NETPORT Modules

Conitec NETPORT modules are intelligent matchbox-sized I/O modules with Ethernet and USB interface. They can be used for PC port extensions, as well as for distributed I/O applications on the network - such as acquiring data, monitoring processes, or operating relays, solenoids, motors or lamps. All NETPORT modules have stand alone capability - they can store and run scripts for control jobs. This way, I/O control or data aquisition tasks can be performed without the need of a PC.

Control through web interface

No additional software is required to set up a NETPORT module connected through the local network - any browser will do. Every NETPORT module contains a HTTP server. When the module is connected to the LAN or to a USB-Port, it can be configured through a web interface, just like a router. On the web interface, all I/O ports can be either controlled, or simple control scripts in C can be loaded up. The scripts are then executed on the module in an endless loop. The interface design can be adapted to customer requirements.

Control by command line

Scripts commands can be sent to a NETPORT module with a simple command line interface, and are then automatically executed on the module.

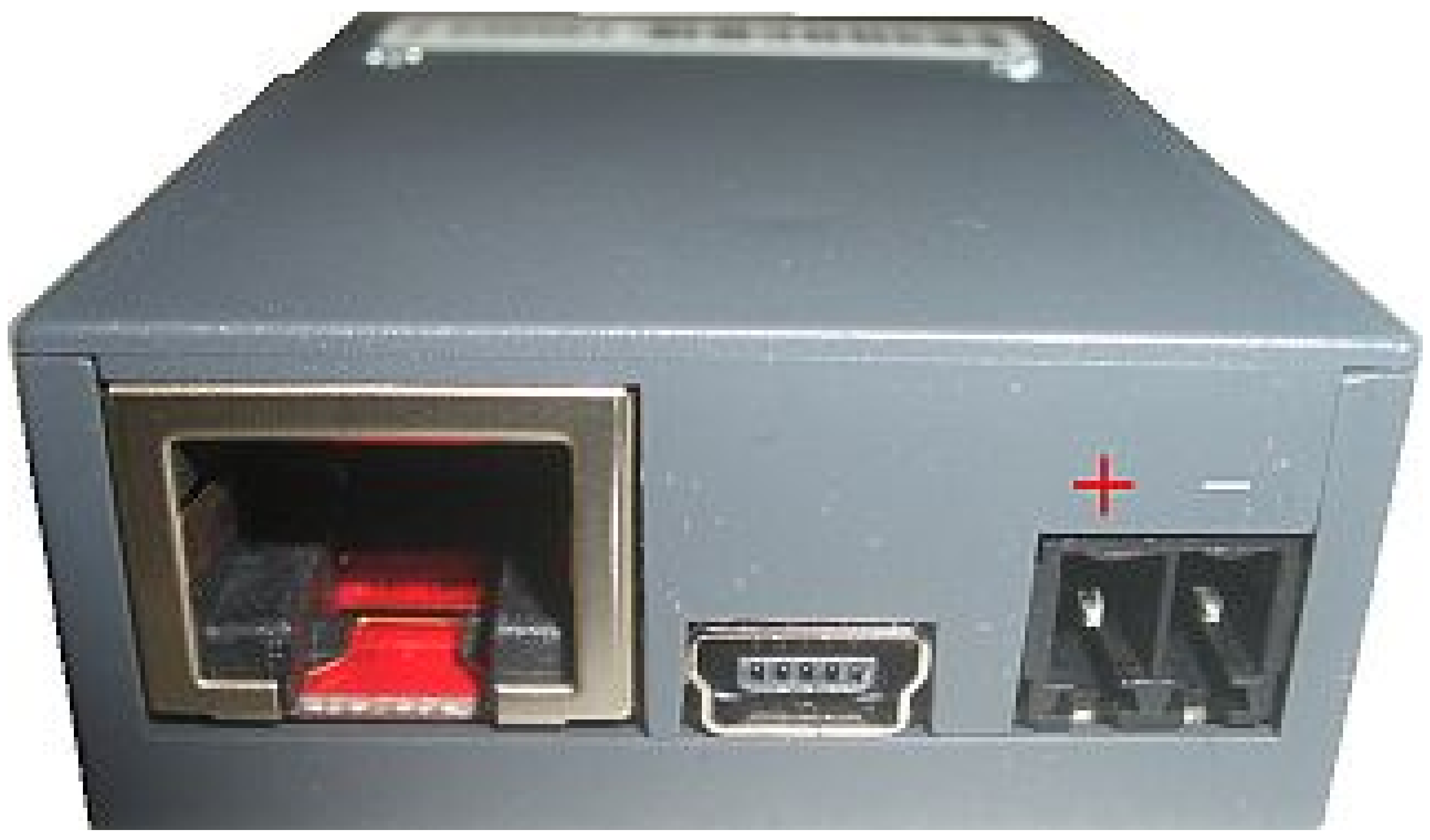
Control by software

For complex control tasks, a driver DLL is available for implementing a NETPORT module in own software. Application development toolkits, such as **Gamestudio**, can directly control NETPORT modules through socket commands.

NETPORT Installation

Every NETPORT module contains a HTTP server. When connected to the LAN or to a USB-Port, it can be configured through a web interface, just like a router. You can access the module's web interface by entering its IP addresse in a web browser. When connected with a LAN, NETPORT modules don't require any driver installation.

Power supply



Power up NETPORT by connecting it to a 12 Volts power adapter, either with the power connector (see above, connector included) or at the DSUB connector. A few seconds later a red LED will blink, indicating the start of the embedded LINUX boot process. After a while a red LED will light steadily. When the boot process is complete, the green LED lights up and a short sound signal can be heard. NETPORT is now ready to run. If a script was stored in Flash memory, it is started now.

Connecting NETPORT to the LAN

After connecting NETPORT with an Ethernet cable to a network and booting up, it can be configured through the **web interface** on IP address **http://192.168.1.12** resp. **http://192.168.1.11**, or alternatively controlled by remote DLL. The IP address can be changed later through the web interface.

Connecting NETPORT to a PC Ethernet port (without network)

Connect NETPORT with an Ethernet crossover cable to the PC's Ethernet port. As your PC is not connected to a normal network now, you need to set up its IP address manually. For this, open TCP/IPv4 Properties of the network connection and set up a fixed IP address (f.i. **192.168.1.1.**). Set the Gateway address at **192.168.1.2.** After booting up NETPORT, you can now configure it through the **web interface** on IP address **http://192.168.1.12** resp. **http://192.168.1.11**, or control it by remote DLL.

Connecting NETPORT to a PC USB port

Connect NETPORT to the PC through an USB cable (if it isn't already). When it is connected via USB the first time, the Windows Hardware Assistant will automatically open for installing the hardware driver. The Windows Hardware Assistant dialogue will vary slightly depending on your Windows version. When asked if you wish to automatically search online for the newest driver version, click [***No***]. In the next dialogues, select the option to install the driver from a specified folder on your PC. In the folder select dialogue, navigate to the **drivers** folder of your NETPORT software. Conclude this step by clicking [***Ok***]. Proceed then by clicking all the following [***Continue***] buttons until the driver installation is complete.
After a successful installation, you'll find an entry for NETPORT in the Windows Network Environment. You can now use any web browser with the IP address **192.168.167.12** to open the NETPORT web Interface.

The USB IP address can be changed later through the web interface.

