NETPORT Module

Conitec NETPORT Module sind intelligente Ein/Ausgabe-Module im Steckergehäuse mit Ethernet- und USB-Anschluss. Sie sind einsetzbar als PC-I/O-Erweiterung, aber auch für eigenständige, verteilte I/O-Anwendungen im Netzwerk - z.B. Datenerfassung, Prozessüberwachung, Ansteuerung von Relais, Motoren, Servos oder Lampen. Alle NETPORT Module sind stand alone-fähig - sie können selbständig Skripte speichern und ausführen. Auf diese Weise lassen sich beliebige Steuer- und Überwachungsaufgaben durchführen, ohne dass ein PC zur Steuerung erforderlich ist.

Steuerung per Web-Interface

Um ein am Netzwerk angeschlossenes Modul zu konfigurieren, ist keine Zusatzsoftware erforderlich - ein Browser genügt. Jedes NETPORT-Modul enthält einen HTTP-Server. Sobald das Modul mit dem LAN oder mit einem USB-Port verbunden wurde, lässt es sich über ein Webinterface konfigurieren, ähnlich wie ein Router. Per Webinterface können I/O-Ports direkt gesetzt oder abgefragt oder einfache Skripte in einer C-Sprache eingegeben werden. Diese Skripte werden dann auf dem Modul in einer Endlosschleife ausgeführt. Das Web-Interface Design kann an eigene Anforderungen angepasst werden.

Steuerung per Befehlszeile

Skripte zur Steuerung lassen sich auch mit einem einfachen Befehlszeileninterface an ein NETPORT-Modul übertragen und dort automatisch ausführen.

Steuerung per Software

Für komplexe Steueraufgaben steht eine DLL zur Verfügung, um NETPORT Module in eigene Software einzubinden. Softwarebaukästen, wie z.B. **Gamestudio**, können NETPORT-Module per Socket-Interface direkt steuern.

NETPORT Inbetriebnahme

Jedes NETPORT-Modul enthält einen HTTP-Server. Sobald das Modul mit dem LAN oder mit einem USB-Port verbunden wurde, lässt es sich über ein Webinterface konfigurieren, ähnlich wie ein Router. Sie erreichen das Modul, indem Sie dessen IP-Adresse in Ihren Browser eingeben. Bei einem Betrieb am Netzwerk ist keine weitere Treiberinstallation erforderlich.

Anschluss der Stromversorgung

Verbinden Sie NETPORT entweder am Netzteilanschluss (s.o., passender Stecker im Lieferumfang) oder am DSUB-Stecker (s. Spezifikation) mit einer 12V-Stromversorgung. Nach einigen Sekunden beginnt die rote LED zu blinken - das interne LINUX System bootet hoch. Nach einer Weile leuchtet die rote LED stetig. Wenn der Boot-Prozess fertig ist, leuchtet die grüne LED und ein kurzes Tonsignal ertönt. NETPORT ist nun betriebsbereit. Falls ein Skript im Flash gespeichert wurde, wird dieses nun gestartet.

Betrieb von NETPORT am Netzwerk

Wenn NETPORT über ein Ethernet-Kabel mit Ihrem Netzwerk verbunden ist, können Sie nach dem Hochbooten per Browser auf der IP-Adresse **http://192.168.1.12** bzw. **http://192.168.1.11** das **Webinterface** öffnen oder NETPORT per Remote-DLL steuern. Die IP-Adresse lässt sich im Webinterface ändern.

Betrieb von NETPORT am PC Ethernet Port (ohne Netzwerk)

Verbinden Sie NETPORT über ein Ethernet-Crossover-Kabel mit dem Ethernet-Port Ihres PC. Da Ihr PC jetzt nicht mit einem normalen Netzwerk verbunden ist, müssen Sie ihm seine IP-Adresse manuell zuweisen. Hierfür öffnen Sie die TCP/IPv4 Eigenschaften der LAN-Verbindung und stellen eine feste IP-Adresse für Ihren PC ein (z.B. **192.168.1.1.**). Setzen Sie die Gateway-Adresse auf **192.168.1.2.** Sie können nun nach dem Hochbooten per Browser auf der IP Adresse **http://192.168.1.12** bzw. **http://192.168.1.11** das Webinterface öffnen oder NETPORT per Remote-DLL steuern.

