Network UPS Tools Developer Guide

Russell Kroll, Arnaud Quette, Charles Lepple, Peter Selinger, Jim Klimov and NUT project community contributors

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.8.1	2023-10-31	Some changes to API, docs and recipes, in particular to simplify local builds and tests (e.g. to help end-users check if current NUT codebase trunk has already fixed an issue they see with a packaged installation). Revived NUT for Windows effort, further improved other OS integrations. NUT became reference for "UPS management protocol", Informational RFC 9271. Documentation files refactored to ease maintenance. More drivers and new driver categories introduced.	JK
2.8.0	2022-04-26	Change of maintainer. Many changes to API, docs (both style and content), and recipes, with a stress on non-regression test-ability, run-time debug-ability, general codebase maintainability, as well as OS integrations (notably nut-driver-enumerator for systemd and SMF service instance maintenance). Added a lot in area of CI support and documented pre-requisite package lists for numerous platforms, and CI agent set-up. Added libusb-1.x support and many new driver categories (and drivers), and daisychain device connection support. Instant commands enhanced with TRACKING to enable protocol-based waiting for completion of a particular INSTCMD or SET operation.	JK
2.7.4	2016-03-09	NUT variables namespace updated, in particular for outlet groups, alarms and thresholds, ATS devices, and battery.charger.status to supersede CHRG and DISCHRG flags published in ups.status readings. NUT network protocol extended with NUMBER type; some API changes.	AQ
2.7.3	2015-04-22	Documentation revised, including some API changes. Added NUT DDL links. NUT variables namespace updated.	AQ
2.7.2	2014-04-17	The nut-website project was offloaded into a separate repository. FreeDesktop HAL support was removed (obsoleted in GNOME consumer). Introduced nutdrv_atcl_usb driver.	AQ

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2.7.1	2013-11-19	NUT source codebase migrated from SVN to Git (and from Debian infrastructure to GitHub source code hosting). jNut binding split into a separate project. Introduced libnutclient (C++ binding), al175, apcupsd-ups and nutdrv_qx drivers, Mozilla NSS support for simpler licensing than OpenSSL, and a newer apcsmart implementation. Documentation support enhanced with a spell checker, contents massively updated to reflect project changes.	CL
2.6.5	2012-08-08	New macosx-ups driver, new implementation of mge-shut driver. NUT variables namespace updated. Docs cleaned up and revised.	AQ
2.6.4	2012-05-31	New NUT network protocol commands (LIST CLIENTS, LIST RANGE and NETVER), and socket protocol commands (ADDRANGE, DELRANGE). NUT variables namespace updated. Introduced nut-recorder tool.	AQ
2.6.3	2012-01-04	No substantial changes to documentation.	AQ
2.6.2	2011-09-15	Introduced nut-scanner tool and nut-ipmipsu driver, systemd support, and a new apcsmart implementation.	AQ
2.6.1	2011-06-01	Introduced default.* and override.* optional settings in ups.conf, an ups.efficiency report, and outlet.0 special handling.	AQ
2.6.0	2011-01-14	First release of AsciiDoc documentation for Network UPS Tools (NUT).	AQ

Contents

1 Introduction **1**

2 NUT design document **1**

2.1 The layering . . . 3
2.2 How information gets around . . . 4
2.2.1 From the equipment . . . 4
2.2.2 From the driver . . . 4
2.2.3 From the server . . . 4
2.3 Instant commands . . . 4
2.4 Setting variables . . . 4
2.5 Example data path . . . 5
2.6 History . . . 6

3 Information for developers **6**

3.1 General stuff — common subdirectory . . . 6
3.1.1 String handling . . . 6
3.1.2 Error reporting . . . 7
3.1.3 Debugging information . . . 7
3.1.4 Memory allocation . . . 7
3.1.5 Config file parsing . . . 7
3.1.6 <time.h> vs. <sys/time.h> . . . 7
3.2 Device drivers — main.c . . . 7
3.3 Portability . . . 7
3.3.1 C comments . . . 8
3.3.2 Variable declarations go on top . . . 8
3.3.3 Variable declaration in loop block syntax . . . 8
3.3.4 Other hints . . . 8
3.4 Continuous Integration and Automated Builds . . . 9
3.4.1 Build automation tools and scripts . . . 9
ci_build.sh . . . 9
Jenkins CI . . . 10
AppVeyor CI . . . 11
CircleCI . . . 11
Travis CI . . . 12
3.4.2 Pre-set warning options . . . 12
3.5 Integrated Development Environments (IDEs) and debugging NUT . . . 12
3.5.1 IDE notes on Windows . . . 13

General settings for builds on Windows . . . 13
GDB on Windows . . . 14
NetBeans on Windows . . . 14
Microsoft VS Code . . . 16
IntelliJ IDEA . . . 17
3.6 Coding style . . . 18
3.6.1 Indenting with tabs vs. spaces . . . 18
3.6.2 Line breaks . . . 19
3.6.3 Un-used variables and function arguments . . . 19
3.7 Miscellaneous coding style tools . . . 20
3.7.1 Finishing touches . . . 20
3.7.2 Switch case fall-through . . . 20
3.7.3 Spaghetti . . . 21
3.7.4 Legacy code . . . 21
3.7.5 Memory leak checking . . . 21
3.7.6 Conclusion . . . 21
3.8 Submitting patches . . . 21
3.9 Patch cohesion . . . 22
3.10 The finishing touches: manual pages and device entry in HCL . . . 22
3.11 Source code management . . . 22
3.11.1 Git access . . . 22
3.11.2 Mercurial (hg) access . . . 23
3.11.3 Subversion (SVN) access . . . 23
3.12 Ignoring generated files . . . 23
3.13 Commit message formatting . . . 23
3.14 Commit sign-off . . . 24
3.15 Repository etiquette and quality assurance . . . 24
3.16 Building the Code . . . 25

4 Creating a new driver to support another device 26
4.1 Smart vs. Contact-closure . . . 26
4.2 Serial vs. USB vs. SNMP and more . . . 26
4.3 Overall concept . . . 27
4.4 Skeleton driver . . . 27
4.5 Essential structure . . . 27
4.5.1 upsdrv_info_t . . . 27
4.6 Essential functions . . . 27
4.6.1 upsdrv_initups . . . 27
4.6.2 upsdrv_initinfo . . . 27

4.6.3 upsdrv_updateinfo . . . 28
4.6.4 upsdrv_shutdown . . . 28
4.7 Data types . . . 28
4.8 Manipulating the data . . . 28
4.8.1 Adding variables . . . 28
4.8.2 Setting flags . . . 28
4.8.3 Status data . . . 29
4.9 UPS alarms . . . 29
4.10 Staleness control . . . 30
4.11 Serial port handling . . . 30
4.12 USB port handling . . . 33
4.12.1 Structure and macro . . . 33
4.12.2 Function . . . 34
4.13 Variable names . . . 34
4.14 Message passing support . . . 34
4.14.1 SET . . . 35
4.14.2 INSTCMD . . . 35
4.14.3 Notes . . . 35
4.14.4 Responses . . . 35
4.15 Enumerated types . . . 35
4.16 Range values . . . 35
4.17 Writable strings . . . 36
4.18 Instant commands . . . 36
4.19 Delays and ser_* functions . . . 36
4.20 Canonical input mode processing . . . 36
4.21 Adding the driver into the tree . . . 37
4.22 Contact closure hardware information . . . 37
4.22.1 Definitions . . . 37
4.22.2 Bad levels . . . 37
4.22.3 Signals . . . 37
4.22.4 New genericups types . . . 38
4.22.5 Custom definitions . . . 38
4.23 How to make a new subdriver to support another USB/HID UPS . . . 39
4.23.1 Overall concept . . . 39
4.23.2 HID Usage Tree . . . 39
4.23.3 Usage macros in drivers/hidtypes.h . . . 40
4.23.4 Writing a subdriver . . . 40
4.23.5 Updating a subdriver . . . 41
4.23.6 Customization . . . 41

4.23.7 Fixing report descriptors . . . 42
4.23.8 Investigating report descriptors . . . 42
4.23.9 Shutting down the UPS . . . 44
4.24 How to make a new subdriver to support another SNMP device . . . 44
4.24.1 Overall concept . . . 44
4.24.2 SNMP data Tree . . . 45
4.24.3 Creating a subdriver . . . 47
mode 1: get SNMP data from a real agent . . . 47
mode 2: get data from files . . . 47
Integrating the subdriver with snmp-ups . . . 48
CUSTOMIZATION . . . 48
4.25 How to make a new subdriver to support another Q* UPS . . . 49
4.25.1 Overall concept . . . 49
4.25.2 Creating a subdriver . . . 49
4.25.3 Writing a subdriver . . . 49
4.25.4 Mapping an idiom to NUT . . . 51
4.25.5 Examples . . . 53
Simple vars . . . 54
Mandatory vars . . . 54
Settable vars . . . 55
Instant commands . . . 56
Information absent in the device . . . 57
Information not yet available in NUT . . . 57
4.25.6 Support functions . . . 60
4.25.7 Armac Subdriver . . . 60
Transfer dumps . . . 61
4.25.8 Notes . . . 63

5 Driver/server socket protocol **64**
5.1 Formatting . . . 64
5.2 Commands used by the drivers . . . 64
5.2.1 SETINFO . . . 64
5.2.2 DELINFO . . . 64
5.2.3 ADDENUM . . . 64
5.2.4 DELENUM . . . 65
5.2.5 ADDRANGE . . . 65
5.2.6 DELRANGE . . . 65
5.2.7 SETAUX . . . 65
5.2.8 SETFLAGS . . . 65

5.2.9 ADDCMD . . . 65
5.2.10 DELCMD . . . 65
5.2.11 DUMPDONE . . . 66
5.2.12 PONG . . . 66
5.2.13 DATAOK . . . 66
5.2.14 DATASTALE . . . 66
5.2.15 TRACKING . . . 66
5.3 Commands sent by the server . . . 66
5.3.1 PING . . . 66
5.3.2 INSTCMD . . . 67
5.3.3 SET . . . 67
5.3.4 DUMPALL . . . 67
5.3.5 NOBROADCAST . . . 67
5.3.6 BROADCAST (NUM) . . . 67
5.4 Design notes . . . 68
5.4.1 Requests . . . 68
5.4.2 Access/Security . . . 68
5.4.3 Command limitations . . . 68
5.4.4 Re-establishing communications . . . 68

6 NUT configuration management with Augeas 68
6.1 Introduction . . . 68
6.2 Requirements . . . 68
6.2.1 Augeas . . . 69
6.2.2 NUT lenses and modules for Augeas . . . 69
6.3 Create a test sandbox . . . 69
6.4 Start testing and using . . . 69
6.4.1 Shell . . . 69
6.4.2 Python . . . 70
6.4.3 Perl . . . 71
6.4.4 Test the conformity testing module . . . 71
6.5 Complete configuration wizard example . . . 71

7 NUT device discovery 72
7.1 Introduction . . . 72
7.1.1 Client access library . . . 72
7.1.2 Configuration helpers . . . 74
7.2 Python . . . 74
7.3 Perl . . . 74
7.4 Java . . . 74

8 Creating new client **74**
8.1 C / C++ . . . 74
8.1.1 Client access library . . . 74
Low-level library: libupsclient . . . 74
High level library: libnutclient . . . 75
8.1.2 Configuration helpers . . . 76
8.2 Python . . . 76
8.3 Perl . . . 77
8.4 Java . . . 77

9 Network protocol information **77**
9.1 Old command removal notice . . . 77
9.2 Command reference . . . 77
9.3 Revision history . . . 78
9.4 GET . . . 78
9.4.1 NUMLOGINS . . . 78
9.4.2 UPSDESC . . . 79
9.4.3 VAR . . . 79
9.4.4 TYPE . . . 79
9.4.5 DESC . . . 80
9.4.6 CMDDESC . . . 80
9.4.7 TRACKING . . . 80
9.5 LIST . . . 80
9.5.1 UPS . . . 81
9.5.2 VAR . . . 81
9.5.3 RW . . . 81
9.5.4 CMD . . . 82
9.5.5 ENUM . . . 82
9.5.6 RANGE . . . 83
9.5.7 CLIENT . . . 83
9.6 SET . . . 83
9.6.1 VAR . . . 83
9.6.2 TRACKING . . . 84
9.7 INSTCMD . . . 84
9.8 LOGOUT . . . 84
9.9 LOGIN . . . 85
9.10 PRIMARY (since NUT 2.8.0) or MASTER (deprecated) . . . 85
9.11 FSD . . . 86
9.12 PASSWORD . . . 86

9.13 USERNAME . . . 86
9.14 STARTTLS . . . 87
9.15 Other commands . . . 87
9.16 Error responses . . . 87
9.17 Future ideas . . . 89
9.17.1 Dense lists . . . 89
9.17.2 Get collection . . . 89

10 NUT developers tools **89**
10.1 Device simulation . . . 89
10.2 Device recording . . . 90

11 NUT core development and maintenance **90**
11.1 NUT-specific autoconf macros . . . 90
11.2 NUT roadmap and ideas for future expansion . . . 92
11.2.1 Roadmap . . . 92
2.6 . . . 92
2.8 . . . 92
3.0 . . . 92
11.2.2 Non-network "upsmon" . . . 92
11.2.3 Completely unprivileged upsmon . . . 92
11.2.4 Chrooted upsmon . . . 93
11.2.5 Monitor program with interpreted language . . . 93
11.2.6 Sandbox . . . 93

A NUT command and variable naming scheme **93**
A.1 Structured naming . . . 94
A.2 Time and Date format . . . 94
A.3 Variables . . . 95
A.3.1 device: General unit information . . . 95
A.3.2 ups: General unit information . . . 95
A.3.3 input: Incoming line/power information . . . 96
A.3.4 output: Outgoing power/inverter information . . . 98
A.3.5 Three-phase additions . . . 98
Phase Count Determination . . . 98
DOMAINs . . . 98
Specification (SPEC) . . . 98
CONTEXT . . . 98
Valid CONTEXTs . . . 99
Valid SPECs . . . 99

A.3.6 EXAMPLES . . . 100
A.3.7 battery: Any battery details . . . 100
A.3.8 ambient: Conditions from external probe equipment . . . 102
A.3.9 outlet: Smart outlet management . . . 103
outlet.group: groups of smart outlets . . . 104
A.3.10 driver: Internal driver information . . . 104
A.3.11 server: Internal server information . . . 105
A.4 Instant commands . . . 105

B NUT daisychain support notes **106**
B.1 Introduction . . . 106
B.2 Implementation notes . . . 106
B.2.1 General specification . . . 106
Devices status handling . . . 106
Devices alarms handling . . . 107
Example . . . 107
B.2.2 Information for developers . . . 107
Base support . . . 108
Templates with multiple definitions . . . 108
Devices alarms handling . . . 108

C NUT libraries complementary information **109**
C.1 Introduction . . . 109
C.2 libupsclient-config . . . 109
C.3 pkgconfig support . . . 110
C.4 Example configure script . . . 110
C.5 Future consideration . . . 111
C.6 Libtool information . . . 111

1 Introduction

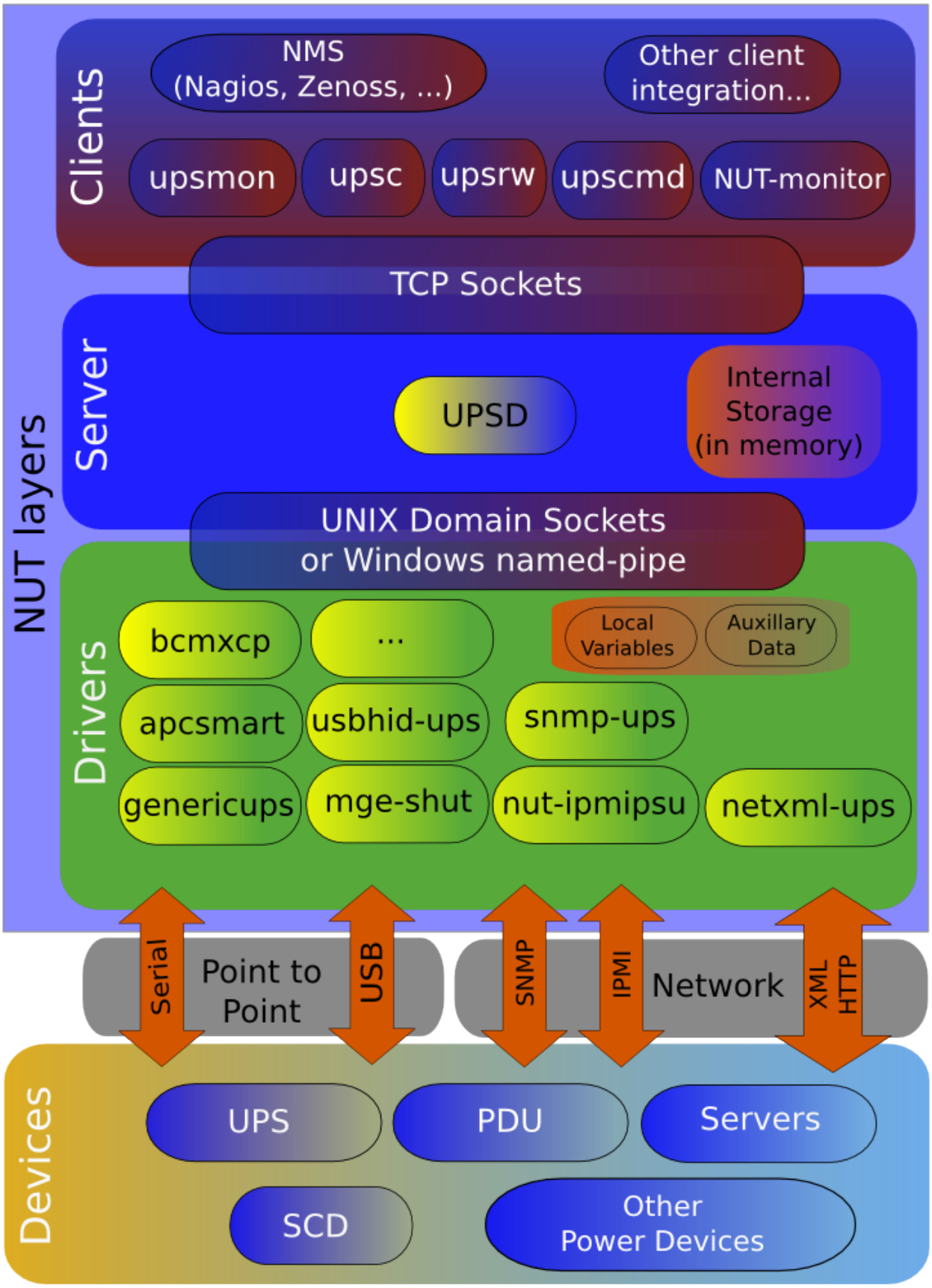
NUT is both a powerful toolkit and framework that provides support for Power Devices, such as Uninterruptible Power Supplies, Power Distribution Units and Solar Controllers.