Specifications NETPORT-48OCA

Mini USB 2.0 and Ethernet 10/100 connector
Size ca. 64 x 42 x 18 mm
DSUB-15 Connector for inputs and outputs
4 open collector outputs, max. 100 mA
2 analog inputs, 10 bits, reference voltage 1.1 / 3.3V
6 universal I/O-lines, configurable for analog/digital I/O
PWM signal generator for motors or servos, jitter < 50 us
Piezo transducer, software control for volume and frequency
Internal ARM9 RISC Processor with Linux kernel
C script language for I/O tasks, directly running on the module
Kit contains: module, power connector, software

Electrical Data

Parameter	Min	Typ	Max	Unit
Supply voltage	8	12	28	V (DC)
Power consumption		1		W
USB current			20	µA
I_{OX}			100	mA
I_{IX}	5		20	mA
fPiezo	800		4000	Hz
SoftPWM Frequency	100		10000	Hz
SoftPWM Jitter			50	µs
Operating temperature	0		60	℃
Storage temperature	-25		75	℃
Type of protection				IP20

Interfaces

Connector		Factory set up	Services
Ethernet	10/100MBPs, Auto Negotiation 00:11:ef:e0:xx:xx bis 00:11:ef:e7:xx:xx	192.168.1.11	FTP, Telnet, Http, MADBridge
USB Mini	Fullspeed (12 MBits/s), Ethernet Emulation, Powered by Device	192.168.167.12	FTP, Telnet, Http, MADBridge
Power	3,81 mm, 2 pin, polarity protected		8..28V

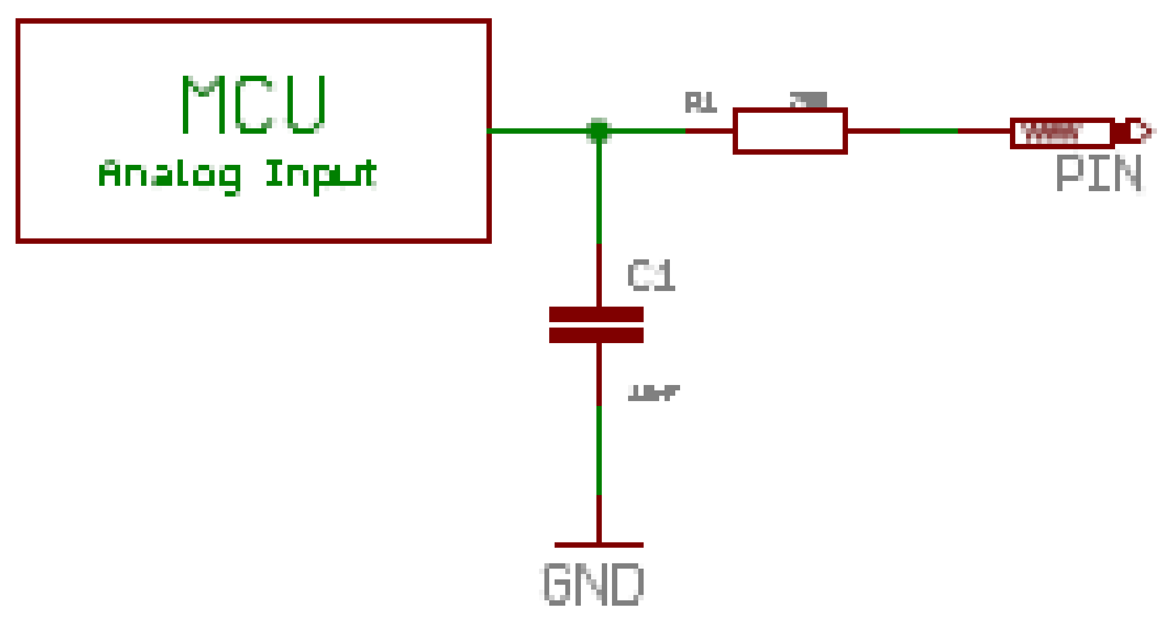
Power supply can be applied through the power connector as well as through pin 8 (VCC) and pin 4 (GND) of the 15-pin DSUB connector.

Inputs / Outputs

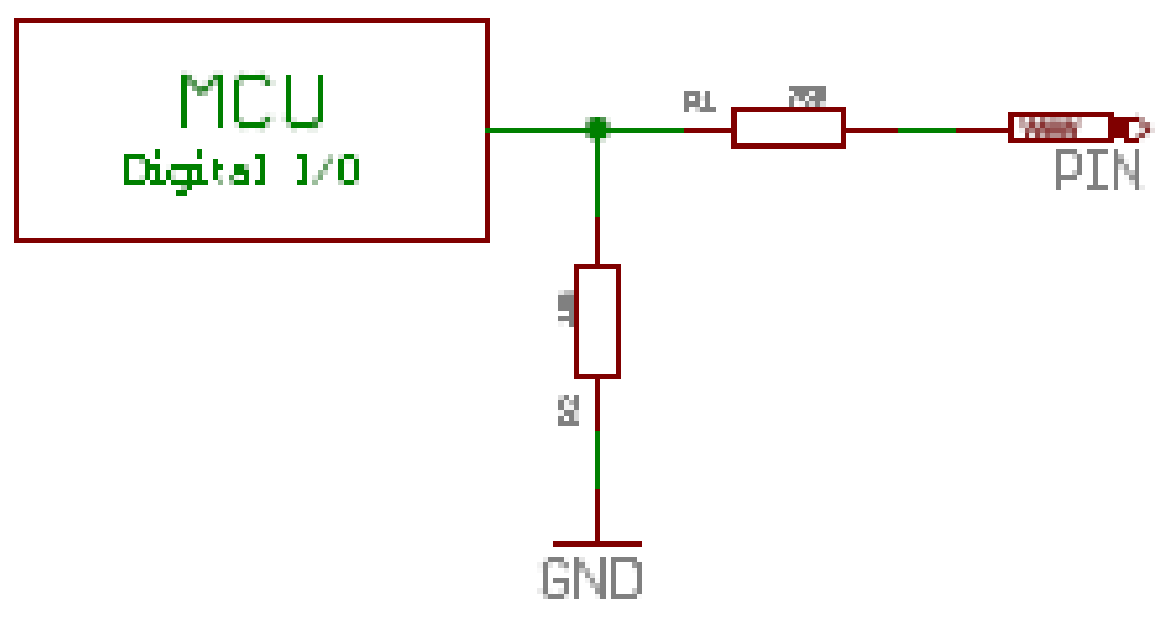
Pin	Signal	Remark
1	AGND	Analog Ground
2	GPIO11 / OC3	Open Collector Output
3	GPIO10 / OC2	Open Collector Output
4	GND	Power GND
5	GPIO9 / OC1	Open Collector Output
6	GPIO8 / OC0	Open Collector Output
7	GPIO0 / AD0	Analog Input
8	VCC	Power VCC
9	GPIO1 / ADIO1	GPIO / Analog Input
10	GPIO2 / ADIO2	GPIO / Analog Input
11	GPIO3 / ADIO3	GPIO / Analog Input
12	GPIO4 / ADIO4	GPIO / Analog Input
13	GPIO5 / ADIO5	GPIO / Analog Input
14	GPIO6 / ADIO6	GPIO / Analog Input
15	GPIO7 / AD7	Analog Input

Circuit diagram open collector output

Circuit diagram analog input GPIO0, GPIO7 with Filter



Circuit diagram input/output GPIO1...GPIO6



Inputs and outputs are not explicitly protected against electrostatic discharges. When handling the module, please follow the same rules as when dealing with sensitive electronic components.

Analog ground and digital ground are internally connected with low resistance. It is recommended that the external circuit keeps the two grounds separate.

For a measurement of analog voltages on **GPIO1** ... **GPIO6** external aliasing filters should be used. For **GPIO0** and **GPIO7** an aliasing filter is already integrated.

