



DLI EPCR6 User's Guide

1.7.23.0

Contents

1	Overview	2
2	Product features	3
3	Package contents	4
4	Basic setup and use	5
5	LCD and keypad	10
6	Emergency stop switch	14
7	Setup page	15
8	Scripting	23
9	Event notification	44
10	Customization page	52
11	External APIs	53
12	Backing up settings	61
13	Firmware upgrade	62
14	Date/time	64
15	AutoPing	65
16	Energy monitor	69
17	Safety shutdown	76
18	Generic input/output	77
19	System log	83
20	Locking down the controller	84
21	Resetting settings to defaults	86
22	Specifications	87
23	Open source code	88
24	Technical support	89
25	Limited five year warranty	90

1 Overview

Congratulations on selecting the DLI Ethernet Power Controller, a smart rack-mount AC power switch.

Check out the main product [features](#). Some more technical [specifications](#) are available as well.

Be sure to check the [package contents](#) when unpacking the unit, then follow the instructions for [basic setup and use](#) if you are a first-time user. The device is controllable using LCD and keypad, which have several [modes](#). This should allow you to access the basic device features. More advanced settings are located on the [Setup page](#).

Ethernet Power Controller 6 can be:

- extended with [user scripts](#);
- [customized](#) to add branding;
- accessed via a growing number of [external APIs](#);
- [upgraded](#) to newer firmware versions.

It can be configured to:

- [ping](#) other devices and take action if they don't respond;
- [shut down](#) outlets to prevent equipment damage;
- [perform input/output](#) via external channels;
- monitor or flexibly drive [input/output pins](#);
- send [notifications](#) when certain events occur;

Ethernet Power Controller 6 utility functions include:

- [monitoring](#) and logging data from miscellaneous sensors;
- reading the [system log](#)
- [setting date/time](#);
- [backing up](#) settings;
- [locking down](#) security-sensitive functionality;
- [resetting](#) settings to defaults.

The product features an [Emergency stop switch](#) which allows to manually shut down switched outlets physically.

The firmware is based on [open-source code](#) which is provided to give you the option to build totally custom firmware.

Please contact [technical support](#) in case of any problems.

We offer a [limited five-year warranty](#) on these units.

2 Product features

Congratulations on selecting the DLI Ethernet Power Controller, a smart rack-mount AC power switch. Its features include:

- **8 Switched Duplex Outlets + 2 Unswitched Outlets**

Eight individual switch control circuits are provided with duplex outlets. Outlets are spaced for plugs and adapters. Unswitched outlets are provided for "always on" devices. Switched outlets are metered in two banks. Unswitched outlets are unmetered.
- **Simple Web Interface**

The internal web server is accessible from any browser. Simply enter an IP. Configuration and control are web-based.
- **AutoPing™ Reboot**

AutoPing continuously monitors an IP address. If a server, router, or other peripheral goes down, AutoPing can automatically reboot it without user intervention. Several devices can be monitored simultaneously.
- **Programmable LCD Display**

A 2x16 LCD displays status for each outlet.
Custom messages can be displayed via user scripts.
- **Multi-User Password Security**

Multi-user authentication limits access to the power controller. The administrator selects which outlets each user can control.
- **Sequenced "On Timer"**

A programmable delay timer allows outlets to be switched on in sequence, rather than simultaneously. Most devices draw a surge of power when initially switched on. Using this timer, more equipment can share a single circuit without overloads. Programmable scripts can be used to create customized power-up and shut-down sequences with variable timing.
- **MOV Surge Suppression**

Dual 3600W metal oxide varistors clamp power surges and spikes, protecting attached devices.
- **Scripting Language, Syslog, and Utilities**

Lua scripting can be used to create custom control and reboot sequences, schedule periodic reboots, etc. Internal and external event logs are provided.
- **New Features**

New features include improved power metering, a light sensor, CPU temperature sensor, longer battery life, HTTPS, WiFi support, web power meters/charts and Lua scripting.
- **Field Upgradeable Firmware**

Firmware is field upgradeable via Ethernet or WiFi.

3 Package contents

- Ethernet Power Controller 5 with NEMA 5-15 plugs and reversible rack ears attached.
- RP-SMA WiFi Antenna.
- Internal web server backup battery (shipped disconnected).

Please contact the freight carrier immediately if your package appears opened or damaged in transit. Call DLI at (408) 330-5599 for tech support, service, and hardware upgrades.

4 Basic setup and use

4.1 Factory defaults

The factory default network configuration is as follows:

- wired network: fixed IP address 192.168.0.100, netmask 255.255.255.0;
- wireless network: fixed IP address 192.168.254.1, netmask 255.255.255.0.

You can log in with username `admin` (lower case) and password `1234`. It is recommended that you change the password. You will be reminded to do so by a big red banner on the top of each page.

To reset to factory defaults, gently press the reset-to-defaults button below the LCD to enter the reset menu, then select a reset option.

4.2 Initial setup

Use these shortcuts if you are an experienced installer. We recommend reading the entire manual for first-time installation.

- Unpack. Save the carton.
- Attach line cords to 15A circuits.
- Attach an Ethernet cable from the controller to your LAN. Switch power on. If you are attached through a switch, you may need to cycle switch power to establish a connection.
- Ping the default address 192.168.0.100 to confirm that a network connection is established. If you don't receive a response, proceed to the [IP setup](#) section below.
- Log in to the power controller using the default user name `admin` and the password `1234`. Note: "admin" must be entered in lower case.
- Click the Settings link to reach the configuration page. Select the safest power-loss configuration for your installation: all OFF, all sequential ON or pre-powerloss state.
- Configure the power switch as described below. After each change, click Submit and wait for the page to refresh before continuing.

Tip: A three-bulb electrical safety-tester is handy for configuring the controller before attaching your equipment.

4.3 IP setup

If your network settings won't access the default IP, use a direct cable connection (temporarily bypass any switch or router) and follow these steps to add a compatible static IP, such as 192.168.0.50.

4.3.1 Windows IP setup

Before adding an IP, close all programs and browsers. After the link is established, you can enable DHCP.

4.3.1.1 Locating IP settings

In Windows, the first step is locating the network adapter TCP/IP properties. The procedure differs for each Windows version:

Windows XP, 2000, 2003:

- Open Start / Control Panel / Network Connections.
- In "classic view", select Start / Settings / Control Panel / Network Connections.
- Right-click on Local Area Network Connection and select Properties.
- Proceed to step 2.

Windows Vista:

- Open Start, right click on Network, then on Properties.
- Double click Network and Sharing Center.
- Click Manage Network Connections. A Network Connections window appears.
- Right click on the network connection to the switch, i.e. Local Area Network.
- Proceed to step 2. Windows 7:
- Open the Start orb, click on Control Panel.
- Click View Network Status and Tasks, then Change Adapter Settings.
- Proceed to step 2

Windows 8:

- Mouse or swipe to the bottom right corner and select Settings.
- Select Control Panel.
- Select Network and Sharing Center.
- Change Adapter Settings.
- Right click on your connected network and select Properties.
- Proceed to step 2

4.3.1.2 Configuring static IP

The second step is adding an IP such as 192.168.0.50. Temporarily disable DHCP while configuring the switch.

- Select Internet Protocol TCP/IP V4 Properties and click Properties.
- Enter a compatible static IP such as 192.168.0.50.
- Click Apply and close windows.
- Ping the power switch to confirm the connection.
- Point your browser to 192.168.0.100
- Log in.

Detailed instructions are at http://digital-loggers.com/ip_setup.html

4.3.2 Mac OS X IP setup

- Turn AirPort off temporarily.
- Click the Apple logo, then System Preferences, then Network.
- Select Built-In Ethernet and then Configure.
- Under the TCP/IP tab, select Manually
- Enter an IP address such as 192.168.0.1 as shown:
- Make changes shown.
- Point a browser to 192.168.0.100 Log in.

Find Mac setup details at http://digital-loggers.com/mac_ip_setup.html

4.4 Windows IP configuration (2000, 2003, XP, Vista)

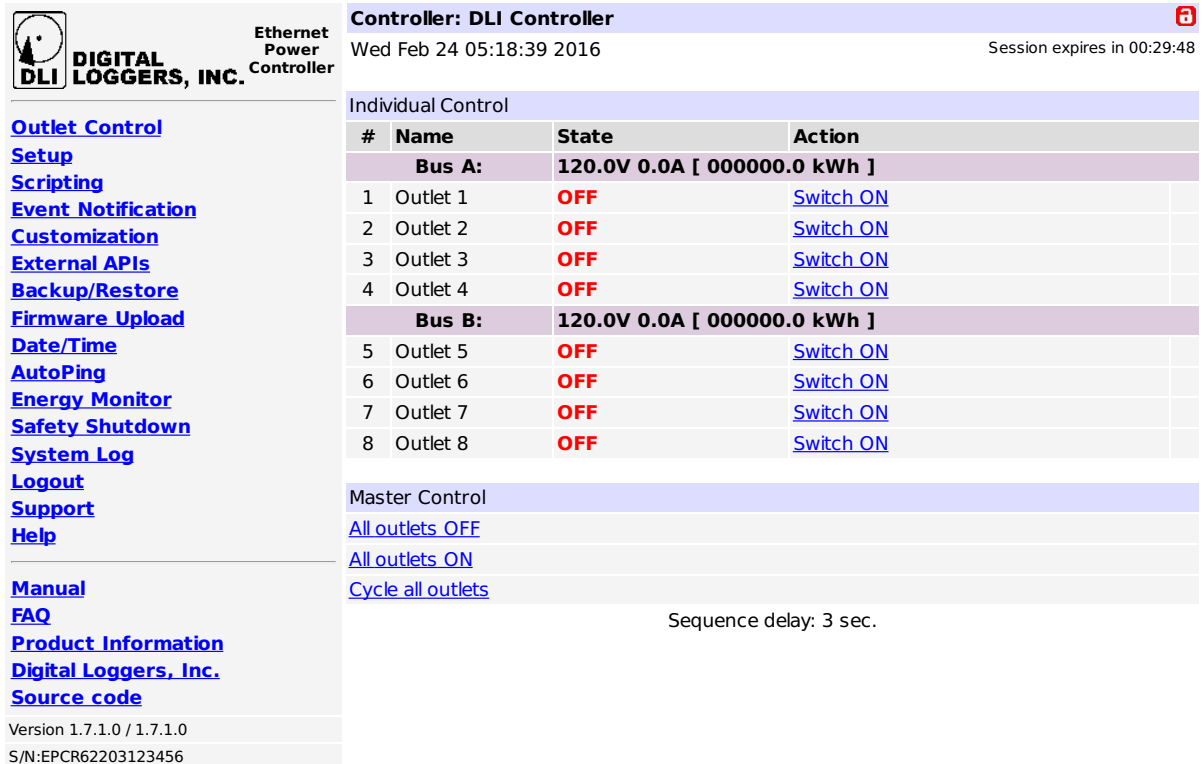
If your default Windows settings won't access the controller, use a crossover cable and follow these steps to reach the controller's IP:

1. Close network programs and browsers
2. Go to Network Settings -> Local Area Network.
3. Use the keyboard shortcut — type "ncpa.cpl" and click OK.
4. Right click on your LAN connection and choose "Properties".
5. Highlight "Internet Protocol" and click the "Properties" button.
6. Click the "Advanced" button.
7. Under the IP Address settings, click the "Add" button.
8. Enter a new IP, such as 192.168.0.10, and a subnet mask of 255.255.255.0.
9. Press the "Add" button; this new IP is added the list.
10. Close all windows for the configuration to take effect.
11. Start your Browser and type 192.168.0.100 in the URL field.

The default user name and password are "admin" (lower case) and "1234".

4.5 Basic switch operation

After power-up, the controller performs a sequence of self-tests to ensure reliability. The controller may then be operated via a web browser. To access the controller, simply enter the IP address in the URL field of your web browser, then log in. You will be presented with a screen similar to this:



Controller: DLI Controller 

Wed Feb 24 05:18:39 2016 Session expires in 00:29:48

Individual Control

#	Name	State	Action
Bus A: 120.0V 0.0A [000000.0 kWh]			
1	Outlet 1	OFF	Switch ON
2	Outlet 2	OFF	Switch ON
3	Outlet 3	OFF	Switch ON
4	Outlet 4	OFF	Switch ON
Bus B: 120.0V 0.0A [000000.0 kWh]			
5	Outlet 5	OFF	Switch ON
6	Outlet 6	OFF	Switch ON
7	Outlet 7	OFF	Switch ON
8	Outlet 8	OFF	Switch ON

Master Control

[All outlets OFF](#)

[All outlets ON](#)

[Cycle all outlets](#)

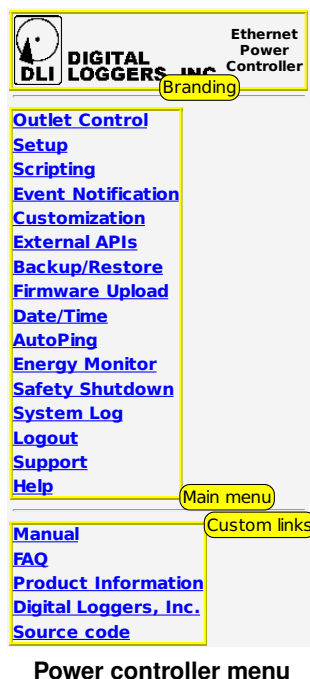
Sequence delay: 3 sec.

Main power controller page

Note that in this and many other pages, logical blocks can be collapsed by clicking  in their right-hand corner, and later expanded by clicking .

4.6 Common page layout

You can navigate the menu links to access the controller's features:



Power controller menu

Custom links can be configured on the [Admin page](#). The branding block can be customized on the [Customization page](#).

4.7 Switching outlets on and off

The outlet control page lets you control outlets. The sequence in which outlets will be switched on is determined by settings on the [Admin page](#). To switch an outlet on or off, simply click to the right of the outlet name or number. Switching an outlet off is immediate. Switching an outlet on may be delayed if a different (or possibly even the same) outlet was recently turned on. The delay acts to protect the device from simultaneous inrush currents and limit cycling rate. You may also "cycle" a device which is connected to the controller. This feature is useful for rebooting Ethernet devices which may interrupt the web link to the controller. Clicking "Cycle" switches power off, waits a few seconds, and then switches power back on. This resets the attached device. You may also "cycle" all outlets using the "Cycle all outlets" button on the bottom of the page. Depending on your web browser settings, you may need to click the "refresh" button to update the on-screen status display after changing settings. A screen refresh setting is provided on the [Setup page](#).

4.8 Logout

Browser logout is automatic when a session is closed or after a time-out period. You can use a menu link to log-out in advance.

5 LCD and keypad

The LCD has 2 lines, 16 character positions each. The displayed data depends on the mode, and possibly also on [user scripting](#).

The keypad has 5 keys:

-  (UP),
-  (DOWN),
-  (ON),
-  (CYCLE),
-  (OFF).

During normal operation, the EPCR6 LCD and keypad interface can be in one of the following modes:

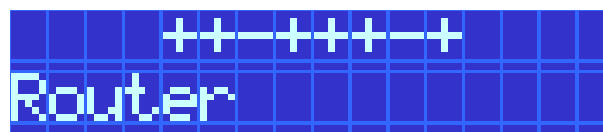
- Outlet mode, which displays status of and allows to manipulate the unit's outlets;
- Meter mode, which allows to read measurements of the EPCR6 meters (bus voltages and currents, temperature and illuminance) and calibrate them;
- Network mode, which displays and allows to configure network settings.

You can cycle through the modes by pressing  and  keys simultaneously.

If the keypad is locked via web UI, the keypad doesn't work, and if you press a key, a message about this is displayed and stays in place until the next update of the LCD data; no action is taken.

5.1 Outlet mode

At boot, EPCR6 starts in Outlet mode.



Outlet mode

In Outlet mode, the  and  keys allow to choose the outlet to display. The first line shows states of all outlets.

The selected outlet is marked by a blinking cursor. Its name is displayed on the second line.

If the outlet is not locked (see below) and its physical state matches the expected state, it is displayed as:

- a plus sign  for outlets that are on, or
- a minus sign  for outlets that are off.

If the outlet's physical state doesn't match the expected state, (e.g. it will be switched on in sequence , or the bus power it is not powered but the outlet is expected to be on), its state is marked by:

- a minus/plus sign  for outlets that are physically off, but should be on, or
- a plus/minus sign  for outlets that are physically on, but should be off (this should be rare).

The  button switches the selected outlet on, likewise the  button switches it off.

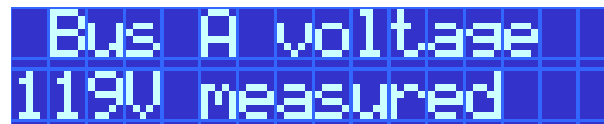
You can press and hold each of the buttons for 3 or more seconds to lock the outlet in the corresponding state. Locked outlets can't be manipulated from web UI or with scripting, and won't be switched by using the hardware  /  keys unless you hold the corresponding key for 3 or more seconds to unlock it. Locked outlets are displayed as:

- a zero sign  for outlets that are locked off, or
- an asterisk  for outlets that are locked on.

The  button cycles the selected outlet unless it's locked.

Managing outlet lock state can only be done using the LCD and keypad (unless you enable SSH). Locked outlets' states cannot be altered, and they are not affected by [power loss recovery mode](#). However, they are still affected by over-current, over-voltage, and low-voltage [safety shutdown](#) configuration.

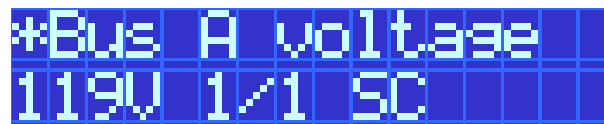
5.2 Meter mode



Meter mode

In Meter mode, the  and  keys allow to choose the selected meter. The name of the selected meter is displayed together with its current measured value.

Meters can be calibrated. Pressing the  button starts the calibration process for the meter. The leading character of the line, previously a whitespace , becomes an asterisk  to indicate that.



Meter calibration submenu

The calibration progress and method are shown on the bottom line. For example, for voltages and currents, single-point scale calibration is used, so "1/1 SC" (first and only point, scale) is displayed.

You can use the ▲ and ▼ buttons to adjust the value displayed to match the reading of an external measurement device (in his case, most likely a digital multimeter). Pressing ↺ will revert the value displayed to the original measurement.

If this is the last point to calibrate, pressing 1 will confirm and store the calibration values and leave calibration mode; the display will be updated in ~2 seconds and you will see a corrected reading. If there are more points to calibrate, pressing 1 saves the current value and lets you input another calibration point.

You can cancel calibration at any time by pressing 0.

You can reset calibration to default by pressing ↺ when not in calibration submenu. You will be asked to confirm this by pressing 1, or decline by pressing 0.

Bus voltage and current meter calibration can also be done using the [web UI](#).

5.3 Network mode



Network mode

In Network mode, the ▲ and ▼ keys allow to select the network parameter. The selected parameter name is displayed together with its current value.



Network configuration submenu

The following parameters are present:

Parameter	Type	Default/sample value
HTTP port	TCP port*	80
HTTPS port	TCP port*	443
SSH port	TCP port	22
SSH enabled	Yes/No	No
Syslog server	IP address	0.0.0.0 (none)
LAN protocol	Static/DHCP	Static
LAN IP	IP address	192.168.0.1
LAN netmask	Netmask	255.255.255.0
LAN gateway	IP address	0.0.0.0 (none)
LAN DNS server	IP address*	0.0.0.0 (none)
LAN MAC	MAC	7C:E1:FF:.....
WiFi enabled	Yes/No	Yes
WiFi SSID	String	DLI....

