Atmel AT91SAM7SE-EK evaluation board
at91sam7x-ek.conf	Atmel AT91SAM7X-EK evaluation board
at91sam9260-ek.conf	Atmel AT91SAM9260-EK evaluation board
charon2.conf	HW group Charon II module
eir10c.conf	Elektor Internet Radio 1.0 Rev-C
ethernut103.conf	Ethernut 1 reference design with Atmega103 CPU
ethernut13f.conf	Ethernut 1.3 Rev-F reference design
ethernut13g.conf	Ethernut 1.3 Rev-G reference design
ethernut13h.conf	Ethernut 1.3 Rev-H reference design
ethernut20a.conf	Ethernut 2.0 Rev-A reference design
ethernut21b.conf	Ethernut 2.1 Rev-B reference design
ethernut30d.conf	Ethernut 3.0 Rev-D reference design
ethernut30e.conf	Ethernut 3.0 Rev-E reference design
ethernut50c.conf	Ethernut 5.0 Rev-C reference design
gbaxport2.conf	Gameboy Advance with CharmedLabs Xport 2
mmnet01.conf	Propox MMnet01 Ethernet Module

Configuration File	Target Hardware
mmnet02_03_04.conf	Propox MMnet01 Ethernet Module
mmnet101.conf	Propox MMnet01 Ethernet Module
mmnet102_103_104.conf	Propox MMnet01 Ethernet Module
olimex-sam7-ex256.conf	Olimex SAM7-EX256 board
stk501.conf	Atmel STK501 evaluation board
xnut-100.conf	proconX XNUT-100 Programmable Gateway
xnut-105c.conf	proconX XNUT-105 Programmable Gateway Rev-C
xnut-105d.conf	proconX XNUT-105 Programmable Gateway Rev-D

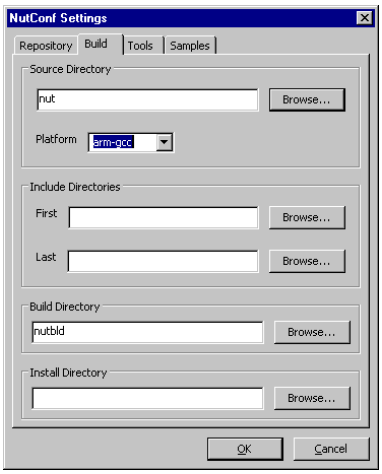
The Configurator's main window is divided into four parts. The Nut/OS module tree on the left side allows to modify the configuration. The upper part on the right side shows specific properties of the currently selected tree item, while additional help is provided in the lower part. A log window available at the bottom may become helpful if things go wrong.



Select *Settings* in the *Edit* menu to enter the Configurator's settings dialog, which is divided into four pages: *Repository, Build, Tools* and *Samples*.



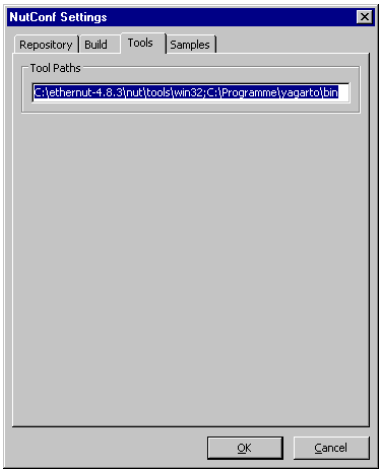
The first page shows the relative path to the repository file. If the main window didn't show the module tree, you may correct this path here and then reload the configuration file. However, most probably you started the Configurator from the wrong directory and this may introduce further problems. Thus, the best way is to exit the Configurator and restart it from the parent directory of nut. If you are concurrently developing for multiple target boards, tick *Enable multiple configurations*. This will link all settings to the configuration file and they don't need to be re-entered when changing the target.



Usually only the *Platform* entry needs to be adjusted on the second page (see table below). The entries for include directories are used when adding custom modules. Header files in the *First* directory can replace existing files, those in the *Last* entry will be added if not found elsewhere. The *Build Directory* will be created when building Nut/OS and the resulting libraries will be placed in the *Install Directory*, which, when left blank, is named lib and located in the build directory. If developing for multiple targets, you may use a more specific name for the *Build Directory*, e.g. *nutbld-enut21b* or *nutbld-eir10c*.

Selected Platform	Use with
arm-gcc	GNU ARM-ELF Toolchain including YAGARTO, recommended for ARM targets
arm-gccdbg	Same as arm-gcc, but includes debug information
avr-gcc	GNU AVR Toolchain including WinAVR, recommended for AVR targets
avr-gccdbg	Same as avr-gcc, but includes debug information
avr-icc	ImageCraft V6 for AVR, do not use this for new projects
avr-icc7	ImageCraft V7 for AVR, recommended for ATmega128 and ICCAVR compiler
avr-iccext	Same as avr-icc, but for ATmega256 targets, not for new projects

Selected Platform	Use with
avr-icc7ext	Same as avr-icc7, recommended for ATmega256 and ICCAVR compiler
avr32-gcc	GNU AVR32 Toolchain (requires Nut/OS 4.9 or later)
avr32-gccdbg	Same as avr32-gcc, but includes debug information
h8-gcc	Lack of maintenance, may no longer work
unix-gcc	Nut/OS emulation on Linux, poorly maintained, but may work



The entry on the third page is required on Windows only and should contain the path to the Nut/OS tools directory. If you deselected the PATH update during compiler installation, then add these paths here. All paths must be separated by semicolons and must not contain spaces. Make sure that the path to the Ethernut tools directory comes first. While based on Linux tools, path components in the Nut/OS configuration are typically separated by slashes. However, in the *Tools Path* you must use backslashes.
On Linux and OS X you should leave this entry empty.

The actual tool path depends on the toolchain and its installation directory. For WinAVR the complete entry may look like this:

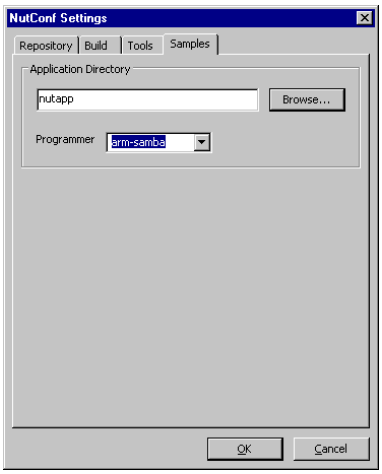
```
C:\ethernut-4.8.3\nut\tools\win32;C:\WinAVR\bin;C:\WinAVR\utils\bin
```

For YAGARTO something like the following should work:

```
C:\ethernut-4.8.3\nut\tools\win32;C:\WinAVR\bin;C:\WinAVR\utils\bin
```

Here's a sample entry for the ImageCraft compiler:

```
C:\ethernut-4.8.3\nut\tools\win32;C:\iccv7avr\bin
```



The items on the last page are used to create an application sample directory. A more advanced application wizard will become available in the future. Since then the Configurator provides the basic functions. Note, that the application directory is tightly linked to the build directory. Thus, if you changed the name of your build directory to *nutbld-eir10c* for example, then you should give the application directory a similar name, like *nutapp-eir10c*.
The selection of a *Programmer* is optional (see table below).
Finally click *OK* to store the settings.

When the correct *Programmer* is selected from the drop down list, applications may later be built and uploaded to the target board in a single step by entering

```
make clean all burn
```

on the command line. The selection mainly depends on the programming software tool and the following options are available:

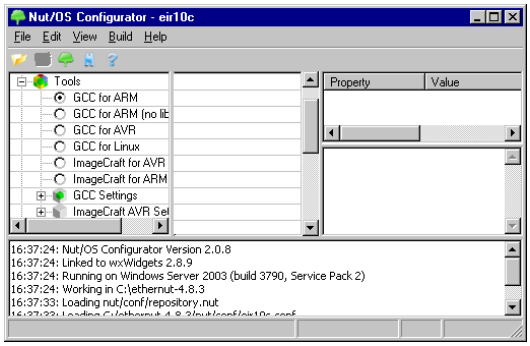
Programmer	Use with
arm-jom	JTAG-O-MAT for ARM targets
arm-samba	Atmel's SAM-BA utility, recommended for ARM targets with SAM-BA boot ROM
avr-dude	AVRDUDE utility, recommended for AVR targets, supports many adapters
avr-jtagice	Atmel's original AVR JTAGICE (RS232 only) and AVR targets
avr-jtagicemkii	Atmel's AVR JTAGICE MKII and AVR targets
avr-uisp-stk500	UISP utility with STK500 compatible adapter, not recommended
avr32-jtagicemkii	Atmel's AVR JTAGICE MKII and AVR32 targets

Programmer	Use with
gba-xport2	CharmedLabs xpcomm utility for GameBoy Advance

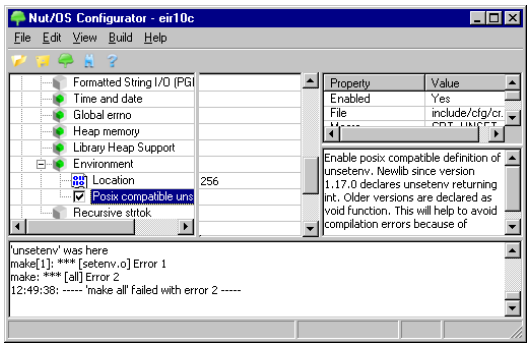
In most cases it will be necessary to edit the related command file. You can find them in the directory nut/app/ with Makeburn as their base name and the programmer as their extension, e.g. Makeburn.avr-dude. If unsure, just leave everything as it is. There are many alternative ways to upload the binaries to the target. Check your hardware manual for additional informations.

If an error message window pops up after clicking *OK*, a path may have been incorrectly entered or a directory does not exist. Please check your paths and directories again, especially in the Tools entry. Look for unintended spaces or any extra characters typed by mistake as well. The paths and directories containing spaces (like the default Program Files directory in several language versions of Windows) have to be enclosed in quotes.