Pin Functions

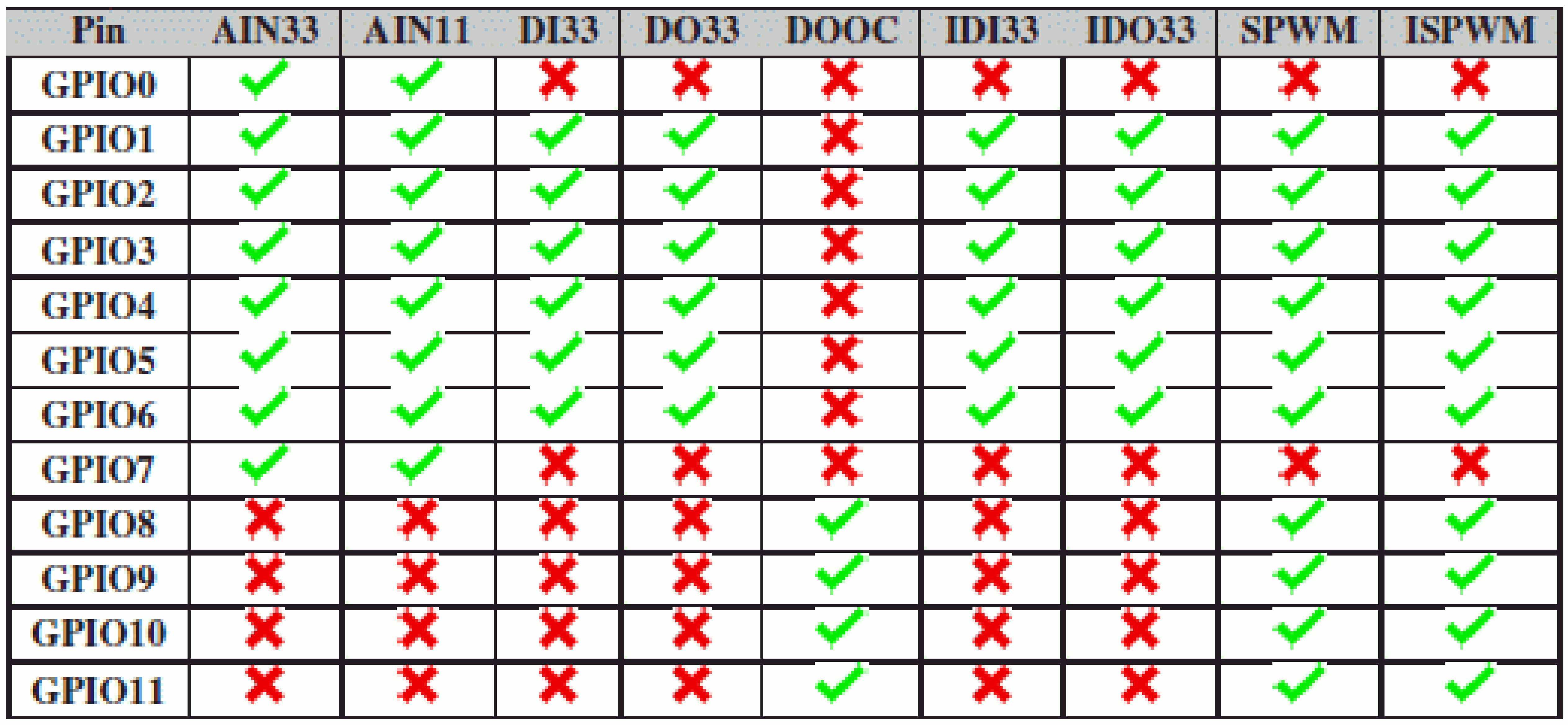
Every I/O Pin can be configured through web interface, script, or remote control. The configuration cosists of:

Pin name
Direction (input or output)
Additional functions (A/D converter, PWM)

The pin name is for documentation only. An I/O pin can have the following functions:

Analog input with 3.3V reference voltage (AIN33)
Analog input with 1.1V reference voltage (AIN11)
Digital input with 3.3V logic level (DI33)
Digital output with 3.3V logic level (DO33)
Digital open collector output (DOOC)
Inverting digital input with 3.3V logic level (IDI33)
Inverting Digital output with 3.3V logic level (IDO33)
Software implemented pulse width modulation (SPWM)
Inverting pulse width modulation (ISPWM)

Not all pins support all functions. The supported pin functions are listed in the following table:



Pin	AIN33	AIN11	DI33	DO33	DOOC	IDI33	IDO33	SPWM	ISPWM
GPIO0	✓	✓	✗	✗	✗	✗	✗	✗	✗
GPIO1	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO2	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO3	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO4	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO5	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO6	✓	✓	✓	✓	✗	✓	✓	✓	✓
GPIO7	✓	✓	✗	✗	✗	✗	✗	✗	✗
GPIO8	✗	✗	✗	✗	✓	✗	✗	✓	✓
GPIO9	✗	✗	✗	✗	✓	✗	✗	✓	✓
GPIO10	✗	✗	✗	✗	✓	✗	✗	✓	✓
GPIO11	✗	✗	✗	✗	✓	✗	✗	✓	✓

Specifications NETPORT-84OCO

Mini USB 2.0 and Ethernet 10/100 connector
Size ca. 64 x 42 x 18 mm
DSUB-15 Connector for inputs and outputs
8 open collector outputs, max. 100 mA
4 opto inputs, 20 mA
PWM signal generator for motors or servos
Piezo transducer, software control for volume and frequency
Internal ARM9 RISC Processor with Linux kernel
C script language for I/O tasks, directly running on the module
Kit contains: module, power connector, software

Electrical Data

Parameter	Min	Typ	Max	Unit
Supply voltage	8	12	28	V (DC)
Power consumption		1		W
USB current			20	µA
I_{OX}			100	mA
I_{IX}	5		20	mA
fPiezo	800		4000	Hz
Operating temperature	0		60	°C
Storage temperature	-25		75	°C
Type of protection				IP20

Interfaces

Connector		Factory set up	Services
Ethernet	10/100MBPs, Auto Negotiation 00:11:ef:e0:xx:xx bis 00:11:ef:e7:xx:xx	192.168.1.12	FTP, Telnet, Http, MADBridge
USB Mini	Fullspeed (12 MBits/s), Ethernet Emulation, Powered by Device	192.168.167.12	FTP, Telnet, Http, MADBridge
Power	3,81 mm, 2 pin, polarity protected		8..28V

Power supply can be applied through the power connector as well as through pin 8 (VCC) and pin 4 (GND) of the 15-pin DSUB connector.