Betrieb von NETPORT am PC USB Port

Wenn NETPORT über ein USB-Kabel das erste Mal mit Ihrem PC verbunden wird, öffnet sich nach dem Hochbooten automatisch der Windows Hardware-Assistent, um den USB-Treiber zu installieren. Dieser Treiber befindet sich im **drivers** Ordner der NETPORT Software. Der Dialog ist je nach Windows-Version leicht unterschiedlich. Die Frage, ob Sie automatisch online nach dem neuesten Treiber suchen wollen, beantworten Sie mit [***Nein***]. In den folgenden Dialogen wählen Sie die Option, den Treiber aus einem Ordner auf Ihrem PC zu installieren. Im folgenden Auswahldialog navigieren Sie zu dem **drivers** Ordner der NETPORT Software. Bestätigen Sie mit [***Ok***]. Bestätigen Sie dann alle weiteren Abfragen mit [***Weiter***], bis die Treiberinstallation komplett ist.

Nach einer erfolgreichen Installation finden Sie einen Eintrag für NETPORT in der Windows Netzwerk-Umgebung. Sie können nun per Browser auf der dem USB Port zugeordneten IP Adresse **http://192.168.167.12** das Webinterface öffnen oder NETPORT per Remote-DLL steuern.

Technische Daten NETPORT-48OCA

Mini USB 2.0 und Ethernet 10/100 Anschluss
Abmessungen ca. 64 x 42 x 18 mm
DSUB-15 Stecker für Ein/Ausgänge
4 Open-Collector Ausgänge, max. 100 mA
2 Analog-Eingänge, 10 Bit, Referenzspannung 1.1 / 3.3V
6 Universal-Ein/Ausgänge, umkonfigurierbar analog/digital
PWM Signalgenerator für Motoren oder Servos, Jitter < 50 us
Piezo Signalgeber, Lautstärke und Frequenz einstellbar
Interner ARM9 RISC Prozessor mit Linux Kernel
C Skriptsprache für I/O Tasks, läuft direkt auf dem Modul

Elektrische Daten und Einsatzbedingungen

Parameter	Min	Typ	Max	Einheit
Versorgungsspannung	8	12	28	V (DC)
Leistungsaufnahme		1		W
Stromaufnahme USB			20	µA
I_{OX}			100	mA
I_{IX}	5		20	mA
fPiezo	800		4000	Hz
SoftPWM Frequenz	100		10000	Hz
SoftPWM Jitter			50	µs
Zul. Betriebstemperatur	0		60	°C
Lagertemperatur	-25		75	°C
Schutzart				IP20

Schnittstellen

Schnittstelle		Werkseinstellung	Dienste
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
Stromversorgung	3,81 mm, 2polig, verpolungssicher		8..28V

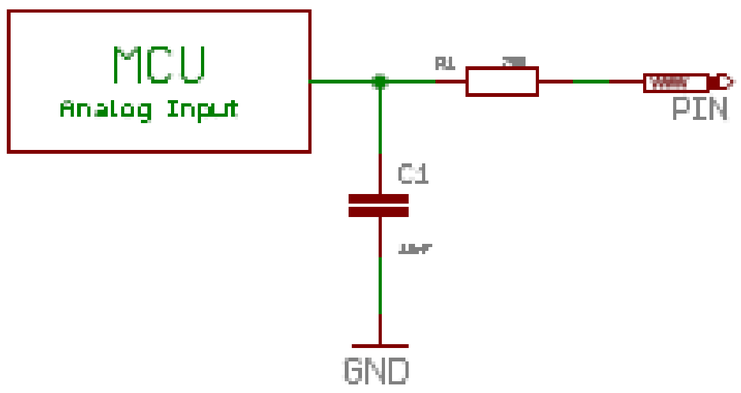
Das Modul kann sowohl über die Stromversorgungsbuchse, als auch über Pin 8 (VCC) und Pin 4 (GND) des 15-poligen DSUB Steckers versorgt werden.