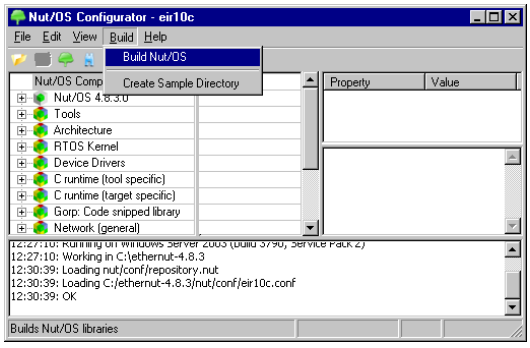
Back in the Configurator's main window it may be required to adjust the compiler selection in the module tree, specifically when using ImageCraft.



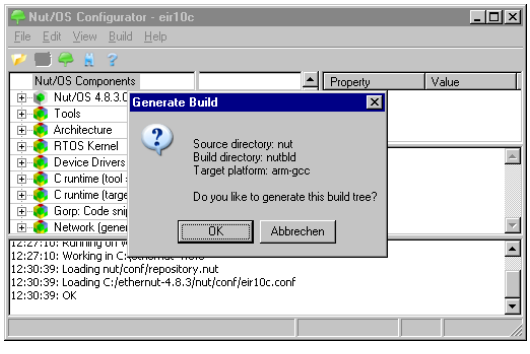
When building for ARM targets and using newlib 1.17 or later, it is further required to select the option *Posix compatible unsetenv*. This is annoying, because the newlib maintainers reject to provide a version entry that can be processed by the compiler. We have to maintain backward compatibility manually. Notice the log window on the left, which shows the related compile error.



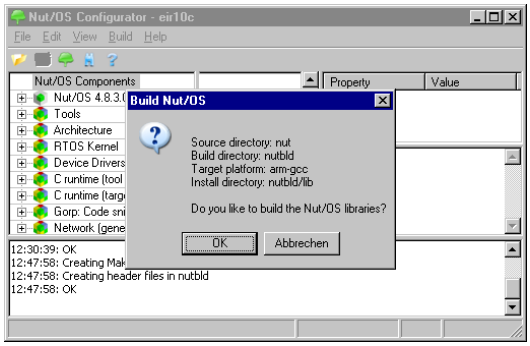
We are finally ready to build Nut/OS. Select *Build Nut/OS* in the *Build* menu or click the tree symbol in the toolbar.



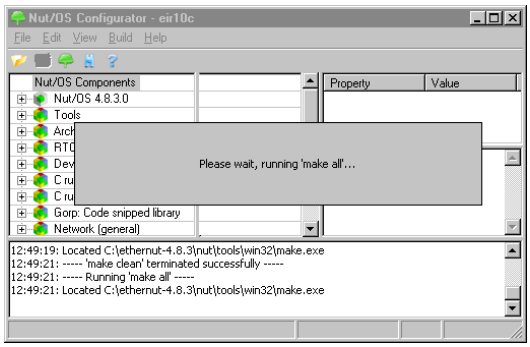
After confirming the message box by clicking *OK*, the Configurator prepares a new build directory or modifies an already existing one. It creates a set of files including some C language header files in a subdirectory named *include/cfg*. These files are included into the Nut/OS source code to tailor the system to your specific target hardware.



A second message box pops up after the build directory has been created or updated.



After confirming the second message box, the Configurator will remove any previously built binaries (make clean), create new libraries (make all) and move them to the installation directory (make install).



When done, the Configurator will have created the following libraries:

- libnutarch.a (Architecture dependent library)
- libnutc.a (Tool specific C runtime library)
- libnutcontrib.a (Contribution library with possibly incompatible licenses)
- libnutcpp.a (C++ runtime extension library)
- libnutcrt.a (Target specific C runtime library)
- libnutdev.a (Device driver library)
- libnutfs.a (File system library)
- libnutgorp.a (Snippet collection library)
- libnutlua.a (Lua language support library)
- libnutnet.a (Network library)
- libnutos.a (RTOS kernel library)
- libnutpro.a (Application protocol library)

If this fails, check your settings again. The log output at the bottom should provide additional hints. If you still can't determine the problem, try the command line to get a more detailed error report. Open a command line (shell on Linux, DOS box on Windows or Terminal on OS X) and change to the build directory. On Windows you must set the PATH environment each time you open a new DOS box:

```
set PATH=<tools path>;%PATH%
```

Replace *<tools path>* with the entry in the Tools settings. For example, with WinAVR you would enter something like:

```
set PATH=C:\ethernut-4.8.3\nut\tools\win32;C:\WinAVR\bin;C:\WinAVR\utils\bin;%PATH%
```

On Linux and OS X this is typically not required.

To manually build the libraries enter

```
make clean all install
```

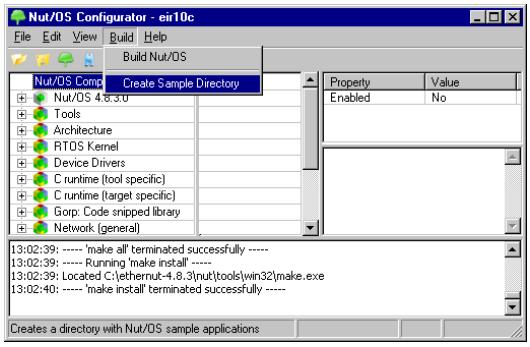
At the end you should see some kind of error message. If you need further help, check

```
http://www.ethernut.de/en/support/index.html
```

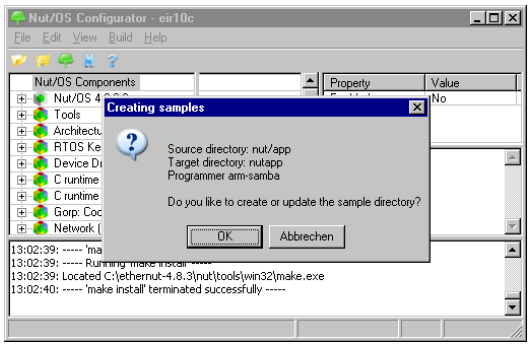
about how to get further support.

When the libraries have been successfully installed, the last task to be done in the Configurator is to create an associated application tree. Although possible, we will not use the application samples in the source directory. Instead we use the Configurator to create a copy of the sample directory for us.

Select *Create Sample Directory* in the *Build* menu.



A message box is displayed to confirm the settings. When clicking *OK* the samples include in the Nut/OS distribution are copied to this application directory. They may also serve as a template for your own applications.



Using the Command Line Configurator

Typically the Configurator is used occasionally, in which case the GUI version is a good choice. However, there are at least two cases, where the command line version is the better choice:

- Building the command line version from the source is much easier and fails less likely than the GUI version.
- The command line version can be used in batch processing.

The name of the executable is nutconfigure. To run it, open a command line shell and change to the parent directory of the source directory nut:

```
cd ~/ethernut/
```

on Linux and OS X, or

```
cd c:\ethernut-4.8.3\
```

on Windows. When called with option *-?*, a short usage help is displayed.

```
nutconfigure -?
Usage: nutconfigure OPTIONS ACTIONS
OPTIONS:
-a<dir>  application directory (./nutapp)
-b<dir>  build directory (./nutbld)
-c<file> configuration file (./nut/conf/ethernut21b.conf)
-i<dir>  first include path ()
-j<dir>  last include path ()
-l<dir>  library directory ()
-m<type> target platform (avr-gcc)
-p<type> programming adapter (avr-dude)
-q       quiet (verbose)
-s<dir>  source directory (./nut)
-r<file> repository (./nut/conf/repository.nut)
ACTIONS:
create-buildtree
create-apptree
```

If this fails, check that your PATH environment is set.

The most important parameter is -c, which specifies the configuration file of the target board. These files are located in *nut/conf/*. Below are the commands to set up a build tree for our four reference boards Ethernut 1.3, Ethernut 2.1, Ethernut 3.0 and Ethernut 5.0.

```
nutconfigure -bnutbld-13h -cnut/conf/ethernut13h.conf \
 -lnutbld-13h/lib -mavr-gcc create-buildtree
nutconfigure -bnutbld-21b -cnut/conf/ethernut21b.conf \
 -lnutbld-21b/lib -mavr-gcc create-buildtree
nutconfigure -bnutbld-30e -cnut/conf/ethernut30e.conf \
 -lnutbld-30e/lib -marm-gcc create-buildtree
nutconfigure -bnutbld-50c -cnut/conf/ethernut50c.conf \
 -lnutbld-50c/lib -marm-gcc create-buildtree
```

Note, that the trailing backslash (line break) will only work on Linux and OS X. On Windows you need to enter each command in one go and let the command line interpreter do an automatic break at the end of the line, e.g.

```
nutconfigure -bnutbld-13h -cnut/conf/ethernut13h.conf -lnutbld-13h/lib -mavr-gcc crea
te-buildtree
```

Each command will create a build directory for the related target board. To build Nut/OS for a specific board, change into the related directory and run *make clean all install*. For Ethernut 2.1 we would enter:

```
cd nutbld-21b/
make clean all install
cd ..
```

In a similar way we create board specific application directories with the command line Configurator. Building the sample application events for Ethernut 2.1 can be done with the following commands:

```
nutconfigure -anutapp-21b -bnutbld-21b -cnut/conf/ethernut21b.conf \
 -lnutbld-21b/lib -mavr-gcc create-apptree
cd nutapp-21b/events
make clean all
```

Of course, we need to call nutconfigure for the first time only and each time the configuration is changed. But how to change the configuration without GUI? A simple text editor will do. All configuration items of the module tree in the GUI version are stored in the *.conf* files in human readable format.

Configuration items, which are modified in the setting dialogs of the GUI version are provided as command line options. For example, in the nutconfigure given above we did not specify the programming adapter. If you check the usage help, you will notice, that this means to stay with the default *avr-dude* (bracketed value). More details about creating Nut/OS applications are presented in the next chapter.

Building Applications on the Command Line

In this chapter we will build the web server application that is included in the Nut/OS distribution. It is assumed, that you built and installed the Nut/OS libraries and that you already created an application sample directory with the Configurator.