Parameter	Type	Default/sample value
WiFi mode	Access Point/Station	Access point
WiFi encryption	None/WPA/WPA2	WPA
WiFi key	Random string	...
WiFi channel	Channel number or auto	11
WiFi protocol	Static/DHCP	Static
WiFi IP	IP address	192.168.254.1
WiFi netmask	Netmask	255.255.255.0
WiFi gateway	IP address	0.0.0.0 (none)
WiFi DNS server	IP address*	0.0.0.0 (none)
WiFi MAC	MAC	7C:E1:FF:.....

Most values can be changed. Pressing the  button allows to change the parameter. The leading character of the first line, previously a whitespace , becomes an asterisk  to indicate that.

Different kinds of values are changed in different ways:

- for selectable options, like static/DHCP IP address mode, WiFi mode and encryption, the  /  keys cycle through the available options;
- for netmasks, the  /  keys decrease or increase the number of set bits in the mask;
- for IP and MAC addresses, a virtual cursor is used to allow modification of individual address bytes; the cursor is initially placed on the last byte, and can be advanced to the preceding byte by pressing ; the  /  keys increase or decrease the current byte's value;
- for the WiFi pre-shared key, any of the ,  or  buttons generates a new key;
- the WiFi SSID can't be changed from keypad.

Options marked with a star (*) correspond to configuration items which can have multiple values (including none) which can be set from the web UI. You can only set a single value from the keypad for these, and . . . is displayed on the LCD if the option doesn't have a single value. For IP addresses, 0.0.0.0 means 'none' (e.g. no default gateway, no syslog server, etc.).

Pressing  saves the current value and starts network reconfiguration if needed. The display isn't updated instantly, and you need to refresh it by pressing  / . You can cancel editing at any time by pressing .

6 Emergency stop switch

To enable the E-Stop switch, remove the yellow ring under the red button by pulling down on the label. To activate E-Stop Shutdown, press the red button. The event will be logged as:

```
emergency shutoff activated, all outlets off
```

Rotate the button clockwise to release and restore power. The event will be logged as:

```
emergency shutoff deactivated, normal outlet operation
```

WARNING! E-STOP DOES NOT SHUT DOWN THE UNSWITCHED (ALWAYS ON) OUTLETS!

The events will not be logged if the EPCR6 is running on battery.

7 Setup page

The setup page allows the administrator to configure the power controller. These settings are supported:

7.1 Controller and outlet names

Unit Names		Confirm
Controller name	DLI Controller	
Outlet 1 name	Outlet 1	☐
Outlet 2 name	Outlet 2	☐
Outlet 3 name	Outlet 3	☐
Outlet 4 name	Outlet 4	☐
Outlet 5 name	Outlet 5	☐
Outlet 6 name	Outlet 6	☐
Outlet 7 name	Outlet 7	☐
Outlet 8 name	Outlet 8	☐

Controller and outlet names

Use the controller name fields to assign a Controller Name to the power controller itself. Examples are "Server Rack Power Strip" or "Plutonium Refinery Control". The Controller Name field appears on the top of the home page. Assign a separate name to each outlet, such as "Missile Launcher" or "Email Server" to make identification of each circuit simple.

You can use characters from the full Unicode character set; they'll be transliterated for display on the LCD if necessary.

7.2 Delays

Delay	
Wrong password lockout	60 minutes. (0-60)
ON sequence delay	3 seconds. (1-255)
Cycle delay	0 seconds. (1-255)
Brown-out re-latch delay	10 seconds. (1-255)
Refresh screen every	1 minutes. (1-255)
Enable screen refresh	☒

Delays

When a time value is entered in the "All ON sequence delay" field, the power controller will pause for a period of time before switching each outlet on in sequence. This delay helps prevent the power surges and blown circuit breakers which can occur when multiple devices are switched on simultaneously. A delay of 60 seconds is suggested for server applications. You may also enter a screen refresh delay in this section. If "Enable screen refresh" is checked, and a delay value is entered, your browser should periodically update the status screen.

7.3 Power loss recovery modes

Power Loss Recovery Mode
⬆

When recovering after power loss

☒ Turn all Outlets off
☐ Turn all Outlets on
☐ Return to pre-powerloss state

Power loss recovery modes

The power loss recovery mode setting has three settings which take effect after a power failure:

1. You can turn all outlets off (all systems will be switched off until manually turned on later) by checking the first box.
2. You can automatically turn all outlets on using the "All ON sequence delay" described above. Check the second option to do this.
3. You can return to the same outlet settings that were used prior to the power loss. The "All ON sequence delay" will also be used in this instance. Click the third option to return to pre-powerless state.

7.4 User-defined links

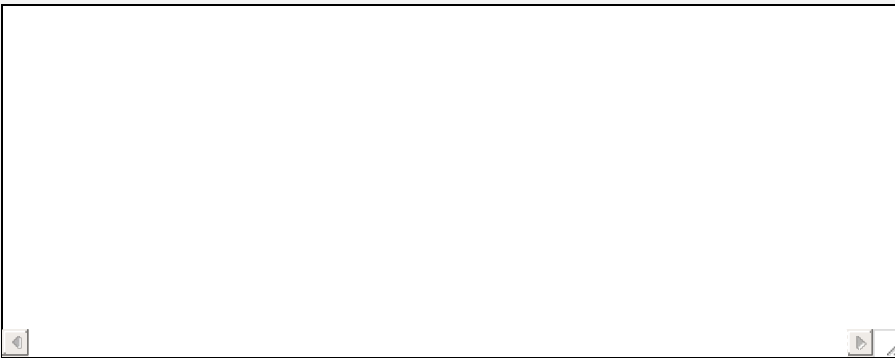
User Defined Links
⬆

#	URL	Description
1	http://www.digital-loggers.co	Digital Loggers, Inc.
2	/src.tar.gz	Source code

User-defined links

You may link to other power controllers, your own web pages, or remote web sites by entering up to four URLs and descriptions in the Setup page. For example, enter "Site Two Power Controller" in the description field with a URL of "http://192.168.0.250/". These links appear on every page of the main web UI.

7.5 Network settings

General Network Settings	
Hostname	power
Location	Rack #3
Contact	Joe Random <joe@random.example>
HTTP Port	80
HTTPS Port	443
Enable SSH Server	☒
SSH Port	22
Syslog Server	192.168.0.2
Syslog severity threshold	Debug
Same subnet access only	☒ FROM 192.168.0.0-192.168.0.255 (wired), 192.168.254.0-192.168.254.255 (wireless) ONLY
Allowed SSH public keys	
Submit	

General network settings

You can adjust the HTTP and HTTPS port bindings. If left empty, the corresponding service is not accessible. It may be a good idea to disable HTTP if HTTPS satisfies your needs. Disabling both for security is possible; you can use the [LCD and keypad](#), or SSH to re-enable them if needed.

Enabling SSH will allow full control over the device, possibly bypassing most of the restrictions, e.g. setting protection. The SSH port is customizable as well. The SSH server also accepts public key authentication for a configurable set of keys (the format is the same as in the `authorized_keys` file).

You can limit the severity of the locally recorded log messages by setting a minimal severity. Note that the local log is circular, with old messages being replaced by newer ones; messages aren't persisted across reboots. For persistent storage, you can configure the unit to send the system log to a syslog server. All messages, regardless of severity, are sent; the receiver is expected to do the filtering.

Same subnet restriction can be used to prevent remote access from outside. ONLY MACHINES IN THE SAME SUBNET WILL CONNECT AFTER ENABLING THIS. If connectivity is lost, use a local connection such as a laptop with a crossover cable to restore your original network settings; you can also use the [LCD and keypad](#) for that.

7.5.1 Wired network settings

LAN Configuration	
MAC Address	7C:E1:FF:00:00:00
Protocol	Dynamic IP/DHCP ▾
IP address	192.168.0.100
Subnet mask	255.255.255.0
Gateway	192.168.0.1
DNS servers, comma-separated	192.168.0.1
Submit	

Wired network settings

The device MAC address is provided for reference only and cannot be changed in this form. If you need to change the MAC address, you may do so via LCD+keypad, SSH or using the REST-like API. Be sure you know what you're doing, as e.g. assigning a conflicting MAC address, or a broadcast MAC address will make an interface unusable.

To configure the unit to use static IP assignment, a fixed IP address and network mask must be entered. If a default gateway is specified it must be on the same subnet as the IP address specified. A number of DNS server IP addresses can be supplied separated by commas, e.g. 192.168.0.1, 8.8.8.8. If DNS servers are available, some other configuration variables can accept hostnames instead of IP addresses.

If you wish to configure the unit to use DHCP IP assignment, you needn't change the IP, network mask, default gateway and DNS servers; rather, after the unit obtains a DHCP lease, the parameters will be displayed for reference.

It is recommended to configure the DHCP server to provide a static lease for the EPCR6 using its MAC address (also displayed).

When changing IP addresses, you may need to restart the unit and your network switch to validate the new IP on an "auto-configuring" switch port.

7.5.2 Wireless network settings

WiFi Configuration	
WiFi module enabled	☒
MAC Address	7C:E1:FF:00:00:00
Protocol	Static IP
Channel	11 (2462 MHz)
Mode	Access Point
SSID	DLI_EPCR656
Encryption	WPA PSK
Encryption Key	••••••••
IP address	192.168.254.1
Subnet mask	255.255.255.0
Gateway	
DNS servers, comma-separated	
Submit	

Wireless network settings

The wireless network adapter has settings similar to those of the wired network adapter (see above), and adds WiFi-specific ones.

It's possible to disable the wireless module entirely by unchecking the "WiFi module enabled" checkbox. The wireless MAC address is configured to match the wired MAC address as the adapters will never be on the same subnet in a regular setup. Use LCD+keypad, SSH access or the REST-like API if you need to change that.

The WiFi module can operate either in Access Point ("server") or Station ("client") mode. Either way, the name of the wireless network to create/connect to must be specified as the SSID.

If the unit is configured to be an Access Point and have a static IP assignment, it starts a DHCP server on the wireless interface.

It is possible to use no encryption on the WiFi channel, or one of the WPA, WPA2 or WPA/WPA2 mixed mode with pre-shared key (the key has to be entered then). WEP encryption is considered insecure and is not supported. Other encryption modes are not supported.

7.5.3 Network settings protection

You may press the "protect" button to lock the network settings (this will also affect the external API settings). Once locked, the network settings cannot be changed except by pressing the physical reset button on the front of the unit.

7.6 Access control

The administrator's username and password can (and should) be changed from the default values. Note that you need to provide the current password for confirmation.

Administrator credentials	
Administrator login	admin
Old administrator password	
New administrator password	
Confirm new administrator password	
Submit	Protect

Administrator credentials

In addition to the administrator, any number of users with individual passwords and outlet permissions may be configured on the setup page. Only the administrator can edit user names and passwords (users can only inspect and switch outlets).

Access control:		Controlled Outlets								Apply
User Name	Password	1	2	3	4	5	6	7	8	
tom		☐	☒	☐	☒	☐	☐	☐	☐	Change
dick		☒	☐	☐	☐	☐	☐	☒	☐	Change
harry		☐	☐	☐	☒	☐	☐	☒	☒	Change
		☐	☐	☐	☐	☐	☐	☐	☐	Change

User access control

Checkboxes to the right of each user name outlet control access privileges. Users can only see and interact with the chosen outlets. For example, user 'harry' would see the following on login:

#	Name	State	OFF / ON	CYCLE
Bus A: 120.0V 0.0A [000000.0 kWh]				
1	Outlet 4	OFF	Switch ON	
Bus B: 120.0V 0.0A [000000.0 kWh]				
2	Outlet 7	OFF	Switch ON	
3	Outlet 8	OFF	Switch ON	
Logout		Help		3 sec.

User outlet control screen

Individual outlets can be manipulated as usual. The top links allow switching all accessible outlets on, off or cycling them.

Notice the outlet numbering difference for non-administrative users vs the previous generations of EPCR/LPC controllers. In EPCR6, non-administrative users see all outlets accessible to them numbered consecutively starting from 1; previously the indices used to match those seen by the administrator. Naive outlet state manipulation links like <http://192.168.0.100/outlet?3=ON> will work differently depending on the user you're logged in as; however, certain use cases may benefit from this change. Consider using the [REST-like API](#) if you need consistent outlet manipulation.

The next group is comprised of miscellaneous settings for access control.

Access settings	
Allow legacy plaintext login methods	☐
Hide user passwords	☐
Hide WiFi password	☐
Disable local keypad	☐
Show device name on login page	☐
Submit	

Access settings

If you need to access the controller with clients supporting Basic authentication, or by browsers without JavaScript, you may need to enable the "Allow legacy plaintext login methods" setting. Those methods transmit passwords over the network and are thus considered insecure. This includes Basic authentication over HTTPS, which is secure relative to Basic authentication over HTTP, but relies solely on TLS for security, which is considered risky by some experts. DLI Ethernet Sender should not require this setting.

The "Hide user passwords" and "Hide WiFi password" settings configure whether clients should be able to read back the relevant values; this may be a security issue if there are untrusted administrator users.

The "Disable local keypad" setting is designed for untrusted physical environments. When enabled, the LCD will briefly indicate that the [keypad](#) is disabled on each keypress and otherwise ignore it.

Note that this still leaves the [emergency stop switch](#) and the [reset button](#) available to an attacker.

The "Show device name on login page" setting can be used to control if unauthenticated users can see the device name (it used to be controllable by a space character preceding the device name in previous controller models, but is now an individual setting). This may be a convenience, but also a possible security issue.

7.7 Power monitor settings

Power monitor	
Power factor Bus A	100 %. (0-100)
Power factor Bus B	100 %. (0-100)
Submit	

Power monitor settings

You can configure the multipliers for bus A and B power values. Note that the values are in percents.

A "Power monitor calibration" link takes you to a page where you can calibrate the bus voltage and measurements.

Power monitor calibration			
Bus A		Bus B	
Voltage	Current	Voltage	Current
120 V Adjust	0 A Adjust	120 V Adjust	0 A Adjust

Power monitor calibration

For example, to calibrate bus A voltage, enter the its actual value (measured externally) into the appropriate input field and press "Adjust".

Meters can also be calibrated using the [keypad](#).

7.8 Miscellaneous settings

The following setting group controls aspects of data presentation of the unit.

Miscellaneous Settings	
Display LCD text in all CAPS	☐
Meter and plot default image format:	PNG
Submit	

Miscellaneous settings

You can force all the text displayed on the LCD to be in CAPS, which may be more legible. However, this won't affect the network settings, as they include the WiFi password which would be useless if capitalized.

You can specify the preferred image format for [plots and meters](#). PNG is the default, as it's supported by most browsers, but SVG can provide a much cleaner result on recent ones.

8 Scripting

On its own, a power switch isn't very smart. Programmers can easily add custom functionality by using the built-in Lua-based scripting language in power controllers.

8.1 Hardware requirements

Lua-based scripting is available in all EPCR6 controllers. Beeper, backlight, LCD, voltage and current monitoring functions are limited to products with appropriate hardware installed.

8.2 Configuration

The scripting server has the following configurable parameters:

Scripting settings	
Script step delay	1 seconds.
User message force display timeout	0 seconds.
User message timeout	seconds.
Start on reboot at	[Disabled]
Trace script	☐

User script configuration

- Script step delay — the time in seconds to wait after execution of a legacy API function (e.g. ON, OFF, see below). Modern API functions (see below) don't have internal delays unless documented; the `delay()` function should be used there.
- User message timeout — the time in seconds after which user messages (displayed e.g. with the DISPLAY command) disappear even if no keys are pressed on the LCD and no changes have been made to the outlet state (leave empty to have the messages displayed indefinitely).
- User message force display timeout — the time in seconds during which user messages are displayed even despite keys being pressed on the LCD or changes to the outlet state (leave empty to have the messages forcefully displayed indefinitely).
- Start on reboot at — the scripting function to start at cold boot.
- Trace script — enable diagnostic output about script progress to system log.

8.3 Entering scripts

First, for a quick overview of the script language visit the [sample scripts page](#) on the Digital Loggers web site. Log in as admin and use the Scripting link to access the programming page.

Scripting is based on the [Lua](#) programming language. A brief introduction is done below, but you may want to consult the general description, especially if you intend to write more complex scripts.

Script code is organized in functions. Configuration items which allow some scripting reaction to an event (reboot, autoping failure, etc.) will ask you for the name of the function to call (you will be offered a list of the functions defined in the script).

Script listing

```

1  --[[ Power controller user script code.
2
3  The scripting implementation has changed, and is no longer compatible
4  with the older BASIC implementation. The most important changes are:
5
6  - Now Lua-based.
7  - No more line numbers, blocks of code identified by functions.
8  - Most of ON, OFF, etc. are kept as legacy functions, which can be called like
9    e.g ON(2345), ON("2345") or ON "2345", your choice.
10
11 Execution is still based on threads. Now threads are more visible and
12 manageable. Try starting some and you'll see them appearing in the
13 list.
14
15 Scripting samples are now available as snippets (below), separate from
16 the main script code. You can copy/paste them from/to the main script.
17
18 Stock snippets have names starting with 'default.'; changing or
19 creating snippets with such names is not recommended as your changes
20 may be erased on an upgrade.
21
22 ]]
23

```

Save

User script source editor

You will need to define your functions to be able to use scripting. Simply putting calls to existing functions in the script will not work. Functions are defined like this:

```

function my_function()
    ... statements go here ...
end

```

All functions defined this way will be usable from the web UI and callable externally. If you want to define a function for internal use, not to be called from outside, prefix it with `local`:

```

local function my_internal_function()
    ... statements go here ...
end

```

Local functions declared this way must be placed before any function that uses it. If you want to move the definition elsewhere and don't need to call the function before the definition, you can use the following structure:

```

local my_internal_function

... functions that use my_internal_function in bodies ...

function my_internal_function()
    ... statements go here ...
end

```

This works as long as all calls to `my_internal_function` occur in other functions not immediately executed. `my_internal_function` is still local (not visible externally). This works because a function is just another type of value, and `function fn() ... end` is just like `fn=function() ... end`, so it works the same way as

```
local x
...
x=5
```

Here, `x` has the value `nil` until it's assigned the value 5. Similarly, `local function x() ... end` is just like `local x; x=function() ... end`.

Functions in Lua are called with their arguments parenthesized, e.g. `func(arg1, arg2)`. Functions with no arguments are called with empty parentheses, like `func()`. However, according to Lua syntax, a single string function argument doesn't need parentheses, thus allowing the BASIC-like commands to avoid them if there's only one argument.

You can also create functions with arbitrary names (including Unicode) like this:

```
_G["Turn on lamp"]=function()
    ... statements go here ...
end
```

All the above declarations are essentially assignments to (global and local) variables. Therefore, if multiple scripts functions have the same name, the one closest to the bottom will effectively mask preceding ones.

When you're prompted for a script function to choose, choices will appear in the order they're specified in the script source. If you wish to change that, you can define a global `ui_order` table with a list of quoted function names to appear at the top of the list, in the desired order (others will be at the bottom of the list in alphabetical order). E.g.

```
ui_order={"important_function", "frequently_useful_function"}
```

Most APIs listed below are only available when used inside functions, not in the global context. That is, attempts to write e.g.

```
ON(1)
```

without an enclosing function will fail (such actions cannot be taken at script load time).