Ein/Ausgänge

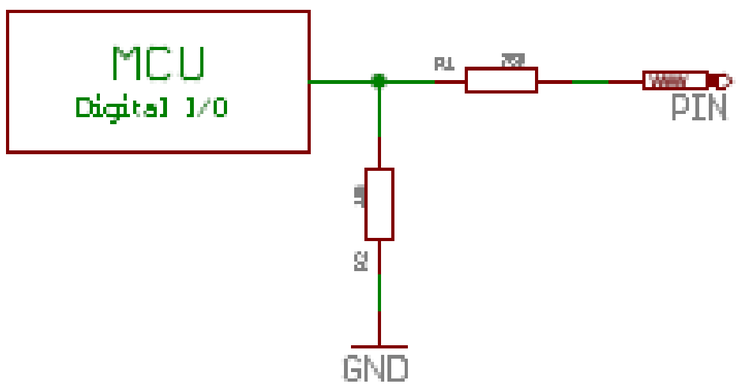
Pin	Signal	Bemerkung
1	AGND	Analog Ground
2	GPIO11 / OC3	Open Collector Ausgang
3	GPIO10 / OC2	Open Collector Ausgang
4	GND	Stromversorgung GND
5	GPIO9 / OC1	Open Collector Ausgang
6	GPIO8 / OC0	Open Collector Ausgang
7	GPIO0 / AD0	Analogeingang mit Filter
8	VCC	Stromversorgung VCC
9	GPIO1 / ADIO1	GPIO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

Innenschaltung Open Collector Ausgang

Innenschaltung Analogeingang GPIO0, GPIO7 mit Filter



Innenschaltung Ein/Ausgang GPIO1...GPIO6



Die Ein- und Ausgänge sind nicht explizit gegen elektrostatische Entladungen gesichert. Bitte beachten Sie beim Umgang mit dem Modul die Regeln, die auch beim Umgang mit empfindlichen elektronischen Bauteilen gelten.

Analogmasse und Digitalmasse sind intern niederohmig verbunden. Es ist bei der externen Beschaltung sinnvoll, beide Massen getrennt zu führen.

Für eine Messung analoger Spannungen an **GPIO1**...**GPIO6** sollte ein externes Aliasingfilter beschaltet werden. Bei **GPIO0** und **GPIO7** ist bereits ein Aliasingfilter integriert.

Pinfunktionen

Jedes I/O Pin kann per Web-Interface, per Skript oder per Fernsteuerung konfiguriert werden. Zur Konfiguration gehören:

Name des Pins
Portrichtung (Ein- oder Ausgang)
Zusatzfunktionen (Analogeingang, SoftPWM)

Der Name eines I/O Pins hat rein dokumentatorischen Wert. Er wird im Webinterface angezeigt und kann frei vergeben werden. Ein I/O Pin kann folgende Funktionen haben:

Analoger Eingang mit 3.3V Referenzspannung (AIN33)
Analoger Eingang mit 1.1V Referenzspannung (AIN11)
Digitaler Eingang 3.3V Pegel (DI33)
Digitaler Ausgang 3.3V Pegel (DO33)
Digitaler Openkollektorausgang (DOOC)
Invertierender digitaler Eingang 3.3V Pegel (IDI33)
Invertierender digitaler Ausgang 3.3V Pegel (IDO33)
Software-implementierte Pulslängen-Modulation (SPWM)
Invertierte Pulslängen-Modulation (ISPWM)

Nicht alle Pins unterstützen alle Funktionen. Folgende Tabelle zeigt die verfügbaren Funktionen der einzelnen Pins:

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

Technische Daten NETPORT-84OCO

Mini USB 2.0 und Ethernet 10/100 Anschluss
Abmessungen ca. 64 x 42 x 18 mm
DSUB-15 Stecker für Ein/Ausgänge
8 Open-Collector Ausgänge, max. 100 mA
4 Optokoppler-Eingänge, 20 mA
PWM Signalgenerator für Motoren oder Servos
Piezo Signalgeber, Lautstärke und Frequenz einstellbar
Interner ARM9 RISC Prozessor mit Linux Kernel
C Skriptsprache für I/O Tasks, läuft direkt auf dem Modul

Elektrische Daten und Einsatzbedingungen

Parameter	Min	Typ	Max	Einheit
Versorgungsspannung	8	12	28	V (DC)
Leistungsaufnahme		1		W
Stromaufnahme USB			20	µA
I_{OX}			100	mA
I_{IX}	5		20	mA
fPiezo	800		4000	Hz
Zul. Betriebstemperatur	0		60	°C
Lagertemperatur	-25		75	°C
Schutzart				IP20

Schnittstellen

Schnittstelle		Werkseinstellung	Dienste
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
Stromversorgung	3,81 mm, 2polig, verpolungssicher		8..28V

DasModul kann sowohl über die Stromversorgungsbuchse, als auch über Pin 8 (VIn) und Pin 4 (GND) des 15-poligen DSUB Steckers versorgt werden.