Basically, Nut/OS applications are built on the command line. This is even true when using an IDE like the ImageCraft IDE or Eclipse, where the command lines are executed internally. When creating a sample directory, the Configurator copies all samples from the source tree to this directory and adds a few more files containing specific settings. As a special highlight, the GUI version even creates project files for ImageCraft. Except for the demo version, building applications with the ImageCraft compiler on the command line is still supported and may have its advantages, but users of this tool may not worry to skip this chapter.

Running make

As usual, open a shell window (aka DOS box on Windows or terminal on OS X) and change to the directory *httpd* in the application sample directory we created with the Configurator. On Linux and OS X something like this should work:

```
cd ~/ethernut/nutapp/httpd
```

Windows users may enter something like

```
c:
cd \ethernut-X.Y.Z\nutapp\httpd
```

and additionally need to set the PATH environment for ARM targets

```
set PATH=c:\ethernut-X.Y.Z\tools\win32;"C:\Program Files\ yagarto\bin";%PATH%
```

or AVR targets with WinAVR

```
set PATH=c:\ethernut-X.Y.Z\tools\win32;C:\WinAVR\bin; C:\WinAVR\utils\bin;%PATH%
```

or AVR targets with ICCV7AVR

```
set PATH=c:\ethernut-X.Y.Z\tools\win32;C:\iccv7avr\bin;%PATH%
```

Note, that these are just examples. You may have installed Nut/OS or your toolchain in different locations and need to adjust the paths accordingly. The PATH setting is valid for the current window only and must be entered each time you open a new command line window. Thus, it is a good idea to store it in a batch file, e.g. *setenv.cmd* and call this instead.

The following commands are the same for all operating systems and targets. Simply enter

```
make clean all
```

to build the target binaries from the source code. This will at least create an Intel hex file *httpserv.hex* in the current directory, which is typically required by your programming software. Most bootloaders, like *BootMon* for Ethernut 3.0, need the raw binary *httpserv.bin*, while debuggers need loadable object files *httpserv.elf* or *httpserv.cof*. If the required format is not created automatically, just call make again with the missing file

```
make httpserv.elf
```

The procedure for uploading the file to your target board depends on the programming adapter or bootloader as well as the tool used on the PC. Refer to the related manuals.

Most programming tools can run on the command line, in which case a single command can re-build the binary and upload it to the target:

```
make clean all burn
```

This requires that you selected the correct *Programmer* on the last page in the Configurator settings notebook. It may be further required to modify the related file *nut/app/Makeburn.**.

Most bootloader tools (like TFTPD32 and SAM-BA) expect the binary in a previously specified directory. Use

```
make clean all install
```

to automatically copy the binaries to *nut/bin/target/*, where *target* will be replaced by the CPU family like avr, arm7tdmi, arm9 etc. If your board supports boot loading over Ethernet (like Ethernut 3), then configure your TFTP server to use this directory for requested files. A simple click on the target's reset button is required to upload and start the newly built binary.

Modifying Nut/OS Samples

By default most Nut/OS samples use DHCP to automatically determine TCP/IP settings. Even without DHCP, typical applications will store these settings in the on-chip EEPROM and use them if DHCP is not available. How this is done in detail depends on your target hardware. In order to not overload this tutorial, we will show how to use hard coded addresses, which will work on any target. Please, check your hardware manual for further information about your board's configuration capabilities.

Open the file httpserv.c in directory *nutapp/httpd* in your favorite text editor (or your IDE, if available). The interesting entries are at the top:

```
#define MY_MAC  "\x00\x06\x98\x30\x00\x35"
#define MY_IPADDR "192.168.192.35"
#define MY_IPMASK "255.255.255.0"
#define MY_IPGATE "192.168.192.1"
```

You probably have to change the IP address and may also modify the IP mask to fit your network environment. Otherwise your web browser won't be able to talk to your board later on. If unsure what to do, better ask someone with IP network experience.

You can change the MAC Address to the one, which you received with your board. If this is not available, you need to at least make sure, that the address is unique in your local network.

When done, run *make clean all install* or *make clean all burn* in this application's directory (*nutapp/httpd*) again and upload the binary to the target hardware.

If you change any items in the Configurator later, it is recommended to create the sample directory again. Fortunately, the Configurator will not overwrite your modified source file.

The easiest way to create your own applications is to create a copy of an existing sample within the same application directory. If you change any source file names or add new ones, you must also update the Makefile. This can be done with any text editor. However, note that tabs have a special meaning in Makefiles and must be preserved.

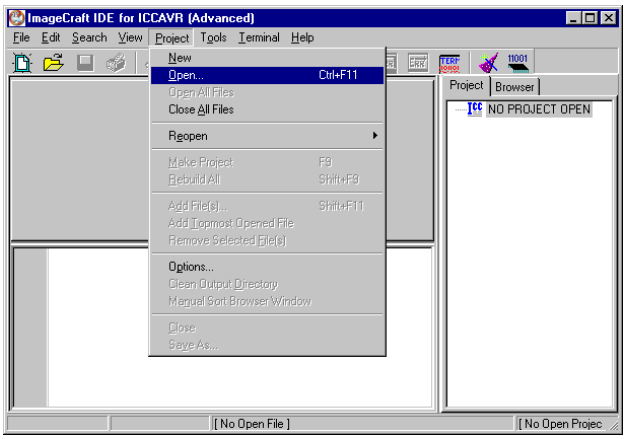
Building Applications with ImageCraft

This chapter will be of use only if you are developing with the ImageCraft IDE.

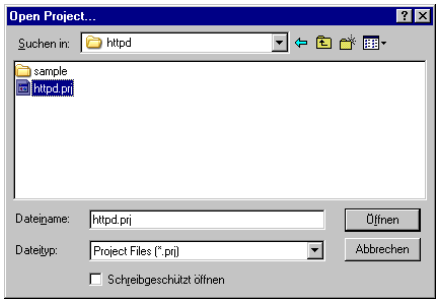
It is assumed that you have built the Nut/OS libraries and created a sample directory with the GUI version of the Configurator.

ImageCraft Configuration

Launch the ImageCraft IDE. If started for the first time, the IDE will appear without any active project.



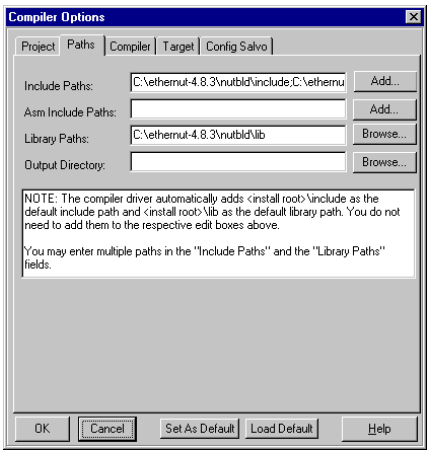
Select Open... from the project menu, navigate to the newly created Nut/OS sample directory and load the project file httpd.prj, which is located in the subdirectory httpd.



The prepared project file contains most required settings. Later on, you will create your own projects and refer to the following steps to configure it.

If the project files in your Nut/OS distribution had been created for a different version of the ImageCraft compiler, you will see a warning message, reminding you to select the correct target. You can ignore this for now, we will get to this step anyway.

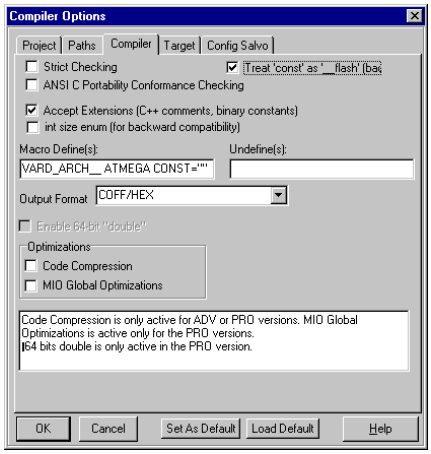
Select Options... from the project menu and click on the tab named *Paths*. Two additional paths were added to the *Include Paths*, *c:\ethernut-4.8.3\nutbld\include* and *c:\ethernut-4.8.3\nut\include*. This way ICCAVR will search C header files in the Nut/OS build directory first, then in the original Nut/OS source directory and finally in its own include directory, which doesn't need to be specified.



In other words, the files located in the build tree will override the original header files in the source tree, which in turn will override the original ICC header files.

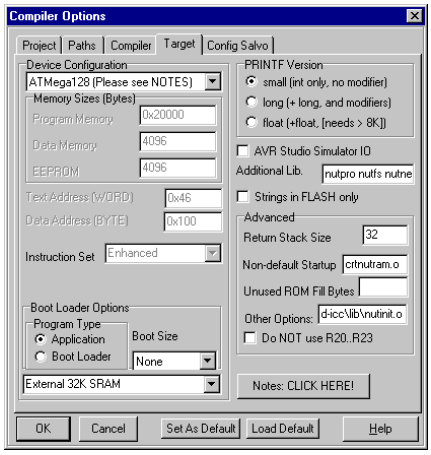
The entry marked Library Path points to the directory, which contains our readily built Nut/OS libraries.

Click the Compiler tab and verify, that all options are correctly set, specifically the *Macro Define(s)*. All *Macro Define(s)* must be separated by spaces. Note, that *CONST=""* must not contain spaces. This macro together with the option *Treat 'const' as '__flash'* maintains backward compatibility with ICCAVR Version 6. However, support for this version may be abandoned in the next Nut/OS release. Check the following table for valid entries.



Macro	Use with
__HARVARD_ARCH__	Mandatory for AVR targets
ETHERNUT1	Ethernut 1.x targets
ETHERNUT2	Ethernut 2.x targets
__MCU_enhanced	Target boards with ATmega128 CPUs
ATMEGA	Mandatory for AVR targets
CONST=""	For backward compatibility

Click the *Target* tab. Again make sure, that the right options are set. Specifically check the *Device Configuration* for ATmega103, ATmega128 or ATmega2561.



The following *Additional Libs* are required for our webserver project:

```
nutpro nutgorp nutfs nutnet nutos nutdev nutcrt nutarch
```

Other Options should contain:

```
-ucrtnutram.o c:\ethernut-4.8.3\nutbld\lib\nutinit.o
```

except for Ethernut 1.3, where Rev-G and any later revision it must contain

```
-ucrtenutram.o c:\ethernut-4.8.3\nutbld\lib\nutinit.o
```

Note the additional letter 'e' after '-ucrt'.