Inputs / Outputs

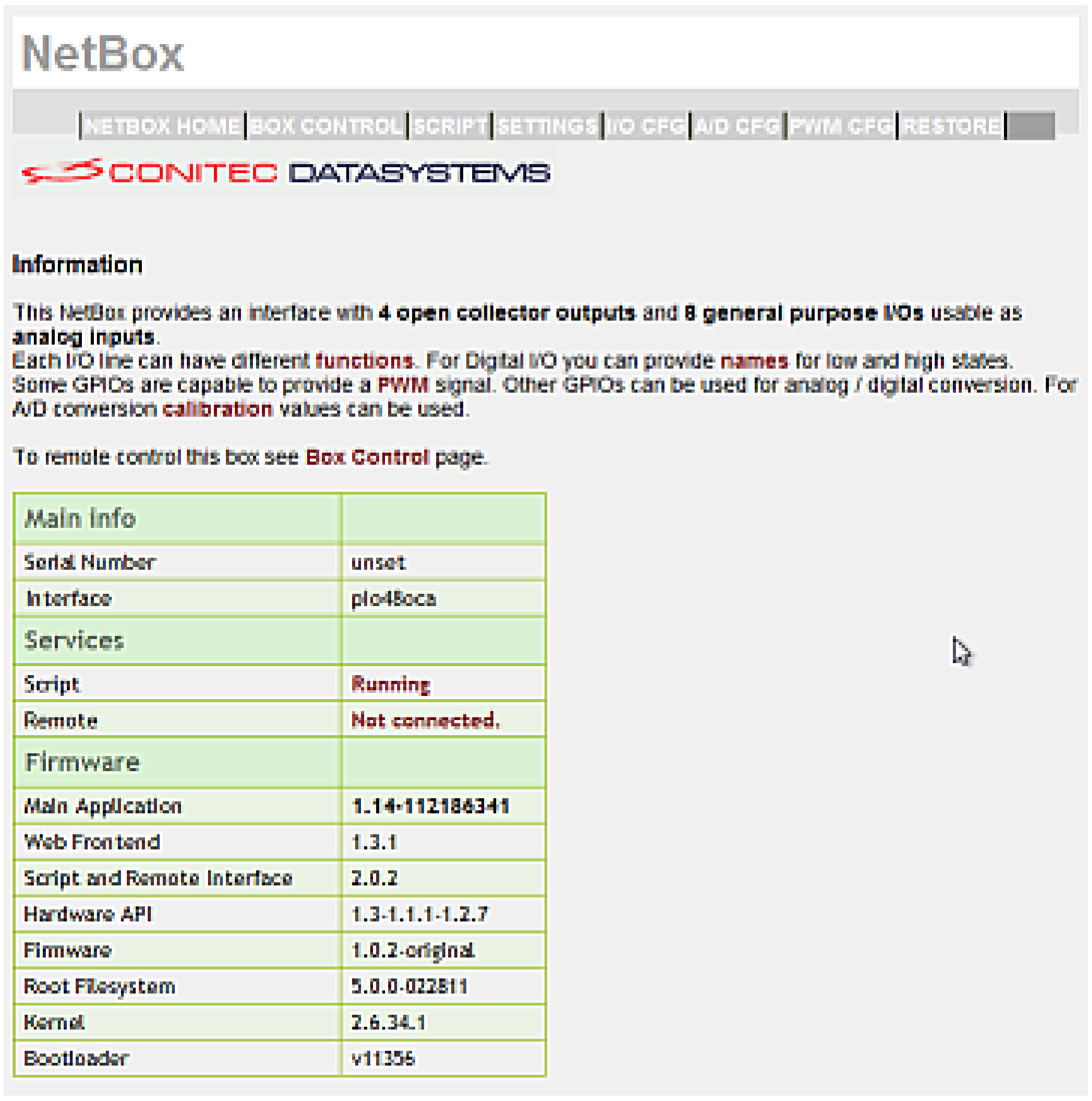
Pin	Signal	Bemerkung
1	OGND	Optocoupler GND
2	IN3	Optocoupler Input
3	IN2	Optocoupler Input
4	GND	Power GND
5	IN1	Optocoupler Input
6	IN0	Optocoupler Input
7	O7	Open Collector Output
8	VCC	Power VCC
9	O6	Open Collector Output
10	O5	Open Collector Output
11	O4	Open Collector Output
12	O3	Open Collector Output
13	O2	Open Collector Output
14	O1	Open Collector Output
15	O0	Open Collector Output

Demo Board

The PIO84 Demo Board allows easy experimenting with the module. It contains 8 LEDs for displaying the state of the outputs, and 4 buttons for feeding the inputs. The board can be equipped with additional power switches and a power supply.

NETPORT Web Interface

NETPORT can be configured through the web interface, similar to a router. You can open the web interface by entering its IP address in your browser (usually **http://192.168.1.12** or **http://192.168.1.11**, see **Installation**).


NetBox
|NETBOX HOME|BOX CONTROL|SCRIPT|SETTINGS|I/O CFG|A/D CFG|PWM CFG|RESTORE|
CONITEC DATASYSTEMS

Information

This NetBox provides an interface with **4 open collector outputs** and **8 general purpose I/Os** usable as **analog inputs**.
Each I/O line can have different **functions**. For Digital I/O you can provide **names** for low and high states.
Some GPIOs are capable to provide a **PWM** signal. Other GPIOs can be used for analog / digital conversion. For A/D conversion **calibration** values can be used.

To remote control this box see **Box Control** page.

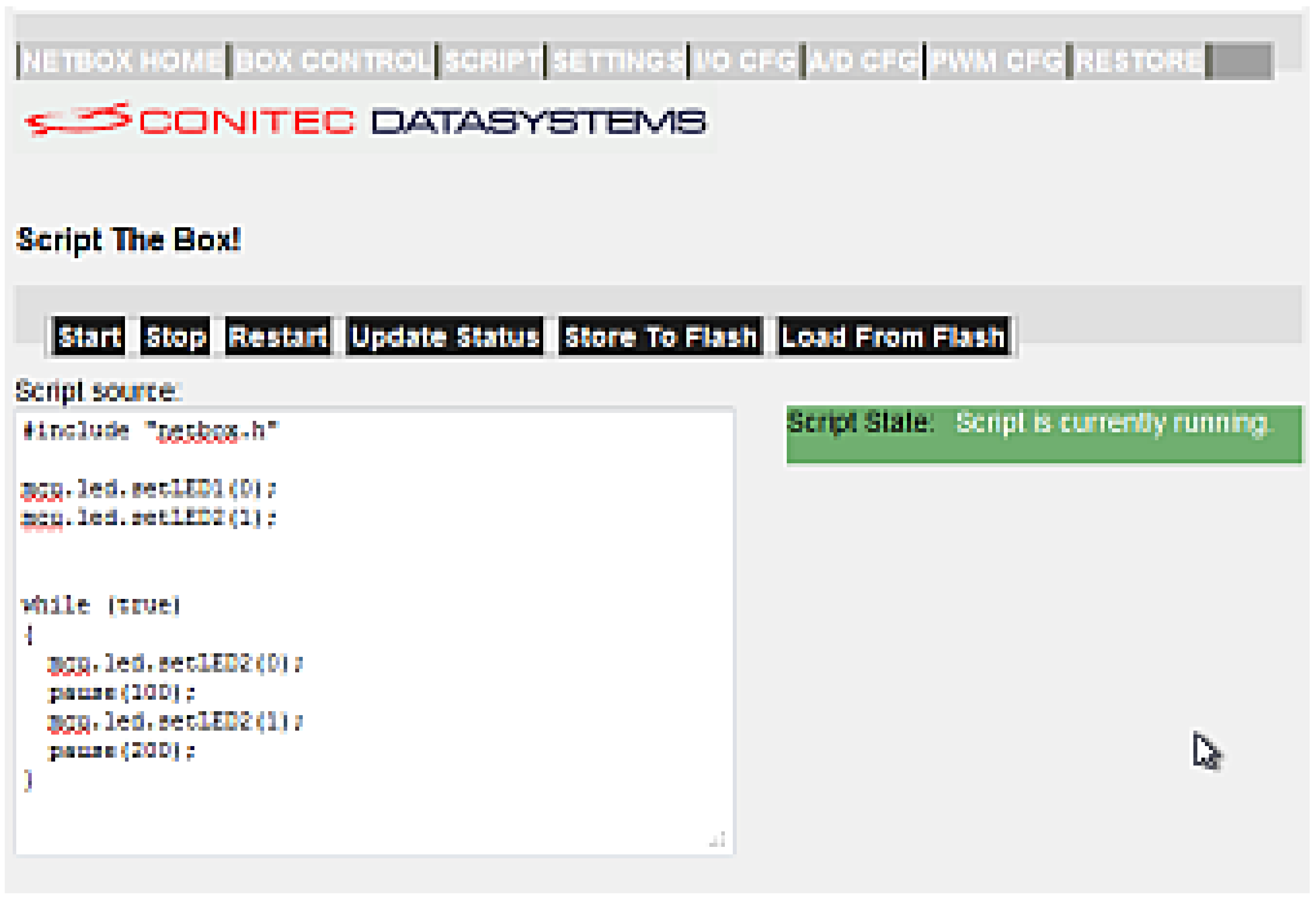
Main Info	
Serial Number	unset
Interface	pio48oca
Services	
Script	Running
Remote	Not connected.
Firmware	
Main Application	1.14-112186341
Web Frontend	1.3.1
Script and Remote Interface	2.0.2
Hardware API	1.3-1.1.1-1.2.7
Firmware	1.0.2-original
Root Filesystem	5.0.0-022811
Kernel	2.6.34.1
Bootloader	v11356

The interface contains the following sections:

Box Control

On this page the outputs can be set (**84oco** only) and the states of the inputs are displayed. Values are updated by clicking on [***Update***]. Depending on ***Settings*** (see below) the signal directions are shown. The manual control of the module is also possible if a script is running on the module or the module is otherwise controlled remotely.