Ein/Ausgänge

Pin	Signal	Bemerkung
1	OGND	Optokoppler Ground
2	IN3	Optokoppler Eingang
3	IN2	Optokoppler Eingang
4	GND	Stromversorgung GND
5	IN1	Optokoppler Eingang
6	IN0	Optokoppler Eingang
7	O7	Open Collector Ausgang
8	VCC	Stromversorgung VCC
9	O6	Open Collector Ausgang
10	O5	Open Collector Ausgang
11	O4	Open Collector Ausgang
12	O3	Open Collector Ausgang
13	O2	Open Collector Ausgang
14	O1	Open Collector Ausgang
15	O0	Open Collector Ausgang

Innenschaltung Optokoppler

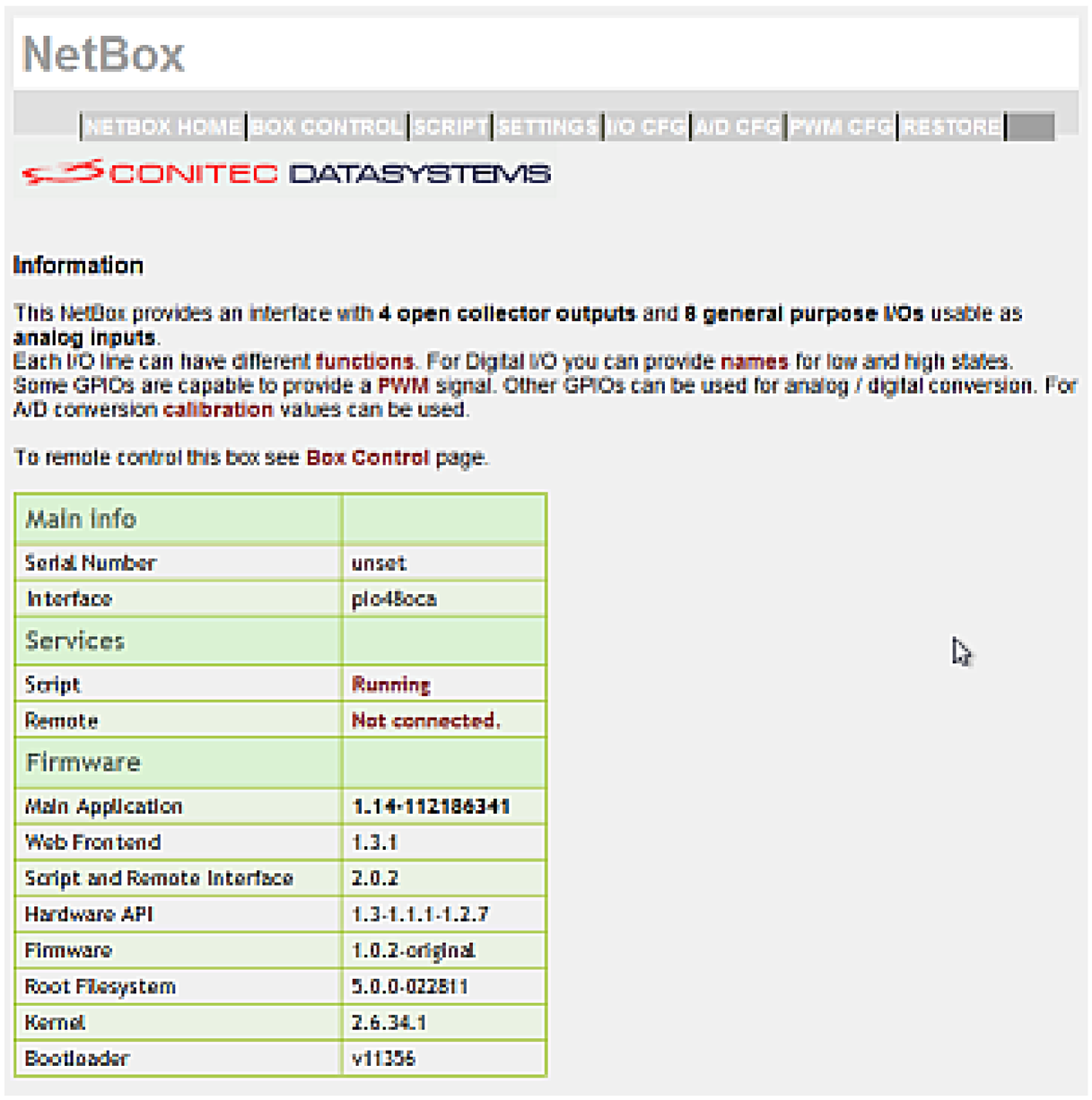
Innenschaltung Open Collector

Demoboard

Das PIO84 Demoboard erlaubt den einfachen Einstieg in die Arbeit mit dem Modul. Es verfügt über acht LEDs, welche die Ausgangszustände anzeigen und vier Taster mit denen die Eingänge stimuliert werden können. Je nach Bestückungsoption können zusätzlich Leistungsschalter, sowie eine schaltbare Spannungsquelle vorhanden sein.