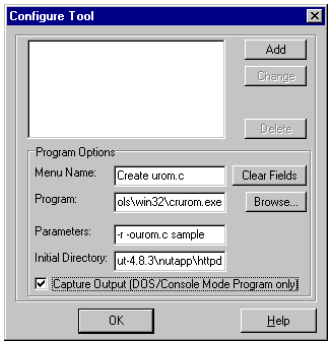
This entry will instruct the compiler to use a Nut/OS specific runtime initialization routine. The main difference to the standard routine is, that NutInit is called before main, so any RTOS specific initialization is hidden from your application code. You can simply start coding main, while the idle thread and the Nut/OS timer are already running in the background. In fact, main is started as a Nut/OS thread. This feature makes Nut/OS applications look like normal C programs and preserves portability.

Most Nut/OS applications will need more than the 4 kBytes of RAM provided internally by the ATmega CPU. Thus, the compiler is intentionally set to External 32k SRAM.

We ignore the last page for Salvo settings. Salvo is another RTOS similar to Nut/OS. Press OK to close the Compiler Option Window.

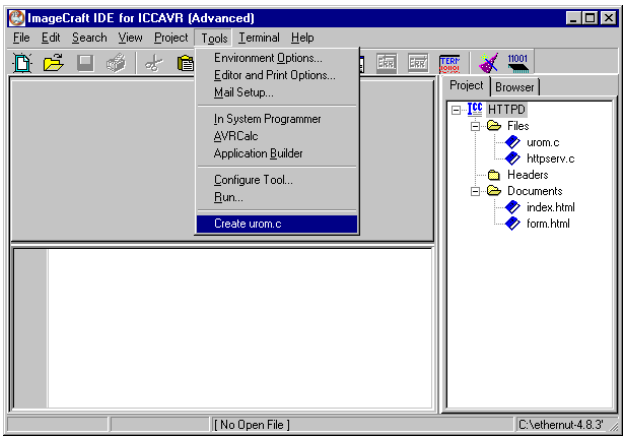
Specially for our HTTP Server project, another entry in the *Tools* menu will be helpful. Select *Configure Tool* from the main menu and add the following entries in the Edit Tool Menu Dialog.

```
Menu Name: Create urom.c
Program: c:\ethernut-4.8.3\nut\tools\win32\crurom.exe
Parameters: -r -ourom.c sample
Initial Directory: c:\ethernut-4.8.3\nutapp\httpd
Activate Capture Output
```

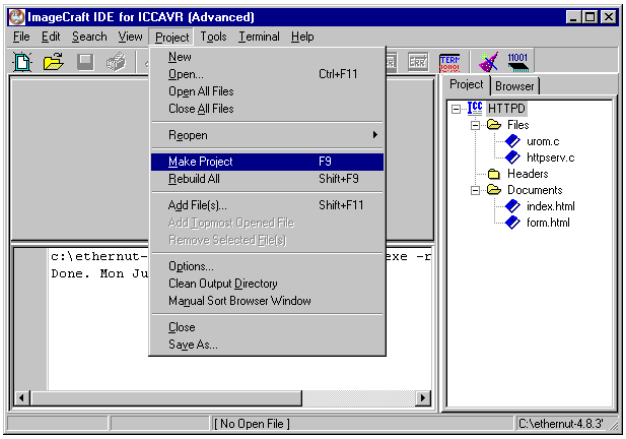


Click *Add* and *OK*.

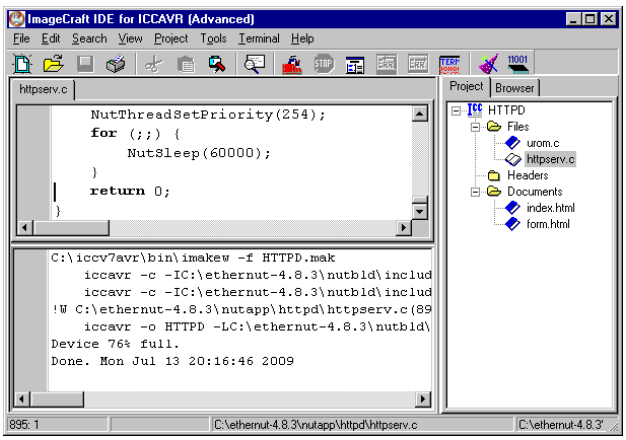
This should have created a new entry in the *Tools* menu. Select this entry now to run the crurom utility, which converts all files in a specified directory (*sample* in our case) to a C source file. This is used to include HTML files, images, Java Applets or other stuff into the Webserver's simple file system. This C source will be compiled and linked to the Nut/OS code.



When selecting Make Project from the project menu, ICCAVR will compile and link the webserver code. Check, that no errors occurred during this process. Refer to the ICCAVR manual for further details.

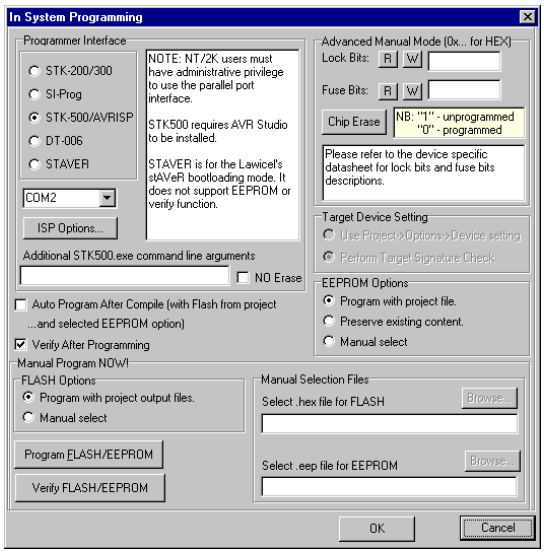


As a result of this step, ICCAVR created several files in the project directory. One is named httpserv.hex and contains the binary code in Intel Hex Format. The second file, httpserv.cof, can be loaded into the AVR Studio Debugger and contains the binary code plus additional debugging information.



Unless debugging is required, you can directly use the ICCAVR IDE to program the ATmega flash memory with an ISP Adapter like the one you received with your Ethernut Starter Kit. For debugging you need a JTAG Adapter with debugging capabilities, for example the ATJTAGICE from Atmel. The JTAG interface of the SP Duo doesn't support debugging.

If not already done, connect the ISP Adapter to the Ethernut Board and the PC. While doing this, you must have switched off the Ethernut's power supply.



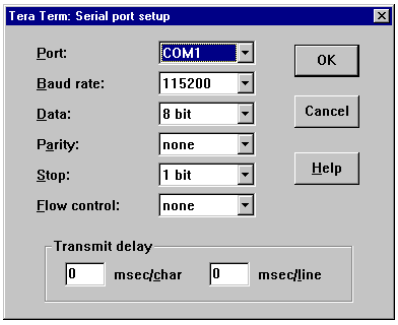
Switch the Ethernut power supply back on and select *In System Programmer* from the *Tools* menu. Then select the file httpserv.hex for the *FLASH* by clicking on the *Browse* button. We do not need to program the EEPROM. Now press the button labeled *Program FLASH/EEPROM* to start in-system programming. This takes some seconds. Some versions of ICCAVR display an error message on empty EEPROM entries. You can ignore this. Finally press the *OK* button to close the *In System Programming* window. Your Ethernut Board will immediately start the webserver application, waiting for a web browser to connect.

Running the Embedded Webserver

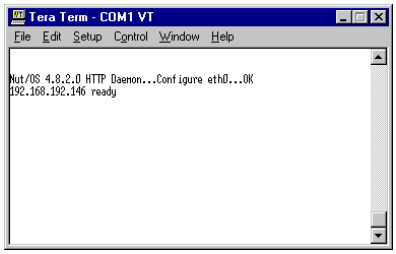
In the previous chapters we compiled the binary for our first Nut/OS application, an embedded webserver, and uploaded it to the target board.

Most application samples use the serial port to provide some feedback about program progress or any kind of errors that may occur. Connect the serial port of your target board to one of the serial ports on your PC. If available, you can use the cable that came with your Ethernut Starter Kit, a 1:1 DB-9 cable for AVR boards or a so called null modem cable for ARM based Ethernuts. There's no need to switch off the target board, serial ports are quite safe and protected against shortcuts or electrical discharge. But remember not to touch any bare contacts on the board before taking some pre-cautions. Dissipate static electricity by touching a grounded metal object.

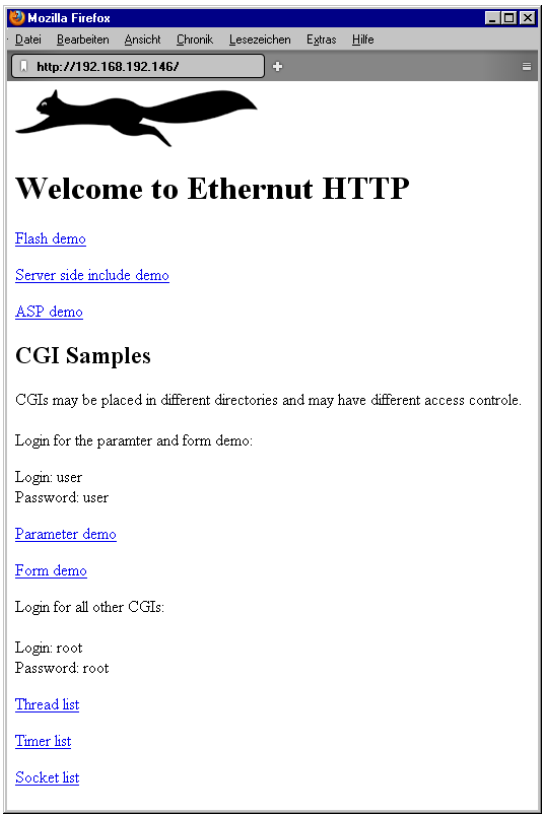
Now start your terminal emulator on the PC. Windows users may use the standard Hyperterm, but we recommend TeraTerm. It is more stable and freely available. For Linux, minicom and gtkterm are good alternatives. OS X users may use the build-in screen command. The required settings for the serial port are 115200 Baud, eight data bits, no parity and one stop bit. All handshakes must be disabled.