8.4 Snippets

Snippets are script fragments not part of the main script, stored for later use or shipped with the device.

Script snippets

default.lua_table_demo Load Save Remove

```
1 function test table manipulations()
2     local a={"1","2","3"}
3     assert(a[3]=="3")
4     assert(a[4]==nil)
5     table.insert(a,"x")
6     assert(a[4]=="x")
7     table.insert(a,1,"y")
8     assert(a[1]=="y")
9     assert(a[2]=="1")
10    assert(a[5]=="x")
11    table.remove(a,2)
12    assert(a[1]=="y")
13    assert(a[2]=="2")
14    assert(a[4]=="x")
15    assert(a[5]==nil)
16    table.sort(a,function(u,v) return u>v end)
17    assert(table.concat(a)=="yx32")
18    LOG("Table tests passed")
19 end
20
```

User script snippet editor

The snippet name selector can be used to choose an existing snippet, or to enter a new name, e.g. to create a new snippet or save the current snippet under a different name. Use the "Load" button to load the snippet with the selected name in the snippet editor. Use the "Save" button to save the currently edited snippet under the selected name. The "Remove" button can be used to erase the selected snippet.

Snippet code can be copied/pasted from/to the main script code. It is not subject to syntax or other checks so it's OK to place incomplete code fragments there.

Snippet names starting with 'default.' are reserved for snippets shipped with the device. It is not recommended to change or create snippets with such names because your changes may be erased on an upgrade.

8.5 Threading

Multiple threads of execution can be running at the same time. Any number of threads may run concurrently.

Thread control
⬆

No threads currently running.

Start thread:

do_some_lua_stuff ▾
Start

User script thread list

Threads can be started from the web UI, via an HTTP request, by an [AutoPing](#) trigger, or from other threads using `thread.run`. They can be explicitly stopped using the web UI or by calling `thread.kill` or `thread.killall` from the script, or implicitly by calling `thread.limit`.

Every thread has an 'origin', which is usually a string identifying the function that started the thread. For instance, when you create a function like this:

```
function my_function()
  ... statements go here ...
end
```

and then start it with the web UI, its origin is the "my_function" string. Threads created by other threads inherit their parent's origin, which can be useful when stopping a group of threads.

8.6 API levels

The scripting engine presents two sets of functions that you can use to write scripts:

- Legacy functions — functions which are designed to resemble the BASIC commands of the previous generations of EPCR/LPC controllers;
- Modern API — functions and objects which are designed to be easier to use.

You can use and even freely mix them as you wish, but only the modern API will receive further development attention. Some features are exposed only via the modern API because they had no corresponding legacy commands.

8.6.1 Legacy functions

Legacy functions (written in CAPS) are executed in sequence with a "step delay" after them. The legacy functions are designed so as to resemble the BASIC commands of the previous generations of EPCR/LPC controllers while remaining compatible with the Lua language.

Arguments to the legacy functions can be written as e.g. `ON(12345678)`, `ON "12345678"` or `ON ("12345678")`.

The supported legacy functions are:

- `ON, OFF, CYCLE, RESTORE` — perform the action on a list of outlets by numbers (as a number or a string);
- `BEEP (ON)` or `BEEP (on)` or `BEEP (true)` — turn beeper on;
- `BEEP (OFF)` or `BEEP (off)` or `BEEP (false)` or `BEEP (0)` — turn beeper off;
- `BEEP (number>0)` — turn beeper on for the specified number of seconds, then off;
- `SLEEP (number[, "unit"])` — suspend execution for the given amount of time (units default to "seconds", but can be "seconds", "minutes", "hours" or "days"; abbreviations like "sec", "h", "d" are also accepted);
- `WAIT "cron time mask"` or `WAIT (minute_mask, hour_mask, day_mask, month_mask, weekday_mask)` — wait for the local time to match the condition (each separate mask element must be a number or "*", and a "cron time mask" must be a string of 5 such elements separated by whitespace);
- `LOG "String"` — write a message to the system log
- `DISPLAY "String"` — display a string on the LCD when it's in outlet mode. The following strings are expanded:
 - `%%` — literal "%";
 - `%a` — Bus A current;
 - `%A` — Bus A voltage;
 - `%b` — Bus B current;
 - `%B` — Bus B voltage;
 - `%o` — state of outlets, in the form "12456" (ON outlets are listed);
 - `%O` — state of outlets, in the form "+-+---";
 - `%n` — serial number;
 - `%f` — firmware version;
 - `%d` — system time/date;
 - `%M` — MAC address of the power controller;
 - `%i` — IP address of the power controller;
 - `%m` — IP network mask;
 - `%g` — IP gateway;
 - `\1` — move cursor to the beginning of line 1;
 - `\2` — move cursor to the beginning of line 2;
 - `\f` — clear screen;
 - `\v` — clear end of current line;
- `WOL "MAC address"` — attempt to wake device with specified MAC address up using Wake-on-LAN protocol (the device has to be in the same LAN segment);
- `TIME "server"` — synchronize time with server specified by IP address or hostname in quotes; you can use `TIME ()` without arguments to synchronize with "pool.ntp.org" if the DNS is configured correctly.

You still need to enclose the function contents in a `function name() ... end` as explained above.

8.6.2 Modern API

Modern API allows more object-oriented approach to scripting. You need to explicitly use the `delay()` function if you use the modern API and need a delay. Note that the outlet power-on sequence delay applies anyway.

Lua objects can have fields (data contained in the object) and methods (functions which affect the object's state). Object fields are accessed with a dot `.`, like `meter.reading`. However, different object implementations in Lua may use the colon `:` or the dot `.` to access the object's methods (`outlet:cycle()` or `outlet.cycle()`). In the modern API, all objects use the dot `.` to access their methods, to prevent confusion.

Modern API objects and functions are grouped into several categories for convenience.

8.6.2.1 Core Lua functions

To make scripting safer, only a limited subset of Lua features is supported by sandboxing. The following Lua standard library features are supported:

Globals: `_VERSION`, `assert`, `error`, `next`, `ipairs`, `pairs`, `pcall`, `xpcall`, `select`, `tonumber`, `tostring`, `type`, `unpack`.

string library: `string.byte`, `string.char`, `string.find`, `string.format`, `string.gmatch`, `string.gsub`, `string.len`, `string.lower`, `string.match`, `string.rep`, `string.reverse`, `string.sub`, `string.upper`.

table library: `table.insert`, `table.concat`, `table.maxn`, `table.remove`, `table.sort`.

math library: `math.abs`, `math.acos`, `math.asin`, `math.atan`, `math.atan2`, `math.ceil`, `math.cos`, `math.cosh`, `math.deg`, `math.exp`, `math.floor`, `math.fmod`, `math.frexp`, `math.huge`, `math.ldexp`, `math.log`, `math.log10`, `math.max`, `math.min`, `math.modf`, `math.pi`, `math.pow`, `math.rad`, `math.random`, `math.sin`, `math.sinh`, `math.sqrt`, `math.tan`, `math.tanh`.

os library: `os.clock`, `os.difftime`, `os.date`, `os.time`.

Additionally, `_G` points to the sandbox environment.

Unlike most APIs, core Lua functions are available in the global context.

8.6.2.2 Delay functions

The `delay` function accepts the number of seconds to wait as an argument (it is assumed to be the script step delay if not specified). Only a single script thread is active at any given time; switching to a different thread is not performed until you call `delay()` (or other function possibly introducing a delay). If a scripting thread doesn't call `delay()` or one of the legacy API functions every now and then, it can't be terminated by `thread.kill` and will eventually be shut down by the runtime.

The `wait_until` function can be used to wait for an arbitrary number of conditions on local time. It may have any number of arguments, each named a condition. It waits for one of the conditions to be satisfied, and returns its 1-based index. If several conditions would be satisfied simultaneously, the first match wins.

A condition is a table which may (but does not have to) contain any set of the following keys:

- `year`, which stands for the year;
- `month`, which stands for month (1 is January);
- `day`, which stands for day of month;

- `wday`, which stands for day of the week (1 is Sunday, 2 is Monday, 7 is Saturday);
- `yday`, which stands for day of the year (1 is January, 1st);
- `hour`, which stands for hours;
- `min`, which stands for minutes;
- `sec`, which stands for seconds;
- `isdst`, which stands for the daylight savings time flag (true or false).

The values corresponding to the keys are the actual restrictions, all of which must be met in order for the condition to be satisfied. A simple value (i.e. a number for anything but the DST flag, and true/false for the DST flag) matches only itself. A function (receiving the field's value as argument) matches anything for which it returns true.

Here are some examples of conditions that can be used:

```
{hour=0,min=0,sec=0} -- matches at midnight;
{hour=7} -- matches anywhere between 7:00:00 and 7:59:59;
{wday=function(d) return d==1 or d==7 end} -- matches weekends.
```

It is not advised to perform exact matches on seconds since delays of internal operations may be greater than 1 second. It is advised to introduce additional delays to avoid triggering the same match again. The following sample switches outlet 1 on at 8:00 and off at 17:00.

```
while true do
  local event=wait_until({hour=8,min=0},{hour=17,min=0})
  if event==1 then
    ON(1)
  else -- event==2
    OFF(1)
  end
  delay (120)
end
```

We match with minute precision here and wait for 2 minutes to avoid double matching.

8.6.2.3 Outlet management

The global variable `outlet` represents a Lua array of outlet objects.

- `outlet[N].on()`: switches the outlet on (affects transient state);
- `outlet[N].off()`: switches the outlet off (affects transient state);
- `outlet[N].cycle()`: cycles the outlet (affects transient state);
- `outlet[N].persistent_state`: boolean representing the persistent requested state of the outlet (writes will affect transient state as well);
- `outlet[N].transient_state`: boolean representing the transient requested state of the outlet;
- `outlet[N].physical_state`: read-only boolean representing the physical state of the outlet;
- `outlet[N].state`: boolean reflecting outlet physical state when read, modifying transient state when written;
- `outlet[N].locked`: read-only boolean indicating if the outlet is locked.

Global constants `on` and `off` are `true` and `false`, respectively, useful to make scripts more readable, like `outlet[1].state=on`;

8.6.2.4 Thread management

Several threads can be executed simultaneously in a pseudo-parallel fashion. The global `thread` table contains these methods:

- `thread.run` can be used to start new threads; it accepts the thread function as argument, and returns the identifier of the resulting thread; it can accept a second string parameter to act as a description of the thread running and a third string parameter to redefine the thread origin, which may be useful for `thread.killall` and `thread.limit`;
- `thread.kill` can be used to stop a thread; it accepts the identifier of the thread as an argument;
- `thread.killall` can be used to stop many threads; it accepts the origin of the threads to kill as an argument (without an argument, all threads are killed, including the calling one);
- `thread.limit` allows to ensure that no more than the specified number of threads with the same origin are present; its first argument is the maximum number of threads, and the second one is one of the strings "this", "earliest" or "latest", indicating which thread(s) should be killed if their count is above the limit (it's possible to specify an array of values, like {"this", "latest"}, instead).

8.6.2.5 User interface

The global `ui` table provides means of configuring the LCD display, backlight and beeper.

Functions `ui.beep` and `ui.blink` configure the beeper and LCD backlight, respectively. Their first argument should be a string of "1"s and "0"s, which specifies the pattern, and their second argument should be the number of seconds after which the preceding behaviour is restored.

The `ui.line` table has two elements `ui.line[1]` and `ui.line[2]`, specifying the custom displayed strings for the LCD rows (or nil for regular operation of said row). This offers more fine-grained control than the `DISPLAY` command above.

As follows from the above, the number of LCD rows is `# ui.line`. Use `ui.column_count` to retrieve the number of LCD columns (usually 16).

8.6.2.6 Configuration access

The global variable `config` represents a partial read-only view on the unit configuration (most of them strings):

- `config.serial`: unit serial number;
- `config.hostname`: unit hostname;
- `config.contact`: primary unit contact;
- `config.contacts`: a table storing contacts related to the unit in different ways;
- `config.location`: physical location of the unit;
- `config.timezone`: system time zone;
- `config.hardware_id`: unit hardware model;
- `config.oid`: object identifier of unit model;
- `config.version`: firmware version;
- `config.outlet_label`: kind of endpoint manipulated by the unit (Outlet/Relay).

8.6.2.7 Transient state management

Local and global variables of scripts are shared between [threads](#) created in a single script environment but are in general separate between separately loaded script environments. In the example below, if you create several threads using `thread_creator`, they will all reference the same instances of `local_var` and `global_var`:

```
local local_var=0
global_var=0

function thread_fn()
  while true do
    ...
    local_var=local_var+1
    global_var=global_var+1
    ...
  end
end

function thread_creator()
  for i=1,10 do
    thread.run(thread_fn)
  end
end
```

Even if you run `thread_creator` several times (e.g. from the web UI) without changing source code, all of the threads will share both `local_var` and `global_var`.

However, if you change the code and launch `thread_creator` again, new instances of `local_var` and `global_var` will be created; the 10 new threads will be completely separate from the old threads.

This makes handling global functions and variables consistent (e.g. if you don't have a global variable in the script, you won't accidentally trip over it if it was there in a script you loaded several edit iterations earlier), but this default behaviour may or may not be what you want.

To store arbitrary data between script edits, you can create entries in the global `sticky` table, like this:

```
sticky["variable"]="some value to save between script edits"
```

As usual in Lua, for identifier-like keys you can alternatively use the dot syntax:

```
sticky.variable="some value to save between script edits"
```

The above example then becomes:

```
function init_sticky()
  sticky.local_var=0
  sticky.global_var=0
end

function thread_fn()
  while true do
    ...
    sticky.local_var=sticky.local_var+1
    sticky.global_var=sticky.global_var+1
    ...
  end
end

function thread_creator()
  for i=1,10 do
    thread.run(thread_fn)
  end
end
```

If you overwrite the whole `sticky` table like this:

```
function init_sticky()
  sticky={local_var=0,global_var=0}
end
```

the changes will only apply from that environment on (threads started in previous environments will be unaffected).

In both cases you'll need to call `init_sticky` once explicitly before calling `thread_creator`, which might be inconvenient. Alternatives include setting default values, if they are not nil, on entry to functions that use the values:

```
function thread_fn()
  sticky.local_var=sticky.local_var or 0
  sticky.global_var=sticky.global_var or 0
  while true do
    ...
    sticky.local_var=sticky.local_var+1
    sticky.global_var=sticky.global_var+1
    ...
  end
end
```

or writing code that it handles default nil values transparently:

```
function thread_fn()
  while true do
    ...
    sticky.local_var=(sticky.local_var or 0)+1
    sticky.global_var=(sticky.global_var or 0)+1
    ...
  end
end
```

Variables created this way are not persisted across reboots; see [below](#) for those that are.

8.6.2.8 Externally accessible state management

Local and global variables of scripts are in general not accessible from outside the scripting engine, e.g. they cannot be manipulated from the REST-like API.

You can create entries in the global `external` table, like this:

```
external["variable"]="some value"
external["variable2"]=12345
```

and have them accessible under `/restapi/script/variables/variablename/`.

You can only store strings, numbers, booleans or nil into this table at nonempty string indices. Attempts to use nonstring keys, or table or function values, or overwrite the `external` table, will result in an error:

```
external={} -- error
external["x"]=outlet -- error
external[5]=external -- error
```

The `external` table otherwise behaves like `sticky`; in particular, variables created this way are not persisted across reboots; see [below](#) for those that are.

8.6.2.9 Persistent state management

User scripts can read and write state variables which persist across power cycles. This is done by modifying the global `persistent` table. For example, the following `log_boot_count` function, if [configured to be started at cold boot](#), can report the current cold boot number (and the `ordinal` function is a convenience for printing strings like "1st", "2nd", "3rd", etc.).

```
local special_suffixes, ordinal

function log_boot_count()
    local boot_count=persistent.boot_count or 0
    boot_count=boot_count+1
    persistent.boot_count=boot_count
    LOG("This is my "..ordinal(boot_count).. " power cycle")
end

special_suffixes={"st", "nd", "rd"}

function ordinal(number)
    local d=number%100
    return tostring(number)..((d<11 or d>19) and special_suffixes[d%10] or "th")
end
```

Here, the "boot_count" key is used to store the number of boots. Like with ordinary Lua tables, nonexistent keys have nil values; to remove a key from `persistent`, write nil to it.

Only strings can be used as keys of `persistent`. Only numbers, strings and booleans can be stored (or nil can be written to erase a value). Some of these limitations may be raised in future.

All keys of `persistent` can be enumerated if necessary using `pairs()`.

8.6.2.10 Meter access

The global `meter` table allows to read the following measured values (where N is 1 bus A or 2 for bus B) :

- `meter.temperature` (the measured internal temperature in degrees Kelvin);
- `meter.illuminance` (the measured illuminance in lux);
- `meter.buses[N].current` (the measured current in amperes);
- `meter.buses[N].voltage` (the measured voltage in volts);
- `meter.buses[N].total_energy` (the measured energy in watts);
- `meter.values`: a table storing properties of different values measured by meters.

Keys of `meter.values` are unique human-readable but possibly platform-specific identifiers (e.g. `power_↔voltage`), and values are tables with the following form:

- `meter.values[key].name`: human-readable name of the measured value;
- `meter.values[key].quantity`: kind of the physical quantity of the value (e.g. "current", "voltage", etc.);
- `meter.values[key].value`: number representing the current value measured by the meter in standard units;
- `meter.values[key].custom`: boolean indicating if the value is custom (user-defined);
- `meter.values[key].bus`: nil if the value is not related to a bus, or the zero-based bus index otherwise;

- `meter.values[key].internal`: boolean indicating if the value is internal (and should not appear in the web UI);
- `meter.values[key].get_history`: function which can be called to obtain historical data points.

The `get_history()` function must be called like this: `get_history(desired_start_time, desired_end_time, desired_step)`. The `desired_start_time` is the timestamp of the start of the desired interval (in seconds since the epoch, 1970-01-01T00:00:00Z). Similarly, `desired_end_time` is the desired end time. `desired_step` is the desired step between adjacent time points. Resulting data may not match the requirements exactly, but will be generated so that it overlaps the desired time range and has the closest time step. The function returns a table with 3 elements `{actual_start, actual_step, data}` (note that it doesn't return 3 values, which is possible for Lua functions, but a single table containing them). `actual_start` is the timestamp of the start of the output interval, `actual_step` is the step between adjacent time points, and `data` is a table whose *i*th element is the (averaged) value of the meter around time `actual_start+(i-1)*actual_step` (mind that indices in Lua are 1-based) or `false` if no reading was obtained within that time interval (e.g. if the device was offline).