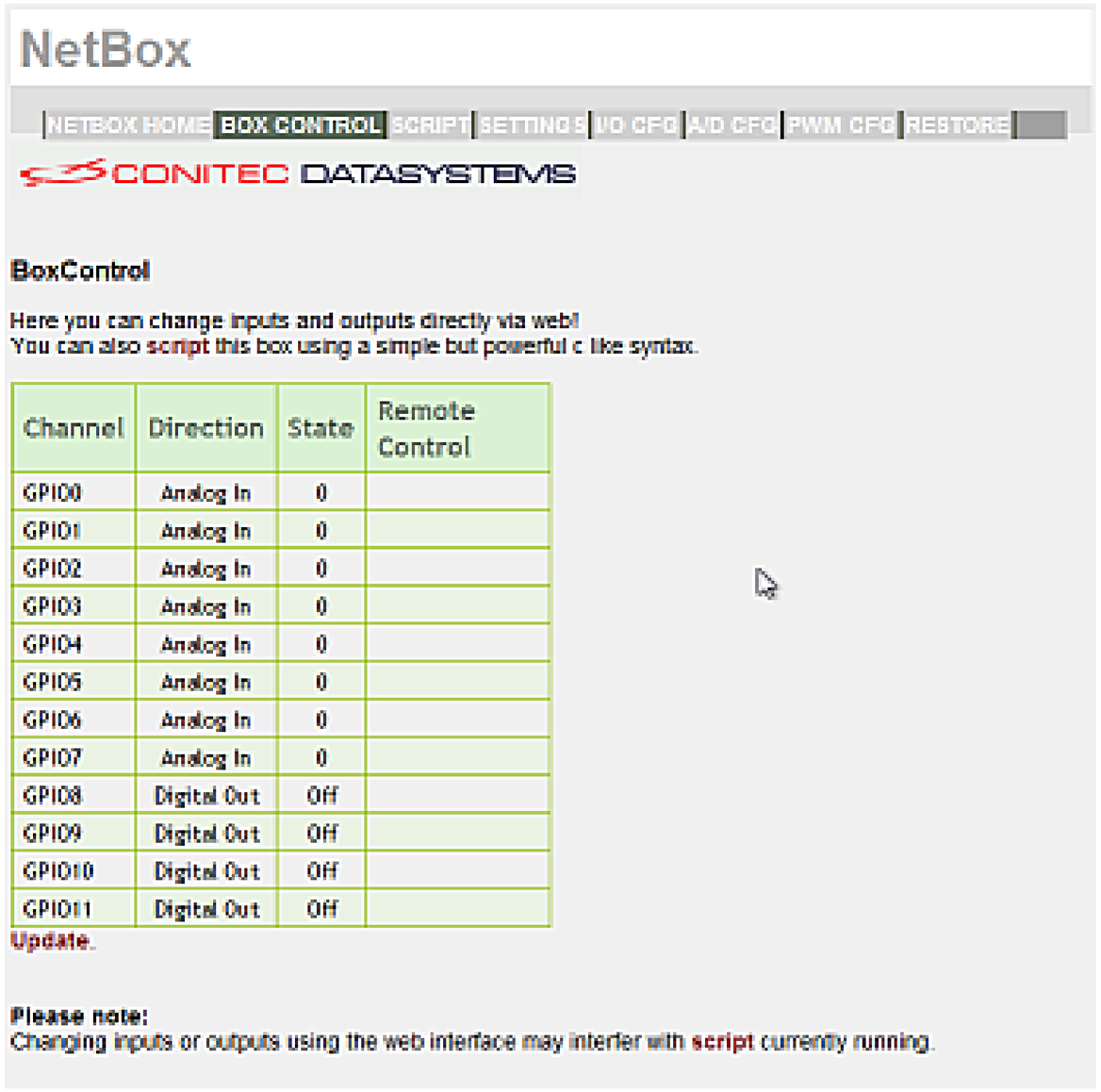
NETPORT Web Interface

Über das Webinterface lässt sich NETPORT konfigurieren, ähnlich wie ein Router. Sie erreichen es, indem Sie die IP-Adresse (normalerweise **http://192.168.1.12** bzw. **http://192.168.1.11**, siehe **Inbetriebnahme**) in Ihren Browser eingeben.



Das Interface ist unterteilt in folgende Seiten:

Box Control



Hier können die Ausgänge gesteuert (nur **84oco**) und der Zustand der Eingänge dargestellt werden. Werte werden durch Klick auf [***Update***] aktualisiert. Je nach Einstellung unter ***Settings*** (s.u.) werden die Signalrichtungen dargestellt. Die manuelle Steuerung des Moduls ist auch dann möglich, wenn ein Skript auf dem Modul ausgeführt oder das Modul anderweitig ferngesteuert wird.

Script

Hier lassen sich einfache C-Skripte auf das Modul laden und ausführen (s. **Skriptbeispiele**). Die Skripte können einfache Abläufe realisieren oder auch komplexe Steuerungsaufgaben erledigen. Sie werden direkt per Weboberfläche eingegeben oder per Copy/Paste aus einer Skriptbibliothek geladen. Skripte lassen sich auf dem NETPORT speichern, so dass sie beim nächsten Start automatisch ausgeführt werden (s. ***Settings***). Syntax- oder Laufzeitfehler werden auf der Weboberfläche dargestellt. Mit den folgenden Buttons wird die Skriptausführung gesteuert:

Start	Startet die Skriptausführung an der aktuellen Position.
Stop	Stoppt die Skriptausführung.
Restart	Startet das Skript am Beginn.
Update Status	Zeigt den aktuellen Status des Skripts an.
Store To Flash	Speichert das Skript im Flash-Speicher des NETPORT. Gespeicherte Skripte können automatisch nach dem Hochbooten ausgeführt werden, so dass NETPORT eigenständig ohne PC-Anschluss arbeiten kann.
Load From Flash	Lädt das Skript aus dem Flash-Speicher des NETPORT und startet es.