Script



Script The Box!

Script source:

```
#include "netbox.h"

mcu.led.setLED1(0);
mcu.led.setLED2(1);


while (true)
{
  mcu.led.setLED2(0);
  pause(100);
  mcu.led.setLED2(1);
  pause(200);
}
```

Script State: Script is currently running.

On this page, simple C scripts can be entered and run on the module (see **script samples**). The scripts can perform simple procedures or perform complex control tasks. They are entered in the text box, either directly or via copy/paste from a script library. Scripts can be stored on the module so that they run automatically when it's started the next time (see ***Settings***). Syntax or runtime errors are also displayed on the Web interface. The following buttons control script execution:

Start	Starts script execution from the current position.
Stop	Stops script execution.
Restart	Starts the script on the first line.
Update Status	Displays the current script state.
Store To Flash	Stores the script in the flash memory of NETPORT. Stored scripts can be run automatically after booting up, so that NETPORT works independently without a PC connection.
Load From Flash	Loads the script from the NETPORT flash memory and starts it.

Settings

On this page the module can be configured. Changed settings are only effective by clicking on [***Apply Settings***]. The following parameters can be set up:

Script	
Autostart	Run the stored script automatically after booting.
When script fails	Behavior in case of an error - reboot or warning.
When script terminates	Reboot at the end of the script.
Services	
Allow telnet	Remote control via Telnet.
Allow remote access	Remote control via NetBridge (GalepX).
Allow remote scripts	Remote control via Socket resp. Remote.DLL.
Load custom pages	Allow the upload of a customized web interface via FTP zulassen.
Show custom pages	Start the customized web interface by default. The original NETPORT web interface can be displayed anytime by entering a direct page URL (f.i. **http://192.168.167.12/nbHome.html**)
Misc	
Beep after booting	Sound signal after booting.
Ethernet / USB	
Use DHCP configuration	Get the IP address from the DHCP server.
IP Address	Ethernet IP address (default: **192.168.1.11**). After changing the address NETPORT must be rebooted.
Netmask	Network mask.
Gateway	Gateway address (default: **192.168.1.2**)
USB IP Address	USB IP-Adresse (default: **192.168.167.12**). After changing the address, NETPORT must be rebooted.
I/O	
Names	Assign names to the I/O pins.
Initial Direction	Set up the I/O pin type (analog, digital, PWM) and the signal direction.
I/O CFG	Assign names and initial states to the I/O pins.
A/D CFG	Sets voltage gradients and offsets for the analog inputs.
PWM CFG	Sets pulse times for the PWM outputs.
Apply Settings	Applies the configuration.
Store Custom Website	Stores the uploaded custom web interface in flash memory.
Download Configuration	Stores the configuration in a file with extension “**.netbox**”. This file contains all settings such as IP address, I/O configuration and script, as well as the serial number of the module.
Set Defaults	Reset all settings to the default values.
Reboot	Restart the NETPORT module.

Restore

In this section a stored configuration can be fully or partially uploaded.

Restore Script	Loads the script from the configuration file (**.netbox**).
Restore Settings	Loads IP address, I/O configuration and all further settings.
Restore When Serial Number Matches	Checks if the moduel serial number is identical to the serial number in the configuration file. The configuration is only loaded when the number match, otherwise an error message is displayed.
Reboot After Restoring	Automatically restarts the module after loading the configuration file.

Custom

In this section a customer specific web interface is displayed. It can be uploaded by FTP to **ftp://netport_ip/custom-html** (see additional documentation). As long as nothing was uploaded, an example page is opened that displaye the states of the inputs and allows setting the outputs. This example page can be used as a template for a customized web interface.

NETPORT Remote Control

A NETPORT module can be remote controlled by script files or software programs through a TCP socket. All I/O functions are accessible through script commands. The script commands are executed by the script interpreter on the module. Any output is sent back through the socket channel. The script syntax follows ANSI C, with some differences. Pointers are not supported, but some C++ elements, such as exceptions, are supported. You'll find details about the script language in the following chapters.

For sending script commands to the module, aside from using the **web interface** there are many other possibilities:

Remote Control through a TCP socket

The module acts as a socket server at TCP Port **1233**. Socket clients can connect to this port, send a script and this way remote control the box. The same scripts are used as on the **web interface** - with two exceptions:

#include <netbox.h> is optional and can be omitted.
The script should not contain an endless loop, so the command can terminate and send back a response.

All scripts are executed in parallel. When multiple clients connect, multiple script commands are executed at the same time. Every script can access all resources such as I/O lines.

Remote control must be enabled under ***Settings*** [***Allow remote script***].

Remote Control by GalepX

The module can be remote controlled through a MAD Bridge f.i. with the **GalepX** device programmer software. An example of such a control script can be found in the GalepX Distribution under **skripts/pio84oco/pio84oco.gxs**. This script opens a window on the GalepX user interface, which displays the input states of the module and allows to set the outputs with check boxes.

Remote Control by Gamestudio

The multimedia development system Gamestudio (**www.3dgamestudio.de**) can control NETPORT modules with the lite-C commands **socket_connect**, **socket_send**, and **socket_receive**. Details can be found in the Gamestudio manual.

Remote Control by Batch File

The program **remote.exe** is provided for sending commands to NETPORT modules through a batch file. F.i. the command line **remote -ip192.168.1.12 test.gxs test.log** reads commands from the script file **test.gxs**, sends them to the NETPORT at IP address **192.168.1.12**, and stores the output in the **test.log** file. If the name of the log file is omitted, the output is displayed in a window. This way remote command scripts can be easily tested. The **remote.exe** program was written with Gamestudio and can be found in the **remote** subfolder.

Remote Control by DLL

The **remote.dll** library is an easy way to implement GALEP remote control in any user software. Its API contains four simple commands:

int RemoteConnect(char* IP,long iPort)
Opens a TCP socket connection to the given IP address and port, and returns **0** when a socket host could not be found.

void RemoteClose()
Closes the socket connection; must be called before terminating the program.

int RemoteSend(void *data,long size)
Sends the content of the ***data** buffer to the socket, with **size** in bytes.

int RemoteReceive(void *data,long size)
Checks if a data packet was received from the socket. In this case the ***data** buffer is filled and the number of bytes is returned. Otherwise **0** is returned.

The **remote.dll** library can be found in the **remote\API** subfolder, together with a small test program **RemoteTest.cpp** that demonstrates how to implement NETPORT remote control into own programs.

Remote Control by Linux

Under Linux, commands can be sent directly to GalepX through shell scripts. The script command **connect 192.168.1.12 port 1233** opens the socket; the following commands are then sent directly to the socket.

Remote sessions

Remote sessions can run in two modes:

command mode - for sending a command or script to the module. The connection is closed after receipt of a command
interactive mode - for this the command **keep** must be sent first. Afterwards any number of commands or scripts can be sent. The connection is terminated by the client.