Make sure, that the target board is connected to your local network. Resetting your target by pressing the reset button will produce some text output in the terminal emulator's window. The text on your system may differ, depending on your network configuration.



Start a web browser on your PC. The URL to request is the IP address, which has been printed on the terminal emulator. If everything went well, you will see the main index page of the Ethernut webserver.



Congratulations, your embedded webserver is working!

The following chapters will introduce Nut/OS and Nut/Net in more detail. Some of you may not be able to follow every part. Don’t worry. Try to code a few basic samples and have a look at the list of links at the end of this manual.

Nut/OS

This chapter will provide a short overview. Check the Nut/OS API (Application Programming Interface) Reference for a detailed description.

Be aware, that this chapter makes no attempts to explain details of the C language. It is assumed that you have a basic knowledge of C programming.

System Initialization

By default, C programs are started with a routine called main. This isn't much different in Nut/OS, however, the operating system requires certain initialization before the application is started. This initialization is included in a module named nutinit. It will initialize memory management and the thread system and start an idle thread, which in turn initializes the timer functions. Finally the application's main routine is called. Because there's nothing to return to, this routine should never do so.

A sample application named simple demonstrates the most simple application that can be built with Nut/OS. It does nothing else than running in an endless loop, consuming CPU time.

```
#include <compiler.h>

int main(void)
{
    for (;;);
}
```

The file compiler.h is included to fix a problem with AVR GCC, which insists on setting the stack pointer again on entry to main(). When using this compiler, main is re-defined to NutAppMain. Other compilers won't be hurt.

Almost all Nut/OS header files do include compiler.h. Thus it is required only when no other header files are used in the application code.

Thread Management

Typically Nut/OS is most useful when 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.

Nut/OS implements cooperative multithreading. That means, that threads are not bound to a fixed time slice. Unless they are waiting for specific event or explicitly yielding the CPU, they can rely on not being stopped unexpectedly. As opposed to pre-emptive multithreading, cooperative multithreading simplifies resource sharing and usually results in faster and smaller code.

Each thread has a priority which is used to determine how urgent it is. This priority ranges from 0 to 254, with the lowest value indicating the most urgent.

As stated earlier, the main application thread is already running as a thread, together with the background idle thread, which is not visible to the application.

Creating a new thread is done by calling NutThreadCreate. The code running as a thread is nothing else than another C routine. To hide platform specific details, applications should use the THREAD macro to declare those routines.

```
THREAD(Thread1, arg)
{
    for (;;) {
        NutSleep(125);
    }
}

int main(void)
{
    NutThreadCreate("t1", Thread1, 0, 512);
    for (;;) {
        NutSleep(125);
    }
}
```

In this example the main thread creates a new thread before entering an endless loop. The new thread will run in a similarly senseless loop.

It is important to keep the cooperative nature of Nut/OS in mind. NutSleep, which will be explained next, stops execution for a specified number of milliseconds. If one of the loops would not call this or any other blocking function like reading from a device or waiting for an event, then the other threads would never gain CPU control.

However, there is an pre-emptive component too, because threads may be interrupted by hardware interrupt signals. Thus, special care is required when writing interrupt handlers and almost all API calls are forbidden when executing in interrupt context, except NutEventPostFromIrq.

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.

Applications can use standard C calls to allocate and release memory blocks:

```
void *malloc(size_t size)
void *calloc(size_t count, size_t size)
void free(void *ptr)
void *realloc(void *ptr, size_t size)
char *strdup(char *str)
```

A program fragment that allocates a buffer of 1024 bytes in size, senselessly filling it with 0xFF and releasing the buffer again, will look like this:

```
char *buffer;

buffer = malloc(1024);
if (buffer) {
    memset(buffer, 0xFF, 1024);
    free(buffer);
} else {
    puts("Out of memory error!");
}
```

You should make intensive use of dynamic memory allocation for two reasons:

First, large local variables will occupy stack space. As each thread gets its own stack, you may waste a lot of memory by reserving large stacks.

Second, large global variables occupy space during the complete lifetime of the application, although they may not be used during that time. For some environments the global variable space may be much more limited than heap space.

The heap manager uses best fit, address ordered algorithm to keep the free-list as defragmented 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.

To determine the total free space, call

```
size_t NutHeapAvailable(void)
```

The size of the largest free block is returned by

```
size_t NutHeapRegionAvailable(void)
```

Sooner or later every C software developer will face dynamic memory issues. Nut/OS offers a set of useful functions to track down such problems. When enabling *Heap Debugging* in the Configurator, the standard routines will be replaced by debug versions, which are slower and produce larger code. The advantage is, that heap management will track memory allocation. Calling

```
void NutHeapDump(void * stream)
```

will then send a detailed list of all memory blocks to the given stream, e.g. stdout. For each allocated block the name of the source file and the line within this file where the allocation took place is listed. This is especially useful to located memory holes.

When *Guard Bytes* are enabled in the Configurator, the heap manager will add extra bytes to each allocation. The application may later call

```
int NutHeapCheck(void)
```

to check that these guard bytes are still valid. This helps to track down buffer overwrites.

Timer Management

On most platforms Nut/OS uses the first on-chip hardware timer of the CPU. Though, there are exceptions to this rule. On several AVR devices timer 2 is used, because this is timer that runs in low power mode, allowing the OS to periodically wake up the system. Most AT91 CPUs provide a periodic timer specifically designed for generating system ticks. The Configurator allows to select this one as a system timer.

All remaining timers are available for the application. You can use them in cases where the system timer resolution, typically 1ms, is insufficient.

Nut/OS provides time related services, allowing application to delay itself for a given number of milliseconds by calling NutSleep(). Furthermore, a time out value can be defined for most blocking operations like input/output functions. Again, time out values are given in milliseconds.

The following minimal application will run in an endless loop, but spend most of the time sleeping for 125 milliseconds. During sleep time, any other thread may take over. If no other thread is ready to run, CPU control is passed to the Nut/OS idle thread.

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

int main(void)
{
    for (;;) {
        NutSleep(125);
    }
}
```

NutSleep guarantees a minimum sleep time, but it may take more time to return to the caller. Often short reproducible delays with higher resolutions are required, in which case

```
void NutMicroDelay(uint32_t us)
```

should be used. The delay time is specified in microseconds. However, this function will not release the CPU and unless the value of NUT_DELAYLOOPS hadn't been exactly configured for your specific hardware, it is based on a rough estimation. Also note, that a fast CPU is required to provide low resolutions of a few microseconds.

Hardware timers are based on hardware clocks, e.g. crystal oscillators. If possible, Nut/OS tries to determine clock frequencies automatically. Otherwise it must be configured, so Nut/OS can calculate the right hardware timer setting for system ticks of one millisecond.

Amongst others, the following timer related API functions may be useful:

```
uint32_t NutClockGet(int idx)
```

returns the specified clock frequency in Hertz. Accepted values are at least NUT_HWCLK_CPU to query the CPU clock and NUT_HWCLK_PERIPHERAL to query the clock that drives on-chip peripherals. Other values may be additionally available for your target.

```
uint32_t NutGetMillis(void)
```

returns the current value of the system's uptime in millisecond and

```
uint32_t NutGetSeconds(void)
```

returns the system's uptime in seconds. Given a system tick of 1ms, both counters will overflow after about 7 years.

```
uint32_t NutGetTickCount(void)
```

returns the current value of the system's tick counter and

```
uint32_t NutGetTickClock(void)
```

returns the number of system ticks per second.

Setting I/O time outs is device dependent. Devices that are used for standard I/O often provide specific ioctl commands. For UART devices we can use

```
UART_SETREADTIMEOUT
UART_SETWRITETIMEOUT
```

The following code fragment opens device “uart0” for binary reading and writing and sets the read timeout to 1000 milliseconds.

```
int com;
unsigned long tmo = 1000;
....
com = _open("uart0", _O_RDWR | _O_BINARY);
_ioctl(com, UART_SETREADTIMEOUT, &tmo);
```

Internally the related drivers use the Nut/OS event management to handle time outs.

Nut/OS doesn't provide non-blocking I/O. A minimum timeout of 1ms must be specified. To disable time outs, set the time value to NUT_WAIT_INFINITE.

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 wait queue, moving the thread from waiting (sleeping) state to the 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 by calling the macro

```
NutEventPostFromIrq(volatile HANDLE *qhp)
```

The event API provides the following functions:

```
int NutEventPost(HANDLE *qhp)
```

posts an event to the specified queue. If no thread is currently waiting, the queue is marked signaled.

```
int NutEventPostAsync(HANDLE *qhp)
```

is the same as NutEventPost, but keeps the current thread running.

```
int NutEventBroadcast(HANDLE *qhp)
```

wakes up all threads waiting on the specified queue. In opposite to NutEventPost and NutEventPostAsync, an empty queue will not be marked signaled. If already signaled, the mark will be removed.

```
int NutEventBroadcastAsync(HANDLE *qhp)
```

is the same as NutEventBroadcast, but, similar to NutEventPostAsync, the current thread continues running.

All posting functions return the number of woken up threads.

```
int NutEventWait(HANDLE *qhp, uint32_t ms)
```

waits for an event. If the queue is marked signaled, the thread will remain ready to run and the signal mark will be removed from the queue. However, if any other thread with equal or higher priority is ready to run, then the calling thread will lose the CPU.

```
int NutEventWaitNext(HANDLE *qhp, uint32_t ms)
```

is the same as NutEventWait, but any signal mark is ignored and silently removed.

All waiting functions return zero, when an event has been posted within the specified number of milliseconds. If no event is received within this time, then the thread is woken up by the timer management and the event wait function returns -1.

A time out value of zero means to wait forever, which is often misinterpreted. You may better use the macro NUT_WAIT_INFINITE instead. Nut/OS currently doesn't provide non-blocking calls.

For using events, a modified version of our simple threading example will look like this.