Settings

Hier können die Einstellungen des Moduls verändert werden. Neue Einstellungen werden erst wirksam durch Klick auf [***Apply Settings***]. Die folgenden Einstellungen stehen zur Verfügung:

Script	
Autostart	Gespeichertes Skript automatisch nach dem Hochbooten starten.
When script fails	Verhalten nach einem Skriptfehler - Neustart und/oder Warnsignal.
When script terminates	Neustart nach Skriptende.
Services	
Allow telnet	Fernsteuerung per Telnet zulassen.
Allow remote access	Fernsteuerung per NetBridge (GalepX) zulassen.
Allow remote scripts	Fernsteuerung per Socket bzw. Remote-DLL zulassen.
Load custom pages	Hochladen von einem eigenem Webinterface per FTP zulassen.
Show custom pages	Anzeige des eigenen Webinterface per Default. Das ursprüngliche NETPORT Webinterface lässt sich immer durch direkte Angabe einer Seiten-URL aufrufen (z.B. **http://192.168.167.12/nbHome.html**)
Misc	
Beep after booting	Tonsignal nach dem Hochbooten.
Ethernet / USB	
Use DHCP configuration	IP-Adresse von DHCP-Server beziehen.
IP Address	Ethernet IP-Adresse (default: **192.168.1.11**). Nach Ändern der Adresse muss NETPORT neu gebootet werden.
Netmask	Netzwerk-Maske.
Gateway	Gateway-Adresse (default: **192.168.1.2**)
USB IP Address	USB IP-Adresse (default: **192.168.167.12**). Nach Ändern der Adresse muss NETPORT neu gebootet werden.
I/O	
Names	Zuweisung von Namen zu den IO-Pins.
Initial Direction	Einstellung des Signaltyps (Analog, Digital, PWM) und der Signalrichtung für die IO-Pins.
I/O CFG	Zuweisung von Namen und Anfangswerten zu den IO-Zuständen.
A/D CFG	Zuweisen von Spannungsgradient und Offset zu den A/D Eingängen.
PWM CFG	Zuweisen von Schaltzeiten zu den PWM-Ausgängen.
Apply Settings	Führt die gewählten Einstellungen aus.
Store Custom Website	Speichert die eigene Website nach dem Hochladen im Flash.