Two examples for a command mode session and an interactive session, under a Linux shell:

Command mode

connect localhost port 1233	(start connection - Linux only)
print("Hallo World");\n	(send command)
	(wait for answer)
#10 Hello World \n	(receive answer)
#00 \n	
	(Module terminates connection)

Interactive

connect localhost port 1233	(start connection - Linux only)
keep \n	(command to hold the connection)
print("Hello World");\n	(send command)
	(wait for answer)
#10 Hello World \n	(receive answer)
#00 \n	
...	(send more commands)
	(client terminates connection)

Script commands

There are two types of script commands: **Syntax commands** are always available. **Remote commands** depend on the state of the software - for instance whether a document is opened and which properties it has. A reference of syntax commands and remote commands can be found in the following sections of this help document.

Script output is directly sent to the client as a character string. A prefix of every string allows to determine whether it's an error, a normal output or the termination of a command. The following prefixes are defined:

#00	Termination of a command.	Sent after any command.
#01	Error	Error message follows.
#10	Output from the **print()** command.	

The following conditions lead to error messages:

Problems establishing the network connection
Problems sending/receiving data
Script syntax errors
Script runtime errors

Connection errors and syntax errors lead to immediate termination of the connection (if there was any), and have to be handled by the client.

Runtime errors can be caught with **try { ... } catch() { ... }** exception handling in the script. It makes sense to insert a print command in the catch block so that the client can react on the error. At the moment, all runtime errors cause an exception of type **string**. If an exception is not caught, the connection is terminated.

? latest version online

Variables and Arrays

Variables store numbers. For defining a variable, use a C style declaration, like this:

```
int name; // uninitialized variable
int name = 123; // initialized variable
```

This declaration creates a variable of type **int** with the given **name**. The name can contain up to 30 characters, and must begin with A..Z, a..z, or an underscore _.

Variable types

Computers always perform their calculations with finite accuracy, so all normal variable types are limited in precision and range:

Type	*Size*	*Range*	**Precision**
piInt64	8 bytes	-9223372036854775808 to 9223372036854775807	1
piUInt64	8 bytes	0 to 18446744073709551616	1
int, piInt32	4 bytes	-2147483648 to 2147483647	1
uint, piUInt32	4 bytes	0 to 4294967296	1
piInt8	1 byte	-128 to 127	1
piUInt8	1 byte	0 to 256	1
double	8 bytes	$-1.8 \cdot 10^{308}$ to $1.8 \cdot 10^{308}$	$> 2.2 \cdot 10^{-308}$

Integer constants in the program - such as character constants (**'A'**), integer numeric constants (**12345**) or hexadecimal constants (**0xabcd**) are treated as **int**. Constants containing a decimal point (**123.456**) are treated as **double**.

Arrays

If you add a "**[ ]**" to the type, you can create a variable group, called an **array**:

```
int[] name; // array definition
```

The **append** function can add elements to the end of the array:

```
int[] my_array;  // define a new array
my_array.append(1,2,3); // the array now contains 3 variables with the numbers 1,, 2, 3
my_array.append(4); // add a fourth variable
```

The elements of an array can be accessed with

array[n] *// get or set the n-th element. n must be smaller than the number of elements in the array!*

The number of elements in an array can be retrieved with

array.size();

Elements can be removed with

array.remove(1); *// remove the first element*

Elements can be inserted at a certain place with

array.insert(0, 10); *// insert an element before the first element, and give it the value 10*

Rather than using **append**, the initial size of an array can be set with

array.setSize(100); ***// generate 100 elements, and remove all prior elements***

For testing if an array contains any elements, use

array.empty(); ***// true: array is empty / false: array contains elements***

Finally, for removing all elements from an array, use

array.clear(); ***// remove all elements***

See also:

Strings, **structs**, **functions**

? latest version online

Strings

Strings are a plain sequence of alphanumerical characters - letters, numbers or symbols - which are mostly used for messages, or for identifiers of objects such as projects (documents), actions, buffers, programmers (end devices), etc. They are defined this way:

string name = "characters";

Defines a string with the given **name** and initializes it to the content **characters** between double quotation marks.

Remarks:

Special characters in the string can be given with a backslash: **\n** = Line feed, **\"** = double quote, \\ = back slash.
Strings can be compared with the == operator, f.i. **if (sDocument == "document0") ...**
Strings can be concatenated with the +, += operators, f.i. **string s = "Hello " + "World";**
Strings can be repeated with the *, *= operators, f.i. **string s = "Hello " * 2;**
Arrays of strings can be defined just as arrays of variables.

Example:

```
string device_name = "device";
string empty_str = "";
string[] MyStringArray;
```

The string class has the following functions:

string.length(): int

returns the number of characters in the string.

string.toInt(): int

returns the integer number represented by the string.

string.toDouble(): double

returns the double floating point number represented by the string.

Strings can be formatted with the **message** function and the **<<** pipe operator. The content between **%..%** is replaced by the pipe in order of appearance, e.g:

```
message("Variable %p1% has the content %p2%") << "Test" << 12;
// results in the string "Variable Test has the content 12"
```

See also:

Variables, **structs**, **functions**, **print**, **message**

? latest version online

Structs

A **struct** is an assembled object that contains variables, functions, or further structs (similar to a C++ class). Members of a struct are individually accessed using the struct name, followed by a '.' and the member name. Example of a counter class:

```
struct tCounter
{
  void count()
  {
    ++miValue;
  }

  int miValue;
};


// We create an object...
tCounter cnt;

cnt.count();
```

See also:

Variables, **strings**, **functions**

? latest version online

Expressions

An expression is an arithmetical operation that delivers a result, which can then be assigned to a variable. The arithmetic expression may be composed of any numbers, further variables, function calls, brackets, and arithmetic operators.

The following operators are available in expressions:

=	Assigns the result right of the '=' to the variable left of the '='.
+-*/	The usual mathematical operators. * and / have a higher priority than + and -.
%	Modulo operator, the integer remainder of a division.
|	Bitwise OR, can be used to set certains bits in a variable.
^	Bitwise exclusive OR, can be used to toggle certain bits in a variable.
~	Bitwise invert, toggles all bits of a variable.
&	Bitwise AND, can be used to reset certains bits in a variable.
>>	Bitwise right shift, can be used to divide a positive integer value by 2.
<<	Bitwise left shift, can be used to multiply a positive integer value by 2.
()	Brackets, for defining the priority of mathematical operations. Always use brackets when priority matters!

Examples:

```
x = (a + 1) * b / c;
z = 10;
x = x >> 2; // divides x by 4
x = x << 3; // multiplies x by 8
x = fraction(x) << 10; // copies the fractional part of x (10 bits) into the integer part
```

Assignment operators

The "="-character can be combined with the basic operators:

+=	Adds the result right of the operator to the variable left of the operator.
-=	Subtracts the result right of the operator from the variable left of the operator.
***=**	Multiplies the variable left of the operator by the result right of the operator.
/=	Divides the variable left of the operator by the result right of the operator.
%=	Sets the variable left of the operator to the remainder of the division by the result right of the operator.
|=	Bitwise OR's the the result right of the operator and the variable left of the operator.
&=	Bitwise AND's the the result right of the operator and the variable left of the operator.
^=	Bitwise excöusive OR's the the result right of the operator and the variable left of the operator.
>>=	Bitwise right shift the variable left of the operator by the result right of the operator.
<<=	Bitwise left shift the variable left of the operator by the result right of the operator.

Increment and decrement operators

By placing a '**++**' at the end of a variable, 1 is added; by placing a '**--**', 1 is subtracted. This is a convenient shortcut for counting a variable up or down.