Its use is best illustrated by an example. Suppose `temperature` is a value of the form:

```
local temperature=meter.values[...]
```

Then the following script could be used for obtaining bounds for the 5-minute-average temperature for the last 24 hours:

```
local now=os.time()
local history_data=temperature.get_history(now-86400,now,300)
local start,step,data=unpack(history_data)
-- or local start,step,data=history_data[1],history_data[2],history_data[3]
local mintime,mintemp,maxtime,maxtemp
for pos,temp in ipairs(data) do
    if temp then
        local time=start+(pos-1)*step
        if mintemp==nil or mintemp>temp then
            mintemp=temp
            mintime=time
        end
        if maxtemp==nil or maxtemp<temp then
            maxtemp=temp
            maxtime=time
        end
    end
end
if mintime then
    LOG(string.format("Min temperature was %gK at %s",mintemp,os.date("%H:%M:%S",mintime)))
    LOG(string.format("Max temperature was %gK at %s",maxtemp,os.date("%H:%M:%S",maxtime)))
else
    LOG("No temperature readings for the last 24 hours!")
end
```

Note that values of `meter.values[key].value` and `meter.values[key].get_history()` are in standard SI units, e.g. degrees Kelvin for temperature.

8.6.2.11 AutoPing integration

The global `autoping` table allows to query and configure AutoPing.

- `autoping.enabled`: boolean variable which allows enabling (`true`) or disabling (`false`) AutoPing;
- `autoping.items[N].enabled`: read-only boolean value indicating if the *N*th AutoPing item is enabled;

- `autoping.items[N].enable`: function to call to attempt to enable (with the argument `true`) or disable (with the argument `false`) the Nth AutoPing;
- `autoping.items[N].addresses`: array of hostnames or IP addresses of the Nth AutoPing item's elements;
- `autoping.items[N].outlets`: array of outlets controlled by the Nth AutoPing item;
- `autoping.items[N].script`: the name of the scripting function run by the Nth AutoPing item when it's triggered (" " to cycle the controlled outlets);
- `autoping.items[N].status`: the run-time status of the Nth AutoPing item;
- `autoping.ping_interval`: ping interval;
- `autoping.ping_timeout`: ping timeout;
- `autoping.post_reboot_delay`: post-reboot delay;
- `autoping.max_reboot_count`: maximum reboot count;
- `autoping.pings_before_enabling`: pings before enabling.

8.6.2.12 Network state access

The global `network` table has two similarly-structured members, `network.wired` and `network.wireless`, which represent partial read-only views on state and configuration of the respective networks (all fields are strings unless indicated otherwise):

- `network.(wired|wireless).online`: boolean value indicating if the interface is online;
- `network.(wired|wireless).protocol`: method for obtaining the IP address (static/dhcp);
- `network.(wired|wireless).ip_address`: IP address;
- `network.(wired|wireless).netmask`: network mask;
- `network.(wired|wireless).gateway`: default gateway;
- `network.(wired|wireless).mac_address`: MAC address;
- `network.wireless.enabled`: boolean value indicating if the wireless interface is enabled;
- `network.wireless.mode`: wireless module mode;
- `network.wireless.ssid`: wireless network name;
- `network.wireless.channel`: wireless channel (channel number as string, e.g. "11", or "auto");
- `network.wireless.encryption`: wireless encryption (none/psk/psk-mixed/psk2).

8.6.2.13 Event APIs

The global `event` table allows limited integration of user scripts with the [notification subsystem](#).

The `event.send` function can be used to send a `dli.script.script_event` event with custom message and data. It takes the event properties `script_message` and `script_data` as arguments. `script_message` must be a string that will be included in the event message. `script_data`, if supplied, must be a table with string keys and scalar (boolean, number or string) values, and can be analyzed by the [rule conditions](#) and/or used by [rule actions](#). Non-string keys or non-scalar values are ignored; if the `script_data` argument is not a table, an empty table is sent. All script-generated events have a (possibly empty) table as `script_data`, and only they have it, so its existence is a distinctive feature of script-generated events.

For example, the following code

```
event.send("coil 1 energized",{coil_index=1,coil_state=true})
```

will send an event that can be matched by, among others, the following notification rules:

```
id=="dli.script.script_event" and script_data.coil_index==1
script_data and script_data.coil_state
```

All script-generated events have INFO severity level.

Events can also be waited for and received, and that is not limited to system-level events discussed above. An event can have several components with integer indices (the components themselves can be any types, e.g. tables). Events generated by the API generally have two components, a numeric timestamp (in seconds since the epoch, 1970-01-01T00:00:00Z) and a property table, but totally custom events are possible.

It is usually essential not to miss an event when processing them, so events are placed into queues waiting to be processed. The `event.stream` function is a Lua generator to be used in a `for` loop like this:

```
for queue_idx,value1,value2,value3... in event.stream(queue1,queue2,queue3...) do
    ...
end
```

`event.stream` accepts any number of queue arguments and waits for events to be received (causing a delay and allowing other threads to run), then extracts and returns their components. Queues are processed in priority order (e.g. `queue1` messages get processed before `queue2` messages).

The `event.queue` function creates a totally custom event queue not bound to any event source. Additional custom events can be placed into any queue using `q[# q+1]={value1,value2,value3...}` regardless of the method used to create the queue; for queues created by `event.queue`, this is also the only way for events to appear there.

The `event.listener` function creates an event queue listening to the global system events (i.e. the ones received by the notification system and generated by other systems and `event.send` function). Events have a timestamp and a property table as components. The property table contains event-type-dependent properties (e.g. the `id` is the event type identifier).

The following example function can be used to display all system events in internal form in the event log:

```
function dump_system_events()
  for i,t,data in event.stream(event.listener()) do
    dump({i,t,data})
  end
end
```

The `event.change_listener` function takes any number of API objects `object1,object2,object3...` as arguments and creates an event queue listening to changes in the API objects' states. Events have a timestamp and a property table as components; the property table contains the following fields:

- `object`: the changed object;
- `index`: the object's number in the argument list;
- `key`: the property of the object that has changed;
- `value`: the new value of the property.

Only a subset of the API's objects support change notifications; they include outlet objects `outlet[i]` , meter values `meter.values[k]` and AutoPing items `autoping.items[i]`.

The following example function can be used to attempt to mirror physical state of outlet 1 to outlet 2:

```
function mirror_outlet_1_to_2()
  outlet[2].state=outlet[1].physical_state
  for i,t,data in event.stream(event.change_listener(outlet[1])) do
    if data.key=="physical_state" then
      outlet[2].state=data.value
    end
  end
end
```

The `event.local_time` and `event.utc_time` functions take any number of time-matching [condition tables](#) as arguments and creates an event queue where events appear in time moments matching the conditions. This can be seen as an improved `wait_until` function which doesn't miss triggers if handling the event takes too long. The most important difference is that new checks are "edge-triggered", not "level-triggered": an event is placed into the queue only when the condition becomes true; no events are created when the condition stays true until it becomes false and then true again. For example, if you start a loop waiting for `{hour=7,min=10}` at 7:10, the event will not be created right away and the loop will not be executed until the next day, whereas `wait_until` would return immediately. This is because a "level-triggered" approach might have to add an infinite number of events to the queue. This also means that you usually won't have to add e.g. `min=0,sec=0'` or similar to the condition to make it start only at the beginning of an hour. Additionally, `wait_until` only handles local time, which corresponds to `event.local_time` behaviour, but you can match UTC time with `event.utc_time`, which has no corresponding `wait_until` option. Events have a timestamp and a property table as components; the property table contains the following fields:

- `index`: the condition's number in the argument list;
- `time`: the time matching the condition (not necessarily exactly one of the timestamp).

The following example function can be used to check outlet 1 state and report if it remains physically off for more than an hour. Note that we do not simply poll the outlet e.g. every minute, but subscribe to state change events instead so that we don't miss any changes. We also create the change event queue before the initial state check so that we don't miss any state changes during initialization.

```
function monitor_1()
  local reported
  local off_since
  local changes=event.change_listener(outlet[1])
  if not outlet[1].physical_state then
    off_since=os.time() -- We don't know that for sure but that's when we start monitoring
  end
  for i,t,data in event.stream(changes,event.utc_time({sec=0})) do
    if i==2 then
      if off_since and t-off_since>3600 then
        if not reported then
          log.warning("Off for too long, since %s",os.date("%c",off_since))
          reported=true
        end
      end
    elseif data.key=="physical_state" then
      if data.value==true then
        off_since=nil
        if reported then
          log.notice("On again, phew")
          reported=true
        end
      elseif off_since==nil then
        off_since=t
      end
    end
  end
end
```

Of course we could have used e.g. `event.send` instead of `log`, or flipped another outlet, etc.

You should not usually poll the same queue from different threads. Every event from a queue will be processed by a single thread only. A single event can be placed into multiple queues though.

Like any other loop, a loop over `event.stream(...)` can be terminated via a `break` statement in the body. The catch is that the loop body isn't executed until a matching event occurs. If you need to terminate a thread's event loop from another thread, you may e.g. create and share an event queue between them, and place an event into it when you want to terminate the loop, like this:

```
local function event_thread(cancel_queue)
  ...
  for i,t,data in event.stream(cancel_queue,...) do
    if i==1 then
      break
    else
      ...
    end
  end
  ...
end

function caller_thread()
  local cancel_queue=event.queue()
  thread.run(function() return event_thread(cancel_queue) end)
  ...
  cancel_queue[#cancel_queue+1]={}
  ...
end
```

There are obviously other ways to share a queue like this. The location of `cancel_queue` in the `event.stream` argument list above affects whether the loop is terminated immediately after receiving the cancellation message or after processing outstanding events in other queues.

You can also just use `thread.kill` if you don't need the looping thread at all any longer.

8.6.2.14 Generic input/output

The global `io` table allows integration of user scripts with the [generic I/O subsystem](#).

Scripts have access to any pins or ports your controller has.

The global `io.ports` table entries represent I/O ports, keyed by their internal identifiers.

- `io.ports[key].name`: user-friendly name of the port;
- `io.ports[key].type`: type of the port (serving as key into `io.known_port_types`);
- `io.ports[key].configuration`: configuration string key-value storage;
- `io.ports[key].send`: function which can be called to send data to the port's channels;
- `io.ports[key].accept`: mask of channels the port is receiving data from;
- `io.ports[key].latest_received`: table representing the latest set of data received from the port.

Note that `io.ports[key].latest_received` is, and `io.ports[key].send` accepts as argument, a table mapping channel identifiers to data in hexadecimal format. E.g. if the port's channel is named "d", the way to send 'Hello' to it would be

```
io.ports[key].send({d="48656c6f"})
```

If you receive 'Hi', `io.ports[key].latest_received` will look like:

```
{d="4869"}
```

This allows for full 8-bit exchange without limitations of UTF-8.

You can obtain the port's type and look it up in the `io.known_port_types` table to obtain details:

- `io.known_port_types[key].data_items`: configuration parameters and their properties (name and regexp);
- `io.known_port_types[key].channels`: table of channels and their names;

E.g. the following function dumps the ports and their current configuration:

```
function describe_ports()
  for id,port in pairs(io.ports) do
    log.notice("Port %s is a %s",id,port.type)
    local t=io.known_port_types[port.type]
    for k,parameter in pairs(t.data_items) do
      log.notice(" %s = %s",parameter.name,port.configuration[k])
    end
  end
end
```

The `io.ports[key].latest_received` changes on every data reception, and can be used together with the [event APIs](#) to work on the data stream being received. For instance, the following will dump all data from an uart port named `uart_ext` to the system log:

```
function dump_external_uart()
  for i,t,data in event.stream(event.change_listener(io.ports.uart_ext)) do
    if data.key=="latest_received" then
      log.notice("Data received from UART:%s",data.value["d"])
    end
  end
end
```

You may use [utility](#) functions `util.hex.encode` and `util.hex.decode` to perform conversions from/to hexadecimal string notation.

The global `io.pins` table entries represent general purpose I/O pins, keyed by their internal identifiers.

- `io.pins[key].name`: user-friendly name of the pin;
- `io.pins[key].input_bits`: number of bits in pin input value (if 0, pin cannot be read so it makes no sense to use it as a source signal of a net);
- `io.pins[key].mode_bits`: number of bits in pin mode value (if 0, pin's mode is locked and cannot be modified);
- `io.pins[key].level_bits`: number of bits in pin level value (if 0, pin's level is locked and cannot be modified);
- `io.pins[key].mode_driver`: driver of the pin's mode (active/tristated);
- `io.pins[key].level_driver`: driver of the pin's level (high/low);
- `io.pins[key].drive_logic_mapping`: a table describing how a pin's mode/level configuration maps to IEEE1164 signal states:

- `io.pins[key].drive_logic_mapping[1]`: signal state for the minimum mode, minimum level configuration;
- `io.pins[key].drive_logic_mapping[2]`: signal state for the minimum mode, maximum level configuration;
- `io.pins[key].drive_logic_mapping[3]`: signal state for the maximum mode, minimum level configuration;
- `io.pins[key].drive_logic_mapping[4]`: signal state for the maximum mode, maximum level configuration.

The IEEE1164 signal states supported are:

- '0', Forcing 0;
- '1', Forcing 1;
- 'L', Weak 0;
- 'H', Weak 1;
- 'Z', High impedance.

Mode / level drivers can assume the following values:

- `true`: maximum mode/level (active / high);
- `false`: minimum mode/level (tristate / low);
- name of a net: the pin's characteristic will depend on the value of the named net's value.

In the most basic usage scenarios, you may simply flip the driver appropriate for the pin in the script:

```
io.pins.out1.mode_driver=true
io.pins.out1.level_driver=true
...
io.pins.out1.level_driver=false
```

(for a high/low pin) or

```
io.pins.out1.mode_driver=false -- tristate
...
io.pins.out1.mode_driver=true -- pull down
```

(for an open collector pin which has `level_driver` locked to `false`).

However, this doesn't make it possible to receive data read from pins or to perform more complex operations. For these cases, signal nets are used.

The global `io.nets` table entries represent general purpose I/O signal nets, keyed by their identifiers. The table is initially empty; it's up to you to create any entries. Entries have the following format:

- `io.nets[key].name`: user-friendly name of the net;
- `io.nets[key].bits`: number of bits in net value;
- `io.nets[key].expression`: the expression governing the net's value;

- `io.nets[key].reporting`: null to disable reporting the net's state for performance, "pull" to perform periodic reads, "push" to send every change;
- `io.nets[key].latest_value`: latest known value of the net.

For instance, you could initially run the following function to create a change-reporting net for each of your input pins:

```
function init_read_nets()
  for k,pin in pairs(io.pins) do
    if pin.input_bits>0 then
      io.nets["input_"..k]={
        name=("Input from %s"):format(k),
        bits=pin.bits,
        expression=("pin[%q]"):format(k),
        tracking="push"
      }
    end
  end
end
```

Note that this isn't a good idea if you have many oscillating input pins as it will slow down processing, so this isn't done by default.

The following sample will monitor the input pin "in1", dumping values to log:

```
function dump_in1_changes()
  io.nets["input_in1"]={name="IN1",bits=1,expression="pin.in1",tracking="push"}
  for i,t,data in event.stream(event.change_listener(io.nets["input_in1"])) do
    if data.key=="latest_value" then
      log.notice("IN1 changed to %d",data.value)
    end
  end
end
```

To also dump the initial value, use e.g. (note the separation of listener creation from enumeration to make sure no updates are missed in between):

```
function dump_in1()
  io.nets["input_in1"]={name="IN1",bits=1,expression="pin.in1",tracking="push"}
  local listener=event.change_listener(io.nets["input_in1"])
  log.notice("IN1 is currently %d",io.nets["input_in1"].latest_value)
  for i,t,data in event.stream(listener) do
    if data.key=="latest_value" then
      log.notice("IN1 changed to %d",data.value)
    end
  end
end
```

The main power of nets lies in their ability to control pin modes and levels. For instance, this sample configures the high/low pin `out1` to be the logical AND of pins `in1` and `in2`. No tracking is configured for performance.

```
function in1_and_in2()
  io.nets["inland2"]={name="IN1 and IN2",bits=1,expression="pin.in1 and pin.in2"}
  io.pins["out1"].level_driver="inland2"
  io.pins["out1"].mode_driver=true -- set active
end
```

You should use `dump(io)` to inspect the capabilities of your device for further details.

8.6.2.15 Utility functions

The global `util` table provides helpful utility constants and functions which usually have no state or side effects.

`util.hex`, `util.base64`, `util.url` are tables with `encode` and `decode` members which perform hex, Base64 and URL component encoding and decoding, respectively; they both accept and accept strings.

The `util.json` table also contains `encode` and `decode` members, which perform JSON encoding and decoding, respectively. `util.json.encode` accepts any regular value (without cycles) and returns a string; `util.json.decode` accepts a string and returns a value corresponding to it.

Due to how Lua handles tables, an empty JSON array `[]` cannot be distinguished from an empty JSON object `{}` once decoded. To help that, `util.json.encode` accepts an optional second argument, which is an "empty array table", that is, a table with keys corresponding to empty Lua tables in the input which represent empty JSON arrays. For example,

```
local data={a={},b={}}
log.notice("%s",util.json.encode(data,{[data.b]=true}))
```

produces:

```
{"a":{}, "b":[]}
```

Conversely, `util.json.decode` returns an empty array table as a second value if it is nonempty (that is, if there were empty arrays in the input).

```
local value,is_empty_array=util.json.decode('{"a":{},"b":[]}',is_empty_array)
log.notice("a: %s",type(value.a),is_empty_array[value.a] and ":array" or "")
log.notice("b: %s",type(value.b),is_empty_array[value.b] and ":array" or "")
```

Thus JSON round-trips, i.e. `util.json.encode(util.json.decode(JSON))` produces a JSON string equivalent to the input JSON even in the presence of empty arrays/objects.

All transcoding functions may throw errors on invalid input; using `pcall` / `xpcall` is advised.

`util.null` is the JSON `null` constant, distinct from, but sometimes interchangeable with, `nil`. It may be useful when dealing with some APIs.

The `util.copy(val[, deep_keys])` function returns a deep plain copy of its argument (which can be or contain an API structure). This may be helpful when working with `util.json.encode` as it doesn't by itself support encoding API structures. An optional argument, `deep_keys`, can make `util.copy` perform deep copying of table keys (in case they're tables themselves), otherwise table keys are left intact.

The `util.equal(val1, val2)` function performs a deep equality comparison of its two arguments (each of which can be or contain an API structure) and returns true if they are equal and false otherwise. Tables are compared structurally, i.e. they are considered equal if they have equal keys with equal values. Unlike `util.copy`, `util.equal` doesn't support comparisons with copied table keys as those are in general undecidable without additional information (consider `{{{}}=1, [[]]=2}` and `{{{}}=2, [[]]=1}`).

The `util.argpack` function compresses its arguments into a table with an extra `n` member indicating the number of arguments. It could be implemented as:

```
util.argpack = function(...)
    return {n=#select("#",...),...}
end
```

The `util.argunpack` function expands a table into a value list with the number of elements taken from the table's `n` member. It could be implemented as:

```
util.argunpack=function(tbl)
    return unpack(tbl,1,tbl.n)
end
```

In combination, `util.argpack` and `util.argunpack` allow perfect function argument forwarding in presence of `nil` arguments.

Unlike most APIs, utility functions are available in the global context.