Download Configuration	Speichert die Konfiguration des Moduls in einer Datei. Diese Datei mit der Endung "**.netbox**" enthält alle Einstellungen wie IP-Adresse, I/O Konfiguration und Skript. Zusätzlich wird die Seriennummer des Moduls gespeichert.
Set Defaults	Setzt alle Einstellungen auf die Defaultwerte zurück.
Reboot	Startet NETPORT neu.

Restore

Hier kann eine gespeicherte Konfiguration oder Teile davon geladen werden.

Restore Script	Ersetzt das aktuelle Skript durch das in der Konfigurationsdatei (**.netbox**) gespeicherte Skript.
Restore Settings	Lädt IP Adresse, I/O Konfiguration und alle weiteren Einstellungen.
Restore When Serial Number Matches	Prüft zunächst, ob die Seriennummer des aktuellen Gerätes mit der Seriennummer der Konfigurationsdatei übereinstimmt. Die Konfiguration wird nur in diesem Fall geladen. Wenn die Seriennummern nicht übereinstimmen, wird eine Fehlermeldung ausgegeben und nichts geändert.
Reboot After Restoring	Startet das Modul nach Laden der Konfiguration automatisch neu.

Custom

Hier wird ein eigenes, angepasstes Webinterface angezeigt, das per FTP an **ftp://netport_ip/custom-html** hochgeladen werden kann (siehe getrennte Dokumentation für Details). Solange nichts hochgeladen wurde, wird eine Beispielseite dargestellt, die den Zustand der Eingänge anzeigt und das Setzen der Ausgänge erlaubt. Diese Seite kann als Vorlage für eigene Webinterfaces dienen.

NETPORT Fernsteuerung

Jedes NETPORT Modul kann mit C-Skriptdateien oder Software-Programmen per TCP-Socket ferngesteuert werden. Alle I/O Funktionen lassen sich durch Skriptbefehle auslösen. Diese Skriptbefehle werden von einem Skript-Interpreter auf dem Modul ausgeführt. Rückgabewerte werden über den Socket-Kanal zurückgesendet. Die Skript-Syntax entspricht ANSI C mit einigen Unterschieden. Pointer werden nicht unterstützt, dafür jedoch einige C++ Elemente wie z.B. Exceptions. Details zur Skriptsprache finden Sie in den folgenden Kapiteln.

Um Skriptbefehle an das Modul zu senden, gibt es außer dem **Webinterface** eine Reihe weiterer Möglichkeiten:

Fernsteuerung per TCP Socket

Das Modul bedient zur Fernsteuerung den TCP Port **1233**. Ein oder mehrere Socket Clients können sich zu diesem Port verbinden, ein Skript senden und somit die Box fernsteuern. Dabei werden die gleichen Skripte verwendet wie per **Webinterface** - mit zwei Ausnahmen:

#include <netbox.h> ist nicht notwendig und kann weggelassen werden.
Das Skript sollte keine Endlosschleife enthalten, damit der Befehl endet und eine Rückgabemeldung gesendt werden kann.

Alle Skripte werden quasiparallel ausgeführt. Wenn sich mehrere Clients verbinden, können also auch mehrere Befehle (z.B. Berechnungen) parallel zueinander ausgeführt werden. Jedes Skript hat Zugriff auf alle Resourcen wie I/O Leitungen.

Die Fernsteuerung über Remote Skripte ist nur aktiv, wenn unter ***Settings*** [***Allow remote script***] aktiviert ist.

Fernsteuerung per GalepX

Das Modul lässt sich per MAD Bridge z.B. mit Hilfe der **GalepX** Programmiergeräte-Software fernsteuern. Ein Beispiel, wie eine solche Fernsteuerung aussehen kann, findet sich in der GalepX Distribution unter **skripts/pio84oco/pio84oco.gxs**. Dieses Skript öffnet auf der GalepX-Oberfläche ein Fenster, welches die Eingangszustände des PIO-Moduls darstellt und eine Steuerung der Ausgänge über Checkboxen erlaubt.