Examples:

```
x = x + 1; // add 1 to x
z += 1; // add 1 to x
x++; // add 1 to x (lite-C only)
```

See also:

Functions, Variables, Comparisons

? latest version online

Comparisons

A comparison is a special type of **expression** that delivers either **true** (nonzero) or **false** (zero) as a result. There are special comparison operators for comparing variables or expressions:

==	True if the expressions left and right of the operator are equal.
!=	True if the expressions left and right of the operator are not equal.
>	True if the expression left of the operator is greater than the expression right of the operator.
>=	True if the expression left of the operator is greater than or equal to the expression right of the operator.
<	True if the expression right of the operator is greater than the expression left of the operator.
<=	True if the expression right of the operator is greater than or equal to the expression left of the operator.
&&	True if the expressions left and right of the operator are both true.
||	True if either of the expressions left and right of the operator is true.
!	True if the expression right of the operator is not true.
()	Brackets, for defining the priority of comparisions. Always use brackets when priority matters!

Remarks:

The "equals" comparison is done with '==', to differentiate it from the assignment instruction with '**=**'. Wrongly using '=' instead of "==" is not noticed by the compiler because it's a valid assignment, but is one of the most frequent bugs in scripts.

Only pointers and integer variables (**int**, **long**, **char** etc., and **var** without decimals) should be compared with '==' or '!='.

Examples:

```
10 < x // true if x is greater than 10
(10 <= x) && (15 => x) // true if x is between 10 and 15
!((10 <= x) && (15 => x)) // true if x is less than 10 or greater than 15 (lite-C only)
```

See also:

Functions, **Variables**, **Expressions**

? latest version online

if (comparison) { instructions... }

else { instructions... }

If the **comparison** between the round brackets is true (i.e. evaluates to non-zero), all instructions between the first pair of winged brackets are executed. It it's not true (i.e. evaluates to zero), the instructions between the second pair of winged brackets following **else** will be executed. The **else** part with the second set of instructions can be omitted. The winged brackets can be omitted when only one instruction is to be executed dependent on the comparison.

Speed:

Fast

Example:

```
if (((x+3)<9) || (y==0))   // set z to 10 if x+3 is below 9, or if y is equal to 0
 z = 10;
else
 z = 5;// set z to 5 in all other cases
```

See also:

comparisons, **while**

while (comparison) { instructions... }

do { instructions... } while (comparison) ;

Repeats all instructions between the winged brackets as long as the **comparison** between the round brackets is true resp. evaluates to non-zero. This repetition of instructions is called a **loop**. The **while** statement evaluates the comparison at the begin, the **do..while** statement at the end of each repetition.

Remarks:

If you want the **loop** to run forever, simply use the value **1** for the comparison - 1 is always true.
Loops can be prematurely terminated by **break**, and prematurely repeated by **continue**.
The winged brackets can be omitted when the loop contains only one instruction.

Example:

```
int x = 0;
while(x < 100) // repeat while x is lower than 100
{
 x += 1;
}
```

See also:

if, **goto**, **break**, **continue**, **comparisons**

? latest version online

for (initialization; comparison; continuation) { instructions... }

Performs the initialization, then evaluates the **comparison** and repeats all instructions between the winged brackets as long as the comparison is true resp. non-zero. The continuation statement will be executed after the instructions and before the next repetition. This repetition of instructions is called a loop. Initialization and continuation can be any **expression** or function call. A **for** loop is often used to increment a counter for a fixed number of repetitions.

Remarks:

Loops can be prematurely terminated by **break**, and prematurely repeated by **continue**.
The winged brackets can be omitted when the loop contains only one instruction.

Example:

```
double x = 3;
for(int i=0; i<5; i++) // repeat 5 times
 x *= x; // calculate the 5th power of x
```

See also:

if, **while**, **goto**, **break**, **continue**, **comparisons**

switch (expression) { case value: instructions... default: instructions... }

The **switch** statement allows for branching on multiple values of a variable or expression. The expression is evaluated and compared with the **case** values. If it matches any of the **case** values, the instructions following the colon are executed. The execution continues until either the closing bracket or a **break** statement is encountered. If the expression does not match any of the **case** statements, and if there is a **default** statement, the instructions following **default:** are executed, otherwise the switch statement ends.

Example:

```
int choice;
...
switch (choice)
{
  case 0:
    print("Zero! ");
    break;
  case 1:
    print("One! ");
    break;
  case 2:
    print("Two! ");
    break;
  default:
    print("None of them! ");
}
```

See also:

if, **while**, **goto**, **break**, **continue**, **comparisons** **? latest version online**

continue

Jumps to the begin of a **while** loop or the continuation part of a **for** loop.

Example:

```
int x = 0;
int y = 0;
while (x < 100)
{
  x+=1;
  if(x % 2) // only odd numbers
  {
    continue; // loop continuing from the start
  }
  y += x; // all odd numbers up to 100 will be sum
}
```

See also:

while, **for**, **break**

? latest version online

break

The **break** statement terminates a loop or a **switch..case** statement, and continues with the first instruction after the closing bracket.

Example:

```
while (x < 100)
{
  x+=1;
  if (x == 50) { break; } // loop will be ended prematurely
}
```

See also:

while, **for**, **switch**, **continue** **? latest version online**

message(string) : string

Returns a string with a placeholder replaced by a number.

Parameters:

string - string with a placeholder between % characters, f.i. **"The value is: %value%"**.

Parameters:

string - formatted string.

Example:

```
print(message("The document ID is: \"%name%\".") << sDocument);
```

See also:

print, **throw**

print(string)

Prints a string through the socket channel.

Parameters:

string - string to print.

Example:

```
print("Test!");
```

See also:

message

pause(int ms)

Does nothing for the given number of millicesonds.

Parameters:

ms - milliseconds to wait.

Example:

```
pause(200);
```

See also:

message

throw object

Throws an exception with the given object. If the exception is not caught, the script is terminated and the object is printed through the output channel.

try { .... } catch(object) { .... }

Catches exceptions with the given object type that occur between the **try { ... }** brackets. When **...** is given for the object type, all remaining exception types are caught.

Example:

```
try {
  if (i == 0)
    throw 1;
  if (i == 2)
    throw "Error";
}
catch(int a)
{
  print(message("Exception %x%!") << a);
}
catch(...)
{
  ...
}
```

See also:

message, **print**

io

Struct for setting outputs, reading inputs, and configuring I/O pins. Not all commands are available for all NETPORT modules.

io.set(int pin_number, bool value)

Sets a digital output pin with the given **pin_number** (**0..11**), to the given **value** (**true, false**).

io.get(int pin_number): bool

Reads a digital input pin in from the given pin number (**0..7**).

io.setPWM(int pin_number, int off_time, int on_time)

Generates a PWM signal on the pin with the given **pin_number** (**0..7**). The **on** and **off time** is given in microseconds. Set both to **0** for disabling the PWM generator.

io.value(int pin_number): int

Reads a universal pin (NETPORT-84oca only), with the given **pin number** (0..7). In case of a digital input, the return value is **0** or **1**; for an analog input it's **0..4095**.

io.out.setMask(int mask)

Sets all output pins from a mask. Bit 0 of the mask corresponds to output **O0**, and so on.

io.in.getMask(): int

Reads all input pins. Bit 0 of the mask corresponds to input **I0**, and so on.

io.setFunction(int pin_number, int mode)

Sets the function of a universal I/O pin (NETPORT-84oca only), with the given **pin number** (0..7). The **mode** is one of the following definitions:

```
enum GPIOConfig
{
   gpioAnalogInput3v3 = 32,
   gpioAnalogInput1v1 = 33,
   gpioDigitalInputLogic = 16,
   gpioInvertedDigitalInputLogic = 272,
   gpioDigitalOutputLogic = 64,
   gpioInvertedDigitalOutputLogic = 320,
   gpioInvertedDigitalOutputLogicSoftPWM = 832,
   gpioDigitalOutputLogicSoftPWM = 576,
   gpioDigitalOutputOC = 65,
   gpioInvertedDigitalOutputOC = 321,
   gpioDigitalOutputOCSoftPWM = 577,
   gpioInvertedDigitalOutputSoftPWM = 833
};
```