This document intend to describe how NUT is designed, and the way to develop new device drivers and client applications.

2 NUT design document

This software is designed around a layered scheme with drivers, a server and clients. These layers communicate with text-based protocols for easier maintenance and diagnostics.

2.1 The layering



2.2 How information gets around

2.2.1 From the equipment

DRIVERS talk to the EQUIPMENT and receive updates. For most hardware this is polled (DRIVER asks EQUIPMENT about a variable), but forced updates are also possible. The exact method is not important, as it is abstracted by the driver.

2.2.2 From the driver

The core of all DRIVERS maintains internal storage for every variable that is known along with the auxiliary data for those variables. It sends updates to this data to any process which connects to the Unix domain socket.

The DRIVERS will also provide a full atomic copy of their internal knowledge upon receiving the "DUMPALL" command on the socket. The dump is in the same format as updates, and is followed by "DUMPDONE". When "DUMPDONE" has been received, the view is complete.

The SERVER will connect to the socket of each DRIVER and will request a dump at that time. It retains this data in local storage for later use. It continues to listen on the socket for additional updates.

This protocol is documented in sock-protocol.txt.

2.2.3 From the server

The SERVER's internal storage maintains a complete copy of the data which is in the DRIVER, so it is capable of answering any request immediately. When a request for data arrives from a CLIENT, the SERVER looks through the internal storage for that UPS and returns the requested data if it is available.

The format for requests from the CLIENT is documented in protocol.txt.

2.3 Instant commands

"Instant commands" is the term given to a set of actions that result in something happening to the UPS. Some of the common ones are `test.battery.start` to initiate a battery test and `test.panel.start` to test the front panel of the UPS.

They are passed to the SERVER from a CLIENT using an authenticated network connection. The SERVER first checks to make sure that the instant command is valid for the DRIVER. If it's supported, a message is sent via a socket to the DRIVER containing the command and any auxiliary information.

At this point, there is no confirmation to the SERVER of the command's execution. This is (still) planned for a future release. This has been delayed since returning a response involves some potentially interesting timing issues. Remember that `upsd` services clients in a round-robin fashion, so all queries must be lightweight and speedy.

Note
FIXME: Wasn't "TRACKING" mechanism for "INSTCMD/SET VAR" introduced to address just this? See https://github.com/-networkupstools/nut/pull/659

2.4 Setting variables

Some variables in the DRIVER or EQUIPMENT can be changed, and carry the FLAG_RW flag. Upon receiving a SET command from the CLIENT, the SERVER first verifies that it is valid for that DRIVER in terms of writability and data type. If those checks pass, it then sends the SET command through the socket, much like the instant command design.

The DRIVER is expected to commit the value to the EQUIPMENT and update its internal representation of that variable.

Like the instant commands, there is currently no acknowledgement of the command's completion from the DRIVER. This, too, is planned for a future release.

Note
FIXME: Wasn't "TRACKING" mechanism for "INSTCMD/SET VAR" introduced to address just this? See https://github.com/-networkupstools/nut/pull/659

2.5 Example data path

Here's the path a piece of data might take through this architecture. The event is a UPS going on battery, and the final result is a pager delivering the alpha message to the admin.

1. EQUIPMENT reports on battery by setting flag in status register
2. DRIVER notices this flag and stores it in the `ups.status` variable as OB. This update gets pushed out to any listeners via the sockets.
3. SERVER `upsd` sees activity on the socket, reads it, parses it, and commits the new data to its local version of the status variable.
4. CLIENT `upsmon` does a routine poll of SERVER for `ups.status` and gets `OB`.
5. CLIENT `upsmon` then invokes its `NOTIFYCMD` which is `upssched`.
6. `upssched` starts up a daemon to handle a timer which will expire about 30 seconds into the future.
7. 30 seconds later, the timer expires since the UPS is still on battery, and so `upssched` calls the `CMDSCRIPT` which is `upssched-cmd`.
8. `upssched-cmd` parses the args and calls `sendmail`.
9. Avian carriers, smoke signals, SMTP, and some magic result in the message getting from the pager company's gateway to a transmitter and then to the admin's pager.

This scenario requires some configuration, obviously:

1. There's an UPS driver running. (Whatever applies for the hardware)
2. `upsd` has a valid UPS entry in *ups.conf* for this UPS.

```
[myups]
        driver = nutupsdrv
        port = /dev/ttySx
```

3. `upsd` has a valid user for `upsmon` in *upsd.users* file.

```
[monuser]
        password = somepass
        upsmon primary
```

4. `upsmon` is set to monitor this UPS with this user in *upsmon.conf* file.

```
MONITOR myups@localhost 1 monuser somepass primary
```

5. `upsmon` is set to `EXEC` the `NOTIFYCMD` for the `ONBATT` condition in *upsmon.conf* file.

```
NOTIFYFLAG ONBATT EXEC
```

6. `upsmon` calls `upssched` as the `NOTIFYCMD` in *upsmon.conf* file.

```
NOTIFYCMD /path/to/upssched
```

7. upssched has a 30 second timer for ONBATT in *upssched.conf* file.

   ```
   AT ONBATT * START-TIMER upsonbatt 30
   ```

8. upssched calls upssched-cmd as the CMDSCRIPT in *upssched.conf*.

   ```
   CMDSCRIPT /path/to/upssched-cmd
   ```

9. upssched-cmd knows what to do with upsonbatt keyword as its first argument (a quick case..esac construct, see the examples)

2.6 History

The oldest versions of this software (1998) had no separation between the driver and the network server, and only supported the latest APC Smart-UPS hardware as a result. The network protocol used brittle binary structs. This had numerous bad implications for compatibility and portability.

After the driver and server were separated, data was shared through the state file concept. Status was written into a static array (the "info array") by drivers, and that array was stored on disk. The upsd would periodically read that file into a local copy of that array.

Shared memory mode was added a bit later, and that removed some of the lag from the status updates. Unfortunately, it didn't have any locking originally, and the possibility for corruption due to races existed.

mmap() support was added at some point after that, and became the default. The drivers and upsd would mmap() the file into memory and read or write from it. Locking was done using the state file as the token, so contention problems were avoided. This method was relatively quick, but it involved at least 3 copies of the data (driver, disk/mmap, server) and a whole lot of locking and unlocking. It could occasionally delay the driver or server when waiting for a lock.

In April 2003, the entire state management subsystem was removed and replaced with a single local socket. The drivers listen for connections and push updates asynchronously to any listeners. They also recognize a few commands. Drivers also dampen updates, and only push them out when something actually changes.

As a result, upsd no longer has to poll any files on the disk, and can just select() all of its file descriptors (fds) and wait for activity. When one of them is active, it reads the fd and parses the results. Updates from the hardware now get to upsd about as fast as they possibly can.

Drivers used to call setinfo() to change the local array, and then would call writeinfo() to push the array onto the disk, or into the mmap/shared memory space. This introduced a lag since many drivers poll quite a few variables during an update.

3 Information for developers

This document is intended to explain some of the more useful things within the tree, and provide a standard for working on the code.

3.1 General stuff — common subdirectory

3.1.1 String handling

Use snprintf(). It's even provided with a compatibility module if the target system doesn't have it natively.

If you use snprintf() to load some value into a buffer, make sure you provide the format string. Don't use user-provided format strings, since that's an easy way to open yourself up to an exploit.

Don't use strcat(). We have a neat wrapper for snprintf() called snprintfcat() that allows you to append to char * with a format string and all the usual string length checking of snprintf() routine.

3.1.2 Error reporting

Don't call `syslog()` directly. Use `upslog_with_errno()` and `upslogx()`. They may write to the syslog, stderr, or both as appropriate. This means you don't have to worry about whether you're running in the background or not.

The `upslog_with_errno()` routine prints your message plus the string expansion of `errno`. The `upslogx()` just prints the message.

`fatal_with_errno()` and `fatalx()` work the same way, but they also `exit(EXIT_FAILURE)` afterwards. Don't call `exit()` directly.

3.1.3 Debugging information

The `upsdebug_with_errno()`, `upsdebugx()`, `upsdebug_hex()` and `upsdebug_ascii()` routines use the global `nut_debug_level`, so you don't have to mess around with `printf()`'s and `if`'s yourself. Use them.

3.1.4 Memory allocation

`xmalloc()`, `xcalloc()`, `xrealloc()` and `xstrdup()` all check the results of the base calls before continuing, so you don't have to. Don't use the raw calls directly.

3.1.5 Config file parsing

The configuration parser, called `parseconf`, is now up to its fourth major version. It has multiple entry points, and can handle many different jobs. It's usually used for parsing files, but it can also take input a line at a time or even a character at a time.

You must initialize a context buffer with `pconf_init()` before using any other `parseconf` function. `pconf_encode()` is the only exception, since it operates on a buffer you supply and is an auxiliary function.

Escaping special characters and quoting multiple-word elements is all handled by the state machine. Using the same code for all config files avoids code duplication.

Note
this does not apply to drivers. Driver authors should use the `upsdrv_makevartable()` scheme to pick up values from *ups.conf* file. Drivers should not have their own config files.

Drivers may have their own data files, such as lists of hardware, mapping tables, or similar. The difference between a data file and a config file is that users should never be expected to edit a data file under normal circumstances. This technique might be used to add more hardware support to a driver without recompiling.

3.1.6 <time.h> vs. <sys/time.h>

This is already handled by autoconf, so just `#include "timehead.h"` and you will get the right headers on every system.

3.2 Device drivers — main.c

The device drivers use `main.c` as their core.

To write a new driver, you create a file with a series of support functions that will be called by main. These all have names that start with `upsdrv_`, and they will be called at different times by main depending on what needs to happen.

See the driver documentation for information on writing drivers, and also refer to the skeletal driver in `skel.c`.

3.3 Portability

Avoid things that will break on other systems. All the world is not an x86 Linux box.

3.3.1 C comments

There are still older systems out there that don't do C++ style comments.

```
/* Comments look like this. */
// Not like this.
```

3.3.2 Variable declarations go on top

Newer versions of gcc allow you to declare a variable inside a function after code, somewhat like the way C++ operates, like this:

```
function do_stuff(void)
{
        check_something();

        int a;

        a = do_something_else();
}
```

While this will compile and run on these newer versions, it will fail miserably for anyone on an older system. That means you must not use it.

Note that `gcc` only warns about this with `-pedantic` flag, and `clang` with a `-Weverything` (possibly `-Wextra`) flag, which can be enabled by developers with `configure --enable-warnings=...` option values (and made fatal with `configure --enable-Werror`), to ensure non-regression of code quality. It was reported that `clang-16` with such options does complain about non-portability to older C language revisions even if explicitly building for a newer revision.

Please note that for the purposes of legacy-compatible variable declarations (on top of their scopes), a `NUT_UNUSED_VARIABLE(varn` counts as code and should be used just below the declarations. Initial assignments to variables (also as return values of methods) may generally happen as part of their declarations.

You can use scoping (e.g. `do { ... } while (0);`) where it makes sense to constrain visibility of temporary variables, such as in `switch/case` blocks.

3.3.3 Variable declaration in loop block syntax

Another feature that does not work on some compilers (e.g. conforming to "ANSI C"/C89/C90 standard) is initial variable declaration inside a *for loop* block, like this:

```
function do_stuff(void)
{
        /* This should declare "int i;" first, then use it in "for" loop: */
        for (int i = 0; i < INT_MAX; ++i) { ... }

        /* Additional loops cause also an error about re-declaring a variable: */
        for (int i = 10; i < 15; ++i) { ... }
}
```

3.3.4 Other hints

Tip
At this point NUT is expected to work correctly when built with a C99 (or rather GNU99 on many systems) or newer standard.

The NUT codebase may build in a mode without warnings made fatal on C89 (GNU89), but the emitted warnings indicate that those binaries may crash. By the end of 2021, NUT codebase has been revised to pass GNU and strict-C mode builds with C89 standard with the GCC toolkit (and on systems that do have the newer features in libraries, just hide them in standard headers); however CLANG toolkit is more restrictive about the C99+ syntax used. If somebody in the community requires to build and run NUT on systems that old, pull requests to fix the offending coding issues are welcome.

Note also that the NUT codebase currently relies on certain features, such as the printf format modifiers for `(s)size_t`, use of `long long`, some nuances about structure/array initializers, variadic macros for debugging, etc. that a pedantic C90 mode compilation warns is not part of the standard but a GNU extension (and part of C99 and newer standard revisions). Many of the "offences" against the older standard actually come from system and third-party header files.

That said, the NUT CI farm does run non-regression builds with GNU C89 and strict C89 standard revisions and minimal passing warnings level, to ensure that codebase is and remains at least basically compliant.

3.4 Continuous Integration and Automated Builds

To ease and automate the build scenarios which were deemed important for quality assurance and non-regression checks of NUT, several solutions were introduced over time.

3.4.1 Build automation tools and scripts

ci_build.sh

This script was originally introduced (following ZeroMQ/ZProject example) to automate CI builds, by automating certain scenarios driven by exported environment variables to set particular `configure` options and `make` some targets (chosen by the `BUILD_TYPE` envvar). It can also be used locally to avoid much typing to re-run those scenarios during development.

Developers can directly use the scripts involved in CI builds to fix existing code on their workstations or to ensure support for new compilers and C standard revisions, e.g. save a local file like this to call the common script with pre-sets:

```
$ cat _fightwarn-gcc10-gnu17.sh
#!/bin/sh

BUILD_TYPE=default-all-errors \
CFLAGS="-Wall -Wextra -Werror -pedantic -std=gnu17" \
CXXFLAGS="-Wall -Wextra -Werror -std=gnu++17" \
CC=gcc-10 CXX=g++-10 \
        ./ci_build.sh
```

... and then execute it to prepare a workspace, after which you can go fixing bugs file-by-file running a `make` after each save to confirm your solutions and uncover the next issue to address :-)

Helpfully, the NUT CI farm build logs report the configuration used for each executed stage, so if some build combination fails — you can just scroll to the end of that section and copy-paste the way to reproduce an issue locally (on an OS similar to that build case).

Note that while spelling out sets of warnings can help in a quest to fix certain bugs during development (if only by removing noise from classes of warnings not relevant to the issue one is working on), there is a reasonable set of warnings which NUT codebase actively tries to be clean about (and checks in CI), detailed in the next section.

For the `ci_build.sh` usage like above, one can instead pass the setting via `BUILD_WARNOPT=...`, and require that all emitted warnings are fatal for their build, e.g.:

```
$ cat _fightwarn-clang9-gnu11.sh
#!/bin/sh
```

```
BUILD_TYPE=default-all-errors \
BUILD_WARNOPT=hard BUILD_WARNFATAL=yes \
CFLAGS="-std=gnu11" \
CXXFLAGS="-std=gnu++11" \
CC=clang-9 CXX=clang++-9 CPP=clang-cpp \
        ./ci_build.sh
```

Finally, for refactoring effort geared particularly for fighting the warnings which exist in current codebase, the script contains some presets (which would evolve along with codebase quality improvements) as `BUILD_TYPE=fightwarn-gcc`, `BUILD_TYPE=fightwarn-clang` or plain `BUILD_TYPE=fightwarn`:

```
BUILD_TYPE=fightwarn-clang ./ci_build.sh
```

As a rule of thumb, new contributions must not emit any warnings when built in GNU99 mode with a `minimal` "difficulty" level of warnings. Technically they must survive the part of test matrix across the several platforms tested by NUT CI and marked in project settings as required to pass, to be accepted for a pull request merge.

Developers aiming to post successful pull requests to improve NUT can pass the `--enable-warnings` option to the `configure` script in local builds to see how that behaves and ensure that at least in some set-up their contribution is viable. Note that different compiler versions and vendors (gcc/clang/...), building against different OS and third-party dependencies, with different CPU architectures and different language specification revisions, might all complain about different issues — and catching this in as diverse range of set-ups as possible is why we have CI tests.

It can be beneficial for serial developers to set up a local BuildBot, Travis or a Jenkins instance with a matrix test job, to test their local git repository branches with whatever systems they have available.

- https://github.com/networkupstools/nut/issues/823

While `autoconf` tries its best to provide portable shell code, sometimes there are builds of system shell that just fail under stress. If you are seeing random failures of `./configure` script in different spots with the same inputs, try telling `./ci_build.sh` to loop configuring until success (instead of quickly failing), and/or tell `./configure` to use another shell at least for the system call-outs, with options like these:

```
SHELL=/bin/bash CONFIG_SHELL=/bin/bash CI_SHELL_IS_FLAKY=true \
./ci_build.sh
```

Jenkins CI

Since mid-2021, the NUT CI farm is implemented by several virtual servers courteously provided by http://fosshost.org

These run various operating systems as build agents, and a Jenkins instance to orchestrate the builds of NUT branches and pull requests on those agents.

This is driven by `Jenkinsfile-dynamatrix` and a Jenkins Shared Library called jenkins-dynamatrix which prepares a matrix of builds across as many operating systems, bitnesses/architectures, compilers, make programs and C/C++ revisions as it can — based on the population of currently available build agents and capabilities which they expose as agent labels.

This hopefully means that people interested in NUT can contribute to the build farm (and ensure NUT is and remains compatible with their platform) by running a Jenkins Swarm agent with certain labels, which would dial into https://ci.networkupstools.org/ controller. Please contact the NUT maintainer if you want to participate in this manner.

The `Jenkinsfile-dynamatrix` recipe allows NUT CI farm to run different sets of build scenarios based on various conditions, such as the name of branch being built (or PR'ed against), changed files (e.g. C/C++ sources vs. just docs), and some build combinations may be not required to succeed.