8.6.2.16 Debugging

- `dump` — useful debugging function which outputs the argument to the system log, can be used to inspect state and even study the modern API itself (try `dump(_G)!`). Note that this function may delay execution while dumping sufficiently large objects. Other threads may run while execution of the dumping thread is suspended. If they modify the object being dumped, the resulting dump output may be inconsistent.
- `log` — contains methods `debug`, `info`, `notice`, etc. which accept a format string argument and any extra arguments it requires, format the string using `string.format` internally and log it at the corresponding severity level, e.g.

```
local i=5 local name="Fridge" log.notice("Switching on %d (%s)",i,name)
```

8.7 Starting scripts

There are a few ways to start scripts:

- On power up. This feature automatically starts a specified script function when power is first applied. The default is not to start any function, so pressing the "reset to defaults" button will disable this feature.
- By another thread. One thread can create another by using the `thread.run` function. For example, `thread.run(func1)` creates a new thread that starts executing the `func1` function. The execution of the parent thread continues.
- By issuing an HTTP request. Follow a link http://Your_IP/script.cgi?run=func to start execution from function `func`. This can be conveniently used by the end users by assigning the programmable web links on the left side of the page a target of the form `script.cgi?run=func`.
- Via AutoPing. The [AutoPing system](#) can be configured to automatically start execution when IP connectivity is lost. Select the script to run from the selection box to the right of the corresponding IP on the AutoPing page.
- By manually clicking the Start button. Execution will start with the selected function.

8.8 Editing scripts

You don't need to disable scripting before editing scripts. If you make a syntactic error, the script won't be modified. Instead, you'll receive an explanatory message pointing to the error.

If you modify a running script, existing threads will continue to run with the code and environment that existed when the script was started. New threads won't interact with old ones directly. Use [transient](#) and [persistent](#) state APIs for that.

8.9 Stopping a thread

A thread terminates automatically when the end of its outermost function is reached. Click "Stop all running threads" to stop everything. You can also stop all scripts via HTTP using http://Your_IP/script.cgi?stop.

8.10 Relay debounce warning

Even with the scripting step delays, it is possible to create a script which will rapidly cycle a relay. This rapid cycling could result in a over current condition, tripped breaker, or stress to the power controller or attached equipment. Please be reasonable!

9 Event notification

Certain power controller events can trigger configurable notifications over a variety of media:

- email;
- XMPP (Jabber, Google Talk, etc.);
- SNMPv1, SNMPv2c and SNMPv3 TRAPs;
- SNMPv2c and SNMPv3 INFORMs;
- webhook.

Notification is based on the [Lua](#) programming language.

9.1 Notification context

When an event happens, a notification thread is started, with event's properties being copied to global variables (and thus constituting the context of further notification code snippets).

The most important of the rule action context (see below) is the `notify` function, which sends the current notification context to matching targets.

All kinds of events share the following properties:

- `id` — the event type identifier,
- `message` — a human-readable message,
- `severity` — the event severity level.

By manipulating the context, rules can check for and adjust the event's properties and prepare it for sending.

For example, you can have a rule with an empty condition and an action altering the 'message' variable like this:

```
message="Server room power: " .. message
```

All the following rules will include the "Server room power:" prefix in the generated notifications.

Some properties, e.g. `message_short` or `message_long`, are supported by notification targets but aren't generated by any events; it's up to the action code to set them if necessary.

It's important to note that rules are processed linearly: adjustment of properties doesn't cause preceding rules to be re-examined.

Additionally, different event types have more specific properties. All event properties can be checked for or adjusted.

The following convenience severity level constants are defined:

- `EMERG, EMERGENCY` — "emergency" severity level,
- `ALERT` — "alert" severity level,
- `CRIT, CRITICAL` — "critical" severity level,
- `ERR, ERROR` — "error" severity level,
- `WARNING, WARN` — "warning" severity level,
- `NOTICE` — "notice" severity level,
- `INFO, INFORMATION, INFORMATIONAL` — "informational" severity level,
- `DEBUG` — "debug" severity level.

Their numeric values are defined so that a higher severity is larger, so condition like `severity>=CRITICAL` behaves like what you'd expect.

The [core Lua functions](#) are accessible from the context as well, should you need them.

Some fields are autofilled for popular email services once you enter the sender's address.

If present, the `message_short` event property is used for the email subject; otherwise, `message` is used.

If present, the `message_long` event property is used for the email body; otherwise, `message` is used.

If present, the `timeout` property can be used to specify the time limit, in seconds, for the notification operation to complete. Consider increasing it, by specifying a higher value in the rule action, if you experience frequent timeouts.

Event properties `client_cert` and `client_key` can be used to specify a pair of client certificate/key files if TLS is used. The `ca_root` event property, if present, can be used to specify the set of trusted CA root certificates (it may point to a single file listing all of them, or to a directory containing the certificate files in OpenSSL format, and defaults to `/etc/ssl/certs`).

9.2.2 XMPP notifications

XMPP notification targets have the following parameters:

- recipient XMPP ID (RFC822-like);
- sender XMPP ID (RFC822-like);
- password (for authenticating to the server).

If present, the `message_short` event property is used for the message; otherwise, `message` is used.

If present, the `timeout` property can be used to specify the time limit, in seconds, for the notification operation to complete. Consider increasing it, by specifying a higher value in the rule action, if you experience frequent timeouts.

Event properties `client_cert` and `client_key` can be used to specify a pair of client certificate/key files if TLS is used. The `ca_root` event property, if present, can be used to specify the set of trusted CA root certificates (it may point to a single file listing all of them, or to a directory containing the certificate files in OpenSSL format, and defaults to `/etc/ssl/certs`).

9.2.3 SNMP notifications

The notification system supports sending:

- SNMPv1 TRAPs;
- SNMPv2c TRAPs and INFORMs;
- SNMPv3 TRAPs and INFORMs.

The difference between a TRAP and an INFORM is that an INFORM requires confirmation of receipt. Thus, the target test function can tell if the message has been delivered.

9.2.3.1 Trap OIDs

The type of a TRAP or INFORM is indicated by its OID. SNMPv2c and SNMPv3 include the full OID in the message; SNMPv1 is different.

SNMPv1 traps are identified by the generic trap type (and correspond to the following trap OIDs):

- 0 — cold start (1.3.6.1.6.3.1.1.5.1),
- 1 — warm start (1.3.6.1.6.3.1.1.5.2),
- 2 — link down (1.3.6.1.6.3.1.1.5.3),
- 3 — link up (1.3.6.1.6.3.1.1.5.4),
- 4 — authentication failure (1.3.6.1.6.3.1.1.5.5),
- 5 — EGP neighbor loss (1.3.6.1.6.3.1.1.5.6).

If the generic trap type is 6, the trap is enterprise-specific (and is usually taken to correspond to OIDs using the template "1.3.6.1.4.1.ENTERPRISE-OID.0.SPECIFIC-TRAP-TYPE").

Unless you send only generic traps or have a OID tree registered to you, you may want to send 'user-configured' traps which carry no additional semantics other than the variable bindings. The `dlinGeneric` user-configured trap type is designed for that. Its OID is 1.3.6.1.4.1.45770.0.1; this can be specified directly for SNMPv2c or SNMPv3, or as an enterprise-specific (generic trap type=6) DLI ("enterprise OID"=45770) trap #1 (specific trap type=1).

9.2.3.2 Security settings

SNMPv1 and SNMPv2c use the 'community' security model which essentially identifies users by a shared secret which is sent over the network in plain text ('public' and 'private' being the most popular 'secrets'). That means that they are very insecure and shouldn't be deployed over an untrusted network. SNMPv3 has a more reasonable security model.

9.2.3.3 Variable bindings

All SNMP TRAPs and INFORMs accept the `snmp_values` event property to send extra values in the message. The property, if not nil, must be an array of the following shape:

```
{ {oid1,value1,type1}, {oid2,value2,type2}, ... }
```

The order may be important; you may want to consult the MIB. The following types are supported:

- "integer",
- "unsigned",
- "counter32",
- "string",
- "hex string",
- "decimal string",
- "nullobj",
- "objid" or "oid",
- "timeticks",
- "ipaddress" or "ip",
- "bits".

Type names are case-insensitive. A type may be omitted, in that case it will be inferred (nil values will be encoded as null objects, strings as octet strings, numbers as integers, "true" values as integer 1, and "false" values as integer 2 as per SMIv2).

9.2.3.4 SNMPv1 settings

SNMPv1 trap targets have the following parameters:

- server address (hostname or IP address of management station),
- community string (the shared secret for authentication to server),
- enterprise OID (number),
- default generic trap type (number 0..6),
- default specific trap type (number 0..2147483647).

SNMP v1 accepts `snmp_enterprise_oid`, `snmp_generic_trap_type` and `snmp_specific_trap_type` event properties to override the trap OID; they must be integers or string representations thereof. It also accepts the `snmp_values` event property as described [above](#).

9.2.3.5 SNMPv2c settings

SNMPv2c TRAP and INFORM targets have the following parameters:

- server address (hostname or IP address of management station),
- community string (the shared secret for authentication to server),
- trap OID (string)

SNMP v2c TRAPs and INFORMs accept an `snmp_trap_oid` event property to override the trap OID; it must be a string. They also accept the `snmp_values` event property as described [above](#).

9.2.3.6 SNMPv3 settings

SNMPv3 TRAP and INFORM targets have the following parameters:

- server address (hostname or IP address of management station),
- security name (username to authenticate as),
- security engine ID (leave empty for default),
- context name (leave empty for default),
- context engine ID (leave empty for default),
- authentication protocol ("MD5" and "SHA" are supported, leave empty "" for no authentication),
- privacy protocol ("DES" and "AES" are supported, leave empty "" for no encryption),
- authentication passphrase (should be at least 8 characters long, or empty if no authentication),
- privacy passphrase (should be at least 8 characters long, or empty if no encryption),
- trap OID (string)

You can either disable authentication and encryption, enable only authentication, or enable both. An encrypted, but not authenticated configuration is invalid.

SNMP v3 TRAPs and INFORMs accept an `snmp_trap_oid` event property to override the trap OID; it must be a string. They also accept the `snmp_values` event property as described [above](#).

9.2.4 Webhook notifications

Web hooks notifications send preconfigured POST HTTP requests to URLs.

- recipient URL — the URL to send a request to (must be an HTTP or HTTPS URL);
- content type — the type of content representation to use (must be "json" for JSON or "urlencoded" for URL-encoded).

The request payload (content) is a key-value map with the following keys:

- `type` — the id of the event type;
- `severity` — the severity label for the event (one of the strings "emergency", "alert", "critical", "error", "warning", "notice", "info" or "debug");
- `message` — the human-readable message of the event type;

If present, the `properties` event property is used to populate the payload with additional items; for instance, an action `properties={time=os.time() }` will result in a `time` key being added. Non-string values will be JSON-encoded (regardless of the content type).

If present, the `headers` event property is used to add HTTP headers to the request being sent.

If present, the `message_short` event property is used for the message; otherwise, `message` is used.

If present, the `timeout` property can be used to specify the time limit, in seconds, for the notification operation to complete. Consider increasing it, by specifying a higher value in the rule action, if you experience frequent timeouts.

Event properties `client_cert` and `client_key` can be used to specify a pair of client certificate/key files if HTTPS is used. The `ca_root` event property, if present, can be used to specify the set of trusted CA root certificates (it may point to a single file listing all of them, or to a directory containing the certificate files in OpenSSL format, and defaults to `/etc/ssl/certs`).

9.3 Notification rules

The rule system is centered around two kinds of entities:

- condition, which determines if a rule is matched, and
- action, which determines what happens if it matches.

Notification rules			
#	Condition	Action	Operations
1	outlet==1	message="The up escalator is down!" severity=CRITICAL	   
2	severity>=CRITICAL	notify("admin")	   
3			  

Notification rule configuration

A condition is a Lua expression, while rules are Lua blocks (may contain several statements).

Roughly, the condition/action table is equivalent to:

```
if condition1 then action1 end
if condition2 then action2 end
if condition3 then action3 end
...
```

However, an important difference is that if a condition would cause an error, the condition is considered false instead; the corresponding action not taken, but the rule check goes on.

Additionally, an empty condition is equivalent to `true`, and the corresponding action is unconditionally taken. To disable an action without deleting it, you can use an explicitly false condition `false` or a condition that's not a valid Lua expression, e.g. `-`. To keep the condition text, you can wrap it with `false and (...)`.

Rules are applied from the first one to the last one, so order is important. The "Operations" column contains buttons which make manipulating rule order easier.

Though you can filter events by the type identifier, it's not necessary in many cases. For example, the condition `auth_allowed==false` will match only `dli.auth.login_denied` events, as `auth_allowed` is set to `false` for those events only. This is notably distinct from a `not auth_allowed` condition, which will match all sorts of messages which don't have an `auth_allowed` property.

Tricks which allow running code in the condition (as opposed to the action) are possible but discouraged.

9.4 Notification event types

Miscellaneous servers expose the types of events they can produce; they are presented on the notification configuration page in a compact form. Here's an example:

Authentication server events		
Severity	Message	Extra properties
NOTICE	login allowed for <u>auth_login@auth_ip</u> (<u>auth_method</u>)	<u>id</u> ="dli.auth.login_allowed" <u>auth_allowed</u> =true
NOTICE	login denied for <u>auth_login@auth_ip</u> (<u>auth_method</u>): <u>auth_reason</u>	<u>id</u> ="dli.auth.login_denied" <u>auth_allowed</u> =false
NOTICE	protection violation attempt: <u>auth_reason</u>	<u>id</u> ="dli.auth.protection_violation"
INFO	<u>auth_login@auth_ip</u> session closed	<u>id</u> ="dli.auth.session_closed"

Notification event types

Underlined items (both in the message and in the 'extra properties' column) specify properties which can be checked for. You can hover them for more detailed descriptions.

10 Customization page

The customization page allows the administrator to configure some user interface aspects of controller behavior.

10.1 Web page layout and branding

Customize page header	
Company name:	
Product name:	
Product URL:	
Logo (image URL):	
Logo width (default 195):	
Logo height (default 65):	
Submit	

Layout and branding settings

The branding block that appears on every web page can be customized:

- Product name: the displayed name of the product;
- Logo: the company logo image URL;
- Product URL: the URL that the image points to.

Custom logo dimensions may be supplied if needed. The logo URL may be absolute or relative (e.g. `/my_company.png`), in which case the related file should be placed in the `/www/static/` subdirectory of the unit's filesystem (probably via SSH).

Company name affects the alternative text for the logo image.

10.2 Measurement units

Customize preferred measurement units	
Illuminance:	(standard) ▾
Temperature:	degree Fahrenheit ▾
Energy:	kilowatt-hour ▾
Submit	

Units

You can choose which units to display temperature, illuminance and energy values in. This affects both textual and graphical web UI display. Note that it's a presentation option only; values are internally stored, and transferred via REST-like API in standard SI units (degrees Kelvin, lux and joules, respectively) regardless of the customization.

11 External APIs

The controller can be accessed programmatically using a number of protocols and APIs, including:

- the REST-like API (over HTTP),
- JSON-RPC (over HTTP),
- SNMP,
- MQTT,
- UPnP

11.1 Common external API settings

External APIs	
Allow JSON-RPC:	☒
Allow REST-style API:	☒
Enable UPnP service:	☐
Enable SNMP service:	☐
Enable MQTT client:	☐
Relax non-HTML method CSRF checks:	☐
Relax non-HTML content type CSRF checks:	☐

Common external API settings

Each of the external APIs can be enabled separately.

HTTP APIs perform cross-site request forgery checking to make sure they are not called by a misguided browser without JavaScript, bypassing browser security checks (a custom header needs to be present in the requests). Browsers can normally issue GET and POST requests with URL-encoded or multipart content types; you can tick the corresponding "relax ... checks" checkboxes to skip the checks in cases where the method or content type indicates that the request couldn't have been sent by a browser without JavaScript.

11.2 REST-like API

Name([view detailed description](#))

User-visible relay name

string

DLI Controller

PUT

- represents a value in persistent storage
- write is denied if not [administrative user](#)
 - no synchronization requirements

[Up to Relay object](#)

Allowed methods:

GET

PUT

PATCH

REST-like API demo

The REST-style API is based on the REST architectural style. It presents the state and configuration as an hierarchy of resources, and relies on HTTP to perform action signaling and content negotiation. Requests with different HTTP headers yield different representations of resources (e.g. plain text, HTML, JSON, etc.). A type description system is used to outline the object model.

Refer to the REST-style API reference for details.

11.3 JSON-RPC

JSON-RPC test (requires JavaScript) 

Parameters	Request	Reply headers	Result
URI: /jsonrpc/relay Method: get "name" Remove Add new parameter: Add Perform RPC	POST /jsonrpc/relay HTTP/1.1 Content-Type: application/json-rpc Accept: application/json-rpc X-Requested-With: XMLHttpRequest Content-Length: 57 {"jsonrpc": "2.0", "id": 1, "method": "get", "params": [{"name"}]}	Connection: close Pragma: no-cache Allow: POST Expires: Mon, 01 Jan 1990 00:00:01 GMT Cache-Control: No-cache, no-store, must-revalidate, max-age=0 Transfer-Encoding: chunked	"DLI Controller"

JSON-RPC demo

JSON-RPC allows to access an object model similar to the one of the REST-like API, but in a different manner which may be more suitable for some integration environments. All composite objects are visible using JSON-RPC, with their field values accessible using "get" (with the field name in the argument) and "set" methods (with the field name and value as arguments). Additionally, containers support "add", "remove" and "list" methods. The "describe" method can be used to output a type description for the object (similar to the REST API "description" relative URI).

11.4 UPnP settings

UPnP outlet binding configuration 

#	Enable	Alternate name	Profile	Unique ID
1	☒	Outlet 1	belkin_wemo_socket ▾	b76cb5710
2	☒	Outlet 2	belkin_wemo_socket ▾	b76cb5711
3	☒	Outlet 3	belkin_wemo_socket ▾	b76cb5712
4	☒	Outlet 4	belkin_wemo_socket ▾	b76cb5713
5	☒	Outlet 5	belkin_wemo_socket ▾	b76cb5714
6	☒	Outlet 6	belkin_wemo_socket ▾	b76cb5715
7	☒	Outlet 7	belkin_wemo_socket ▾	b76cb5716
8	☒	Outlet 8	belkin_wemo_socket ▾	b76cb5717

UPnP settings

The unit's outlets can be exposed via UPnP as devices with different profiles. The currently supported profile is a Belkin WeMo socket.

11.5 SNMP settings

SNMP (simple network management protocol) exposes the control variables as a set of hierarchical resources identified by object identifiers (OIDs). An object identifier is roughly a sequence of non-negative integers (called arcs), separated by dots ('.'). A leading dot may be used to emphasize that it's an absolute OID; however, all of the OIDs configurable in EPCR6 are absolute unless otherwise stated explicitly, and the leading dot is not needed, therefore, it's not supported.

SNMP OID subtrees				
ID	Description	Root OID	Read security level	Write security level
energyObject	energyObject MIB (RFC	1.3.6.1.2.1.229	Authenticated and encrypted	Authenticated and encrypted

SNMP OID subtree properties

SNMP v3 introduces a user-based security model, where a number of different users can exist whose requests can be signed, and possibly encrypted, and who can have different access rights to the OID tree.

The specified root OIDs and their children will be exposed over SNMP. All OIDs must be absolute but not preceded by a dot.

The root OIDs are actually treated as masks, indicating to set of roots to apply the permission to. In addition to the standard OID syntax, all but the first two arcs of an OID mask may contain:

- an asterisk "*", which means that any value in this position will match, e.g. "1.2.*.1" will match both "1.2.1.1" and "1.2.100.1";
- a dash-delimited range, e.g. "1.2.8.1-3" will match both "1.2.8.1" and "1.2.8.2";
- a comma-separated list of arcs, possibly including ranges, e.g. "1.2.8,9" will match both "1.2.8" and "1.2.9", and "1.2.1,6-8" will match both "1.2.1" and "1.2.7".