```
HANDLE evt_h;

THREAD(Thread1, arg)
{
    for (;;) {
        NutSleep(125);
        NutPostEvent(&evt_h);
    }
}

int main(void)
{
    NutThreadCreate("t1", Thread1, 0, 192);
    for (;;) {
        NutEventWait(&evt_h, NUT_WAIT_INFINITE);
    }
}
```

Thread1 is running an endless loop of sleeps. The main thread waits for an event posted to an event queue. Event queues are specified by variables of type HANDLE. The endless loop of the main thread is blocked in NutEventWait() and woken up each time an event is posted by Thread1.

Btw. it had not been very smart to introduce the variable type HANDLE. But it's there since the very early releases of Nut/OS and most people don't care much.

Stream I/O

Typical C applications make extensive use of the standard I/O library. In most cases the runtime library that comes with your compiler provides the related API. However, only very simple devices are typically supported, or, in case of newlib, a Linux-like kernel is expected. Therefore Nut/OS provides its own library nutcrt, which replaces almost all related functions of the compiler's runtime library.

For a tiny operating system like Nut/OS, the list of available functions is quite impressive:

```
void clearerr(FILE * stream);
int fclose(FILE * stream);
void fcloseall(void);
FILE *_fdopen(int fd, CONST char *mode);
int feof(FILE * stream);
int ferror(FILE * stream);
int fflush(FILE * stream);
int fgetc(FILE * stream);
char *fgets(char *buffer, int count, FILE * stream);
int _fileno(FILE * stream);
void _flushall(void);
FILE *fopen(CONST char *name, CONST char *mode);
int fprintf(FILE * stream, CONST char *fmt, ...);
int fpurge(FILE * stream);
int fputc(int c, FILE * stream);
int fputs(CONST char *string, FILE * stream);
size_t fread(void *buffer, size_t size, size_t count, FILE * stream);
FILE *freopen(CONST char *name, CONST char *mode, FILE * stream);
int fscanf(FILE * stream, CONST char *fmt, ...);
int fseek(FILE * stream, long offset, int origin);
long ftell(FILE * stream);
size_t fwrite(CONST void *data, size_t size, size_t count, FILE *stream);
int getc(FILE * stream);
int getchar(void);
int kbhit(void);
char *gets(char *buffer);
int printf(CONST char *fmt, ...);
int putc(int c, FILE * stream);
```

```
int putchar(int c);
int puts(CONST char *string);
int scanf(CONST char *fmt, ...);
int sprintf(char *buffer, CONST char *fmt, ...);
int sscanf(CONST char *string, CONST char *fmt, ...);
int ungetc(int c, FILE * stream);
int vfprintf(FILE * stream, CONST char *fmt, va_list ap);
int vfscanf(FILE * stream, CONST char *fmt, va_list ap);
int vsprintf(char *buffer, CONST char *fmt, va_list ap);
int vsscanf(CONST char *string, CONST char *fmt, va_list ap);
```

In addition, the following low level I/O functions are available:

```
int _close(int fd);
int _open(CONST char *name, int mode);
int _read(int fd, void *buffer, size_t count);
int _write(int fd, CONST void *buffer, size_t count);
int _ioctl(int fd, int cmd, void *buffer);
long _filelength(int fd);
```

This set of functions allows to port many existing PC applications without too much effort. Nevertheless, there is an important difference to standard C applications written for the PC. Typical embedded systems do not pre-define devices for stdin, stdout and stderr. Thus, the well known standard C sample 'Hello world'

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
}
```

requires a few additional lines of code on Nut/OS.

```
#include <stdio.h>
#include <dev/usartavr.h>

int main(void)
{
    unsigned long baud = 115200;

    /* Register the device we want to use. */
    NutRegisterDevice(&devUsartAvr0, 0, 0);
    /* Assign the device to stdout. */
    freopen("uart0", "w", stdout);
    /* Optionally set the baudrate of the serial port. */
    _ioctl(_fileno(stdout), UART_SETSPEED, &baud);

    printf("Hello world!\n");
}
```

Btw., a typical pitfall is to mix Nut/OS and compiler libraries for standard C functions. If the linker reports duplicates or undefined syscalls, or if your application hangs in stdio functions or crashes when calling them, then check your linker map file to make sure, that these functions are loaded from Nut/OS libraries. If not, you may have forgotten to add nutcrt to the list of libraries or your application may reference a function that is not available in Nut/OS.

File Systems

Neither Nut/OS nor Nut/Net require a file system, but for example Webservers are designed with a file system in mind. To make things easier for the programmer, Nut/OS provides a very simple read-only file system named UROM, where files are located in flash memory. A special tool, crurom (create UROM) is needed to convert the complete contents of a directory on your PC into a C source file, which is then compiled and linked to your target code.

Two other low level file systems are available. The PNUT file system is a volatile RAM file system, where the contents is lost on reboot. The lately introduced RAWFS doesn't support directories. It assigns a complete storage device to a single file. It may be used to access an external Flash memory chip with high level I/O functions. Both, PNUT and RAWFS support read and write access.

More advanced application may need to exchange data with other systems, using removable media like MultiMedia or SD cards, where FAT is the quasi standard file system to which the Nut/OS PHAT file system is compatible.

Applications can use standard C functions for file system access. With PHAT and PNUT the following functions are provided to access directory entries:

```
DIR *opendir(CONST char *);
struct dirent *readdir(DIR *);
int closedir(DIR *);
```

While directories and subdirectories are supported with UROM, PNUT and PHAT, there is no current working directory. Files must be specified with their full path, prefixed with the name of the file system driver instance. The following code fragment opens a file, writes to it and finally closes the file:

```
FILE *fp = fopen("PHAT0:/TEST.TXT", "w");
if (fp) {
    fprintf(fp, "HelloFile");
    fclose(fp);
} else {
    /* File open failed. */
}
```

Like other devices, the system will not include any file system support by default. Though, UROM and PNUT file systems are immediately available after the file system driver has been registered. Use

```
#include <dev/urom.h>
...
NutRegisterDevice(&devUrom, 0, 0);
```

for UROM and

```
#include <dev/pnut.h>
...
NutRegisterDevice(&devPnut, 0, 0);
```

for the PNUT file system.

PHAT volumes must be explicitly mounted.

```
#include <dev/board.h>
#include <fs/phatfs.h>

… more code …

int hvol;
/* Register the file system driver. */
NutRegisterDevice(&devPhat0, 0, 0);
/* Register the block device driver. */
NutRegisterDevice(&DEV_MMCARD, 0, 0);
/* Mount the file system on the specified device. */
hvol = _open(DEV_MMCARD_NAME ":1/PHAT0", _O_RDWR | _O_BINARY);

… more code …

/* Unmount the file system if no longer used. */
_close(hvol);
```

Different block device drivers are available on different target platforms. The example above makes use of the header file *board.h*, which defines the default DEV_MMCARD and its related DEV_MMCARD_NAME for the currently used target hardware. This ensures, that the code will compile on all supported targets without change. For the Ethernut 3 board *devNplMmc0* is the block device used to access the MMC socket. The mount name for this board will be

```
"MMC0:1/PHAT0"
```

where *"MMC0"* is the name of the block device driver, *"1"* specifies the number of the partition to mount and *"PHAT0"* is the name of the file system driver instance.

Device Drivers

As opposed to desktop computers, tiny embedded systems do not provide a memory management hardware unit to protect memory areas or I/O port access. As a result, there is no special 'kernel mode' for device drivers to run in. Thus, there is no requirement for a device driver at all. To phrase it favorably, application code can freely use any kind of resource.

One advantage of device drivers is still left for Nut/OS: Abstraction. If a device driver exists for, let's say, an LCD display, it may be used with stdio streams like other devices. Device usage can be switched easily by changing a few lines of code. Look at our 'Hello world' output going to an LCD:

```
#include <stdio.h>
#include <dev/hd44780.h>
#include <dev/term.h>

int main(void)
{
    /* Register the device we want to use. */
    NutRegisterDevice(&devLcd, 0, 0);
    /* Assign the device to stdout. */
    freopen("lcd", "w", stdout);

    printf("Hello world!\n");
}
```

All device drivers are based on the NUTDEVICE structure:

```
struct _NUTDEVICE {
    NUTDEVICE *dev_next;
    char dev_name[9];
    uint8_t dev_type;
    uintptr_t dev_base;
    uint8_t dev_irq;
    void *dev_icb;
    void *dev_dcb;
    int (*dev_init) (NUTDEVICE *);
    int (*dev_ioctl) (NUTDEVICE *, int, void *);
    int (*dev_read) (NUTFILE *, void *, int);
    int (*dev_write) (NUTFILE *, CONST void *, int);
#ifdef __HARVARD_ARCH__
    int (*dev_write_P) (NUTFILE *, PGM_P, int);
#endif
    NUTFILE * (*dev_open) (NUTDEVICE *, CONST char *, int, int);
    int (*dev_close) (NUTFILE *);
    long (*dev_size) (NUTFILE *);
};
```

The most important members of this structure are the 7 or 8 function pointers at its end. Each device driver instance provides a global variable of this structure, where at least some of the function pointers point to the related device driver functions. Note, that the dev_write_P member is available on targets with Harvard Architecture only.

Ethernut 3 offers two UARTs, USART0 and USART1. Both can be used with the debug device driver. The driver instance that uses USART0 provides the following global NUTDEVICE structure:

```
NUTDEVICE devDebug0 = {
    0,            /* dev_next */
    {'u', 'a', 'r', 't', '0', 0, 0, 0}, /* dev_name */
    0,            /* dev_type */
    USART0_BASE,  /* dev_base */
    0,            /* dev_irq */
    NULL,         /* dev_icb */
    &dbgfile0,    /* dev_dcb */
    Debug0Init,   /* dev_init */
    Debug0IOCtl,  /* dev_ioctl */
    NULL,         /* dev_read */
    DebugWrite,   /* dev_write */
    DebugOpen,    /* dev_open */
    DebugClose,   /* dev_close */
    NULL          /* dev_size */
};
```

When an application calls

```
NutRegisterDevice(&devDebug0, 0, 0);
```

then Nut/OS calls the function that *dev_init* points to, *Debug0Init* in this case. On success, the driver is added to the linked list of registered drivers.