For example, the main development branch and pull requests against it must cleanly pass all specified builds and tests on various platforms with the default level of warnings specified in the `configure` script. These are balanced to not run too many build scenarios overall, but just a quick and sufficiently representative set.

As another example, there is special handling for "fightwarn" pattern in the branch names to run many more builds with varying warning levels and more variants of intermediate language revisions, and so expose concerns deliberately missed by default warnings levels in "master" branch builds (the bar moves over time, as some classes of warnings become extinct from our codebase).

Further special handling for branches named like `fightwarn.*89.*` regex enables more intensive warning levels for a GNU89 build specifically (which are otherwise disabled as noisy yet not useful for supported C99+ builds), and is intended to help develop fixes for support of this older language revision, if anyone would dare.

Many of those unsuccessful build stages are precisely the focus of the "fightwarn" effort, and are currently marked as "may fail", so they end up as "UNSTABLE" (seen as orange bubbles in the Jenkins BlueOcean UI, or orange cells in the tabular list of stages in the legacy UI), rather than as "FAILURE" (red bubbles) for build scenarios that were not expected to fail and usually represent higher-priority problems that would block a PR.

Developers whose PR builds (or attempts to fix warnings) did not succeed in some cell of such build matrix, can look at the individual logs of that cell. Beside indication from the compiler about the failure, the end of log text includes the command which was executed by CI worker and can be reproduced locally by the developer, e.g.:

```
22:26:01  FINISHED with exit-code 2 cmd:  (
22:26:01  [ -x ./ci_build.sh ] || exit
22:26:01
22:26:01  eval BUILD_TYPE="default-alldrv" BUILD_WARNOPT="hard" \
    BUILD_WARNFATAL="yes" MAKE="make"  CC=gcc-10 CXX=g++-10 \
    CPP=cpp-10 CFLAGS='-std=gnu99 -m64' CXXFLAGS='-std=gnu++11 -m64' \
    LDFLAGS='-m64' ./ci_build.sh
22:26:01  )
```

or for autotools-driven scenarios (which prep, configure, build and test in separate stages — so for reproducing a failed build you should also look at its configuration step separately):

```
22:28:18  FINISHED with exit-code 0 cmd:  ( [ -x configure ] || exit; \
    eval  CC=clang-9 CXX=clang++-9 CPP=clang-cpp-9 CFLAGS='-std=c11 -m64' \
    CXXFLAGS='-std=c++11 -m64' LDFLAGS='-m64' time ./configure )
```

To re-run such scenario locally, you can copy the line from `eval` (but without the `eval` keyword itself) up to and including the executed script or tool, into your shell. Depending on locally available compilers, you may have to tweak the `CC`, `CXX` and `CPP` arguments; note that a `CPP` may be specified as `/path/to/CC -E` for GCC and CLANG based toolkits at least, if they lack a standalone preprocessor program (e.g. IntelCC).

Note
While NUT recipes do not currently recognize a separate `CXXCPP`, it would follow similar semantics.

Some further details about the NUT CI farm workers are available in config-prereqs.txt and ci-farm-lxc-setup.txt documents.

AppVeyor CI

Primarily used for building NUT for Windows on Windows instances provided in the cloud — and so ensure non-regression as well as downloadable archives with binary installation prototype area, intended for enthusiastic testing (proper packaging to follow). NUT for Windows build-ability was re-introduced soon after NUT 2.8.0 release.

This relies on a few prerequisite packages and a common NUT configuration, as coded in the `appveyor.yml` file in the NUT codebase.

CircleCI

Primarily used for building NUT for MacOS on instances provided in the cloud, and so ensure non-regression across several Xcode releases.

This relies on a few prerequisite packages and a common NUT configuration, as coded in the `.circleci/config.yml` file in the NUT codebase.

Travis CI

See the `.travis.yml` file in project sources for a detailed list of third party dependencies and a large matrix of `CFLAGS` and compiler versions last known to work or to not (yet) work on operating systems available to that CI solution.

Note
The cloud Travis CI offering became effectively defunct for open-source projects in mid-2021, so the `.travis.yml` file in NUT codebase is not actively maintained.
Local private deployments of Travis CI are possible, so if anybody does use it and has updated markup to share, they are welcome to post PRs.

The NUT project on GitHub has integration with Travis CI to test a large set of compiler and option combinations, covering different versions of gcc and clang, C standards, and requiring to pass builds at least in a mode without warnings (and checking the other cases where any warnings are made fatal).

3.4.2 Pre-set warning options

The options chosen into pre-sets that can be selected by `configure` script options are ones we use for different layers of CI tests.

Values to note include:

- `--enable-Werror(=yes/no)` — make warnings fatal;
- `--enable-warnings(=.../no)` — enable certain warning presets:
  - `gcc-hard`, `clang-hard`, `gcc-medium`, `clang-medium`, `gcc-minimal`, `clang-minimal`, `all` — actual definitions that are compiler-dependent (the latter just adds `-Wall` which may be relatively portable);
  - `hard`, `medium` or `minimal` — if current compiler is detected as CLANG or GCC, apply corresponding setting from above (or `all` otherwise);
  - `gcc` or `clang` — apply the set of options (regardless of detected compiler) with default "difficulty" hard-coded in `configure` script, to tweak as our codebase becomes cleaner;
  - `yes`/`auto` (also takes effect if `--enable-warnings` is requested without an `=ARG` part) — if current compiler is detected as CLANG or GCC, apply corresponding setting with default "difficulty" from above (or `all` otherwise).

Note that for backwards-compatibility reasons and to help filter out introduction of blatant errors, builds with compilers that claim GCC compatibility can enable a few easy warning presets by default. This can be avoided with an explicit argument to `--disable-warnings` (or `--enable-warnings=no`).

All levels of warnings pre-sets for GCC in particular do not enforce the `-pedantic` mode for builds with C89/C90/ANSI standard revision (as guesstimated by `CFLAGS` content), because nowadays it complains more about the system and third-party library headers, than about NUT codebase quality (and "our offenses" are mostly something not worth fixing in this era, such as the use of `__func__` in debug commands). If there still are practical use-cases that require builds of NUT on pre-C99 compiler toolkits, pull requests are of course welcome — but the maintainer team does not intend to spend much time on that.

Hopefully this warnings pre-set mechanism is extensible enough if we would need to add more compilers and/or "difficulty levels" in the future.

Finally, note that such pre-set warnings can be mixed with options passed through `CFLAGS` or `CXXFLAGS` values to your local `configure` run, but it is up to your compiler how it interprets the resulting mix.

3.5 Integrated Development Environments (IDEs) and debugging NUT

Much of NUT has been coded using classic editors of developers' preference, like `vi`, `nano`, Midnight Commander `mcedit`, `gedit`/`pluma`, NotePad++ and tools like `meld` or WinMerge for file comparison and merge.

Modern IDEs however do offer benefits, specifically for live debugging sessions in a more convenient fashion than with command-line `gdb` directly. They also simplify writing AsciiDoc files with real-time rendering support.

Note
Due to use of `libtool` wrappers in "autotools" driven projects, it may be tricky to attach the debugger (mixing the correct `LD_LIBRARY_PATH` or equivalent with a binary under a `.libs` subdirectory; on some platforms you may be better off copying shared objects to the directory with the binary being tested).

IDEs that were tested to work with NUT development and real-time debugger tracing include:

- Sun NetBeans 8.2 on Solaris, Linux (including local and remote build and debug ability);
- Apache NetBeans 17 on Windows with MSYS2 support (as MinGW toolkit);
- Visual Studio Code (VSCode) on Windows with MSYS2 support.

Some supporting maintenance and development is doable with IntelliJ IDEA, making some things easier to do than with a simple Notepad, but it does not handle C/C++ development as such.

3.5.1 IDE notes on Windows

General settings for builds on Windows

When working in a native Windows environment with MSYS2 (providing MinGW x64 among other things), you may need to ensure certain environment variables are set before you start the IDE (shortcuts and wrappers that start your console apply them via shell).



Warning
If you set such environment variables system-wide for your user profile (or wrap the IDE start-up by a script to set them), it may compromise your ability to use **other** MSYS2 profiles and/or other builds of these toolkits (packaged by e.g. Git for Windows or PERL for Windows projects) generally, or in the same IDE session, respectively. You may want to do this in a dedicated user account!

Examples below assume you installed MSYS2 into `C:\msys64` (by default) and are using the "MinGW X64" profile for GCC builds (nuances may differ for 32-bit, CLANG, UCRT and other profile variants).

Also keep in mind that not all dependencies and tools involved in a fully-fledged NUT build are easily available or usable on Windows (e.g. the spell checker). See the config-prereqs.txt for better detailed package lists for different operating systems including Windows, and feel welcome to post pull requests with suggestions about new tool-chains that might fare better than those already tried and documented.

- Make sure its tools are in the `PATH`:

  Control Panel ⇒ "Edit the system environment variables" ⇒ "Environment variables..." (button) ⇒ "Edit..." or create "New..." `Path` setting ("User variable" level suffices) ⇒

  - Make sure `C:\msys64\mingw64\bin` and `C:\msys64\usr\bin` are both there.
  - Depending on further installed toolkits, you may want to add `C:\Program Files\Git\cmd` or `C:\Program Files\Micr` `VS Code\bin` (preferably use deployment-dependent spellings without white-space like `Progra~1` to err on the safe side of variable expansions later).

- Make sure that MSYS2 (and tools which integrate with it) know its home:

  Open Environment variables window as above, and "Edit..." or create "New..." `MSYS_HOME` setting ⇒ Set to `C:\msys64\mingw6`
  * Restart the IDE (if already running) for it to acknowledge the system configuration change.

Otherwise, NetBeans for example claims there is no shell for it to run `make` or open Terminal pane windows, and fails to start the built programs due to lack of DLL files they were linked against (such as `libssl` usually needed for any networked part of the codebase).

You might still have to fiddle with DLL files built in other directories of the NUT project, when preparing to debug certain programs, e.g. for `dummy-ups` testing you may need to:

```
:; cp ./clients/.libs/libupsclient-6.dll ./drivers/.libs/
```

To ensure builds with debug symbols, you may add `CFLAGS` and `CXXFLAGS` set to `-g3 -gdwarf-2` or similar to `configure` options, or if that confuses the cross-build (it tends to assume those values are part of GCC path), you may have to hack them into your local copy of `configure.ac`, after the `AM_INIT_AUTOMAKE([subdir-objects])` line:

```
CFLAGS="$CFLAGS -g3 -gdwarf-2"
CXXFLAGS="$CXXFLAGS -g3 -gdwarf-2"
```

... and re-run the `./autogen.sh` script.

GDB on Windows

Examples below assume that whichever IDE you are using, the primary goal is to debug some issues with NUT on that platform.

This may require you to craft a configuration file for the GNU Debugger, e.g. `C:\Users\abuild\.gdbinit` for the examples below. One is not required however, and may be missing.

Another thing to keep in mind is that with `libtool` involved, the actual binary for testing would be in a `.libs` subdirectory and you may have some fun with ensuring that DLLs are found to start them — see the notes above.

NetBeans on Windows

When you install newer Apache NetBeans releases (14, 17 as of this writing), you may need to enable the use of "NetBeans 8.2 Plugin Portal" (check under Tools/Plugins/Settings) and install the "C/C++" plugin only available there at the moment. In turn, that older build of a plugin package may require that your system provides the `unpack200(.exe)` tool which was shipped with JDK11 or older (you may have to install that just to get the tool, or copy its binary from another system).

Under Tools/Options menu open the C/C++ tab and further its Build Tools sub-tab.

Note
NetBeans allows you to easily define different Tool Collections, including those associated with a different build host (accessible over SSH and source/build paths optionally shared over NFS or similar technology, or copied over). This allows you to run the IDE on your desktop while debugging a build running on a server or embedded system.

Make sure you have a MinGW Tool Collection for the "localhost" build host with such settings as:

Option name	Sample value
Family	GNU MinGW
Encoding	UTF-8
Base Directory	`C:\msys64\mingw64\bin`
C Compiler	`C:\msys64\mingw64\bin\gcc.exe`
C++ Compiler	`C:\msys64\mingw64\bin\g++.exe`
Assembler	`C:\msys64\mingw64\bin\as.exe`
Make Command	`C:\msys64\usr\bin\make.exe`
Debugger Command	`C:\msys64\mingw64\bin\gdb.exe`

In the Code Assistance sub-tab check that there are toolkit-specific and general include paths, e.g. both C and C++ Compiler settings might involve:

`C:\msys64\mingw64\lib\gcc\x86_64-w64-mingw32\12.2.0\include`
`C:\msys64\mingw64\include`
`C:\msys64\mingw64\lib\gcc\x86_64-w64-mingw32\12.2.0\include-fixed`
`C:\msys64\mingw64\x86_64-w64-mingw32\include`

On top of that, C++ Compiler settings may include:

`C:\msys64\mingw64\include\12.2.0`
`C:\msys64\mingw64\include\12.2.0\x86_64-w64-mingw32`
`C:\msys64\mingw64\include\12.2.0\backward`

In the "Other" sub-tab, set default standards to C99 and C++11 to match common NUT codebase expectations.

Finally, open/create a "nut" project pointing to your git checkout workspace.

Next part of configuration regards build/debug configurations, which you can find on the toolbar or as File / Project Properties.

The main configuration for debugging a particular binary (and NUT has tons of those, good luck in case you want to debug several simultaneously) is in the **Run** and **Debug** categories. You may want to define different Configuration profiles to track the individual Run/Debug settings for different tested binaries, while the Build/Make settings would remain the same. Alternatively, you may set the **Make** category's "Build Result" as the path to the binary you would test, and use `${OUTPUT_PATH}` variable as its name in the "Run Command" (still likely need custom arguments) and "Symbol File" below.

When you investigate interactions of two or more programs, but only want to debug (step through) just one of them, you are advised to run each of the others from a dedicated terminal session, and just bump their debug verbosity.

- In the **Build** category, set the Build Host (localhost) and Tool Collection (MinGW). In expert part of the settings, un-check "platform-independent" and revise that the `TOOLS_PATH=C:\msys64\mingw64\bin` while the `UTILITIES_PATH=C:\msys`
- In the **Pre-Build** category likely keep the Working Directory as `.` and the `Pre-Build First` generally unchecked (so only enable it to reconfigure the project, which takes time and is not needed for every rebuild iteration), but you may still pre-set the Command line to something like the following (on one line):

```
bash -c "rm -f configure Makefile; ./autogen.sh &&
    ./configure CC='${IDE_CC}' CXX='${IDE_CXX}'
        --with-all=auto --with-docs=skip"
```

  In some cases, NOT specifying the `CC`, `CXX` and the flags actually succeeds while passing their options fails the configuration ("Compiler can not create executables" etc.) probably due to path resolution issues between the native and MinGW environments.

  Note
  In practice, you may have an easier time using NUT `./ci_build.sh` helper or running a more specific `./autogen.sh && ./configure ...` spell similar to the above example or customized otherwise, in the MinGW x64 console window to actually configure a NUT source code setup, than to maintain one via the IDE. Running (re-)builds with the IDE (as you just edit non-recipe sources and iterate with a debugger) using externally configured Makefiles works fine.

- In the **Make** category you may want to customize for parallelized builds on multi-CPU systems with something like:
  - Build Command: `${MAKE} -j 6 -f Makefile`
  - Clean Command: `${MAKE} -f Makefile clean`
- In the **Run** category you should set the "Run Command" to point to your binary (note the `.libs` sub-directory, and see comments above regarding possibly needed copies of shared objects) and its arguments (all on one line), e.g.:

```
C:\Users\abuild\Desktop\nut\drivers\.libs\usbhid-ups.exe -s ups -x port=auto
    -d1 -DDDDDD
```

  Other useful settings may be to keep "Build First" checked, and if the "Internal Terminal" does not work for you as the debugged program's console — set the "Console Type" to "External Terminal" of type "Command Window". Unfortunately, NetBeans on Windows may have issues running terminal tabs unless CygWin is installed.

- In the **Debug** category you should set the "Symbol File" to point to your tested binary (e.g. `C:\Users\abuild\Desktop\nut\d` to match the "Run Command" example above) and specify "Follow Fork Mode" as "child" and "Detach On Fork" as "off". "Reverse Debugging" may be useful too in some situations. Finally, select your "Gdb Init File" if you have one, e.g. `C:\Users\abuild\.gdbinit`.