This can be used to implement fine-grained access to states of individual outlets and buses (see below).

SNMP users					
Is allowed	Username	Authentication		Privacy	
☐	powerAdmin	SHA	Leave unchanged	AES	Leave unchanged
☒	powerReader	SHA	Leave unchanged	AES	Leave unchanged
☐	anotherUser	SHA	Leave unchanged	AES	Leave unchanged
☐		MD5	Leave unchanged	DES	Leave unchanged
Submit					
energyObject: Full root: Forbidden					
energyObject: Read-only root: Forbidden					
energyObject: Read-only root: Read-only					
energyObject: Forbidden root: Forbidden					

SNMP user table

The engine ID identifies the device, and plays an important role in SNMPv3, in particular in authentication and encryption. It will normally be autodetected by management software (SNMP clients), but you may save it for future reference (the default value is based on the device factory MAC address). You can even change it; however, if you do, all passwords for SNMPv3 users will be invalidated as they are stored in a localized form to improve security.

SNMP v1 and v2c do not have a notion of 'users'. Instead, a 'community string', acting as a shared secret, is transferred in requests in plain text. The following table allows to configure how community strings are mapped to the above users.

SNMP communities			
Community	IP address	Netmask	Mapped username
private	192.168.0.0	255.255.255.0	powerAdmin
public	192.168.0.0	255.255.255.0	powerReader
			powerAdmin

Submit

SNMP community-to-user mapping configuration

In this example, requests with the 'private' community string will be serviced as though they were made by the 'powerAdmin' user if they come from the 192.168.0.x subnet, and denied otherwise. Likewise, requests with the 'public' community string coming from the same subnet will be served as the 'powerReader' user.

11.6 SNMP energy object MIB support overview

The Net-SNMP agent included in EPCR6 has built-in support for several well-known MIBs, but none of them deal with power control. The power-control-related ENERGY-OBJECT-MIB is described in RFC 7460, and is supported in the following manner:

- the root of the OID tree is at 1.3.6.1.2.1.229 as per RFC;
- the objects are outlets, with indices starting at 1 , and buses, with indices starting at 257 ;
- the power consumed is indicated in the `eoPower` table (1.3.6.1.2.1.229.1.2.1.1) for buses only;
- the current actual power states are indicated in the `eoPowerOperState` (1.3.6.1.2.1.229.1.2.1.9) table;
- the assigned (expected) power states can be manipulated in the `eoPowerAdminState` (1.3.6.1.2.1.229.1.2.1.8) table;
- supported power states are `ieee1621Off` (257) and `ieee1621On` (259) only ; bus objects have `unknown` (255) power state as they are not directly controllable

Additionally, among others, the following potentially useful parts of the above MIB are implemented:

- `eoPowerStateTotalTime`;
- `eoPowerStateEnterCount`.

These accumulate outlet state statistics. Note that those don't persist across device reboots.

The following parts of the above MIB are NOT implemented:

- eoEnergyParametersTable;
- eoEnergyTable;
- eoMeterCapabilitiesTable.

The following related MIBs are NOT supported:

- ENTITY-MIB;
- ENERGY-OBJECT-CONTEXT-MIB.

Additionally, modifying the user permissions via SNMP is NOT supported as they are generated from the configuration described above and the process is not easily reversible.

In the default configuration, the security level for accessing the energy object MIB subtree is high. You can set the access level to 'Minimal' to interact with the device using SNMPv2c and SNMPv1, or use SNMPv3 instead, which is the recommended and more secure alternative.

11.7 SNMP sample commands

These examples assume you have your EPCR6 at 192.168.0.100 with SNMPv3 user `powerAdmin` configured with SHA1 for authentication and AES for encryption, with password `powerAdminPassword` for both authentication and encryption. Requests with the `private` community string are assumed to be serviced as though they were made by the `powerAdmin` user.

You'll need Net-SNMP to run these samples; analogous commands should be available for other management software. The matching of requests vs SNMP protocol version is really arbitrary and is only used to demonstrate different ways of performing requests. Lines are broken using `\\` for readability. We use `-On` to force numeric OID output, and omit the leading `'.'` in output OIDs for simplicity.

An SNMPv2 SET to turn outlet #3 on:

```
$ snmpset -On -v 2c -c private 192.168.0.100 1.3.6.1.2.1.229.1.2.1.8.3 i 259
```

Output:

```
1.3.6.1.2.1.229.1.2.1.8.3 = INTEGER: 259
```

An SNMPv3 GET to get outlet #5 status:

```
$ snmpget -On -v 3 -u powerAdmin -l authPriv -a SHA -x AES \
-A powerAdminPassword -X powerAdminPassword 192.168.0.100 \
1.3.6.1.2.1.229.1.2.1.9.5
```

Output:

```
1.3.6.1.2.1.229.1.2.1.9.5 = INTEGER: 257
```

257 is `ieee1621Off`, so now you know the outlet is physically off.

An SNMPv3 SET to turn outlet #5 on:

```
$ snmpset -On -v 3 -u powerAdmin -l authPriv -a SHA -x AES \
-A powerAdminPassword -X powerAdminPassword 192.168.0.100 \
1.3.6.1.2.1.229.1.2.1.8.5 i 259
```

Output:

```
1.3.6.1.2.1.229.1.2.1.8.5 = INTEGER: 259
```

Using SNMPv1 to enumerate the actual power states table:

```
$ snmpwalk -On -v 1 \
-c private \
192.168.0.100 \
1.3.6.1.2.1.229.1.2.1.9
```

Output:

```
1.3.6.1.2.1.229.1.2.1.9.1 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.2 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.3 = INTEGER: 259
1.3.6.1.2.1.229.1.2.1.9.4 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.5 = INTEGER: 259
1.3.6.1.2.1.229.1.2.1.9.6 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.7 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.8 = INTEGER: 257
1.3.6.1.2.1.229.1.2.1.9.257 = INTEGER: 255
1.3.6.1.2.1.229.1.2.1.9.258 = INTEGER: 255
```

You see that outlets 3 and 5 are on, and all others are off ; both buses show 255=unknown ; you can change outlet states as described above.

11.8 MQTT API

MQTT is an event-oriented protocol with a centralized publish/subscribe model, which makes it a bit awkward to use for controlling devices; however, an implementation is included due to its popularity.

11.8.1 MQTT client settings

MQTT client configuration	
Broker address:	192.168.0.5
Broker port:	1883
Use SSL:	☐
Username:	
Password:	
Topic root:	pcr12345
Last Will and Testament topic:	
Last Will and Testament payload:	
Last Will and Testament QoS:	At most once ▾
Submit	

General MQTT settings

EPCR6 can function as an MQTT client, so you need to have a configured MQTT broker which it could connect to; then, other MQTT clients connected to the same broker could communicate with it (multi-broker configurations are also possible but out of scope of this document). SSL and username/password authentication are supported (leave empty to disable). The default port is 8883 when SSL is enabled, and 1883 otherwise.

Setting and reporting outlet state are performed by means of MQTT messages. MQTT messages carry topic markers to identify their type. The topics are arranged in a '/'-separated hierarchy.

The 'Topic root' setting allows to prepend a common string to topics related to all outlets, e.g. to group messages related to the same controller. It is advisable that you set a different topic root for every controller that you connect to the same MQTT broker; otherwise, you'll get collisions unless you set different topic subtrees for their outlets.

Last Will and Testament functionality is included which allows the broker to notify you if the EPCR6 unexpectedly goes offline. Note that the topic of the message is not prefixed by the topic root.

11.8.2 MQTT outlet bindings

MQTT client outlet binding configuration 

#	Topic subtree	Allow read	Allow write	QoS
1	outlets/0	☒	☒	At least once ▾
2	outlets/1	☒	☒	At least once ▾
3	outlets/2	☒	☒	At least once ▾
4	outlets/3	☒	☒	At least once ▾
5	outlets/4	☒	☒	At least once ▾
6	outlets/5	☒	☒	At least once ▾
7	outlets/6	☒	☒	At least once ▾
8	outlets/7	☒	☒	At least once ▾

MQTT outlet bindings

Every outlet can be mapped to an MQTT topic subtree, which will be prefixed by the topic root and '/'; the outlet will report its state with a message with that topic if 'Allow read' is checked and honor requests to change its state if it receives a message with that topic suffixed by '/set' and 'Allow write' is checked. The quality of service determines the mode of delivery to request from the broker for messages on reporting outlet state; not all brokers may support all delivery modes; the default should be sufficient for most purposes.

Note that there is no explicit way to request the states of outlets. MQTT brokers are expected to keep track of the most recent payload of the topics.

11.8.3 MQTT payload formats

MQTT payload format is not defined by a standard, so we explicitly define it here. Everything that can be controlled via MQTT in EPCR6 is an outlet state, which can be ON or OFF. We encode ON as '1' (the single ASCII character '1') and OFF as '0'. For compatibility, in addition to decoding '1' as ON and '0' as OFF, we accept '\0' (the ASCII NUL character) and the strings 'off' and 'false' as OFF, and '\1' (the ASCII SOH character) and the strings 'on' and 'true' as ON. Additionally, the empty string "" is used to indicate topic erasure (on topic changes or read access revocation).

11.8.4 MQTT sample commands

These examples assume you have an MQTT broker at 192.168.0.5, it doesn't require authentication and you have the EPCR6 set up like shown on screenshots above.

You'll need the mosquitto MQTT client to run these samples; analogous commands should be available for other clients. mosquitto also has an MQTT broker implementation.

If you run the command:

```
mosquitto_sub -h 192.168.0.5 -C 1 -t pcr12345/outlets/0
```

it'll print the current state of the first outlet as known to the broker (as '0' or '1') and exit. The '-C 1' flags disable waiting for more state changes; if you run the command with them removed:

```
mosquitto_sub -h 192.168.0.5 -t pcr12345/outlets/0
```

you'll see the current state, but the program will wait for more state messages and print the states as they arrive; if you flip the outlet, you'll see output like this:

```
0
1
0
1
```

etc.

To change the state of the first outlet, use the mosquitto_pub command, e.g. to switch it on use:

```
mosquitto_pub -h 192.168.0.5 -t pcr12345/outlets/0/set -m 1
```

and to switch it off use:

```
mosquitto_pub -h 192.168.0.5 -t pcr12345/outlets/0/set -m 0
```

As mentioned [above](#), alternate value forms, e.g.

```
mosquitto_pub -h 192.168.0.5 -t pcr12345/outlets/0/set -m true
mosquitto_pub -h 192.168.0.5 -t pcr12345/outlets/0/set -m on
```

will also work as expected.

12 Backing up settings

The setting backup/restore system operates on a file level. It allows saving and restoring most configuration items, including those which have been done manually, e.g. via SSH. Settings can be backed up and restored selectively.

Download backup

Select objects to back up:

...

☐ Authentication configuration *(can contain sensitive data)*

☒ AutoPing configuration

...

☒ HTTP server configuration

...

Download

Backup setting selection (sample)

Modified files are highlighted in green. Choose the setting files you want to save (unknown files are shown as "File "+filename) and click "Download".

You may protect the security-sensitive parts of the configuration from being stored in a backup. Pressing the hardware reset button will be required to unlock.

13 Firmware upgrade

The controller's firmware can be upgraded to a newer version by first uploading it, and then committing the upgrade.

13.1 Uploading the firmware

Upload new firmware		
Firmware file:	Choose File No file selected	
	OR	Upload
Firmware URL:		

Firmware upload page

You can upload a file or specify a URI where it can be downloaded from. Be sure to disable the [same subnet restriction](#) if you intend to download firmware from a server not in your local network.

By default, the unit beeps and blinks during update. You can disable this using the following form.

Configure update process	
☐	Beep during upgrade
☐	Blink during upgrade
Save	

Firmware update configuration page

You may protect the current firmware from modification. Pressing the hardware reset button will be required to unlock.

13.2 Committing the firmware upgrade

After the firmware has been uploaded, you are presented with a form to perform the upgrade.

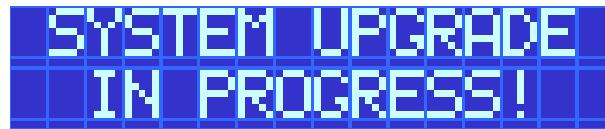
Firmware check OK
Upgrading from 1.7.1.0 to 1.7.2.0.

Perform firmware update	
Update	Cancel

Firmware upgrade page

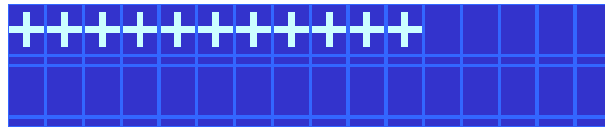
You can double-check the firmware version. If you wish to cancel the upgrade, it's best to explicitly press 'cancel' in the form so that the uploaded file could be cleaned up.

When you start an upgrade, a message about this briefly appears on the LCD.



Firmware upgrade start indication

Further activity is indicated by a cycling pattern of  signs.



Firmware upgrade progress indication

After an upgrade has been completed, the unit is rebooted. The first boot may take longer than usual due to configuration updates.



Post-firmware upgrade initialization indication

14 Date/time

The Date / Time page allows the administrator to set the internal clock and time zone. The clock may be set within the browser or synchronized with an NTP server.

Date and Time

Date

6

/

15

/

2016

(MM/DD/YYYY)

Time

14

:

1

:

57

(HH:MM:SS)

Submit

Sync with Computer Clock

Date/time

The default OpenWrt configuration is to use the NTP servers `(0|1|2|3).openwrt.pool.ntp.org`. See [Scripting](#) scripting to perform manual synchronization.

The preferred time zone can be customized as well.

Time Zone

UTC+3

Submit

Timezone

The selected timezone is used for header date/time display and formatting time in plots.

Internally, the time zone is stored in a format different from the display (it has a different meaning for + and -). You should take that into account when interpreting related REST API values and log output.

15 AutoPing

AutoPing can monitor a network device and perform a task if the device stops responding. It can also monitor a group of devices, the task will be executed if none of the group members respond. The task is either a list of outlets to reboot or a script to execute.

15.1 Common configuration

AutoPing Properties		
Enable AutoPing:	☒	
Time between pings:	30	seconds. (2-3600)
Ping timeout to reboot:	150	seconds. (2-3600)
Ping responses to enable autoping:	5	pings. (0-100)
Times to attempt reboot:	5	tries. (1-255)
Device reboot delay:	120	seconds. (1-43200)

Common AutoPing settings

Be sure to enable AutoPing operation by ticking the "Enable AutoPing" checkbox. Certain [reset procedures](#) may turn it off automatically.

The following parameters are used for AutoPing operation:

- **Time between pings:** This is the time between each ping check of an address. 60 seconds should be useful for most applications.
- **Ping timeout to reboot:** This sets the maximum time that sequential communication attempts may fail. Any failure beyond this time limit will cause the task to be executed. For example, when set to 300 seconds and a time between pings is 30 seconds, if a target system fails to any pings for 330 seconds, the task will be executed. The ping that occurred after 300 seconds came at 330 seconds and still failed. Since occasional network overloads and missed packets can occur during normal network operation, be sure to choose a reasonable time. AutoPing may handle certain failures immediately instead of waiting for the timeout if configured to (see below).
- **Ping responses to enable autoping:** To ensure a reliable connection, autoping will only be enabled after this many successful pings. We do not recommend changing this (10 is default) unless you must configure your controller before connecting it to the target devices.
- **Times to attempt reboot:** If you have an unreliable target device, limit the number of times it will be rebooted by entering that value here. For example, entering 5 will execute the task up to 5 times before giving up.
- **Device reboot delay:** After rebooting a device with a cold-boot power-off, a waiting period should occur before the IP address is re-checked by AutoPing. This delay allows the device to reboot. Windows and Linux servers can force automatic file system checks which may take several minutes to complete. Enter a safe value here, for example entering 600 would cause the power controller to start checking the server for normal operation 10 minutes after reboot. If a script is to be triggered, any delays contained in the code being executed should be considered in determining the delay setting here so that the thread completes before the delay elapses. This timer starts at the execution of the thread started.

- Handle failures immediately instead of waiting for timeout: Enabling this feature may make sense for handling certain AutoPing target types which may return an explicit error (TCP RST, HTTP 500, etc.) by invoking the task immediately instead of waiting for the timeout to pass (during which the error condition could have disappeared and no action would have been taken). Consider the setup and AutoPing action when enabling this option (e.g. you shouldn't enable it if the AutoPing action is to power cycle a server, you need to shut one of its services down temporarily for maintenance and it's the only target of the AutoPing entry).
- Activate enabled entries without trial on service restoration: By default enabled entries still need to wait for a certain number of successful ping responses on initial power-up before AutoPing actions are taken to make sure the targets have come online as well (in the assumption that they might have suffered a power failure as well and may need time to recover). This option can be used to disable this additional check.

15.2 Ping target configuration

To actually use AutoPing, add one or more AutoPing targets (IP addresses) to the list. The  button is used to remove a target from the list.

Below is an example autoping configuration with four targets:

AutoPing														
IP(s)		Reboot Outlets								Script	Action	Stats		
		1	2	3	4	5	6	7	8			TX	RX	HIT
☐	74.125.87.103											790	578	
☒	67.122.199.250	☒	☐	☒	☐	☐	☒	☐	☐	switch_off		642	583	2
												0	0	
☐	192.168.0.92	☐	☐	☒	☐	☒	☒	☐	☐	[Cycle]		215	41	5
												0	0	
☒	192.168.0.93	☐	☐	☐	☐	☐	☐	☐	☐	toggle_stuff_and_log		823	822	0
												0	0	
☐		☐	☐	☐	☐	☐	☐	☐	☐	[Cycle]		0	0	0

Individual AutoPing settings

The checkbox to the left of the IP address is used to start/stop target monitoring. Confirm your action with  button. This button is also used to link a list of outlets or a script line to the autoping target.

You can select the outlets to perform trigger action on by ticking their respective checkboxes.

You can select a scripting action to perform when the AutoPing item triggers (by default the selected outlets are cycled). The action must be a function defined in the scripting server, like

```
function action_to_perform()
.
.
.
end
```

or e.g.

```
function action_to_perform(selected_outlets)
    .
    .
    .
end
```

In the second form the argument `selected_outlets` (any other name will do) will receive a table of the 1-based indices of outlets selected (e.g. `{1, 3, 6}`). The order of outlets in the table is unspecified; use `table.sort` in the script function if you rely on a particular order.

The stats column shows some statistics:

- *TX* — the number of pings sent to the target IP address;
- *RX* — the number of pongs received back so far;
- *HIT* — the number of times the trigger action was executed.

On the sample image, three targets are being monitored (74.125.87.103, 67.122.199.250, and 192.168.0.93). 192.168.0.93 seems to be a very reliable/well-connected device: 823 pings were sent to it and 822 pongs received back. Chances are very good, the 823rd pong will arrive soon. The reboot task (script function `toggle_stuff_and_log`) was never executed.

Looks like 192.168.0.92 failed hard. The task (cycle outlets 3,5,6) was executed 5 times in a row but the target did not respond. Monitoring was automatically disabled.

74.125.87.103 and 67.122.199.250 form a group, the trigger task will be performed if they both lose 5 sequential packets simultaneously. This has happened 2 times so far. Monitoring a group of several external spatially separated reliable IP addresses (in this example they belong to Google and Digital Loggers respectively) may become very useful to detect a stuck ADSL modem or some other no-Internet condition.