Sooner or later the application will call

```
FILE *fp = fopen(”uart0”, ”w”);
```

or similar to open the device. Nut/OS will now retrieve the NUTDEVICE structure from the linked list, of which dev_name contains the string “uart0” and then call the function the dev_open member points at. Hence, DebugOpen is called, which returns a pointer to a NUTFILE structure. Among other things, this structure contains a pointer to the NUTDEVICE structure.

The system will now create a stream variable of type FILE, store the NUTFILE pointer in it and return the FILE pointer back to the application.

When the application calls

```
fprintf(fp, “Hello world!\n”);
```

then the system is able to retrieve the both, the NUTFILE pointer and the NUTDEVICE pointer. The latter is used to call DebugWrite via the dev_write member, passing the NUTFILE pointer and the string to print to the driver. The driver will then do the required work to send out character by character, using the USART0 hardware registers.

You may have noticed, that two function pointers contain NULL. The debug device is write-only, thus no dev_read function exists. The system will return an error when an application tries to read from this device. The dev_size entry is a legacy item. It had been used in early days to retrieve the size of a file and was re-used later to query the number of bytes available in the input buffer. In general drivers should use ioctl commands instead to implement specific functions.

Usage of the remaining entries dev_base, dev_irq, dev_icb and dev_dcb is typically up to the driver code. Specifically the last two are often used to store pointers to some local data. This way, the driver is also able to use the same function for all of its instances. For example, the second debug device instance for USART1 on Ethernut 3 is

```
NUTDEVICE devDebug1 = {
    0,            /* dev_next. */
    {'u', 'a', 'r', 't', '1', 0, 0, 0}, /* dev_name */
    0,            /* dev_type */
```

```
    USART1_BASE,  /* dev_base */
    0,            /* dev_irq */
    NULL,         /* dev_icb */
    &dbgfile1,    /* dev_dcb */
    Debug1Init,   /* dev_init */
    Debug1IOCtl,  /* dev_ioctl */
    NULL,         /* dev_read */
    DebugWrite,   /* dev_write */
    DebugOpen,    /* dev_open */
    DebugClose,   /* dev_close */
    NULL          /* dev_size */
};
```

Here the same routines are used for dev_open, dev_write and dev_close.

In some cases the system will verify dev_type to make sure that the right device driver is used. Currently known types are

```
#define IFTYP_RAM       0 /* RAM device */
#define IFTYP_ROM       1 /* ROM device */
#define IFTYP_STREAM    2 /* Stream device */
#define IFTYP_NET       3 /* Net device */
#define IFTYP_TCPSOCK   4 /* TCP socket */
#define IFTYP_CHAR      5 /* Character stream device */
#define IFTYP_CAN       6 /* CAN device */
#define IFTYP_BLKIO     7 /* Block I/O device */
#define IFTYP_FS       16 /* File system device */
```

but, as you can see in the devDebug structures above, they are not used consistently.

Driver instances may have the same name if they refer to the same hardware. For example, instead of limited devDebug0 instance the Ethernut 3 application may register devUart0, which is fully interrupt driven and also provides a dev_read routine for data input. Both instances have the same name, because both use the same USART0 hardware. Only one of them can be used, because NutRegisterDevice will reject a device instance, if one with the same name is already registered.

Not every hardware device fits well, though. For example, SPI is very special. In order to receive a byte, one byte has to be sent out. Even worse, this interface may run in master or slave mode, may change the mode dynamically and multiple different devices may be attached to SPI. A similar situation exists with I2C, or in general, with all devices that implement busses.

Especially for SPI a special device driver type named NUTSPIBUS has been implemented, which completely differs from the NUTDEVICE type. However, white-bread device driver instances may be attached to it using

```
int NutRegisterSpiDevice(NUTDEVICE * dev, NUTSPIBUS * bus, int cs)
```

The following code fragment attaches the MP3 decoder driver devSpiVsCodec0 to the first SPI bus on an AT91 CPU and sends a buffer with encoded MP3 data to the decoder:

```
NutRegisterSpiDevice(devSpiVsCodec0, spiBus0At91, 0);

... more code ...

int ad = _open("audio0", _O_WRONLY | _O_BINARY);
_write(ad, buf, len);
_close(ad);
```

There are other devices, for which the NUTDEVICE structure will not make sense at all, like a watchdog device or in general timers and counters. In order to hide hardware details from the application, several device helper functions are included. Although part of the nutdev library, these are not device drivers in strict interpretation simply because they do not provide a global NUTDEVICE structure variable.

Another kind of helper routines are so called driver frameworks. A typical one is used by the UART driver. The hardware specific part itself provides the NUTDEVICE structure, but the device function pointers are implemented by the framework. The driver itself is reduced to a few hardware specific routines, which are called by the framework. A similar method is used by the terminal driver framework, which interprets VT52 control sequences and calls specific hardware function of the display driver. This way a framework avoids duplicate code, which otherwise would have been included in each device driver over and over again.

Error Handling

While many operating systems extensively check parameters passed from applications to the system API, Nut/OS intentionally relies on good application code for good reasons:

- Due to the lack of memory protection, tiny embedded systems are quite defenseless and have to rely on responsible application code anyway.
- Parameter checking in API functions is often redundant. Most checks have to be done in application code anyway.

This doesn't mean, that the application programmer will be left alone. Nut/OS offers a number of plausibility checks at run time, but they must be explicitly enabled and will be disabled again later, when the system had been fully tested.

Two of these features had been discussed with heap management: Heap dump and memory guards.

Another one is NUTASSERT. This preprocessor macro is expanded only, when NUTDEBUG_USE_ASSERT is defined. At the time of this writing this option is not available in the Configurator, but can be manually set in UserConf.mk.

```
HWDEF+=-DNUTDEBUG_USE_ASSERT
```

Add this line to both files, the one located in the system build directory (nutbld) and the one in the application directory (nutapp). Then rebuild the system libraries and your application code.

The NUTASSERT macro is usually located at the API function entries and will check the plausibility of the parameters passed from application code. If something is wrong, NUTFATAL will be called, which in turn calls NUTPANIC to print out a detailed error message on stdout. For example, assume that an application misses to check the result of fopen

```
fp = fopen("UROM:CONF.TXT", "rb");
fseek(fp, 100, SEEK_SET)
```

If the file doesn't exist or if UROM wasn't registered, fseek will be called with a NULL pointer. Not checking this may result in all kind of problems, typically not directly related to this code. When NUTDEBUG_USE_ASSERT is enabled, the following message is displayed:

```
fseek.c:72: Fatal: Expected stream != NULL in fseek
```

It is also possible to implement NUTFATAL directly in the application and use other methods to report errors. The function prototype is

```
void NUTFATAL(CONST char *func, CONST char *file, int line, CONST char *expected)
```

One of the most subtle challenges for application programmers is to determine the proper stack size of threads. The typical approach is trial and error. Adding

```
HWDEF+=-DNUTDEBUG_CHECK_STACK
```

to UserConf.mk allows to list the maximum stack usage of all threads by using a simple loop:

```
NUTTHREADINFO *tdp;

for (tdp = nutThreadList; tdp; tdp = tdp->td_next) {
    printf("%s: %lu bytes stack available\n", tdp->td_name,
            (unsigned long)NutThreadStackAvailable(tdp->td_name));
```

Internally the system fills the stack on thread creation with a specific pattern. The NutThreadStackAvailable then walks from the stack's bottom until it finds the first location that no longer contains the original pattern to determine the number of unused stack bytes. If you try to make sure, that all parts of your application had been executed and that the application was running long enough, you can use the values with a certain safety margin to fine tune your stack sizes.

More advance debugging capabilities are offered for target platforms, which at least offer a minimal protection such as hardware exceptions. Currently Nut/OS adds support based on hardware exception for ARM CPUs only.

When adding

```
$(LIBDIR)/arm-da.o
```

in front of the *LIBS* entry in your Makefile, e.g.

```
LIBS =  $(LIBDIR)/arm-da.o $(LIBDIR)/nutinit.o -lnutos -lnutdev -lnutarch -lnutcrt
```

then a data abort will print out a back trace on stdout.

There is also

- arm-pfa.o for prefetch abort
- arm-swi.o for software interrupt
- arm-udf.o for undefined instructions

Note, that back trace requires to select arm-gccdbg as the platform in the Configurator settings.

You may enhance various checks with the back trace by adding the following NUTPANIC function to your application:

```
void NUTPANIC(CONST char *fmt, ...)
{
    va_list ap;

    NutEnterCritical();
    va_start(ap, fmt);
    vfprintf(stdout, fmt, ap);
    NutHeapDump(stdout);
    ListStacks();
    for(;;);
}
```

This will replace the default handling and provide a detailed analysis of the system status.

Nut/Net

The code and buffer sizes required by traditional TCP/IP implementations make them less suitable for tiny microcontrollers. 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 Configuration

Before using any Nut/Net function, the application must register the network device driver by calling NutRegisterDevice. The include file *board.h* defines the default network device for all supported boards, so the following code should work on most targets:

```
#include <dev/board.h>

NutRegisterDevice(&DEV_ETHER, 0, 0);
```

In a next step the IP interface must be configured. To do this, the following information is needed:

- Hardware address of the interface. Ethernet uses a 6 byte MAC address.
- Local IP address and a related network mask.
- Optional routing information, typically the IP address of a default gateway.