Example:

```
// running light script
#include <netbox.h>
void setMask(int iMask)
{
  io.set(1, bool(iMask & 1));
  io.set(2, bool(iMask & 2));
  io.set(3, bool(iMask & 4));
  io.set(4, bool(iMask & 8));
  io.set(5, bool(iMask & 16));
  io.set(6, bool(iMask & 32));
}

while (true)
{
  for (int i = 0; i < 5; ++i)
  {
    setMask(1 << i);
    pause(100);
  }
  for (int i = 0; i < 5; ++i)
  {
    setMask(1 << (5-i));
    pause(100);
  }
}
```

See also:

mcu

Timer

Timer class, for periodic functions or returning the current time.

timer.time(): int

Returns the number of ms since the creation of the timer.

timer. setUTCSecSinceEpoc(piInt64 n)

Sets the number of seconds elapsed since 1.1.1970. Required for the **second**, **minute**, **hour**, **day**, **month**, and **year** functions.

timer.second(): int

Returns the current second (0..59).

timer.minute(): int

Returns the current minute (0..59).

timer.hour(): int

Returns the current hour (0..23).

timer.day(): int

Returns the current day of the month (1..31).

timer.month(): int

Returns the current month (1..12).

timer.year(): int

Returns the current year.

timer.setOnTimeOut(string name)

Returns the number of ms since the creation of the timer.

timer.start(int ms)

Returns the number of ms since the creation of the timer.

timer.stop()

Returns the number of ms since the creation of the timer.

Sets the second LED on or off.

Example:

```
// fading out acoustic signal
#include "netbox.h"
for (int i = 0; i < 100; ++i)
```

```
{
  mcu.piezo.beep(1000, 100-i);
  pause(1);
}
mcu.piezo.off();
// Wait forever...
while (true) pause();
```

See also:

mcu, **pause**

mcu

Struct for setting peripherals, such as LED and piezo speaker.

mcu.piezo.beep(int frequency, int volume)

Generates a sound signal with the given **frequency** (kHz) and **volume** (0..100).

mcu.piezo.off()

Switches off the piezo beeper.

mcu.led.setLED1(bool value)

Sets the first LED on or off.

mcu.led.setLED2(bool value)

Sets the second LED on or off.

Example:

```
// fading out acoustic signal
#include "netbox.h"
for (int i = 0; i < 100; ++i)
{
  mcu.piezo.beep(1000, 100-i);
  pause(1);
}
mcu.piezo.off();
// Wait forever...
while (true) pause();
```

See also:

mcu

Configuration functions

The following functions are available from MainApplication version **1.20** resp. firmware version **2.0.6** or above. They can be used for exchanging data between the main script running on the box, and additional scripts sent through a remote control channel.

getVariable(string name) : string

Returns the string representation of the content of the global variable with the given **name**. If the variable does not exist, an empty string is returned.

setVariable(string name, string value)

Sets the global variable with the given **name** to the given **value**. If the variable does not exist, it is created.

getConfig(string name) : string

Returns the string representation of the content of the configuration variable with the given **name**. If the variable does not exist, an empty string is returned.

setConfig(string name, string value)

Sets the configuration variable with the given **name** to the given **value**. If the variable does not exist, it is created. It's not yet stored in Flash memory; for this the function **storeConfig()** must be called.

storeConfig()

Stores all configuration variables in Flash memory.

loadConfig()

Loads all configuration variables from Flash memory.

Example:

See also:

message, **print**

NETPORT Script Examples

The following example scripts can be directly copied and tested in the web interface:

Blinker

```
#include <netbox.h>

// blink output O0 (100 ms on, 200 ms off)
while(true)
{
  io.set(0,true);
  pause(100);
  io.set(0,false);
  pause(200);
}
```

Running light

```
#include <netbox.h>

int iDir = 1;
int iIdx = 0;
while (true)
{
  iIdx += iDir;
  if ((iIdx >= 7) ||(iIdx == 0))
  iDir *= -1;
  io.out.setMask(1 << iIdx);
  pause(100);
}
```

Copy input I0 to output O0

```
#include <netbox.h>

// All off.
io.out.setMask(0);
while (true)
{
  bool bo = io.get(0);
  io.set(0, bo);
}
```

Set output O0 to (I0 or I1)

```
#include <netbox.h>

// All off.
io.out.setMask(0);
enum Switches
{
  sw1 = (1 << 0),
  sw2 = (1 << 1),
};

while (true)
{
  bool bo = io.in.getMask() & (sw1 |sw2);
  io.set(0,bo);
}
```

Read an analog input (48oca only)

```
#include <netbox.h>

while (true)
{
  bool boState = io.value(2) > 800;
  io.set(1, boState); // set output when input voltage > 800 units
}
```

Generate PWM signals

```
#include <netbox.h>

// 1ms = 1000us
const int us = 1;
const int ms = 1000;
io.setPWM(0, 15*ms, 100*us);
io.setPWM(1, 15*ms, 200*us);
io.setPWM(2, 15*ms, 400*us);
io.setPWM(3, 15*ms, 1*ms);
io.setPWM(4, 15*ms, 2*ms);
io.setPWM(5, 15*ms, 3*ms);
io.setPWM(6, 15*ms, 4*ms);
io.setPWM(7, 15*ms, 5*ms);
// wait forever...
while (true) pause();
```

Configure inputs and outputs (48oca only)

```
#include <netbox.h>

io.setFunction(0, gpioAnalogInput3v3);
io.setFunction(1, gpioDigitalInputLogic);
io.setFunction(2, gpioDigitalOutputLogic);
io.setFunction(3, gpioDigitalInputLogic);
io.setFunction(4, gpioDigitalInputLogic);
io.setFunction(5, gpioDigitalInputLogic);
io.setFunction(6, gpioDigitalInputLogic);
io.setFunction(7, gpioAnalogInput3v3);
```

Generate a short sound (0.2 sec 1kHz)

```
#include <netbox.h>

mcu.piezo.beep(1000,100);
pause(200);
mcu.piezo.off();
// Wait forever...
while (true) pause();
```

Generate a fading sound

```
#include <netbox.h>

for (int i = 0; i < 100; ++i)
{
  mcu.piezo.beep(1000, 100-i);
  pause(1);
}
mcu.piezo.off();
// Wait forever...
while (true) pause();
```

Function to store current output states to flash

```
void storeOutputState()
{
  int iCurrentState = io.out.getMask();
  string s = message("%d%") << iCurrentState;
  setVariable("OutputState", s);
  storeConfig();
}
```

Function that returns the output states stored in flash

```

int storedOutputState()
{
  int iOldPrevious = 0;
  try {
    iOldPrevious = getVariable("OutputState").toInt();
  }
  catch(...)
  {
     // Variable was not defined or not a valid integer...
  }
  return iOldPrevious;
}
```

Functions to update the flash with the output state

```
// Call this function at the beginning of your script to restore output states.
void restoreOutputState()
{
  int iOld = storedOutputState();
  io.out.setMask(iOld);
}

// Call this function periodically or each time you change the output states.
// When output states has been changed we store this persistently.
void storeWhenChanged()
{
  int iCurrentState = io.out.getMask();
  int iOld = storedOutputState();

  if (iOld != iCurrentState)
  {
     print(message("Different output states: %old% / %current%! Will store values...") << iOld << iC
     storeOutputState();
  }
}
```