15.3 Action on local network failures

AutoPing is designed to control operation of remote hosts. You usually don't want to e.g. cycle power to all servers if you turn on same subnet restriction. So AutoPing tries not to trigger if there might be a problem local to the unit itself. For example, if you detach the Ethernet cable from the unit, you'll see messages similar to the following:

```
kernel: eth0: link down
config.net: Interface "eth0" is down
autoping: ping x.y.z.t: no usable route to host, ..., not considered a failure
```

and no actions will be performed. A similar situation will occur if you reconfigure the controller to use a new IP network from which old addresses are unreachable.

Use the `link://` [scheme](#) to check for local link loss.

15.4 Advanced ping targets

AutoPing targets don't have to be IP addresses. If you enter a hostname, it will be resolved before sending each request. If the name resolution fails, it is assumed to be a local error and, as described [above](#), no action is taken. If a name is resolved to multiple IP addresses, a random one is chosen.

AutoPing defaults to checking targets using the ICMP protocol by default. A variety of other ping target kinds can be used if you specify a URL instead of simply an IP address or hostname. Supported URL schemes include:

- `icmp` — this is explicit specification of the "regular" ping protocol, e.g. `icmp://192.168.0.1` is equivalent to `192.168.0.1` (note that no trailing slash is used);
- `link` — this allows to check if the physical link is present on the wired (`link://eth0`) or wireless (`link://wlan0`) interface (which is useful as higher-level targets will usually [ignore](#) link loss);
- `tcp` — this causes AutoPing to try to establish a TCP connection to the given port, e.g. `tcp://192.168.0.1:22` can be used to check that there's a service listening on TCP port 22 (usually SSH) of `192.168.0.1` (note that no trailing slash is used);
- `http` and `https` — this causes AutoPing to perform a HTTP/HTTPS GET request for the given URL, e.g. `http://www.digital-loggers.com/index.html` can be used to check that the web server is responding and can serve its main page.

15.5 AutoPing events

The most often encountered AutoPing events are:

- `pinging ... (timeout)`
- `ping ... succeeded (time)`
- `ping ... failed (time)`

The time is request round-trip time, in seconds. Note that it's purely informative and can't be used as a measure of target response time unless it has order of hundreds of milliseconds and above.

Several failures in a row trigger AutoPing actions which are reported with corresponding events:

- `item ... (addresses...) failed [failures/max]`
- `item ... (addresses...) failed over (max) times in a row, disabling`

As described [above](#), local network failures don't count toward failure count, but generate these notifications instead:

- `no usable route to host, possibly due to local network outage, not considered a failure (when a request isn't being sent)`
- `ping ... not received (time), possibly due to local network outage, not considered a failure (when an outage occurred after a request has been sent)`

The events associated with item trial before enabling are self-explanatory:

- `item ... (addresses...) enable approved`
- `item ... (addresses...) enable cancelled`
- `item ... (addresses...) trial restarted due to address list changes`

16 Energy monitor

The energy monitor page displays values of measurements in different forms. The AC input current on each bus, AC input voltage, battery voltage, and CPU / Relay voltages are metered. A thermistor monitors CPU and control board temperature, which should not exceed 185F. The 5V meter monitors the control board VCC, the 12V meter indicates control relay power, and battery voltage can be used as a discharge indication.

Batteries are fully charged at 5.60V and 90% discharged at 4.60V. Note that if you detach the battery, the reading will be at or above 5.60V, so you should make sure the battery is in fact connected before relying on this indicator.

The image format can be configured on the [setup page](#).

Bus	Voltage, V	Current, A	Total energy, kWh	Reset energy meter
Bus A	120.0	0.0	0.0	Reset
Bus B	120.0	0.0	0.0	Reset
Temperature, F	69.5			
Illuminance, lx	0.1			
Power voltage, V	5.3			
Battery voltage, V	5.7			
Relay voltage, V	13.9			

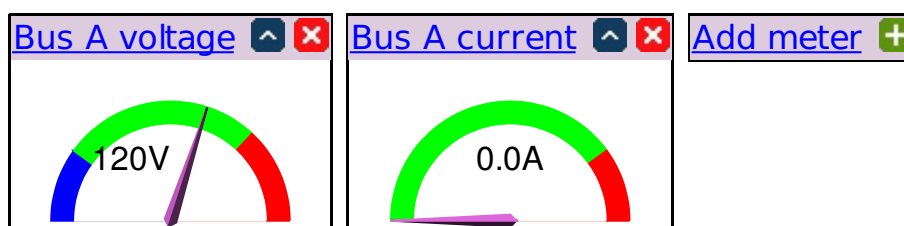
Currently measured values

The rest of the page is occupied by user-configurable meters and plots. You can use the  icon to add a new meter or plot, or the  icon to delete an existing one.

Use the  /  icon to toggle visibility of a plot or meter. The setting persists (unlike visibility of configuration blocks).

Plots and meters can be configured in detail by clicking on them.

16.1 Meters



Meters

Clicking on a meter allows to configure it.

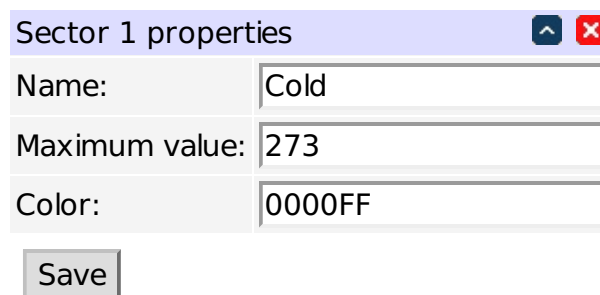
Meter properties	
Title:	Temperature
Data source:	Temperature
Minimal value:	244
Width:	128
Height:	64
Pointer color:	
Scale:	linear
Number of decimal places:	0
Save	

Basic meter configuration

The basic meter properties include:

- the title;
- the width in pixels;
- the height in pixels;
- the main color of the pointer, in hex web notation without '#';
- the minimum (leftmost) value to display;
- the gauge's scale (linear or logarithmic);
- the number of digits to display after decimal point;
- the data source.

A meter should have one or more sectors, which define different ranges of the value to indicate. The end of one sector's range is the beginning of the next one's range; for the start of the first sector's range, the "minimum value to display" setting is used.



Sector 1 properties	
Name:	Cold
Maximum value:	273
Color:	0000FF

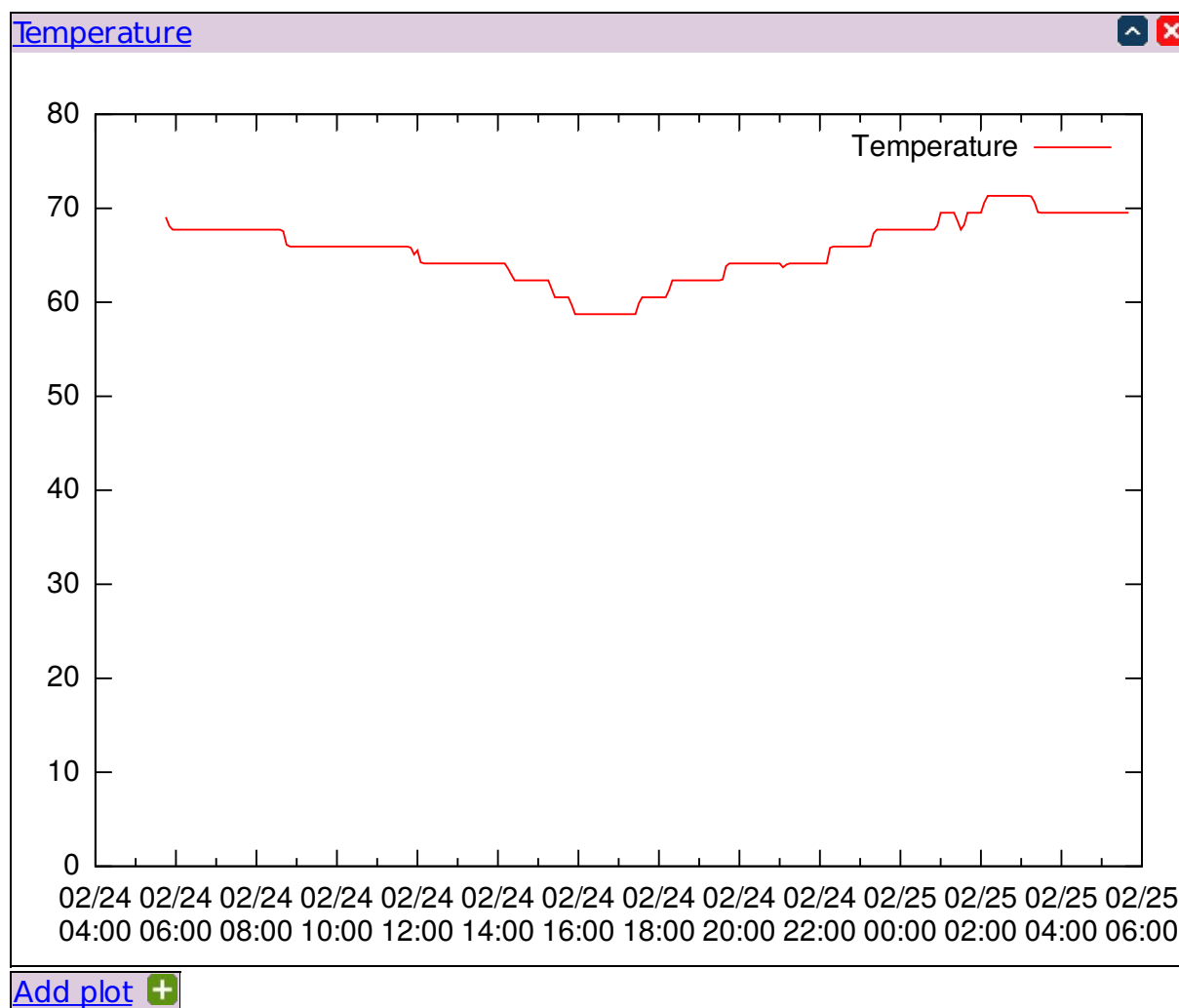
Save

Meter sector configuration

The sector properties include:

- the name of the sector (currently unused);
- the maximum value for the sector (and also the minimum value for the next sector);
- the sector color, in hex web notation without '#'.

16.2 Plots



Plots

Clicking on a plot allows to configure it.

Plot properties	
Title:	Bus A status
Width:	640
Height:	480
Save	

Basic plot configuration

The basic meter properties include:

- the title;
- the width in pixels;
- the height in pixels.

The rest of meter configuration is centered around axes and plot lines which are drawn on them.

There are 4 axes: x_1 and y_1 are the usual ones; x_2 is at the top, and y_2 is at the right side.

Axis y_1 properties	
Minimum value:	0
Maximum value:	
Soft minimum value:	
Soft maximum value:	
Major tic interval:	
Minor tics per major tic:	

Save

Configuration of an axis

Each axis has the following properties:

- minimum/maximum values: these specify the exact plotting value range;
- soft minimum/maximum values: these specify a range with limits which can be exceeded;
- major tic interval/minor tics per major tic: configure the axis tic behaviour;

The interaction between hard and soft limits is as follows:

- if no limit is specified, the plotting range is determined by the data to be plotted;
- if only a hard limit is specified, it's obeyed;
- if only a soft limit is specified, it's obeyed unless data exceed it, then data are obeyed;
- if both limits are specified, the soft limit (which should be less strict than the hard limit) is obeyed unless data exceeds it, then data are obeyed unless they exceed the hard limit (which should be stricter than the soft limit), then the hard limit is obeyed.

Several plot lines can be drawn on the same plot (they don't have to be proper lines but may have various shapes, see below).

Data line 1 properties
⬆️ ✖️

Title:	Bus A voltage
Data source:	Bus A voltage ▾
Color:	FF0000
Line kind:	Lines only ▾
Line type:	Solid ▾
Line width:	1
Marker type:	Plus ▾
Marker size:	0
Base axes:	X1, Y1 ▾

Save

Configuration of a plot line

Each plot line can be one of the following styles:

- lines only;
- markers only;
- lines and markers;
- small dots;
- step lines;
- boxes;
- spline smoothed lines;
- approximated spline smoothed lines;
- Bezier smoothed lines;
- vertical lines.

Solid/dotted/dashed lines are supported. Data points can be displayed using a variety of markers:

- empty/filled circle;
- empty/filled square;
- empty/filled triangle;
- empty/filled inverse triangle;
- empty/filled rhombus;
- plus;
- dot;
- cross.

Line width and marker size can be configured. Colors have to be specified in hex web notation without '#'.

Each plot line is plotted on a combination of x and y axes. Plot lines which are plotted against the same axis must have matching measurement units along that axis; e.g. if you want voltage and current on the same plot, they need to be plotted against different y axes but can share the same x axis (time).

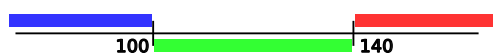
16.3 Alarms

Every measured value can trigger 'alarms' when it is in a certain value interval; those alarms can be made periodic. Alarms carry a 'level', which is an administratively assigned number; the range of these numbers is entirely up to the user; it may be used in [notification rule](#) conditions (e.g. 'value_alarm_level>2'); it's intended that alarm levels for different values correspond to the associated severity/risk value so that the notification rules can be made generic (and independent of concrete values).

Configuration of meter value alarms is done not on the notification page, but on the energy monitor page. You can configure a meter value's alarms by clicking the  icon next to it. This will open the interval alert configuration page for it.

Every measured value is a just a number, or 'none' for some meters on some devices which are detachable, so it makes sense to say there's no value.

The real line of the possible values can be partitioned into intervals using threshold points: e.g. voltage can be below 100V, between 100V and 140V or above 140V, so there are two points, 100V and 140V, splitting the real line into 3 intervals. The idea is similar to [configuring sectors of a meter's visual appearance](#).



Naive interval configuration

Each of the intervals can be assigned an alarm level, which can then be analyzed by notification server conditions (you can also 'none' if you explicitly wish to send no events for that interval).

It is often desirable to keep some level of hysteresis between two adjacent intervals, so that there is some threshold the value needs to cross before we consider that it has left one interval and entered a different one, to avoid needless notifications. This is accomplished by further splitting the threshold points into the top and bottom values. If a value increases and crosses the interval boundary, we only consider that it has switched intervals after it's above the top value. Likewise, if a value decreases and crosses the interval boundary, we only consider that it has switched intervals after it's below the bottom value. E.g. you could use 95V and 105V for the bottom and top values of the 100V threshold point in the previous examples for a 10V hysteresis.



Interval configuration with hysteresis

Each interval is characterized by its lower threshold point (with its bottom and top values), its alarm level and period. The exception is the lowest interval which has no threshold point (but still has a level and a period). The "none" value is considered so distinct from all regular values that no hysteresis is possible, so it is also characterized by a level and a period only.

The above sample configuration might look like this:

Interval alert configuration for Bus A voltage

Data absence alarm level:	None
Data absence alarm period:	None
Lowest interval alarm level:	1
Lowest interval alarm period:	60

Interval 1 properties

Lower bound bottom:	95
Lower bound top:	105
Alarm level:	None
Alarm period:	None

Interval 2 properties

Lower bound bottom:	135
Lower bound top:	145
Alarm level:	2
Alarm period:	20

Add interval

Lower bound bottom:	None
Lower bound top:	None
Alarm level:	None
Alarm period:	None

Sample interval alert configuration

Here, we trigger a level 1 alarm every minute if the voltage is below 100V and a level 2 alarm every 20 seconds if the value is above 140V.

17 Safety shutdown

Over-current, over-voltage, and low-voltage shutdown are provided on the safety shutdown page.

Safety Shutdown
⌵

Over- and Undervoltage	Switch OFF	☒ Outlet 1	☒ Outlet 2	☒ Outlet 3	☒ Outlet 4	when Bus A voltage exceeds		V	or goes under	60	V
	Switch OFF	☒ Outlet 5	☒ Outlet 6	☒ Outlet 7	☒ Outlet 8	when Bus B voltage exceeds		V	or goes under	60	V
Overcurrent	Switch OFF	☐ Outlet 1	☐ Outlet 2	☐ Outlet 3	☐ Outlet 4	when Bus A current exceeds	14	A			
	Switch OFF	☐ Outlet 5	☐ Outlet 6	☐ Outlet 7	☐ Outlet 8	when Bus B current exceeds	14	A			

Behaviour on limit hit: ☐ Hold outlets **OFF** for seconds
☒ Latch outlets **OFF** until manual relatch

Safety shutdown settings

When one of the limits is violated, the unit will shut off the selected outlets. When the error condition is raised, the unit may either:

- hold those outlets off for a configurable amount of time, or
- keep those outlets off until manual intervention (toggling via [keypad](#)).

Keeping undervoltage shutdown enabled is essential to correct partial power loss detection and handling. Without it enabled, the unit won't be able to detect and report loss of power on one of the buses.

Take care when using this feature so as not to create an oscillating condition.

18 Generic input/output

The EPCR6 has user-defined input/output facilities which can be used from the web UI, REST API or user scripts. The "Input/Output" main menu item leads you to the page allowing you to control the pins and ports (if they're present on your power controller model).

18.1 I/O ports

I/O ports			
ID	Name	Configuration	Exchange
uart_ext	uart_ext ☒	Baud rate 115200 Character size in bits 8 Parity setting N Number of stop bits 1 Reception mask ☒ Data ☒	Console XTerm

I/O port configuration

An I/O port is one or more communication channels sharing a common configuration structure. A typical example is a UART.

You can assign a descriptive name to ports. You can also choose which channels, if any, you want to receive data from. To send and receive data in the web UI, follow the "Console" (or "XTerm" if available for the I/O port type) links.

Session expires in 00:29:39

Welcome to

DLI BA

generic I/O

000A

☐ Hex ☒ CRLF

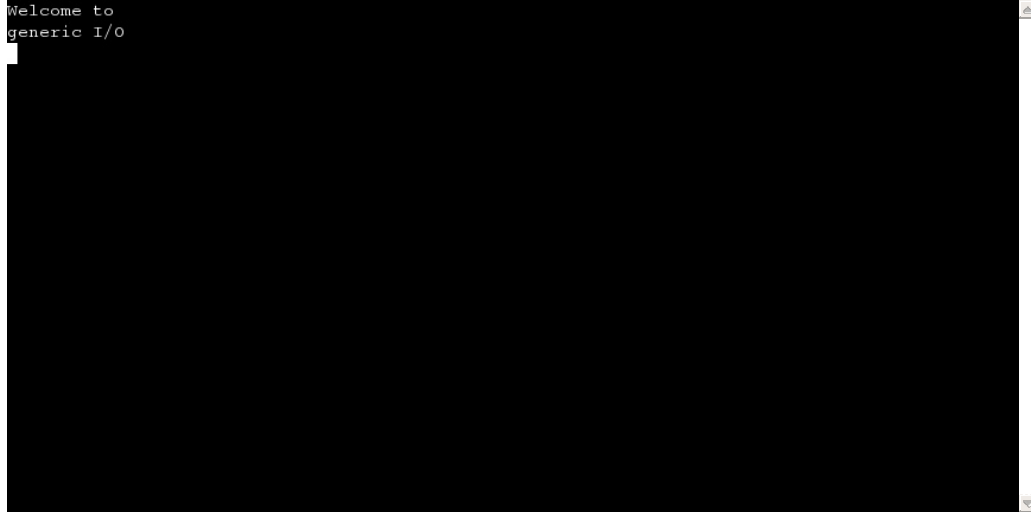
Send

I/O port console

The console interface is more generic and applicable to all I/O port types. It allows you to send textual data or binary sequences encoded in hex and optionally followed by CR+LF (0D 0A). It displays received ASCII data as text and non-ASCII data as hex sequences. You can hover a received message to see the arrival timestamp.