Nut/Net supports IPv4, where IP addresses fit into 4 bytes. Thus, the 32 bit type uint32_t is used for the binary representation. The conversion functions inet_ntoa() and inet_addr() are provided to convert the binary version into human readable format and vice versa. The following code fragment may be used as a template to configure an Ethernet interface including a default gateway.

```
uint8_t mac[6] = { 0x00, 0x06, 0x98, 0x00, 0x00, 0x00};
uint32_t ip_addr = inet_addr("192.168.0.5");
uint32_t ip_mask = inet_addr("255.255.255.0");
uint32_t ip_gate = inet_addr("192.168.0.1");

NutNetIfConfig(DEV_ETHER_NAME, mac, ip_addr, ip_mask);
NutIpRouteAdd(0, 0, ip_gate, &DEV_ETHER);
```

For real world applications hard coded IP addresses are not practical, though. We can use non-volatile memory to have these settings available on each reboot. Nut/OS offers standard routines to store and load the basic network configuration:

```
int NutNetSaveConfig(void);
int NutNetLoadConfig(CONST char *name);
```

These functions refer to a global structure variable *confnet* of the following type.

```
struct _CONFNET {
    uint8_t cd_size;
    char cd_name[9];
    uint8_t cdn_mac[6];
    uint32_t cdn_ip_addr;
    uint32_t cdn_ip_mask;
    uint32_t cdn_gateway;
    uint32_t cdn_cip_addr;
} CONFNET;
```

Using this capability, we can rewrite the hard coded version as

```
NutNetLoadConfig(DEV_ETHER_NAME);
NutNetIfConfig(DEV_ETHER_NAME, confnet.cdn_mac, confnet.cdn_ip_addr,
               confnet.cdn_ip_mask);
if (confnet.cdn_gateway)
    NutIpRouteAdd(0, 0, confnet.cdn_gateway, &DEV_ETHER);
```

Setting up a virgin system highly depends on the application. You may provide some kind of user input to set the confnet variable and then call NutSaveConfig to store the settings in non-volatile memory. For AVR targets the Basemon program may be used to configure your board.

Today DHCP is available in most networks and it is supported by Nut/Net as well. However, at least the MAC address must be available before DHCP can be used, either hard coded or read from non-volatile memory. The hard coded version is:

```
uint8_t mac[6] = { 0x00, 0x06, 0x98, 0x00, 0x00, 0x00};
NutDhcpIfConfig(DEV_ETHER_NAME, mac, 60000);
```

DHCP may take time to finish or the DHCP server may not be reachable at all. In order not to block the system forever and let the application act on failure, a third parameter is used to specify a time out in milliseconds, 60 seconds in this case.

Instead of a hard coded IP address, a NULL pointer may be passed a the second parameter. The function will then internally call NutNetLoadConfig to fill the global confnet variable. It then decides about further actions.

An overview about the CONFNET structure elements and their meaning is given by the following table.

Name	Type	Default	Description
cd_size	Byte value	31	Total size of the configuration structure. Used to check validity.
cd_name	Character array	”eth0”	Name of the network interface. Maximum size is 8 characters.
cdn_mac	Byte array	0x000698000000	6 bytes unique MAC address.
cdn_ip_addr	IP address	0.0.0.0	Last used IP address.
cdn_ip_mask	IP address	255.255.255.0	Configured IP mask.
cdn_gateway	IP address	0.0.0.0	Default gateway IP.
cdn_cip_addr	IP address	0.0.0.0	Configured IP address.

The contents is accepted by NutDhcpIfConfig, if

- cd_size is valid
- cd_name matches the name given as the first parameter
- cdn_mac is a unicast address

If the contents is not accepted, the function will immediately return -1 to indicate an error.

If the contents of confnet is accepted and if cdn_cip_addr is not zero, then the network interface will be configured to this local IP address and the function returns zero.

If cdn_cip_addr is zero, then NutDhcpIfConfig will start a DHCP client thread to query the network configuration from a DHCP server. On success, cdn_ip_addr, cdn_ip_mask and cdn_gateway will be updated and stored in non-volatile memory. The function returns zero to indicate success.

If the DHCP query fails, and if cdn_ip_addr is not zero, then the interface will be configured to this IP address and the function will return zero to indicate success.

Finally, if DHCP fails and cdn_ip_addr is zero, then NutDhcpIfConfig will give up and return -1 to indicate and error.

As you can see, NutDhcpIfConfig will automatically work correctly in most environments and is the preferred way to configure the Ethernet interface.

Socket API

On top of the protocol stack Nut/Net provides an easy to use 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 traditional computers and differs in some aspects from the standard Berkeley interface. However, programmers used to develop TCP/IP applications for desktop systems 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 is to create a socket by calling NutTcpCreateSocket or NutUdpCreateSocket.

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

A simple TCP client looks like this:

```
TCPSOCKET *sock = NutTcpCreateSocket();

NutTcpConnect(sock, inet_addr("192.168.192.2"), 80);
NutTcpSend(sock, "Hello\r\n", 7);
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpCloseSocket(sock);
```

A Nut/Net TCP server looks similar:

```
TCPSOCKET *sock = NutTcpCreateSocket();

NutTcpAccept(sock, 80);
NutTcpReceive(sock, buff, sizeof(buff));
NutTcpSend(sock, "2U2\r\n", 5);
NutTcpCloseSocket(sock);
```

Furthermore, Nut/OS allows to attach TCP sockets to standard I/O streams, which makes our code look more familiar. The following line is used to attach a connected socket to a Nut/OS stream.

```
stream = _fdopen((int)((uptr_t) sock), "r+b");
```

The code can use fprintf, fputs, fscanf, fgets and other stdio functions to talk TCP.

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 currently unused port number greater than 1023. Data is transferred by calling NutUdpSendTo and NutUdpReceiveFrom, quite similar to the well known BSD functions sendto() and recvfrom(). NutUdpDestroySocket may be called to release all memory occupied by the UDP socket structure.

While UDP read provides a timeout parameter, TCP read doesn't. At least not directly, but some special ioctrls are available, including one to set the read timeout. BSD calls them socket options, and so does Nut/Net.

```
unsigned long to = 1000; /* millisecs */
NutTcpSetSockOpt(sock, SO_RCVTIMEO, &to, sizeof(to));
```

Like with most TCP/IP stacks, there is no provision to set any connection timeout value and NutTcpConnect() may block for half a minute or more. That's how TCP/IP is designed. Most people experience this from the webbrowser on the desktop PC, when trying to connect a webserver that doesn't respond. Nevertheless, this seems to be unacceptable for embedded systems and may be changed in later Nut/Net releases.

Another socket option allows to modify the TCP maximum segment size.

```
unsigned long to = 1024;
NutTcpSetSockOpt(sock, TCP_MAXSEG, &mss, sizeof(mss));
```

It is also possible to define an initial TCP window size.

```
unsigned long win = 8192;
NutTcpSetSockOpt(sock, SO_RCVBUF, &win, sizeof(win));
```

The maximum segment size (MSS) and the initial window size may become useful when you need the maximum TCP throughput. The 8-bit AVR CPU running at 14.7 MHz on the Ethernut easily reaches 2 MBit with the default values.

One last note to a frequently asked question. The number of concurrent connections is limited by memory space only. It is generally no problem to run several TCP and/or UDP server and/or client connections on a single Ethernut. The application may even use the same socket in more than one thread.

HTTP

The Hypertext Transfer Protocol (HTTP) is based on TCP. Nut/Net offers a set of helper APIs to simplify writing Embedded Webservers, which are described in detail in the Nut/OS API Reference.

Just a few lines of code are sufficient to implement a simple webserver:

```
FILE *stream;
TCPSOCKET *sock = NutTcpCreateSocket();

NutTcpAccept(sock, 80);
stream = _fdopen((int) ((uptr_t) sock), "r+b");
NutHttpProcessRequest(stream);
fclose(stream);
NutTcpCloseSocket(sock);
```

Almost all the work is done by the Nut/Net library function NutHttpProcessRequest. The file system is an essential part of a webserver. By default UROM is used, because it is fully hardware independent and available for all targets. Other file systems may be used as well by registering a different HTTP root directory. Here is an example for PHAT:

```
NutRegisterHttpRoot("PHAT0:/");
```

Make sure that the file system is properly mounted.

Basic CGI support is available too. It allows to call C functions to create dynamic HTML content. Here is a simple CGI routine.

```
int ShowParms(FILE *stream, REQUEST *req)
{
    NutHttpSendHeaderTop(stream, req, 200, "Ok");
    NutHttpSendHeaderBottom(stream, req, "text/html", -1);

    fputs("<HTML><BODY><H1>Show Parameters</H1>", stream);

    if (req->req_query) {
        char *name;
        char *value;
        int i;
        int count;

        count = NutHttpGetParameterCount(req);
        for (i = 0; i < count; i++) {
            name = NutHttpGetParameterName(req, i);
            value = NutHttpGetParameterValue(req, i);
            fprintf(stream, "%s: %s<BR>\r\n", name, value);
        }
    }
    fputs("</BODY></HTML>", stream);
    fflush(stream);

    return 0;
}
```

To activate this function for CGI, it must be registered:

```
NutRegisterCgi("showparms.cgi", ShowParms);
```

The function *ShowParms* will be executed when cgi-bin/showparms.cgi is requested by the webbrowser.

Furthermore, basic support is available for password protection, Server Side Includes and Active Server Pages. Please refer to the HTTP application sample that is included in the Nut/OS distribution.

ICMP

The Internet Control Message Protocol.

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.

ARP

The Address Resolution Protocol is used by IP over Ethernet to resolve IP and MAC address relations.

PPP

The Point to Point is as an alternative to the Ethernet protocol and can be used with serial ports, modems, GPRS etc.

Nut/Net provides PPP client mode only. That means, other nodes can be actively connected, but Nut/Net can’t listen to incoming connection attempts. This is related to establishing PPP connections and should not be confused with TCP/UDP client and server capabilities, which are fully available over PPP.

Network Buffers

Nut/Net uses a special internal representation of TCP/IP packets, which is designed for minimal memory allocation and minimal copying when packets are passed between layers. Application programmers don't need to care about this internal stuff. It is included here in case you might become interested in looking into the Nut/Net source code. Network buffers are one of the central data structures within Nut/Net.

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 associated 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.

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 Berkeley 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 project page
http://www.egnite.de	Ethernut hardware vendor

egnite GmbH
Erinstrasse 9
44575 Castrop-Rauxel
Germany

Phone: +49 (0)23 05-44 12 56
Fax: +49 (0)23 05-44 14 87

E-Mail: info@egnite.de

http://www.egnite.de
http://www.ethernut.de