Fernsteuerung per Gamestudio

Das Software-Baukastensystem **Gamestudio** (**www.3dgamestudio.de**) kann NETPORT-Module mit den lite-C Befehlen **socket_connect**, **socket_send** und **socket_receive** direkt steuern. Details finden Sie im Gamestudio-Handbuch in der Dokumentation der lite-C Befehle.

Fernsteuerung per Batch-Datei

Das Gamestudio-Programm **remote.exe** sendet eine Skriptdatei direkt an das NETPORT Modul und kann per Batch-File aufgerufen werden. Z.B. die Batch-Befehlszeile

remote -ip192.168.1.12 test.gxs test.log

liest die Skriptdatei **test.gxs**, sendet sie zum NETPORT auf der IP-Adresse **192.168.1.12** und speichert die Rückgabemeldungen in der Datei **test.log**. Wenn der Name der Logdatei weggelassen wird, wird die Rückgabemeldung im Fenster angezeigt. Die Befehlszeilenoption **-nwnd** unterdrückt die Ausgabe im Fenster.

Das **remote.exe** Programm dient normalerweise der Batch-Steuerung von GALEP-Programmiergeräten, es lassen sich jedoch beliebige Socket-Kanäle damit steuern. Es befindet sich zusammen mit dem einfachen Testskript **test.gxs**, das den NETPORT-Lautsprecher piepsen lässt, im **remote** Ordner der Distribution. Den Sourcecode des Programms finden Sie dort ebenfalls.

Fernsteuerung per DLL

Die **remote.dll** Library ist ein einfacher Weg, um eine NETPORT Fernsteuerung in eigene Software zu implementieren. Die API enthält vier Funktionen:

int RemoteConnect(char* IP,long iPort)
Öffnet eine Verbindung zum TCP Socket auf der angebenen IP-Adresse und Port. Liefert **0** zurück, wenn kein Socket Host gefunden wurde.

void RemoteClose()
Schließt die Verbindung zum Socket; muss vor Programmende aufgerufen werden.

int RemoteSend(void *data,long size)
Sendet den Inhalt des ***data** Puffers zum Socket, mit Pufferlänge **size** in Bytes.

int RemoteReceive(void *data,long size)
Prüft, ob ein Datenpaket vom Socket empfangen wurde. In diesem Fall wird der ***data** Puffer mit dem Inhalt gefüllt und die Zahl der Bytes zurückgeliefert. Andernfalls wird **0** zurückgeliefert.

Die **remote.dll** library befindet sich im **remote\API** Ordner, zusammen mit einem einfachen Testprogramm **RemoteTest.cpp**, das die Benutzung der Library demonstriert.

Fernsteuerung per Linux

Unter Linux können Befehle direkt zum Modul per Shell-Skript gesendet werden. Der Shell-Befehl **connect 192.168.1.12 port 1233** öffnet den Socket auf der angegebenen IP-Adresse; die folgenden Befehle werden dann direkt zum Socket gesendet.

Fernsteuer-Praxis

Sobald eine Verbindung zum Modul hergestellt wurde, kann es auf zwei verschiedene Arten gesteuert werden:

Befehlsorientiert - Hierbei wird nur ein Befehl (oder eine kurze Sequenz) an das Modul gesendet. Nach

Abarbeitung der Sequenz wird die Verbindung durch das Modul beendet.
Interaktiv - Hierzu wird als erstes der Befehl **keep** gesendet. Danach können beliebig viele Befehle oder Sequenzen gesendet werden. Die Verbindung wird durch den Client beendet.

Die folgenden Beispiele verdeutlichen das Vorgehen in einer Linux-Shell und der Modul-Adresse **192.168.1.12**:

Befehlsorientiert

socket 192.168.1.12 port 1233	(öffne Verbindung - nur Linux)
print("Hallo World");\n	(sende Skriptbefehl)
	(warte auf Antwort)
#10 Hello World \n	(empfange Antwort)
#00 \n	
	(Modul beendet Verbindung)

Interaktiv

socket 192.168.1.12 port 1233	(öffne Verbindung - nur Linux)
keep \n	(halte Verbindung)
print("Hallo World");\n	(sende Skriptbefehl)
	(warte auf Antwort)
#10 Hello World \n	(empfange Antwort)
#00 \n	
...	(sende mehr Befehle)
	(Client beendet Verbindung)

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:

```c
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**

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();
  }
}
```