Session expires in 00:29:37

Welcome to
generic I/O



I/O port terminal

The terminal window is applicable to UART ports only. It displays an xterm-compatible terminal window and allows you to interact with it.

Ports are also available to [scripting](#).

18.2 I/O pins and nets

The EPCR6 has pins with customizable behaviour which may be connected by nets to perform real-time operations.

18.2.1 I/O pin configuration

I/O pins				
ID	Name	Properties	Drivers	
out1	Open collector output ✓	Bits Input 0 Mode 1 Level 0 M L Low High Tristate Z Z Active 0 0	Level: Low ▾ Mode: Tristate ▾ ✓	
out2	Regular output ✓	Bits Input 0 Mode 0 Level 1 M L Low High Tristate 0 1 Active 0 1	Level: High ▾ Mode: Active ▾ ✓	

I/O pin configuration

I/O pins have mode and level 'driver' parameters. Those dictate how the pin's active/tristate mode and high/low level are calculated. Some of the drivers can be locked by the hardware (e.g. open collector outputs have their level fixed at 'low', you can only switch the mode from tristate to active) so you won't be able to change them. Otherwise, they can be set to Active (High for level) or Tristate (Low for level) or to a name of a signal net (see below); then the pin's characteristic will depend on the value of the named net's value.

The input/mode/level signal bits table in pin properties indicates pin's capabilities. If input bit count is 0, the pin is output-only so it makes no sense trying to read it. If mode or level bit counts are 0, the corresponding configuration value is locked by the hardware and cannot be modified.

Usually, pins in Tristate High state are pulled up; pins in Tristate Low state aren't. Pin properties give a summary of what configurations the pin can assume, i.e. how the mode and level drivers are mapped to IEEE1164 signal states '0' (Forcing 0), '1' (Forcing 1), 'L' (Weak 0), 'H' (Weak 1) and 'Z' (High impedance).

18.2.2 Signal net configuration

I/O signal nets				
ID	Name	Configuration	Value	Actions
net1	New net	Expression: Reporting: None ▾	0	+

I/O signal net configuration

Nets are what you use when you want to have more complex pin behaviour (and also to read input pin levels as a useful side effect). A net has a value which can be assigned manually or computed using pin input values and values of nets in the previous cycle. Pins' levels and modes can be controlled by nets, with built-in transition safety (i.e. switching from pullup to active low and back will avoid the active high state). You may also receive change notifications for signal nets in [user scripts](#).

A net's name can consist of alphanumeric, the underscore '_' and the pound sign '#' (useful for marking inverted logic).

Net names starting with the double underscore ('__') are reserved and should not be referenced or modified by users.

Net expressions are essentially characteristic equations for the nets in question.

Net values, and intermediate values appearing in a net expression, are 32-bit integers. They are interpreted by arithmetic and comparison operations to be signed in two's complement notation.

Net expression syntax is a subset of Lua syntax with the following limitations stemming from the nature of signal net definition which has to do with manipulating multibit signals:

- the whole net is described by a single expression, without any control structures;
- defining functions is not supported;
- creating tables is not supported;
- the standard library is not supported;
- 'and' and 'or' operators do not have shortcut semantics: both operands are always evaluated.

The following details concern arithmetic operators and are intended to make operation fully defined and predictable as well as compatible with Lua and expected behaviour:

- saturation arithmetic: overflow in arithmetic (but not e.g. shift) operations results in maximum/minimum representable value;
- division rounds the result down (toward negative infinity, e.g. $1/2 = 0$ and $-1/2 = -1$);
- modulus sign matches divisor sign;
- division or modulus with zero divisor are tolerated; $0/0 = 0$, otherwise division by zero results in maximum/minimum representable value; $a\%0 = 0$.

The API is comprised of literals `nil` and `false` (corresponding to the integer value 0) and `true` (corresponding to the integer value 1) and the following operators and globals:

Operator	Global	Evaluates to
x and y	land(x,y)	logical AND of x and y
x or y	lor(x,y)	logical OR of x and y
	lxor(x,y)	logical XOR of x and y
	band(x,y)	bitwise AND of x and y
	bor(x,y)	bitwise OR of x and y
	bxor(x,y)	bitwise XOR of x and y
	shl(x,y)	x, shifted left by y bits
	shr(x,y)	x, shifted right by y bits
x + y	add(x,y)	x + y

<code>x - y</code>	<code>sub(x,y)</code>	<code>x - y</code>
<code>x * y</code>	<code>mul(x,y)</code>	<code>x * y</code>
<code>x / y</code>	<code>div(x,y)</code>	<code>x / y</code>
<code>x % y</code>	<code>mod(x,y)</code>	<code>x % y</code>
<code>x == y</code>	<code>eq(x,y)</code>	1 if <code>x == y</code> and 0 otherwise
<code>x < y</code>	<code>lt(x,y)</code>	1 if <code>x < y</code> and 0 otherwise
<code>x > y</code>	<code>gt(x,y)</code>	1 if <code>x > y</code> and 0 otherwise
<code>x <= y</code>	<code>le(x,y)</code>	1 if <code>x <= y</code> and 0 otherwise
<code>x >= y</code>	<code>ge(x,y)</code>	1 if <code>x >= y</code> and 0 otherwise
<code>not x</code>	<code>lnot(x)</code>	1 if <code>x==0</code> and 0 otherwise
	<code>bnot(x)</code>	bitwise NOT of <code>x</code> (all bits inverted)
	<code>neg(x)</code>	0-s complement of <code>x</code> (all bits inverted+1)
	<code>pin[name]</code>	the value of the named pin (name must be a string constant)
	<code>net[name]</code>	the value of the named net (name must be a string constant)
	<code>apin[name]</code>	the value of the named pin interpreted as an analog signal (see below)
	<code>anet[name]</code>	the value of the named net interpreted as an analog signal (see below)

So, every net can reference pins (reading their input values), or nets (including itself). It is not a syntax error to reference a nonexistent net (otherwise complex schemes wouldn't be possible to enter a net at a time) or a nonexistent or unreadable pin, but such references return the value 0 (until, in the case of a net, it appears).

Standard Lua syntax rules apply to `net [ ]`, etc. indexing operations, so you can address net `q1` as `net [ "q1" ]` or `net .q1`, but net `#R` only as `net [ "#R" ]`.

Every net output is latched, that is, it is stored in a memory area on every step. When a net's value is read to compute another net's value, the previous step's value is used, so e.g. an expression '`not net.Q`' for net `Q` makes sense, and produces a rapidly oscillating output. On creation, all net values are initialized with the value 0.

This latching behaviour allows for edge detection, e.g. if you have net `Q` you can create a helper net `Q_` which copies it and introduces a one-step delay as a side effect:

```
Q_: net.Q
```

and add edge-detecting nets:

```
Q_rise: net.Q and not net["Q_"]
```

```
Q_fall: not net.Q and net["Q_"]
```

Due to this latching behaviour, situations which have a diminishing probability in real-life circuits can happen with certainty in emulated circuits. For instance, consider that you already have nets `#R` and `#S` and want to produce an SR latch on two NAND logical elements. If you naively follow the classic schematic and create the two nets:

```
Q: not (net["#S"] and net["#Q"])
#Q: not (net["#R"] and net["Q"])
```

at once, the following will happen (considering `#R` and `#S` are both 1, i.e. hold state):

- both `Q` and `#Q` start out with value 0;
- on the next step, both `Q` and `#Q` assume the value 1 because `not (1 and 0) = 1`;
- on the next step, both `Q` and `#Q` assume the value 0 because `not (1 and 1) = 0`;

- etc.

The result is not state being held, but an oscillation! If you have configured the nets in question for change reporting, the EPCR6 may become slow to respond. In a real device, the signal propagation times would never be exactly equal so the situation would be resolved into one of the stable states, but this is not so in emulation.

One possible advice on avoiding such situations is to use the minimum number of nets. E.g. a similar latch could be produced by the following net (replacing `net["#Q"]` in the expression for `Q` with the expression for `#Q`):

```
Q: not (net["#S"] and not (net["#R"] and net["Q"]))
```

or, equivalently,

```
Q: (net["#R"] and net["Q"]) or not net["#S"]
```

It has a single positive feedback loop and doesn't exhibit the above problem. It also explicitly makes `#S` dominate `#R` (in case of conflict, `Q` is 1). If you want `#R` to dominate, all you need is move the parentheses:

```
Q: net["#R"] and (net["Q"] or not net["#S"])
```

18.2.3 Analog signal handling

Analog signals (e.g. ones read from an ADC channel or an external sensor) are commonly presented in a fixed-point format as signed two's complement integers with the upper half storing the integer part, and the lower half storing the fractional part. The values of the readings match those in e.g. `scripting meter.values[key].value`, that is, they are the physical values measured by the channel or sensor, expressed in SI units (e.g. volts for voltage, degrees Kelvin for temperature, etc.). To handle pin or net signals in this format, reference them using `apin[]` or `anet[]` instead of `pin[]` and `net[]`, respectively.

Analog signal representation matters when analog signals are operands of arithmetic or comparison operators. To ensure proper operation, it's required to use the `apin[]/anet[]` ('a' for 'analog') symbols to address the pins or nets you want to treat as such.

You may get a warning if you address a pin or net incorrectly (using e.g. `pin` instead of `apin` or vice versa), but otherwise this distinction is not enforced, since you may have reasons to reinterpret a value.

Additionally, analog pins represent Not-a-Number (in turn representing `readings` from a disconnected sensor) as all-zeros (binary value 0). It is false in a boolean context, which simplifies net coding. All other values are nonzero; zero is mapped to the binary value 1, corresponding to the epsilon value, below error margin of most sensors. The value is, however, not handled specially by any built-in functions or operators; any special handling must be implemented manually.

19 System log

Most events occurring during controller operation are logged into the system log.

```
System Log:
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: compile time options: IPv6 GNU-getopt no-DBus no-i18n no-IDN DHCP no-DHCPv6 no-Lua TFTP no-contrack no-ipset no-auth no-DNSSEC no-ID loop-detect inotify
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq-dhcp[4061]: DHCP, IP range 192.168.254.200 -- 192.168.254.248, lease time 12h
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: using local addresses only for domain lan
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: reading /tmp/resolv.conf.auto
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: using local addresses only for domain lan
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: using nameserver 192.168.0.1#53
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: read /etc/hosts - 2 addresses
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq[4061]: read /tmp/hosts/dhcp - 1 addresses
Fri Jan 1 00:02:31 2018 daemon.info dnsmasq-dhcp[4061]: read /etc/ethers - 0 addresses
Fri Jan 1 00:02:32 2018 user.notice firewall: Reloading firewall due to ifup of wan (wlan0)
Fri Jan 1 00:02:34 2018 user.notice upnp[4136]: Starting...
Fri Jan 1 00:02:39 2018 user.notice www[4291]: Starting...
Fri Jan 1 00:02:46 2018 daemon.info procd: - init complete -
```

System log

Note that the system log buffer has a fixed size, and old entries are removed automatically as new ones appear. The display is periodically updated.

You can use the `logread` command to read the same data in an SSH session. `logread -f` will display new entries in real time.

20 Locking down the controller

20.1 Intended locking use cases

In some cases it's required to grant administrative access to multiple, possibly not completely trusted, parties. The settings described below are designed to somewhat limit what an administrator can do to the device.

20.2 Protection bits

The following operations available to the administrator pose an increased risk and can be protected:

- changing administrator credentials;
- changing networking settings;
- changing notification settings;
- backing up private settings (passwords, keys, etc.);
- restoring settings from backup;
- upgrading firmware;
- entering maintenance mode.

20.3 Protection status indication

Overall protection status is indicated in the top right corner of [each page](#):

-  — no protection bits active;
-  — some protection bits active;
-  — all protection bits active.

Clicking on the icon gives more detailed info:

Firmware protection is disabled
Notification settings protection is disabled
Private configuration protection is enabled but ineffective: SSH is enabled, which could be used to bypass protection Firmware upload is allowed, specially crafted firmware could be used to bypass protection
Administrator credentials protection is disabled
Maintenance mode lock is enabled but ineffective: SSH is enabled, which could be used to bypass protection Firmware upload is allowed, specially crafted firmware could be used to bypass protection
Protection from restore from backup is enabled but ineffective: SSH is enabled, which could be used to bypass protection Firmware upload is allowed, specially crafted firmware could be used to bypass protection
Network settings protection is disabled

Protection status details

20.4 Unlocking protection

You should use [the reset button](#) and select the "Clear lock bits" reset mode to clear protection bits. This, of course, requires physical access to the unit.

21 Resetting settings to defaults

The device's settings can be reset to defaults by pressing the reset button in the hole under the Ethernet jack.

You may want to take a [backup](#) of your settings first.

The display displays an overview of possible actions with a ticker on the second LCD line. You may interrupt it by pressing ,  or .

Use  and , or short presses of the reset button, to select a reset mode. A description of the currently selected mode is displayed on the second LCD line.

The following reset modes are available:

1. Clear lock bits: Clear protection bits only. Other settings are preserved.
2. Reset network and scripting: Clear protection bits, reset network settings and admin login, disable autoping and scripts. Other settings are preserved.
3. Reset network and scripting + enable WiFi: Clear protection bits, reset network settings and admin login, disable autoping and scripts, and enable open WiFi access. Other settings are preserved.
4. Complete wipe: Reset all settings to factory defaults and remove any user files. All settings will be lost!
5. Complete wipe + enable WiFi: Reset all settings to factory defaults and remove any user files, then enable open WiFi access. All settings are lost!

To activate the selected reset mode, press  or the reset button and hold it.

If all settings are reset (the two last 'wipe' reset modes), the Subnet Restriction will be enabled to prevent remote access using the default password. ONLY MACHINES IN THE SAME SUBNET WILL BE ABLE TO CONNECT AFTER RESETTING TO DEFAULTS. If connectivity is lost, use a local connection such as a laptop with a crossover cable to restore your original network settings.

22 Specifications

Alert Beeper	73dBa at 12". Programmable.
Applications	Commercial, industrial, and residential remote power control and reboot. Indoor use only.
Circuit Breakers	Manual reset, 15A Thermal, UL Supplemental
Clock / RTC	15 year Li battery
Controls / Display	Reset-to-factory-default switch, 2x16 Backlit LCD w/ PowerSave, 5 button keypad, E-Stop
Enclosure	Steel, double grounded. Vented 4 sides. Fanless.
Ethernet Interface	10/100 autosensing, Static IP, TCP port selectable, 8 pin RJ-45 w/ internal FCC filtering
Humidity	8-80% RH Operating
Input Power Cord	Fixed 14AWG with 5-15 plug standard
Inlet and Outlet Rating	UL, CSA 15A, 120VAC only
Input Frequency	Power supply - DC-400Hz
Metering Accuracy	+/- 2V, +/- .5A when calibrated 50-60Hz only.
Operating Temperature	-30° to 170°F, -34° to 77°C
Options - Factory	Input cord length and 120V plug style
Power Supply Rating	90-240V, AC/DC Autosensing
Password Transmission	Encrypted, base 64 or HTTPS
Power Dissipation	4.9W Typ Max (all on) <3 W idle
Power Fail Hold-Over	350ms minimum (all relays on)
Power-Up Modes	Last used settings, all power on or off, sequential on or run user-script ~30s after power-up
Relay Contact Spec	15-25A AC/DC, 1/2HP
Surge Protection	Dual 3600W Metal Oxide Varistors
Size	3.5x5.5x19" 8.9x14x48cm 2-U
Weight (packed)	Single unit 10.3lbs 4.7kg
WiFi	Atheros 9331 2.4G 802.11n RP-SMA

FCC Note:

The EPCR6 may only be used with

- the manufacturer supplied antenna (Gain: 2.0dBi),or
- a 50 Ohm antenna of equal or lesser gain.

23 Open source code

In compliance with the spirit of the GNU Public License, source code is provided together with the firmware itself (accessible using one of the user-configurable links). Note that it is placed on the read/write firmware partition, so certain operations (like full factory reset) may remove it.

Purchasing a TLA and signing an NDA from Atheros are highly recommended before attempting any custom development; however, they aren't required to build the firmware (only the bootloader).

DLI cannot provide warranty or technical support for modified units; this includes units with custom firmware.

24 Technical support

Please register. Painless on-line registration gets you:

- free tech support,
- access to firmware updates,
- and information when updates and new features become available.

To save time, please have a look at the product FAQ page solutions. You may FAX questions to (408) 541-8459 or email: support@digital-loggers.com.

For phone support, call (408) 330-5599 with the following so we can better serve you:

- The firmware version level installed in the power switch. This information can be found on the lower left corner of the outlet control page.
- A description of the Ethernet devices connected to your unit, for example, a 10/100 PC and crossover cable.
- A description of the WiFi devices connected to your unit, i.e. their manufacturers and model numbers.

25 Limited five year warranty

The terms of this warranty may be legally binding. If you do not agree to the terms listed below, return the product immediately in original unopened condition for a full refund. The purchaser assumes the entire risk as to the results and performance of the unit.

DLI warrants this power controller to be free from major defects. No agency, country, or local certifications are included with this unit. It is the responsibility of the user to obtain such certifications if necessary for the customer's application. Buyer acknowledges and agrees that he is solely responsible for proper use, certification and safety testing of components supplied by DLI. DLI's entire liability and exclusive remedy as to defective hardware shall be, at DLI's option, either (a) return of the purchase price or (b) replacement or repair of the hardware that does not meet DLI's quality control standards and has been returned through proper RMA procedures. DLI's liability for repair or replacement is to DLI's customer ONLY.

WARRANTY SERVICE DOES NOT COVER DAMAGE TO SCREW TERMINALS FROM EXCESSIVE TORQUE OR DAMAGE DUE TO EXPOSURE TO WATER OR VIBRATION.

NO SUPPORT IS PROVIDED FOR MODIFIED FIRMWARE. MODIFICATION OF FIRMWARE VOIDS ALL WARRANTY.

Warranty service requires an original invoice from DLI and an RMA number provided by technical support. RMA material must be shipped prepaid to DLI. RMA numbers are valid for 15 days from date of issue. This warranty does not cover products which are modified (including firmware modifications), subjected to rough handling, or used in applications for which they were not originally intended. Batteries are not covered under warranty. Physical damage caused by customer or in transit to DLI is not covered under warranty. Please insure your shipments.

No oral advice or verbal warranties made by DLI's employees, dealers, or distributors shall in any way increase the scope of this warranty. DLI makes no warranty as to merchantability or fitness for any particular purpose. DLI assumes no liability for incidental or consequential damages arising from the use or inability to use this product. This warranty gives you specific legal rights. You may also have other rights that vary from state to state. Since some states do not allow the exclusion of liability for consequential damages, some of the above limitations may not apply to you. This product is not qualified or intended for mobile, airborne, medical or aerospace use or FDA Class III applications.