Microsoft VS Code

With this IDE you can benefit from numerous Extensions from its Marketplace, the ones found useful for NUT development and debugging include:

- AsciiDoc (by asciidoctor)
- EditorConfig for VS Code (by EditorConfig)
- C/C++ (by Microsoft)
- C/C++ Extension pack (by Microsoft)
- Makefile tool (by Microsoft)
- MSYS2/Cygwin/MinGW/Clang support (by okhlybov)
- Native Debug (GDB, LLDB ... Debugger support; by WebFreak)

Configurations are tracked locally in JSON files where you would need to add some entries. Examples below highlight the needed keys and values; your files may have others:

- `.vscode/launch.json` (can create one via Run/Add Configuration... menu defines ways to launch the debug session for a program:

```
{
    "configurations": [
        {
            "name": "CPPDBG GDB usbhid-ups",
            "type": "cppdbg",
            "request": "launch",
            "program": "C:\\Users\\abuild\\Desktop\\nut\\drivers\\.libs\\usbhid-ups.exe",
            "additionalSOLibSearchPath": "C:\\Users\\abuild\\Desktop\\nut\\.inst\\mingw64 ↩
                \\bin",
            "stopAtConnect": true,
            "args": ["-s", "ups", "-DDDDDD", "-d1", "-x", "port=auto"],
            "stopAtEntry": false,
            "cwd": "C:\\Users\\abuild\\Desktop\\nut",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "C:\\msys64\\mingw64\\bin\\gdb.exe",
            "targetArchitecture": "x64",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "make usbhid-ups"
        },
        {
            // Alternately with LLDB (clang), the rest looks like above:
            "name": "CPPDBG LLDB usbhid-ups",
            "MIMode": "lldb",
            "miDebuggerPath": "C:\\msys64\\usr\\bin\\lldb.exe",
```

```
        },
        ...
    ]
}
```

- `.vscode/tasks.json` defines other tasks, such as the `preLaunchTask` mentioned above (assuming you have configured the build externally in the MinGW x64 terminal session):

```
{
    "tasks": [
        {
            "type": "shell",
            "label": "make usbhid-ups",
            "command": "C:\\msys64\\usr\\bin\\make usbhid-ups",
            "options": {
                "cwd": "${workspaceFolder}/drivers"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        },
        ...
    ]
}
```

- `.vscode/c_cpp_properties.json` defines general compiler settings, e.g.:

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**",
                "C:\\msys64\\mingw64\\include\\libusb-1.0",
                "C:\\msys64\\mingw64\\include",
                "C:\\msys64\\usr\\include"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "C:\\msys64\\mingw64\\bin\\gcc.exe",
            "cStandard": "c99",
            "cppStandard": "c++11",
            "intelliSenseMode": "windows-gcc-x64",
            "configurationProvider": "ms-vscode.makefile-tools"
        }
    ],
    "version": 4
}
```

IntelliJ IDEA

It is worth mentioning IntelliJ IDEA as another free (as of Community Edition) and popular IDE, however it is of limited use for NUT development.

Its ecosystem does feature a good AsciiDoc plugin, Python and of course the Java/Groovy support, so IDEA is helpful for maintenance of NUT documentation, helper scripts and CI recipes.

It lacks however C/C++ language support (allegedly a different product in the IntelliJ portfolio is dedicated to that), so for the core NUT project sources it is just a fancy text editor (with `.editorconfig` support) without syntax highlighting or codebase cross-reference aids, build/run/debug support, etc.

Still, it is possible to run builds and tests in embedded or external terminal session — so it is not worse than editing with legacy tools, and navigation or code-base-wide search is arguably easier.

3.6 Coding style

This is how we do things:

```
int open_subspace(char *ship, int privacy)
{
        if (!privacy)
                return insecure_channel(ship);

        if (!init_privacy(ship))
                fatal_with_errno("Can't open secure channel");

        return secure_channel(ship);
}
```

The basic idea is that we try to group things into functions, and then find ways to drop out of them when we can't go any further. There's another way to program this involving a big else chunk and a bunch of braces, and it can be hard to follow. You can read this from top to bottom and have a pretty good idea of what's going on without having to track too much `{ }` nesting and indenting.

We don't really care for `pretentiousVariableNamingSchemes`, but you can probably get away with it in your own driver that we will never have to touch. If your function or variable names start pushing important code off the right margin of the screen, expect them to meet the byte chainsaw sooner or later.

All types defined with typedef should end in `_t`, because this is easier to read, and it enables tools (such as indent and emacs) to display the source code correctly.

3.6.1 Indenting with tabs vs. spaces

Another thing to notice is that the indenting happens with tabs instead of spaces. This lets everyone have their personal tab-width setting without inflicting much pain on other developers. If you use a space, then you've fixed the spacing in stone and have really annoyed half of the people out there.

Note that tabs apply only to **indenting**. Alignment of text after any non-tab character has appeared on the line must be done by spaces in order for it to remain at the same alignment when someone views tabs at a different widths.

One common example for this is multi-line if condition:

```
        if (something &&
            something_else) {
```

which may be written without mixing tabs and spaces to indent, as:

```
        if (something
        &&  something_else
        ) {
```

Another example is tables of definitions that are better aligned with (non-leading) spaces at least between names and values not too many characters wide; it still helps to align the columns with spaces at offsets divisible by 4 or 8 (consistently for the whole table):

```
#define SHORT_MACRO                             1   /* flag comment */
#define SOMETHING_WITH_A_VERY_LONG_NAME     255 /* flag comment */
```

If you write something that uses leading spaces, you may get away with it in a driver that's relatively secluded. However, if we have to work on that code, expect it to get reformatted according to the above.

Patches to existing code that don't conform to the coding style being used in that file will probably be dropped. If it's something we really need, it will be grudgingly reformatted before being included.

When in doubt, have a look at Linus's take on this topic in the Linux kernel — Documentation/CodingStyle. He's done a far better job of explaining this.

3.6.2 Line breaks

It is better to have lines that are longer than 80 characters than to wrap lines in random places. This makes it easier to work with tools such as `grep`, and it also lets each developer choose their own window size and tab setting without being stuck to one particular choice.

Of course, this does not mean that lines should be made unnecessarily long when there is a better alternative (see the note on `pretentiousVariableNamingSchemes` above). Certainly there should not be more than one statement per line. Please do not use

```
if (condition) break;
```

but use the following:

```
if (condition) {
        break;
}
```

Note
Earlier revisions of coding style might suggest avoiding braces if just one line is added as condition/loop/etc. handling code. Current approach is to welcome them even for single lines: on one hand, this confirms the intention that only this line is the conditional code; on another, this minimizes the context differences for later code comparisons, relocation, refactoring, etc.

3.6.3 Un-used variables and function arguments

Whenever a function needs to satisfy a particular API, it can end up taking arguments that are not used in practice (think a too-trivial signal handler). While some compilers offer the facility of decorations like `__attribute__(unused)`, this proved not to be a portable solution. Also the abilities of newer C++ standard revisions are of no help to the vast range of existing systems that run NUT today and expect to be able to do so tomorrow (hence the required C99+ support noted above).

In NUT codebase we prefer to mark un-used variables explicitly in the body of the function (or an `#ifdef` branch of its code) using the `NUT_UNUSED_VARIABLE(varname)` as a routine call inside a function body, referring to the macro defined in `common.h`.

Please note that for the purposes of legacy-compatible variable declarations (on top of their scopes), `NUT_UNUSED_VARIABLE(varna` counts as code and should happen below the declarations.

To display in a rough example:

```
        static void signal_X_handler(int signal_X) {
                NUT_UNUSED_VARIABLE(signal_X);
                /* We have explicitly got nothing to do if we catch signal X */
                return;
        }
```

3.7 Miscellaneous coding style tools

NUT codebase includes an `.editorconfig` file which should be supported by most of the IDEs and text editors nowadays. Many support this format specification (at least partially) out of the box, possibly with some configuration toggle in the GUI. Others may need a plugin, see more at https://editorconfig.org/#pre-installed page. There are also command-line tools to verify and/or enforce compliance of source files to configuration.

You can go a long way towards converting your source code to the NUT coding style by piping it through the following command:

```
indent -kr -i8 -T FILE -l1000 -nhnl
```

This next command does a reasonable job of converting most C++ style comments (but not URLs and DOCTYPE strings):

```
sed 's#\(^\|[ \t]\)//[ \t]*\(.*\)[ \t]*#/* \2 */#'
```

Emacs users can adjust how tabs are displayed. For example, it is possible to set a tab stop to be 3 spaces, rather than the usual 8. (Note that in the saved file, one indentation level will still correspond to one tab stop; the difference is only how the file is rendered on screen). It is even possible to set this on a per-directory basis, by putting something like this into your `.emacs` file:

```
;; NUT style

(defun nut-c-mode ()
 "C mode with adjusted defaults for use with the NUT sources."
 (interactive)
 (c-mode)
 (c-set-style "K&R")
 (setq c-basic-offset 3)  ;; 3 spaces C-indentation
 (setq tab-width 3))      ;; 3 spaces per tab

;; apply NUT style to all C source files in all subdirectories of nut/

(setq auto-mode-alist (cons '(".*/nut/.*\\.[ch]$". nut-c-mode)
                       auto-mode-alist))
```

3.7.1 Finishing touches

We like code that uses `const` and `static` liberally. If you don't need to expose a function or global variable to the outside world, `static` is your friend. If nobody should edit the contents of some buffer that's behind a pointer, `const` keeps them honest.

We always compile with `-Wall`, so things like `const` and `static` help you find implementation flaws. Functions that attempt to modify a constant or access something outside their scope will throw a warning or even fail to compile in some cases. This is what we want.

3.7.2 Switch case fall-through

While C standards allow to write `switch` statements to "fall through" from handling one case into another, modern compilers frown upon that practice and spew warnings which complicate detecting real bugs in the code (and also looking back at some of the cases written decades ago, it is not trivial to state whether the fall-through was intentional or really is a bug).

Compilers which detect such problem usually offer ways to decorate the code with comments or attributes to keep it quiet it in cases where the jump is intentional; also C++17 introduces special keywords for that in the standard. NUT aiming to be portable and independent of compilers as much as possible, prefers the arguably clearer and standards-based way of using `goto` into the next intended operation, even though it is a couple of lines away, e.g.:

```
int uppercase = 0;
switch (char_opt) {
        case 'U':
                uppercase = 1;
```

```
                goto fallthrough_case_u_option;
        case 'u':
        fallthrough_case_u_option:
                process_u_option(uppercase);
                break;
}
```

In trivial cases, like falling through to `default` which just returns, it may be clearer and more maintainable (adding other option cases in the future) to just `return same_result` in the code block that would fall through otherwise and avoid `goto` statements altogether.

3.7.3 Spaghetti

If you use a `goto` that jumps over long distances (see "Switch case fall-through" section above), expect us to drop it when our head stops spinning. It gives us flashbacks to the very old code we wrote. We've tried to clean up our act, and you should make the effort as well.

We're not making a blanket statement about gotos, since everything probably has at least one good use. There are a few cases where a goto is more efficient than any other approach, but you probably won't encounter them very often in this software.

3.7.4 Legacy code

There are parts of the source tree that do not yet conform to these specs. Part of this is due to the fact that the coding style has been evolving slightly over the course of the project. Some of the code you see in these directories is 5 years old, and things have gotten cleaner since then. Don't worry — it'll get cleaned up the next time something in the vicinity gets a visit.

3.7.5 Memory leak checking

We can't say enough good things about valgrind. If you do anything with dynamic memory in your code, you need to use this. Just compile with `gcc -g` and start the program inside `valgrind`. Run it through the suspected area and then exit cleanly. valgrind will tell you if you've done anything dodgy like freeing regions twice, reading uninitialized memory, or if you've leaked memory anywhere.

For more information, refer to the Valgrind project.

3.7.6 Conclusion

The summary: please be kind to our eyes. There's a lot of stuff in here, and many people have put a lot of time and energy to improve it.

3.8 Submitting patches

Current preference for suggesting changes is to open a pull request on GitHub for the https://github.com/networkupstools/nut/ project.

For some cases, small patches that arrive by mailing list in unified format (`diff -u`) as plain text attachments with no HTML and a brief summary at the top are easy to handle, but sadly also easy to overlook.

If a patch is sent to the nut-upsdev mailing list, it stands a better chance of being seen immediately. However, it is likely to be dropped if any issues cannot be resolved quickly. If your code might not work for others, or if it is a large change, your best bet is to submit a pull request or create an issue on GitHub.

The issue tracker allows us to track the patches over a longer period of time, and it is less likely that a patch will fall through the cracks. Posting a reminder to the developers (via the nut-upsdev list) about a patch on GitHub is fair game.

3.9 Patch cohesion

Patches should have some kind of unifying element. One patch set is one message, and it should all touch similar things. If you have to edit 6 files to add support for neutrino detection in UPS hardware, that's fine.

However, sending one huge patch that does massive separate changes all over the tree is not recommended. That kind of patch has to be split up and evaluated separately, assuming the core developers care enough to do that instead of just dropping it.

If you have to make big changes in lots of places, send multiple patches — one per item.

3.10 The finishing touches: manual pages and device entry in HCL

If you change something that involves an argument to a program or configuration file parsing, the man page is probably now out of date. If you don't update it, we have to, and we have enough to do as it is.

If you write a new driver, send in the man page when you send us the source code for your driver. Otherwise, we will be forced to write a skeletal man page that will probably miss many of the finer points of the driver and hardware.

The same remark goes for device entries: if you add support for new models, please remember to also complete the hardware compatibility list, present in data/driver.list.in. This will be used to generate both textual, static HTML and dynamic searchable HTML for the website.

Finally, don't forget about fame and glory: if you added or substantially updated a driver, your copyright belongs in the heading comment (along with existing ones). For vendor backed (or sponsored) contributions we welcome an entry in the docs/acknowledgements.txt file as well, to track and know the industry players who help make NUT better and more useful.

It is nice to update the NEWS file for significant development to be seen as part of next release, as well as to update the UPGRADING file for potentially breaking changes and similar heads-up notes for third-party teams (distribution packagers, clients and bindings, etc.)

3.11 Source code management

We currently use a Git repository hosted at GitHub to track changes to the NUT source code. This allows you to clone the repository (or fork, in GitHub parlance), make changes, and post them online for peer review prior to integration.

To obtain permission to commit directly to the common upstream NUT repository, you must be prepared to spend a fair amount of time contributing to the NUT codebase. Most developers will be well served by committing to their own forked Git repository (preferably in a uniquely named branch for each new contribution), and having the NUT team merge their changes using pull requests.

Git offers a little more flexibility than the `svn update` command. You may fetch other developers' changes into your repository, but hold off on actually combining them with your branch until you have compared the two branches (for instance, with `gitk --all`). Git also allows you to accumulate more than one commit worth of changes before pushing to another repository. This allows development to continue without a constant network connection.

For a quick change to a file in the Git working copy, you can use `git diff` to generate a patch to send to the nut-upsdev mailing list. If you have more extensive changes, you can use `git format-patch` on a complete commit or branch, and send the resulting series of patches to the list.

If you use GitHub's web-based editor to make changes, it tends to create lots of small commits, one per change per file. Unless there is reason to keep the intermediate history, we will probably collapse (or "squash" in Git parlance) the entire branch into one commit with a `git rebase -i` before merging.

The GitSvnCrashCourse wiki page has some useful information for long-time users of Subversion.

3.11.1 Git access

Anonymous Git checkouts are possible:

```
git clone git://github.com/networkupstools/nut.git
```

or

```
git clone https://github.com/networkupstools/nut.git
```

if it is necessary to get around a pesky firewall that blocks the native Git protocol.

For a quicker checkout (when you don't need the entire repository history), you can limit the depth of the clone:

```
git clone --depth 1 git://github.com/networkupstools/nut.git
```

3.11.2 Mercurial (hg) access

There are those who prefer the simplicity and self-consistency of the Mercurial SCM client over the hodgepodge of unique commands which make up Git. Rather than debate the merits of each system, we will gently guide you towards the hg-git project which would theoretically be a transparent bridge between the central Git repository, and your local Mercurial working copy.

Other tools for hg/git interoperability are sure to exist. We would welcome any feedback about this process on the nut-upsdev mailing list.

3.11.3 Subversion (SVN) access

If you prefer to check out the NUT source code using an SVN client, GitHub has a SVN interface to Git repositories hosted on their servers. You can fork a copy of the NUT repository and commit to your fork with SVN.

Be aware that the examples in the GitHub blog post might result in a checkout that includes all of the current branches, as well as the trunk. You are most likely interested in a command line similar to the following:

```
svn co https://github.com/networkupstools/nut/trunk nut-trunk-svn
```

3.12 Ignoring generated files

The NUT repository generally only holds files which are not generated from other files. This prevents spurious differences from being recorded in the repository history.

If you add a driver, it is recommended that you add the driver executable name to the `.gitignore` file in that directory. Similarly, files generated from `*.in` and `*.am` source templates should be ignored as well. We try to include a number of generated files in the tarball releases with `make dist` hooks in order to minimize the number of dependencies for end users, but the assumption is that a developer can install the packages needed to regenerate those files.

3.13 Commit message formatting

From the `git commit` man page:

> Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout git.

If your commit is just a change to one component, such as the HCL, upsd or a specific driver, prefix your commit message in a way that matches similar commits. This helps when searching the repository or tracking down a regression.

Referring to previous commits can be tricky. If you are referring to the immediate parent of a given commit, it suffices to say "the previous commit". (Are you correcting a typo in the previous commit? If you haven't pushed yet, consider using the `git commit --amend` command instead of creating a new commit.) For other commits, even though tools like gitk and GitHub's repository viewers recognize Git hashes and create links automatically, it is best to add some context such as the commit title or a date.

You may notice that some older commits have `[[SVN:####]]` tags and Fossil-ID footers. These were lifted from the old SVN commit messages using reposurgeon, and should **not** be used as a guide for future commits.

3.14 Commit sign-off

Please also note that since 2023 we explicitly ask for contributions to be "Signed Off" according to "Developer Certificate of Origin" as represented in the `LICENSE-DCO` file in the root of NUT source tree (verbatim copy of Version 1.1 of DCO published at https://developercertificate.org/ web site). This is exactly the same one created and used by the Linux kernel developers.

This is a developer's certification that he or she has the right to submit the patch for inclusion into the project. Simply submitting a contribution implies this agreement, however, please include a "Signed-off-by" tag in every patch (this tag is a conventional way to confirm that you agree to the DCO). In other words, this tag certifies that committer has the rights to submit this work under the same license as the project and agrees to the terms of a Developer Certificate of Origin.

Note that while git commit hook tricks are available to automatically sign off all commits, these signatures are intended to be a conscious (legally meaningful) act — hence they are not automated in git core with an easy configuration option.

For more details see:

- https://github.com/networkupstools/nut/issues/1994
- https://stackoverflow.com/questions/1962094/what-is-the-sign-off-feature-in-git-for
- https://stackoverflow.com/questions/15015894/git-add-signed-off-by-line-using-format-signoff-not-working

You are also encouraged to set up a PGP key, make its public part known, and use it to sign your git commits (in addition to the `Signed-Off-By` tag) by also passing a `-S` option or calling `git config commit.gpgsign true` once. Numerous public articles can walk you through this ordeal, including:

- https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits
- https://docs.github.com/en/authentication/managing-commit-signature-verification/telling-git-about-your-signing-key
- https://www.kernel.org/doc/html/v4.19/process/maintainer-pgp-guide.html

3.15 Repository etiquette and quality assurance

For developers who have commit access to the common upstream NUT repository: Please keep the Git "master" branch in working condition at all times. The "master" branch may be used to generate daily tarballs, it provides the baseline for new contributions, and occasionally is tagged for a new release. It should not contain broken code. If you need to commit incremental changes that leave the system in a broken state, please do so in a separate branch and merge the changes back into "master" once they are complete.

To help keep the codebase ever-green, we run a number of CI tests and builds in various conditions, including older compilers, different C/C++ standard revisions, and an assortment of operating systems; a section below elaborates on this in more detail.

You are encouraged to use `git rebase -i` on your private Git branches to separate your changes into logical changes.

From there, you can generate patches for the issue tracker, or the nut-upsdev mailing list.

Note that once you rebase a branch, anyone else who has a copy of this branch will need to rebase on top of your rebased branch. Obviously, this hinders collaboration. In this case, we recommend that you rebase only in your private repository, and push when things are ready for discussion. Merging instead of rebasing will help with collaboration, but please do not turn the repository history into a pile of spaghetti by merging unnecessarily. (Test merges can be done on integration branches, which can be discarded if the merge is trivial.) Be sure that your commit messages are descriptive when merging.

If you haven't created a commit out of your local changes yet, and you want to fetch the latest code, you can also use `git stash` before pulling, then `git stash pop` to apply your saved changes.

Here is an example workflow:

```
git clone -o central git://github.com/networkupstools/nut.git

cd nut
git remote add -f username git://github.com/username/nut.git
```

```
git checkout master
git branch my-new-feature
git checkout my-new-feature

# Hack away

git add changed-file.c
git commit -s

# Fix a typo in a file or commit message:

git commit -s -a --amend

# Someone committed something to the central repository. Fetch it.

git fetch central
git rebase central/master

# Publish your branch to your GitHub repository:

git push username my-new-feature
```

If you are new to Git, but are familiar with SVN, some of the following links may be of use:

- Git - SVN Crash Course (archived)
- Git and Other Systems - Migrating to Git
- Switching from Subversion to Git
- Migrate from SVN to Git

3.16 Building the Code

For a developer, the NUT build process starts with `./autogen.sh`.

This script generates the `./configure` script that end users typically invoke to build NUT. If you are making a number of changes to the NUT source tree, configuring with the `--enable-maintainer-mode` flag will ensure that after you change a `Makefile.am`, nearby `Makefile.in` and `Makefile` get regenerated. At a minimum, you will need at least:

- autoconf
- automake
- libtool
- Python
- Perl

Note
See the config-prereqs.txt for better detailed package lists for different operating systems.
See `ci_build.sh` for automating many practical scenarios, for easier iterations.
It is optional, but highly recommended, to have Python 2.x or 3.x, and Perl, to generate some files included into the `configure` script whose presence is checked by autotools when it is generated. Neutered files can be just "touched" to pass the `autogen.sh` if these interpreters are not available, and effectively skip those parts of the build later on — `autogen.sh` will then advise which special environment variables to `export` in your situation and re-run it.

Even if you do not use your distribution's packages of NUT, installing the distribution's list of build dependencies for NUT can reduce the amount of trial-and-error when installing dependencies. For instance, in Debian, you can run `apt-get build-dep nut` to install all of the auto* tools as well as any development libraries and headers.

After running `./autogen.sh`, you can pass your local configuration options to `./configure` and run `make` from the top-level directory. To avoid the need for root privileges when testing new NUT code, you may wish to use `--prefix=$HOME/local/nu --with-statepath=/tmp`. You can also keep compilation times down by only building the driver which you are currently working on: `--with-drivers=driver1,dummy-ups`.

Before pushing your commits upstream, please run `make distcheck-light`. This checks that the Makefiles are not broken, that all the relevant files are distributed, and that there are no compilation or installation errors. Note that unless you specifically pass `--with-doc=skip` to `configure`, this requires all of the dependencies necessary to build the documentation to be locally installed on your system, including `asciidoc`, `a2x`, `xsltproc`, `dblatex` and any additional XSL stylesheets.

Running `make distcheck-light` is especially important if you have added or removed files, or updated `configure.ac` or some `Makefile.am` file. Remember: simply adding a file to Git does not mean it will be distributed. To distribute a file, you must update the corresponding `Makefile.am` with `EXTRA_DIST` entry and possibly other recipe handling.

There is also `make distcheck`, which runs an even stricter set of tests than `make distcheck-light`, but will not work unless you have all of the optional third-party libraries and features installed.

Finally note, that since 2017 the GitHub upstream project is monitored by Travis CI (in addition to earlier multi-platform buildbots which occasionally do not work), replaced since 2021 by a dedicated NUT CI farm. This means that if your posted improvements are based on current NUT "master" branch, the resulting pull request should get tested for a number of scenarios automatically. If your code adds a substantial feature, consider extending the `Jenkinsfile-dynamatrix` and/or `ci_build.sh` scripts in the workspace root to add another `BUILD_TYPE` to the matrix of tests run in parallel.

4 Creating a new driver to support another device

This chapter will present the process of creating a new driver to support another device.

Since NUT already supports many major power device protocols through several generic drivers (genericups, usbhid-ups, snmp-ups, blazer_* and nutdrv_qx), creation of new drivers has become rare.

So most of the time, this process will be limited to completing one of these generic drivers.

4.1 Smart vs. Contact-closure

If your UPS only does contact closure readings over an RS-232 serial port, then go straight to the Contact closure hardware chapter for information on adding support. It's a lot easier to add a few lines to a header file than it is to create a whole new driver.

4.2 Serial vs. USB vs. SNMP and more

If your UPS connects to your computer via a USB port, then it most likely appears as a USB HID device (this is the simplest way for the vendor to write a Windows control program for it). What comes next depends on whether the vendor implemented the HID PDC (Power Device Class) specification, or simply used the HID protocol to transport serial data to the UPS microcontroller.

A rough heuristic is to check the length of the HID Descriptor length (`wDescriptorLength` in `lsusb -v` output). If it is less than 200 bytes long, the UPS probably has a glorified USB-to-serial converter built in. Since the query strings often start with the letter `Q`, this family of protocols is often referred to as `Q*` in the NUT documentation. See the Q* UPS chapter for more details.

Otherwise, if the HID Descriptor is longer, you can go to the HID subdrivers chapter. You can probably add support for your device by writing a new subdriver to the existing usbhid-ups driver, which is easier (and more maintainable) than writing an entire new driver.

If your USB UPS does not appear to fall into either of these two categories, feel free to contact the `nut-upsdev` mailing list with details of your device.

Similarly, if your UPS connects to your computer via an SNMP network card, you can probably add support for your device by adding a new subdriver to the existing snmp-ups driver. Instructions are provided in the SNMP subdrivers chapter.

4.3 Overall concept

The basic design of drivers is simple. `main.c` handles most of the work for you. You don't have to worry about arguments, config files, or anything else like that. Your only concern is talking to the hardware and providing data to the outside world.

4.4 Skeleton driver

Familiarize yourself with the design of `skel.c` in the drivers directory. It shows a few examples of the functions that `main.c` will call to obtain updated information from the hardware.

4.5 Essential structure

4.5.1 upsdrv_info_t

This structure tracks several description information about the driver:

- **name**: the driver full name, for banner printing and "driver.name" variable.
- **version**: the driver's own version. For sub driver information, refer below to sub_upsdrv_info. This value has the form "X.YZ", and is published by main as "driver.version.internal".
- **authors**: the driver's author(s) name. If multiple authors are listed, separate them with a newline character so that it can be broken up by author if needed.
- **status**: the driver development status. The following values are allowed:
  - DRV_BROKEN: setting this value will cause main to print an error and exit. This is only used during conversions of the driver core to keep users from using drivers which have not been converted. Drivers in this state will be removed from the tree after some period if they are not fixed.
  - DRV_EXPERIMENTAL: set this value if your driver is potentially broken. This will trigger a warning when it starts so the user doesn't take it for granted.
  - DRV_BETA: this value means that the driver is more stable and complete. But it is still not recommended for production systems.
  - DRV_STABLE: the driver is suitable for production systems, but not 100 % feature complete.
  - DRV_COMPLETE: this is the gold level! It implies that 100 % of the protocol is implemented, and a full QA pass.
- **subdrv_info**: array of upsdrv_info_t for sub driver(s) information. For example, this is used by usbhid-ups.

This information is currently used for the startup banner printing and tests.

4.6 Essential functions

4.6.1 upsdrv_initups

Open the port (`device_path`) and do any low-level things that it may need to start using that port. If you have to set DTR or RTS on a serial port, do it here.

Don't do any sort of hardware detection here, since you may be going into upsdrv_shutdown next.

4.6.2 upsdrv_initinfo

Try to detect what kind of UPS is out there, if any, assuming that's possible for your hardware. If there is a way to detect that hardware and it doesn't appear to be connected, display an error and exit. This is the last time your driver is allowed to bail out.

This is usually a good place to create variables like `ups.mfr`, `ups.model`, `ups.serial`, and other "one time only" items.

4.6.3 upsdrv_updateinfo

Poll the hardware, and update any variables that you care about monitoring. Use `dstate_setinfo()` to store the new values.

Do at most one pass of the variables. You MUST return from this function or upsd will be unable to read data from your driver. main will call this function at regular intervals.

Don't spent more than a couple of seconds in this function. Typically five (5) seconds is the maximum time allowed before you risk that the server declares the driver stale. If your UPS hardware requires a timeout period of several seconds before it answers, consider returning from this function after sending a command immediately and read the answer the next time it is called.

You must never abort from upsdrv_updateinfo(), even when the UPS doesn't seem to be attached anymore. If the connection with the UPS is lost, the driver should retry to re-establish communication for as long as it is running. Calling `exit()` or any of the `fatal*()` functions is specifically not allowed anymore.

4.6.4 upsdrv_shutdown

Do whatever you can to make the UPS power off the load but also return after the power comes back on. You may use a different command that keeps the UPS off if the user has requested that with a configuration setting.

You should attempt the UPS shutdown command even if the UPS detection fails. If the UPS does not shut down the load, then the user is vulnerable to a race if the power comes back on during the shutdown process.

This method should not directly `exit()` the driver program (neither should it call `fatalx()` nor `fatal_with_errno()` methods). It can `upslogx(LOG_ERR, ...)` or `upslog_with_errno(LOG_ERR, ...)`, and then `set_exit_flag(N)` if required (`-1` for `EXIT_FAILURE` and `-2` for `EXIT_SUCCESS` which would be handled in the standard driver loop or `forceshutdown()` method of `main.c`).

4.7 Data types

To be of any use, you must supply data in `ups.status`. That is the minimum needed to let upsmon do its job. Whenever possible, you should also provide anything else that can be monitored by the driver. Some obvious things are the manufacturer name and model name, voltage data, and so on.

If you can't figure out some value automatically, use the `ups.conf` options to let the user tell you. This can be useful when a driver needs to support many similar hardware models, but can't probe to see what is actually attached.

4.8 Manipulating the data

All status data lives in structures that are managed by the dstate functions. All access and modifications must happen through those functions. Any other changes are forbidden, as they will not pushed out as updates to things like upsd.

4.8.1 Adding variables

```
dstate_setinfo("ups.model", "Mega-Zapper 1500");
```

Many of these functions take format strings, so you can build the new values right there:

```
dstate_setinfo("ups.model", "Mega-Zapper %d", rating);
```

4.8.2 Setting flags

Some variables have special properties. They can be writable, and some are strings. The ST_FLAG_* values can be used to tell upsd more about what it can do.

```
dstate_setflags("input.transfer.high", ST_FLAG_RW);
```

4.8.3 Status data

UPS status flags like on line (OL) and on battery (OB) live in ups.status. Don't manipulate this by hand. There are functions which will do this for you.

```
status_init() -- before doing anything else

status_set(val) -- add a status word (OB, OL, etc)

status_commit() -- push out the update
```

Possible values for status_set:

```
OL      -- On line (mains is present)
OB      -- On battery (mains is not present)
LB      -- Low battery
HB      -- High battery
RB      -- The battery needs to be replaced
CHRG    -- The battery is charging
DISCHRG -- The battery is discharging (inverter is providing load power)
BYPASS  -- UPS bypass circuit is active -- no battery protection is available
CAL     -- UPS is currently performing runtime calibration (on battery)
OFF     -- UPS is offline and is not supplying power to the load
OVER    -- UPS is overloaded
TRIM    -- UPS is trimming incoming voltage (called "buck" in some hardware)
BOOST   -- UPS is boosting incoming voltage
FSD     -- Forced Shutdown (restricted use, see the note below)
```

Anything else will not be recognized by the usual clients. Coordinate with the nut-upsdev list before creating something new, since there will be duplication and ugliness otherwise.

Note

- upsd injects `FSD` by itself following that command by a primary upsmon process. Drivers must not set that value, apart from specific cases (see below).
- As an exception, drivers may set `FSD` when an imminent shutdown has been detected. In this case, the "on battery + low battery" condition should not be met. Otherwise, setting status to `OB LB` should be preferred.
- the `OL` and `OB` flags are an indication of the input line status only.
- the `CHRG` and `DISCHRG` flags are being replaced with `battery.charger.status`. See the NUT command and variable naming scheme for more information.

4.9 UPS alarms

These work like ups.status, and have three special functions which you must use to manage them.

```
alarm_init() -- before doing anything else

alarm_set() -- add an alarm word

alarm_commit() -- push the value into ups.alarm
```

Note
the ALARM flag in ups.status is automatically set whenever you use alarm_set. To remove that flag from ups.status, call alarm_init and alarm_commit without calling alarm_set in the middle.

You should never try to set or unset the ALARM flag manually.

If you use UPS alarms, the call to status_commit() should be after alarm_commit(), otherwise there will be a delay in setting the ALARM flag in ups.status.

There is no official list of alarm words as of this writing, so don't use these functions until you check with the upsdev list.

Also refer to the NUT daisychain support notes chapter of the user manual and developer guide for information related to alarms handling in daisychain mode.

4.10 Staleness control

If you're not talking to a polled UPS, then you must ensure that it is still out there and is alive before calling dstate_dataok(). Even if nothing is changing, you should still "ping" it or do something else to ensure that it is really available. If the attempts to contact the UPS fail, you must call dstate_datastale() to inform the server and clients.

- dstate_dataok()

  You must call this if polls are succeeding. A good place to call this is the bottom of upsdrv_updateinfo().

- dstate_datastale()

  You must call this if your status is unusable. A good technique is to call this before exiting prematurely from upsdrv_updateinfo().

Don't hide calls to these functions deep inside helper functions. It is very hard to find the origin of staleness warnings, if you call these from various places in your code. Basically, don't call them from any other function than from within upsdrv_updateinfo(). There is no need to call either of these regularly as was stated in previous versions of this document (that requirement has long gone).

4.11 Serial port handling

Drivers which use serial port functions should include serial.h and use these functions (and cross-platform data types) whenever possible:

- TYPE_FD

Cross-platform data type to represent a serial-port connection.

- ERROR_FD_SER

Macro value representing an invalid serial-port connection.

- VALID_FD_SER(TYPE_FD_SER fd)

This macro evaluates to `true` if `fd` currently has a "valid" value (e.g. represents a connected device). You should invalidate the `fd` when you initialize the variable or close the connection, by assigning `fd = ERROR_FD`.

- INVALID_FD_SER(TYPE_FD_SER fd)

This macro evaluates to `true` if `fd` does not currently have a "valid" value.

- TYPE_FD_SER ser_open(const char *port)

This opens the port and locks it if possible, using one of fcntl, lockf, or uu_lock depending on what may be available. If something fails, it calls fatal for you. If it succeeds, it always returns the fd that was opened.

- TYPE_FD_SER ser_open_nf(const char *port)

This is a non-fatal version of ser_open(), that does not call fatal if something fails.

- int ser_set_speed(TYPE_FD_SER fd, const char *port, speed_t speed)

This sets the speed of the port and also does some basic configuring with tcgetattr and tcsetattr. If you have a special serial configuration (other than 8N1), then this may not be what you want.

The port name is provided again here so failures in tcgetattr() provide a useful error message. This is the only place that will generate a message if someone passes a non-serial port /dev entry to your driver, so it needs the extra detail.

- int ser_set_speed_nf(TYPE_FD_SER fd, const char *port, speed_t speed)

This is a non-fatal version of ser_set_speed(), that does not call fatal if something fails.

- int ser_set_dtr(TYPE_FD_SER fd, int state)
- int ser_set_rts(TYPE_FD_SER fd, int state)

These functions can be used to set the modem control lines to provide cable power on the RS232 interface. Use state = 0 to set the line to 0 and any other value to set it to 1.

- int ser_get_dsr(TYPE_FD_SER fd)
- int ser_get_cts(TYPE_FD_SER fd)
- int ser_get_dcd(TYPE_FD_SER fd)

These functions read the state of the modem control lines. They will return 0 if the line is logic 0 and a non-zero value if the line is logic 1.

- int ser_close(TYPE_FD_SER fd, const char *port)

This function unlocks the port if possible and closes the fd. You should call this in your upsdrv_cleanup handler.

- ssize_t ser_send_char(TYPE_FD_SER fd, unsigned char ch)

This attempts to write one character and returns the return value from write. You could call write directly, but using this function allows for future error handling in one place.

- ssize_t ser_send_pace(TYPE_FD_SER fd, useconds_t d_usec, const char *fmt, ...)

If you need to send a formatted buffer with an intercharacter delay, use this function. There are a number of UPS controllers which can't take commands at the full speed that would normally be possible at a given bit rate. Adding a small delay usually turns a flaky UPS into a solid one.

The return value is the number of characters that was sent to the port, or -1 if something failed.

- ssize_t ser_send(TYPE_FD_SER fd, const char *fmt, ...)

Like ser_send_pace, but without a delay. Only use this if you're sure that your UPS can handle characters at the full line rate.

- ssize_t ser_send_buf(TYPE_FD_SER fd, const void *buf, size_t buflen)

This sends a raw buffer to the fd. It is typically used for binary transmissions. It returns the results of the call to write.

- ssize_t ser_send_buf_pace(TYPE_FD_SER fd, useconds_t d_usec, const void *buf, size_t buflen)

This is just ser_send_buf with an intercharacter delay.

- ssize_t ser_get_char(TYPE_FD_SER fd, void *ch, time_t d_sec, useconds_t d_usec)

This will wait up to d_sec seconds + d_usec microseconds for one character to arrive, storing it at ch. It returns 1 on success, -1 if something fails and 0 on a timeout.

Note
the delay value must not be too large, or your driver will not get back to the usual idle loop in main in time to answer the PINGs from upsd. That will cause an oscillation between staleness and normal behavior.

- ssize_t ser_get_buf(TYPE_FD_SER fd, void *buf, size_t buflen, time_t d_sec, useconds_t d_usec)

Like ser_get_char, but this one reads up to buflen bytes storing all of them in buf. The buffer is zeroed regardless of success or failure. It returns the number of bytes read, -1 on failure and 0 on a timeout.

This is essentially a single read() function with a timeout.

- ssize_t ser_get_buf_len(TYPE_FD_SER fd, void *buf, size_t buflen, time_t d_sec, useconds_t d_usec)

Like ser_get_buf, but this one waits for buflen bytes to arrive, storing all of them in buf. The buffer is zeroed regardless of success or failure. It returns the number of bytes read, -1 on failure and 0 on a timeout.

This should only be used for binary reads. See ser_get_line for protocols that are terminated by characters like CR or LF.

- ssize_t ser_get_line(TYPE_FD_SER fd, void *buf, size_t buflen, char endchar, const char *ignset, time_t d_sec, useconds_t d_usec)

This is the reading function you should use if your UPS tends to send responses like "OK\r" or "1234\n". It reads up to buflen bytes and stores them in buf, but it will return immediately if it encounters endchar. The endchar will not be stored in the buffer. It will also return if it manages to collect a full buffer before reaching the endchar. It returns the number of bytes stored in the buffer, -1 on failure and 0 on a timeout.

If the character matches the ignset with strchr(), it will not be added to the buffer. If you don't need to ignore any characters, just pass it an empty string — `""`.

The buffer is always cleared and is always `null`-terminated. It does this by reading at most `(buflen - 1)` bytes.

Note
any other data which is read after the endchar in the serial buffer will be lost forever. As a result, you should not use this unless your UPS uses a polled protocol.

Let's say your endchar is `\n` and your UPS sends `"OK\n1234\nabcd\n"`. This function will `read()` all of that, find the first `\n`, and stop there. Your driver will get `"OK"`, and the rest is gone forever.

This also means that you should not "pipeline" commands to the UPS. Send a query, then read the response, then send the next query.

- ssize_t ser_get_line_alert(TYPE_FD_SER fd, void *buf, size_t buflen, char endchar, const char *ignset, const char *alertset, void handler(char ch), time_t d_sec, useconds_t d_usec)

This is just like ser_get_line, but it allows you to specify a set of alert characters which may be received at any time. They are not added to the buffer, and this function will call your handler function, passing the character as an argument.

Implementation note: this function actually does all of the work, and ser_get_line is just a wrapper that sets an empty alertset and a NULL handler.

- ssize_t ser_flush_in(TYPE_FD_SER fd, const char *ignset, int verbose)

This function will drain the input buffer. If verbose is set to a positive number, then it will announce the characters which have been read in the syslog. You should not set verbose unless debugging is enabled, since it could be very noisy.

This function returns the number of characters which were read, so you can check for extra bytes by looking for a nonzero return value. Zero will also be returned if the read fails for some reason.

- int ser_flush_io(TYPE_FD_SER fd)

This function drains both the in- and output buffers. Return zero on success.

- void ser_comm_fail(const char *fmt, ...)

Call this whenever your serial communications fail for some reason. It takes a format string, so you can use variables and other things to clarify the error. This function does built-in rate-limiting so you can't spam the syslog.

By default, it will write 10 messages, then it will stop and only write 1 in 100. This allows the driver to keep calling this function while the problem persists without filling the logs too quickly.

In the old days, drivers would report a failure once, and then would be silent until things were fixed again. Users had to figure out what was happening by finding that single error message, or by looking at the repeated complaints from upsd or the clients.

If your UPS frequently fails to acknowledge polls and this is a known situation, you should make a couple of attempts before calling this function.

Note
this does not call dstate_datastale. You still need to do that.

- void ser_comm_good(void)

This will clear the error counter and write a "re-established" message to the syslog after communications have been lost. Your driver should call this whenever it has successfully contacted the UPS. A good place for most drivers is where it calls dstate_dataok.

4.12 USB port handling

Drivers which use USB functions should include usb-common.h and use these:

4.12.1 Structure and macro

You should use the usb_device_id_t structure, and the USB_DEVICE macro to declare the supported devices. This allows the automatic extraction of USB information, to generate the Hotplug, udev and UPower support files.

The structure allows to convey `uint16_t` values of VendorID and ProductID, and an optional matching-function callback to interrogate the device in more detail (constrain known supported firmware versions, OEM brands, etc.)

For example:

```
/* SomeVendor name */
#define SOMEVENDOR_VENDORID             0xXXXX

/* USB IDs device table */
static usb_device_id_t sv_usb_device_table [] = {
        /* some models 1 */
        { USB_DEVICE(SOMEVENDOR_VENDORID, 0xYYYY), NULL },

        /* various models */
        { USB_DEVICE(SOMEVENDOR_VENDORID, 0xZZZZ), NULL },
        { USB_DEVICE(SOMEVENDOR_VENDORID, 0xAAAA), NULL },

        /* Terminating entry */
        { 0, 0, NULL }
};
```

4.12.2 Function

- is_usb_device_supported(usb_device_id_t *usb_device_id_list, USBDevice_t *device)

Call this in your device opening / matching function. Pass your usb_device_id_t list structure, and a set of VendorID, DeviceID, as well as Vendor, Product and Serial strings, possibly also Bus, bcdDevice (device release number) and Device name on the bus strings, in the USBDevice_t fields describing the specific piece of hardware you are inspecting.

This function returns one of the following value:

- NOT_SUPPORTED (0),
- POSSIBLY_SUPPORTED (1, returned when the VendorID is matched, but the DeviceID is unknown),
- or SUPPORTED (2).

For implementation examples, refer to the various USB drivers, and search for the above patterns.

Note
This set of USB helpers is due to expand is the near future...

4.13 Variable names

PLEASE don't make up new variables and commands just because you can. The new dstate functions give us the power to create just about anything, but that is a privilege and not a right. Imagine the mess that would happen if every developer decided on their own way to represent a common status element.

Check the NUT command and variable naming scheme section first to find the closest fit. If nothing matches, contact the upsdev list, and we'll figure it out.

Patches which introduce unlisted names may be modified or dropped.

4.14 Message passing support

upsd can call drivers to store values in read/write variables and to kick off instant commands. This is how you register handlers for those events.

The driver core (drivers/main.c) has a structure called upsh. You should populate it with function pointers in your upsdrv_initinfo() function. Right now, there are only two possibilities:

- setvar = setting UPS variables (SET VAR protocol command)
- instcmd = instant UPS commands (INSTCMD protocol command)

4.14.1 SET

If your driver's function for handling variable set events is called my_ups_set(), then you'd do this to add the pointer:

```
upsh.setvar = my_ups_set;
```

my_ups_set() will receive two parameters:

```
const char * -- the variable being changed
const char * -- the new value
```

You should return either STAT_SET_HANDLED if your driver recognizes the command, or STAT_SET_UNKNOWN if it doesn't. Other possibilities will be added at some point in the future.

4.14.2 INSTCMD

This works just like the set process, with slightly different values arriving from the server.

```
upsh.instcmd = my_ups_cmd;
```

Your function will receive two args:

```
const char * -- the command name
const char * -- (reserved)
```

You should return either STAT_INSTCMD_HANDLED or STAT_INSTCMD_UNKNOWN depending on whether your driver can handle the requested command.

4.14.3 Notes

Use strcasecmp. The command names arriving from upsd should be treated without regards to case.

4.14.4 Responses

Drivers will eventually be expected to send responses to commands. Right now, there is no channel to get these back through upsd to the client, so this is not implemented.

This will probably be implemented with a polling scheme in the clients.

4.15 Enumerated types

If you have a variable that can have several specific values, it is enumerated. You should add each one to make it available to the client:

```
dstate_addenum("input.transfer.low", "92");
dstate_addenum("input.transfer.low", "95");
dstate_addenum("input.transfer.low", "99");
dstate_addenum("input.transfer.low", "105");
```

4.16 Range values

If you have a variable that support values comprised in one or more ranges, you should add each one to make it available to the client:

```
dstate_addrange("input.transfer.low", 90, 95);
dstate_addrange("input.transfer.low", 100, 105);
```

4.17 Writable strings

Strings that may be changed by the client should have the ST_FLAG_STRING flag set, and a maximum length (in bytes) set in the auxdata.

```
dstate_setinfo("ups.id", "Big UPS");
dstate_setflags("ups.id", ST_FLAG_STRING | ST_FLAG_RW);
dstate_setaux("ups.id", 8);
```

If the variable is not writable, don't bother with the flags or the auxiliary data. It won't be used.

4.18 Instant commands

If your hardware and driver can support a command, register it.

```
dstate_addcmd("load.on");
```

4.19 Delays and ser_* functions

The new ser_* functions may perform reads faster than the UPS is able to respond in some cases. This means that your driver will call select() and read() numerous times if your UPS responds in bursts. This also depends on how fast your system is.

You should check your driver with `strace` or its equivalent on your system. If the driver is calling read() multiple times, consider adding a call to usleep before going into the ser_read_* call. That will give it a chance to accumulate so you get the whole thing with one call to read without looping back for more.

This is not a request to save CPU time, even though it may do that. The important part here is making the strace/ktrace output easier to read.

```
write(4, "Q1\r", 3)                     = 3
nanosleep({0, 300000000}, NULL)         = 0
select(5, [4], NULL, NULL, {3, 0})      = 1 (in [4], left {3, 0})
read(4, "(120.0 084.0 120.0   0 60.0 22.6"..., 64) = 47
```

Without that delay, that turns into a mess of selects and reads. The select returns almost instantly, and read gets a tiny chunk of the data. Add the delay and you get a nice four-line status poll.

4.20 Canonical input mode processing

If your UPS uses "\n" and/or "\r" as endchar, consider the use of Canonical Input Mode Processing instead of the ser_get_line* functions.

Using a serial port in this mode means that select() will wait until a full line is received (or times out). This relieves you from waiting between sending a command and reading the reply. Another benefit is, that you no longer have to worry about the case that your UPS sends "OK\n1234\nabcd\n". This will be broken up cleanly in "OK\n", "1234\n" and "abcd\n" on consecutive reads, without risk of losing data (which is an often forgotten side effect of the ser_get_line* functions).

Currently, an example how this works can be found in the safenet and upscode2 drivers. The first uses a single "\r" as endchar, while the latter accepts either "\n", "\n\r" or "\r\n" as line termination. You can define other termination characters as well, but can't undefine "\r" and "\n" (so if you need these as data, this is not for you).

4.21 Adding the driver into the tree

In order to build your new driver, it needs to be added to `drivers/Makefile.am`. At the moment, there are several driver list variables corresponding to the general protocol of the driver (`SERIAL_DRIVERLIST`, `SNMP_DRIVERLIST`, etc.). If your driver does not fit into one of these categories, please discuss it on the nut-upsdev mailing list.

There are also `*_SOURCES` and optional `*_LDADD` variables to list the source files, and any additional linker flags. If your driver uses the C math library, be sure to add `-lm`, since this flag is not always included by default on embedded systems.

When you add a driver to one of these lists, pay attention to the backslash continuation characters (`\\`) at the end of the lines.

The `automake` program converts the `Makefile.am` files into `Makefile.in` files to be processed by `./configure`. See the discussion in Section 3.16 about automating the rebuild process for these files.

4.22 Contact closure hardware information

This is a collection of notes that apply to contact closure UPS hardware, specifically those monitored by the genericups driver.

4.22.1 Definitions

"Contact closure" refers to a situation where one line is connected to another inside UPS hardware to indicate some sort of situation. These can be relays, or some other form of switching electronics. The generic idea is that you either have a signal on a line, or you don't. Think binary.

Usually, the source for a signal is the host PC. It provides a high (logic level 1) from one of its outgoing lines, and the UPS returns it on one or more lines to communicate. The rest of the time, the UPS either lets it float or connects it to the ground to indicate a 0.

Other equipment generates the high and low signals internally, and does not require cable power. These signals just appear on the right lines without any special configuration on the PC side.

4.22.2 Bad levels

Some evil cabling and UPS equipment uses the transmit or receive lines as their reference points for these signals. This is not sufficient to register as a high signal on many serial ports. If you have problems reading certain signals on your system, make sure your UPS isn't trying to do this.

4.22.3 Signals

Unlike their smarter cousins, this kind of UPS can only give you very simple yes/no answers. Due to the limited number of serial port lines that can be used for this purpose, you typically get two pieces of data:

1. "On line" or "on battery"
2. "Battery OK" or "Low battery"

That's it. Some equipment actually swaps the second one for a notification about whether the battery needs to be replaced, which makes life interesting for those users.

Most hardware also supports an outgoing signal from the PC which means "shut down the load immediately". This is generally implemented in such a way that it only works when running on battery. Most hardware or cabling will ignore the shutdown signal when running on line power.

4.22.4 New genericups types

If none of the existing types in the genericups driver work completely, make a note of which ones (if any) manage to work partially. This can save you some work when creating support for your hardware.

Use that information to create a list of where the signals from your UPS appear on the serial port at the PC end, and whether they are active high or active low. You also need to know what outgoing lines, if any, need to be raised in order to supply power to the contacts. This is known as cable power. Finally, if your UPS can shut down the load, that line must also be identified.

There are only 4 incoming and 2 outgoing lines, so not many combinations are left. The other lines on a typical 9 pin port are transmit, receive, and the ground. Anything trying to do a high/low signal on those three is beyond the scope of the genericups driver. The only exception is an outgoing BREAK, which we already support.

When editing the genericups.h, the values have the following meanings:

Outgoing lines:

- line_norm = what to set to make the line "normal" — i.e. cable power
- line_sd = what to set to make the UPS shut down the load

Incoming lines:

- line_ol = flag that appears for on line / on battery
- val_ol = value of that flag when the UPS is on battery
- line_bl = flag that appears for low battery / battery OK
- val_bl = value of that flag when the battery is low
- line_rb = flag that appears for battery health
- val_rb = value of that flag when the battery needs a replacement
- line_bypass = flag that appears for battery bypass / battery protection active
- val_bypass = value of that flag when the battery is bypassed / missing

This may seem a bit confusing to have two variables per value that we want to read, but here's how it works. If you set line_ol to TIOCM_RNG, then the value of TIOCM_RNG (0x080 on my box) will be anded with the value of the serial port whenever a poll occurs. If that flag exists, then the result of the and will be 0x80. If it does not exist, the result will be 0.

So, if line_ol = foo, then val_ol can only be foo or 0.

As a general case, if *line_ol == val_ol*, then the value you're reading is active high. Otherwise, it's active low. Check out the guts of upsdrv_updateinfo() to see how it really works.

4.22.5 Custom definitions

Late in the 1.3 cycle, a feature was merged which allows you to create custom monitoring settings without editing the model table. Just set upstype to something close, then use settings in ups.conf to adjust the rest. See the genericups(8) man page for more details.

4.23 How to make a new subdriver to support another USB/HID UPS

4.23.1 Overall concept

USB (Universal Serial Port) devices can be divided into several different classes (audio, imaging, mass storage etc). Almost all UPS devices belong to the "HID" class, which means "Human Interface Device", and also includes things like keyboards and mice. What HID devices have in common is a particular (and very flexible) interface for reading and writing information (such as X/Y coordinates and button states, in the case of a mouse, or voltages and status information, in the case of a UPS).

The NUT "usbhid-ups" driver is a meta-driver that handles all HID UPS devices. It consists of a core driver that handles most of the work of talking to the USB hardware, and several sub-drivers to handle specific UPS manufacturers (MGE, APC, and Belkin are currently supported). Adding support for a new HID UPS device is easy, because it requires only the creation of a new sub-driver.

There are a few USB UPS devices that are not true HID devices. These devices typically implement some version of the manufacturer's serial protocol over USB (which is a really dumb idea, by the way). An example is the original Tripplite USB interface (USB idProduct = 0001). Its HID descriptor is only 52 bytes long (compared to several hundred bytes for a true PDC HID UPS). Such devices are **not** supported by the usbhid-ups driver, and are not covered in this document. If you need to add support for such a device, read new-drivers.txt and see the "tripplite_usb" driver for inspiration.

4.23.2 HID Usage Tree

From the point of view of writing a HID subdriver, a HID device consists of a bunch of variables. Some variables (such as the current input voltage) are read-only, whereas other variables (such as the beeper enabled/disabled/muted status) can be read and written. These variables are usually grouped together and arranged in a hierarchical tree shape, similar to directories in a file system. This tree is called the "usage" tree. For example, here is part of the usage tree for a typical APC device. Variable components are separated by ".". Typical values for each variable are also shown for illustrative purposes.

UPS.Battery.Voltage	11.4 V
UPS.Battery.ConfigVoltage	12 V
UPS.Input.Voltage	117 V
UPS.Input.ConfigVoltage	120 V
UPS.AudibleAlarmControl	2 (enabled)
UPS.PresentStatus.Charging	1 (yes)
UPS.PresentStatus.Discharging	0 (no)
UPS.PresentStatus.ACPresent	1 (yes)

As you can see, variables that describe the battery status might be grouped together under "Battery", variables that describe the input power might be grouped together under "Input", and variables that describe the current UPS status might be grouped together under "PresentStatus". All of these variables are grouped together under "UPS".

This hierarchical organization of data has the advantage of being very flexible; for example, if some device has more than one battery, then similar information about each battery could be grouped under "Battery1", "Battery2" and so forth. If your UPS can also be used as a toaster, then information about the toaster function might be grouped under "Toaster", rather than "UPS".

However, the disadvantage is that each manufacturer will have their own idea about how the usage tree should be organized, and usbhid-ups needs to know about all of them. This is why manufacturer specific subdrivers are needed.

To make matters more complicated, usage tree components (such as "UPS", "Battery", or "Voltage") are internally represented not as strings, but as numbers (called "usages" in HID terminology). These numbers are defined in the "HID Usage Tables", available from http://www.usb.org/developers/hidpage/. The standard usages for UPS devices are defined in a document called "Usage Tables for HID Power Devices" (the Power Device Class [PDC] specification).

For example:

```
0x00840010 = UPS
0x00840012 = Battery
0x00840030 = Voltage
0x00840040 = ConfigVoltage
```

```
0x0084001a = Input
0x0084005a = AudibleAlarmControl
0x00840002 = PresentStatus
0x00850044 = Charging
0x00850045 = Discharging
0x008500d0 = ACPresent
```

Thus, the above usage tree is internally represented as:

```
00840010.00840012.00840030
00840010.00840012.00840040
00840010.0084001a.00840030
00840010.0084001a.00840040
00840010.0084005a
00840010.00840002.00850044
00840010.00840002.00850045
00840010.00840002.008500d0
```

To make matters worse, most manufacturers define their own additional usages, even in cases where standard usages could have been used. for example Belkin defines `00860040` = ConfigVoltage (which is incidentally a violation of the USB PDC specification, as `00860040` is reserved for future use).

Thus, subdrivers generally need to provide:

- manufacturer-specific usage definitions,
- a mapping of HID variables to NUT variables.

Moreover, subdrivers might have to provide additional functionality, such as custom implementations of specific instant commands (load.off, shutdown.restart), and conversions of manufacturer specific data formats.

4.23.3 Usage macros in drivers/hidtypes.h

The `drivers/hidtypes.h` header provides a number of macro names for entries in the standard usage tables for Power Device `USAGE_POW_<SOMETHING>` and Battery System `USAGE_BAT_<SOMETHING>` data pages.

If NUT codebase would ever need to refresh those macros, here is some background information (based on NUT issue #1189 and PR #1290):

These data were parsed from (a very slightly updated version of) https://github.com/abend0c1/hidrdd/blob/master/rd.conf file, which incorporates the complete USB-IF usage definitions for Power Device and Battery System pages (among many others), so we didn't have to extract the names and values from the USB-IF standards documents (did check it all by eye though).

The file was processed with the following chain of commands:

```
:; grep -e '^0084' -e '^0085' rd.conf \
   | sed 's/,.*$//;s/ *$//' \
   | sed 's/ /_/g;s/_/ /' \
   | tr '[:lower:]' '[:upper:]' \
   | sed 's/\(0085.... \)/\1USAGE_BAT_/;s/\(0084.... \)/\1USAGE_POW_/;s/\([A-Z_]*\)_PAGE/ ↩
      PAGE_\1/' \
   | awk '{print "#define "$2" 0x"$1}'
```

4.23.4 Writing a subdriver

In preparation for writing a subdriver for a device that is currently unsupported, run usbhid-ups with the following command line:

```
drivers/usbhid-ups -DD -u root -x explore -x vendorid=XXXX \
        -x port=auto -s ups
```

(substitute your device's 4-digit VendorID instead of "XXXX"). This will produce a bunch of debugging information, including a number of lines starting with "Path:" that describe the device's usage tree. This information forms the initial basis for a new subdriver.

You should save this information to a file, e.g.:

```
drivers/usbhid-ups -DD -u root -x explore -x vendorid=XXXX \
        -x port=auto -s ups -d1 2>&1 | tee /tmp/info
```

You can now create an initial "stub" subdriver for your device by using helper script `scripts/subdriver/gen-usbhid-subdriv`

Note
this only creates a driver code "stub" which needs to be further customized to be actually useful (see "Customization" below).

Use the script as follows:

```
scripts/subdriver/gen-usbhid-subdriver.sh < /tmp/info
```

where /tmp/info is the file where you previously saved the debugging information.

This script prompts you for a name for the subdriver; use only letters and digits, and use natural capitalization such as "Belkin" (not "belkin" or "BELKIN"). The script may prompt you for additional information.

You should put the generated files into the drivers/ subdirectory, and update `usbhid-ups.c` by adding the appropriate `#include` line and by updating the definition of `subdriver_list` in `usbhid-ups.c`. You must also add the subdriver to USB-HID_UPS_SUBDRIVERS in `drivers/Makefile.am` and call `autoreconf` and/or `./configure` from the top-level NUT directory. You can then recompile `usbhid-ups`, and start experimenting with the new subdriver.

4.23.5 Updating a subdriver

You may have a device from vendor (and maybe model) whose support `usbhid-ups` already claims. However, you may feel that the driver does not represent all data points that your device serves. This may be possible, as vendors tend to use the same identifiers for unrelated products, as well as produce revisions of devices with same marketed name but different internals (due to chip and other components availability, cost optimization, etc.) Even without sinister implications, UPS firmwares evolve and so bugs and features can get added, fixed and removed over time with truly the same hardware being involved.

In this case you should follow the same instructions as above for "Writing a subdriver", but specify the same subdriver name as the one which supports your device family already.

Then compare the generated source file with the one already committed to NUT codebase, paying close attention to `..._hid2nut[]` table which maps "usage" names to NUT data points. There may be several "usage" values served by different device models or firmware versions, that provide same information for a NUT data point, such as `input.voltage`. For the `hid2nut` mapping tables, first hit wins (so you may e.g. prefer to check values with better precision first).

Using a GUI tool with partial-line difference matching and highlighting, such as Meld or WinMerge, is recommended for this endeavour.

For new data points in `hid2nut` tables be sure to not invent new names, but use standard ones from `docs/nut-names.txt` file. Temporarily, the `experimental.*` namespace may be used. If you need to standardize a name for some concept not addressed yet, please do so via nut-upsdev mailing list discussion.

4.23.6 Customization

The initially generated subdriver code is only a stub, and will not implement any useful functionality (in particular, it will be unable to shut down the UPS). In the beginning, it simply attempts to monitor some UPS variables. To make this driver useful, you must examine the NUT variables of the form "unmapped.*" in the hid_info_t data structure, and map them to actual NUT variables and instant commands. There are currently no step-by-step instructions for how to do this. Please look at the files to see how the currently implemented subdrivers are written:

- apc-hid.c/h
- belkin-hid.c/h
- cps-hid.c/h
- explore-hid.c/h
- libhid.c/h
- liebert-hid.c/h
- mge-hid.c/h
- powercom-hid.c/h
- tripplite-hid.c/h

Note
To test existing data points (including those not yet translated to standard NUT mappings conforming to NUT command and variable naming scheme), you can use custom drivers built after you `./configure --with-unmapped-data-points`. Production driver builds must not include any non-standard names.

4.23.7 Fixing report descriptors

It is a fact of life that fellow developers make mistakes, and firmware authors do too. In some cases there are inconsistencies about bytes seen on the wire vs. their logical values, such value range and signedness if interpreting them according to standard.

NUT drivers now include a way to detect and fix up known issues in such flawed USB report descriptors, side-stepping the standard similarly where deemed needed. A pointer to such hook method is part of the `subdriver_t` structure detailing each `usbhid-ups` subdriver nuances, defaulting to a `fix_report_desc()` trivial implementation.

For some practical examples, see e.g. `apc_fix_report_desc()` method in the `drivers/apc-hid.c` file, and `cps_fix_repo` in `drivers/cps-hid.c` file.

Finally note that such fix-ups may be not applicable to all devices or firmware versions for what they assume their target audience is. If you suspect that the fix-up method is actually causing problems, you can quickly disable it with `disable_fix_report_desc` driver option for `usbhid-ups`. If the problem does dissipate, please find a way to identify your "fixed" hardware/firmware vs. those models where existing fix-up method should be applied, and post a pull request so the NUT driver would handle both cases.

4.23.8 Investigating report descriptors

Beside looking for problems with report descriptor processing in NUT code, it is important to make sure what data the device actually serves on the wire, and if it is logically consistent with the protocol requirements.

While here, keep in mind that USB protocol on the wire has a specified order of bytes involved, while processing on your computer may lay them out differently due to bitness and endianness of the current binary build. General NUT codebase (`libhid.c`, `hidparser.c`) aims to abstract this, so application code like drivers can deal with their native numeric data types, but when troubleshooting, do not rule out possibility of flaws there as well. And certainly do not code any assumptions about ordered multiple-byte ranges in a protocol buffer.

For a deep dive into the byte stream, you will need additional tools:

- get/build/install regina-rexx
- get/install HIDRDD (uses REXX as the interpreter)

Typical troubleshooting of suspected firmware/protocol issues goes like this:

- Turn the NUT `usbhid-ups` driver debug verbosity level up to 5 (or more) and restart the driver, so it would record the HEX dump of report descriptor
- Look for reports from the driver of any problems it has already detected and possibly amended (LogMin/LogMax, report descriptor fix-ups)
- Extract the HEX dump of the report descriptor from USB driver output from the first step above, and run it through HIDRDD (and/or REXX directly, per example below).
- Look at the HIDRDD output, with reference to any documents related to your device and the USB/HID power devices class available in NUT documentation, e.g. at https://www.networkupstools.org/ups-protocols.html
- Especially look for inconsistencies in the USB HID report descriptors (RD):
- between the min/max (logical and physical) values,
- the sizes of the report fields they apply to,
- the expected physical values (e.g., supply and output voltages, over-voltage/under-voltage transfer points, ...)
- If you're seeing unexpected values for particular variables, look at the raw data that is being sent, decide whether it makes sense in the context of the logical and physical min/max values from the report descriptor.
- Read the NUT code, tracing through how each value gets processed looking for where the result deviates from expectations...
- Think, code, test, rinse, repeat, post a PR :)

Example 4.1 Example direct use of REXX

Example adapted from https://github.com/networkupstools/nut/issues/2039
Run a NUT `usb-hid` driver with at least debug verbosity level 3 (`-DDD`) to get a report descriptor dump starting with a line like this:

```
   3.670755     [D3] Report Descriptor: (909 bytes) => 05 84 09 04 a1 01 ...
```

...and copy-paste those reported lines as input into `rexx` tool, which would generate a C source file including human-worded description and a relevant data structure:

```
:; rexx rd.rex -d --hex 05 84 09 04 a1 01 85 01 09 18 ... 55 b1 02 c0 c0 c0

//------------------------------------------------------------------------------
// Decoded Application Collection
//------------------------------------------------------------------------------

/*
05 84        (GLOBAL) USAGE_PAGE         0x0084 Power Device Page
09 04        (LOCAL)  USAGE              0x00840004 UPS (Application Collection)
A1 01        (MAIN)   COLLECTION         0x01 Application (Usage=0x00840004: Page=Power  ↩
    Device Page, Usage=UPS, Type=Application Collection)
85 01          (GLOBAL) REPORT_ID          0x01 (1)
09 18          (LOCAL)  USAGE              0x00840018 Outlet System (Physical Collection)
...
*/

// All structure fields should be byte-aligned...
#pragma pack(push,1)

//------------------------------------------------------------------------------
// Power Device Page featureReport 01 (Device <-> Host)
//------------------------------------------------------------------------------

typedef struct
{
  uint8_t  reportId;                                // Report ID = 0x01 (1)
```

```
                                                    // Collection: CA:UPS CP:OutletSystem ↩
                                                        CP:Outlet
  int8_t   POW_UPSOutletSystemOutletSwitchable;      // Usage 0x0084006C: Switchable, Value ↩
       =  to
  int8_t   POW_UPSOutletSystemOutletDelayBeforeStartup; // Usage 0x00840056: Delay Before ↩
      Startup, Value = -1 to 60
  int8_t   POW_UPSOutletSystemOutletDelayBeforeShutdown; // Usage 0x00840057: Delay Before ↩
      Shutdown, Value = -1 to 60
  int8_t   POW_UPSOutletSystemOutletDelayBeforeReboot; // Usage 0x00840055: Delay Before ↩
      Reboot, Value = -1 to 60
  int8_t   POW_UPSOutletSystemOutletSwitchable_1;    // Usage 0x0084006C: Switchable, Value ↩
       = -1 to 60
  int8_t   POW_UPSOutletSystemOutletDelayBeforeStartup_1; // Usage 0x00840056: Delay Before ↩
       Startup, Value = -1 to 60
  int8_t   POW_UPSOutletSystemOutletDelayBeforeShutdown_1; // Usage 0x00840057: Delay ↩
      Before Shutdown, Value = -1 to 60
  int8_t   POW_UPSOutletSystemOutletDelayBeforeReboot_1; // Usage 0x00840055: Delay Before ↩
      Reboot, Value = -1 to 60
} featureReport01_t;

#pragma pack(pop)
```

4.23.9 Shutting down the UPS

It is desirable to support shutting down the UPS. Usually (for devices that follow the HID Power Device Class specification), this requires sending the UPS two commands. One for shutting down the UPS (with an *offdelay*) and one for restarting it (with an *ondelay*), where offdelay < ondelay. The two NUT commands for which this is relevant, are *shutdown.return* and *shutdown.stayoff*.

Since the one-to-one mapping above doesn't allow sending two HID commands to the UPS in response to sending one NUT command to the driver, this is handled by the driver. In order to make this work, you need to define the following four NUT values:

```
ups.delay.start    (variable, R/W)
ups.delay.shutdown (variable, R/W)
load.off.delay     (command)
load.on.delay      (command)
```

If the UPS supports it, the following variables can be used to show the countdown to start/shutdown:

```
ups.timer.start    (variable, R/O)
ups.timer.shutdown (variable, R/O)
```

The `load.on` and `load.off` commands will be defined implicitly by the driver (using a delay value of *0*). Define these commands yourself, if your UPS requires a different value to switch on/off the load without delay.

Note that the driver expects the `load.off.delay` and `load.on.delay` to follow the HID Power Device Class specification, which means that the `load.on.delay` command should NOT switch on the load in the absence of mains power. If your UPS switches on the load regardless of the mains status, DO NOT define this command. You probably want to define the `shutdown.return` and/or `shutdown.stayoff` commands in that case. Commands defined in the subdriver will take precedence over the ones that are composed in the driver.

When running the driver with the *-k* flag, it will first attempt to send a `shutdown.return` command and if that fails, will fallback to `shutdown.reboot`.

4.24 How to make a new subdriver to support another SNMP device

4.24.1 Overall concept

The SNMP protocol allow for a common way to interact with devices over the network.

The NUT "snmp-ups" driver is a meta-driver that handles many SNMP devices, such as UPS and PDU. It consists of a core driver that handles most of the work of talking to the SNMP agent, and several sub-drivers to handle specific device manufacturers. Adding support for a new SNMP device is easy, because it requires only the creation of a new sub-driver.

4.24.2 SNMP data Tree

From the point of view of writing an SNMP subdriver, an SNMP device consists of a bunch of variables, called OIDs (for Object IDentifiers). Some OIDs (such as the current input voltage) are read-only, whereas others (such as the beeper enabled/disabled/-muted status) can be read and written. OID are grouped together and arranged in a hierarchical tree shape, similar to directories in a file system. OID components are separated by ".", and can be expressed in numeric or textual form. For example:

```
.iso.org.dod.internet.mgmt.mib-2.system.sysObjectID
```

is equivalent to:

```
.1.3.6.1.2.1.1.2.0
```

Here is an excerpt tree, showing only two OIDs, sysDescr and sysObjectID:

```
.iso
        .org
                .dod
                        .internet
                                .mgmt
                                        .mib-2
                                                .system
                                                        .sysDescr.0 = STRING: Dell ↩
                                                            UPS Tower 1920W HV
                                                        .sysObjectID.0 = OID: .iso ↩
                                                           .org.dod.internet. ↩
                                                           private.enterprises ↩
                                                           .674.10902.2
                                                        (...)
                                                .upsMIB
                                                        .upsObjects
                                                                .upsIdent
                                                                        . ↩
                                                                           upsIdentModel ↩
                                                                            = ↩
                                                                           STRING: ↩
                                                                            "Dell ↩
                                                                           UPS ↩
                                                                           Tower ↩
                                                                           1920W ↩
                                                                           HV"
                                                                        (...)
                                .private
                                        .enterprises
                                                .674
                                                        .10902
                                                                .2
                                                                        .100
                                                                                .1.0 ↩
                                                                                   ↩
                                                                                = ↩
                                                                                   ↩
                                                                                STRING
```

```
: ↩
  ↩
" ↩
Dell ↩
  ↩
UPS ↩
  ↩
Tower ↩
  ↩
1920 ↩
W ↩
  ↩
HV ↩
" ↩

(...) ↩
```

As you can see in the above example, the device name is exposed three times, through three different MIBs:

- Generic MIB-II (RFC 1213):

```
.iso.org.dod.internet.mgmt.mib-2.system.sysDescr.0 = STRING: Dell UPS Tower 1920W ↩
    HV
.1.3.6.1.2.1.1.1.0 = STRING: Dell UPS Tower 1920W HV
```

- UPS MIB (RFC 1628):

```
.iso.org.dod.internet.mgmt.mib-2.upsMIB.upsObjects.upsIdent.upsIdentModel =  ↩
   STRING: "Dell UPS Tower 1920W HV"
.1.3.6.1.2.1.33.1.1.2.0 = STRING: "Dell UPS Tower 1920W HV"
```

- DELL SNMP UPS MIB:

```
.iso.org.dod.internet.private.enterprises.674.10902.2.100.1.0 = STRING: "Dell UPS ↩
    Tower 1920W HV"
```

But only the two last can serve useful data for NUT.

An highly interesting OID is **sysObjectID**: its value is an OID that refers to the main MIB of the device. In the above example, the device points us at the Dell UPS MIB. **sysObjectID**, also called "sysOID" is used by snmp-ups to find the right mapping structure.

For more information on SNMP, refer to the Wikipedia article, or browse the Internet.

To be able to convert values, NUT SNMP subdrivers need to provide:

- manufacturer-specific sysOID, to determine which lookup structure applies to which devices,
- a mapping of SNMP variables to NUT variables,
- a mapping of SNMP values to NUT values.

Moreover, subdrivers might have to provide additional functionality, such as custom implementations of specific instant commands (load.off, shutdown.restart), and conversions of manufacturer specific data formats. At the time of writing this document, snmp-ups doesn't provide such mechanisms (only formatting ones), but it is planned in a future release.

4.24.3 Creating a subdriver

In order to create a subdriver, you will need the following:

- the "MIB definition file. This file has a ".mib" extension, and is generally available on the accompanying disc, or on the manufacturer website. It should either be placed in a system directory (/usr/share/mibs/ or equivalent), or pointed using **-M** option,
- a network access to the device
- OR information dumps.

You can create an initial "stub" subdriver for your device by using the helper script **scripts/subdriver/gen-snmp-subdriver.sh**. Note that this only creates a "stub" which MUST be customized to be useful (see CUSTOMIZATION below).

You have two options to run gen-snmp-subdriver.sh:

mode 1: get SNMP data from a real agent

This method requires to have a network access to the device, in order to automatically retrieve the needed information.

You have to specify the following parameters:

- **-H** host address: is the SNMP host IP address or name
- **-c** community: is the SNMP v1 community name (default: public)"

For example:

```
$ gen-snmp-subdriver.sh -H W.X.Y.Z -c foobar -n <MIB name>.mib
```

mode 2: get data from files

This method does not require direct access to the device, at least not for the one using gen-snmp-subdriver.sh.

The following SNMP data need to be dumped first:

- sysOID value: for example *.1.3.6.1.4.1.705.1*
- a numeric SNMP walk (OIDs in dotted numeric format) of the tree pointed by sysOID. For example:

  ```
  snmpwalk -On -c foobar W.X.Y.Z .1.3.6.1.4.1.705.1 > snmpwalk-On.log
  ```

- a textual SNMP walk (OIDs in string format) of the tree pointed by sysOID. For example:

  ```
  snmpwalk -Os -c foobar W.X.Y.Z .1.3.6.1.4.1.705.1 > snmpwalk-Os.log
  ```

Note
if the OID are only partially resolved (i.e, there are still parts expressed in numeric form), then try using **-M** to point your .mib file.

Then call the script using:

```
$ gen-snmp-subdriver.sh -s <sysOID value> <numeric SNMP walk> <string SNMP walk>
```

For example:

```
$ gen-snmp-subdriver.sh -s .1.3.6.1.4.1.705.1 snmpwalk-On.log snmpwalk-Os.log
```

This script prompts you for a name for the subdriver if you don't provide it with **-n**. Use only letters and digits, and use natural capitalization such as "Camel" (not "camel" or "CAMEL", apart if it natural). The script may prompt you for additional information.

Integrating the subdriver with snmp-ups

Beside of the mandatory customization, there are a few things that you have to do, as mentioned at the end of the script:

- edit drivers/snmp-ups.h and add #include "<HFILE>.h", where <HFILE> is the name of the header file, with the **.h** extension,
- edit drivers/snmp-ups.c and bump DRIVER_VERSION by adding "0.01".
- also add "&<LDRIVER>" to snmp-ups.c:mib2nut[] list, where <LDRIVER> is the lower case driver name
- add "<LDRIVER>-mib.c" to snmp_ups_SOURCES in drivers/Makefile.am
- add "<LDRIVER>-mib.h" to dist_noinst_HEADERS in drivers/Makefile.am
- copy "<LDRIVER>-mib.c" and "<LDRIVER>-mib.h" to ../drivers/
- finally call the following, from the top level directory, to test compilation:

```
$ autoreconf && configure && make
```

You can already start experimenting with the new subdriver; but all data will be prefixed by "unmapped.". You will now have to customize it.

CUSTOMIZATION

The initially generated subdriver code is only a stub (mainly a big C structure to be precise), and will not implement any useful functionality (in particular, it will be unable to shut down the UPS). In the beginning, it simply attempts to monitor some UPS variables. To make this driver useful, you must examine the NUT variables of the form "unmapped.*" in the hid_info_t data structure, and map them to actual NUT variables and instant commands. There are currently no step-by-step instructions for how to do this. Please look at the files to see how the currently implemented subdrivers are written:

- apc-mib.c/h
- baytech-mib.c/h
- bestpower-mib.c/h
- compaq-mib.c/h
- cyberpower-mib.c/h
- eaton-*-mib.c/h
- ietf-mib.c/h
- mge-mib.c/h
- netvision-mib.c/h
- powerware-mib.c/h
- raritan-pdu-mib.c
- huawei-mib.c/h

To help you, above each entry in <LDRIVER>-mib.c, there is a comment that displays the textual OID name. For example, the following entry:

```
/* upsMIB.upsObjects.upsIdent.upsIdentModel = STRING: "Dell UPS Tower 1920W HV" */
{ "unmapped.upsidentmodel", ST_FLAG_STRING, SU_INFOSIZE,
    ".1.3.6.1.4.1.2254.2.4.1.1.0", NULL, SU_FLAG_OK, NULL },
```

Many times, only the first field will need to be modified, to map to an actual NUT variable name.

Check the NUT command and variable naming scheme section first to find a name that matches the OID name (closest fit). If nothing matches, contact the upsdev list, and we'll figure it out.

In the above example, the right NUT variable is obviously "device.model".

The MIB definition file (.mib) also contains some description of these OIDs, along with the possible enumerated values.

Note
To test existing data points (including those not yet translated to standard NUT mappings conforming to NUT command and variable naming scheme), you can use custom drivers built after you `./configure --with-unmapped-data-points`. Production driver builds must not include any non-standard names.

4.25 How to make a new subdriver to support another Q* UPS

4.25.1 Overall concept

The NUT "**nutdrv_qx**" driver is a meta-driver that handles Q* UPS devices.

It consists of a core driver that handles most of the work of talking to the hardware, and several sub-drivers to handle specific UPS manufacturers.

Adding support for a new UPS device is easy, because it requires only the creation of a new sub-driver.

4.25.2 Creating a subdriver

In order to develop a new subdriver for a specific UPS you have to know the "idiom" (dialect of the protocol) spoken by that device.

This kind of devices speaks idioms that can be summed up as follows:

- We send the UPS a query for one or more information
  - If the query is supported by the device, we'll get a reply that is mostly of a fixed length, therefore, in most cases, each information starts and ends always at the same indexes
- We send the UPS a command
  - If the command is supported by the device, the UPS will either take action without any reply or reply us with a device-specific answer signaling that the command has been accepted (e.g. `ACK`)
- If the query/command isn't supported by the device we'll get either the query/command echoed back or a device-specific reply signaling that it has been rejected (e.g. `NAK`)

To be supported by this driver the idiom spoken by the UPS must comply to these conditions.

4.25.3 Writing a subdriver

You have to fill the `subdriver_t` structure:

```
typedef struct {
        const char      *name;
        int             (*claim)(void);
        item_t          *qx2nut;
        void            (*initups)(void);
        void            (*initinfo)(void);
        void            (*makevartable)(void);
```

```
        const char      *accepted;
        const char      *rejected;
#ifdef TESTING
        testing_t       *testing;
#endif  /* TESTING */
} subdriver_t;
```

Where:

name
Name of this subdriver: name of the `protocol` that will need to be set in the `ups.conf` file to use this subdriver plus the internal version of it separated by a space (e.g. "`Megatec 0.01`").

claim
This function allows the subdriver to "claim" a device: return `1` if the device is supported by this subdriver, else `0`.

qx2nut
Main table of vars and instcmds: an array of `item_t` mapping a UPS idiom to NUT.

initups (optional)
Subdriver-specific `upsdrv_initups`. This function will be called at the end of nutdrv_qx's own `upsdrv_initups`.

initinfo (optional)
Subdriver-specific `upsdrv_initinfo`. This function will be called at the end of nutdrv_qx's own `upsdrv_initinfo`.

makevartable (optional)
Function to add subdriver-specific `ups.conf` vars and flags. Make sure not to collide with other subdrivers' vars and flags.

accepted (optional)
String to match if the driver is expecting a reply from the UPS on instcmd/setvar in case of success. This comparison is done after the answer we got back from the UPS has been processed to get the value we are searching, so you don't have to include the trailing carriage return (`\r`) and you can decide at which index of the answer the value should start or end setting the appropriate `from` and `to` in the `item_t` (see Mapping an idiom to NUT).

rejected (optional)
String to match if the driver is expecting a reply from the UPS in case of error. Note that this comparison is done on the answer we got back from the UPS before it has been processed, so include also the trailing carriage return (`\r`) and whatever character is expected.

testing
Testing table (an array of `testing_t`) that will hold the commands and the replies used for testing the subdriver.

`testing_t`:

```
typedef struct {
        const char      *cmd;
        const char      answer[SMALLBUF];
        const int       answer_len;
} testing_t;
```

Where:

cmd
Command to match.

answer
Answer for that command.

> **Note**
> If `answer` contains inner `\0`s, in order to preserve them, `answer_len` as well as an `item_t`'s `preprocess_answer()` function must be set.

answer_len
Answer length:

- if set to `-1` → auto calculate answer length (treat `answer` as a null-terminated string),
- otherwise → use the provided length (if reasonable) and preserve inner `\0`s (treat `answer` as a sequence of bytes till the `item_t`'s `preprocess_answer()` function gets called).

For more information, see Mapping an idiom to NUT.

4.25.4 Mapping an idiom to NUT

If you understand the idiom spoken by your device, you can easily map it to NUT variables and instant commands, filling `qx2nut` with an array of `item_t` data structure:

```
typedef struct item_t {
        const char      *info_type;
        const int       info_flags;
        info_rw_t       *info_rw;
        const char      *command;
        char            answer[SMALLBUF];
        const int       answer_len;
        const char      leading;
        char            value[SMALLBUF];
        const int       from;
        const int       to;
        const char      *dfl;
        unsigned long   qxflags;
        int             (*preprocess_command)(struct item_t *item, char *command, const  ↩
            size_t commandlen);
        int             (*preprocess_answer)(struct item_t *item, const int len);
        int             (*preprocess)(struct item_t *item, char *value, const size_t  ↩
            valuelen);
} item_t;
```

Where:

info_type
NUT variable name, otherwise, if `QX_FLAG_NONUT` is set, name to print to logs and if both `QX_FLAG_NONUT` and `QX_FLAG_SETVAR` are set, name of the var to retrieve from `ups.conf`.

info_flags
NUT flags (`ST_FLAG_*` values to set in `dstate_addinfo`).

info_rw
An array of `info_rw_t` to handle r/w variables:

- If `ST_FLAG_STRING` is set in `info_flags` it'll be used to set the length of the string (in `dstate_setaux`)
- If `QX_FLAG_ENUM` is set in `qxflags` it'll be used to set enumerated values (in `dstate_addenum`)
- If `QX_FLAG_RANGE` is set in `qxflags` it'll be used to set range boundaries (in `dstate_addrange`)

Note
If `QX_FLAG_SETVAR` is set the value given by the user will be checked against these infos.

`info_rw_t`:

```
typedef struct {
        char    value[SMALLBUF];
        int     (*preprocess)(char *value, const size_t len);
} info_rw_t;
```

Where:

value
Value for enum/range, or length for `ST_FLAG_STRING`.

preprocess(value, len)
Optional function to preprocess range/enum `value`.

This function will be given `value` and its `size_t` and must return either `0` if `value` is supported or `-1` if not supported.

command
Command sent to the UPS to get answer, or to execute an instant command, or to set a variable.

answer
Answer from the UPS, filled at runtime.

> **Note**
> If you expect a non-valid C string (e.g.: inner `\0`s) or need to perform actions before the answer is used (and treated as a null-terminated string), you should set a `preprocess_answer()` function.

answer_len
Expected minimum length of the answer. Set it to `0` if there's no minimum length to look after.

leading
Expected leading character of the answer (optional), e.g. `#`, `(` ...

value
Value from the answer, filled at runtime (i.e. `answer` in the interval [`from` to `to`]).

from
Position of the starting character of the info we're after in the answer.

to
Position of the ending character of the info we're after in the answer: use `0` if all the remaining of the line is needed.

dfl
printf format to store value from the UPS in NUT variables. Set it either to `%s` for strings or to a floating point specifier (e.g. `%.1f`) for numbers.

Otherwise:

- If `QX_FLAG_ABSENT` → default value
- If `QX_FLAG_CMD` → default command value

qxflags
Driver's own flags.

`QX_FLAG_STATIC`	Retrieve this variable only once.
`QX_FLAG_SEMI_STATIC`	Retrieve this info smartly, i.e. only when a command/setvar is executed and we expect that data could have been changed.
`QX_FLAG_ABSENT`	Data is absent in the device, use default value.
`QX_FLAG_QUICK_POLL`	Mandatory vars.
`QX_FLAG_CMD`	Instant command.
`QX_FLAG_SETVAR`	The var is settable and the actual item stores info on how to set it.
`QX_FLAG_TRIM`	This var's value need to be trimmed of leading/trailing spaces/hashes.
`QX_FLAG_ENUM`	Enum values exist.

QX_FLAG_RANGE	Ranges for this var are available.
QX_FLAG_NONUT	This var doesn't have a corresponding var in NUT.
QX_FLAG_SKIP	Skip this var: this item won't be processed.

Note
The driver will run a so-called QX_WALKMODE_INIT in initinfo walking through all the items in qx2nut, adding instant commands and the like. From then on it'll run a so-called QX_WALKMODE_QUICK_UPDATE just to see if the UPS is still there and then it'll do a so-called QX_WALKMODE_FULL_UPDATE to update all the vars.
If there's a problem with a var in QX_WALKMODE_INIT, the driver will automagically set QX_FLAG_SKIP on it and then it'll skip that item in QX_WALKMODE_QUICK_UPDATE/QX_WALKMODE_FULL_UPDATE, provided that the item has not the flag QX_FLAG_QUICK_POLL set, in that case the driver will set datastale.

preprocess_command(item, command, commandlen)
Last chance to preprocess the command to be sent to the UPS (e.g. to add CRC, ...). This function is given the currently processed item (item), the command to be sent to the UPS (command) and its size_t (commandlen). Return -1 in case of errors, else 0. command must be filled with the actual command to be sent to the UPS.

preprocess_answer(item, len)
Function to preprocess the answer we got from the UPS before we do anything else (e.g. for CRC, decoding, ...). This function is given the currently processed item (item) with the answer we got from the UPS unmolested and already stored in item's answer and the length of that answer (len). Return -1 in case of errors, else the length of the newly allocated item's answer (from now on, treated as a null-terminated string).

preprocess(item, value, valuelen)
Function to preprocess the data from/to the UPS: you are given the currently processed item (item), a char array (value) and its size_t (valuelen). Return -1 in case of errors, else 0.

- If QX_FLAG_SETVAR/QX_FLAG_CMD is set then the item is processed before the command is sent to the UPS so that you can fill it with the value provided by the user.

  Note
  In this case value must be filled with the command to be sent to the UPS.

- Otherwise the function will be used to process the value we got from the answer of the UPS before it'll get stored in a NUT variable.

  Note
  In this case value must be filled with the processed value already compliant to NUT standards.



Important
You must provide an item_t with QX_FLAG_SETVAR and its boundaries set for both ups.delay.start and ups.delay.shutdown to map the driver variables ondelay and offdelay, as they will be used in the shutdown sequence.

Tip
In order to keep the data flow at minimum you should keep together the items in qx2nut that need data from the same query (i.e. command): doing so the driver will send the query only once and then every item_t processed after the one that got the answer, provided that it's filled with the same command and that the answer wasn't NULL, will get that answer.

4.25.5 Examples

The following examples are from the voltronic subdriver.

Simple vars

We know that when the UPS is queried for status with `QGS\r`, it replies with something like `(234.9 50.0 229.8 50.0 000.0 000 369.1 ---.- 026.5 ---.- 018.8 100000000001\r` and we want to access the output voltage (the third token, in this case `229.8`).

```
> [QGS\r]
< [(234.9 50.0 229.8 50.0 000.0 000 369.1 ---.- 026.5 ---.- 018.8 100000000001\r]
   012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
   0         1         2         3         4         5         6         7
```

Here's the `item_t`:

```
{ "output.voltage", 0, NULL, "QGS\r", "", 76, '(', "", 12, 16, "%.1f", 0, NULL, NULL, NULL  ↩
    },
```

`info_type`	`output.voltage`
`info_flags`	`0`
`info_rw`	`NULL`
`command`	`QGS\r`
`answer`	Filled at runtime
`answer_len`	`76`
`leading`	`(`
`value`	Filled at runtime
`from`	`12` → the index at which the info (i.e. `value`) starts
`to`	`16` → the index at which the info (i.e. `value`) ends
`dfl`	`%.1f` We are expecting a number, so at first the core driver will check if it's made up entirely of digits/points/spaces, then it'll convert it into a double. Because of that we need to provide a floating point specifier.
`qxflags`	`0`
`preprocess_command`	`NULL`
`preprocess_answer`	`NULL`
`preprocess`	`NULL`

Mandatory vars

Also from `QGS\r`, we want to process the 9th status bit `10000000`**`0`**`001` that tells us whether the UPS is shutting down or not.

```
> [QGS\r]
< [(234.9 50.0 229.8 50.0 000.0 000 369.1 ---.- 026.5 ---.- 018.8 100000000001\r]
   012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
   0         1         2         3         4         5         6         7
```

Here's the `item_t`:

```
{ "ups.status", 0, NULL, "QGS\r", "", 76, '(', "", 71, 71, "%s", QX_FLAG_QUICK_POLL, NULL,  ↩
    NULL, voltronic_status },
```

`info_type`	`ups.status`
`info_flags`	`0`
`info_rw`	`NULL`
`command`	`QGS\r`
`answer`	Filled at runtime
`answer_len`	`76`
`leading`	`(`
`value`	Filled at runtime

from	71 → the index at which the info (i.e. value) starts
to	71 → the index at which the info (i.e. value) ends
dfl	%s Since a preprocess function is defined for this item, this could have been NULL, however, if we want — like here — we can use it in our preprocess function.
qxflags	QX_FLAG_QUICK_POLL → this item will be polled every time the driver will check for updates. Since this item is mandatory to run the driver, if a problem arises in QX_WALKMODE_INIT the driver won't skip it and it will set datastale.
preprocess_command	NULL
preprocess_answer	NULL
preprocess	voltronic_status This function will be called **after** the command has been sent to the UPS and we got back the answer and stored the value in order to process it to NUT standards: in this case we will convert the binary value to a NUT status.

Settable vars

So your UPS reports its battery type when queried for QBT\r; we are expecting an answer like (01\r and we know that the values can be mapped as follows: 00 → "Li", 01 → "Flooded" and 02 → "AGM".

```
> [QBT\r]
< [(01\r       <- 00="Li", 01="Flooded" or 02="AGM"
   0123
   0
```

Here's the item_t:

```
{ "battery.type", ST_FLAG_RW, voltronic_e_batt_type, "QBT\r", "", 4, '(', "", 1, 2, "%s",
  QX_FLAG_SEMI_STATIC | QX_FLAG_ENUM, NULL, NULL, voltronic_p31b },
```

info_type	battery.type
info_flags	ST_FLAG_RW → this is a r/w var
info_rw	voltronic_e_batt_type The values stored here will be added to the NUT variable, setting its boundaries: in this case Li, Flooded and AGM will be added as enumerated values.
command	QBT\r
answer	Filled at runtime
answer_len	4
leading	(
value	Filled at runtime
from	1 → the index at which the info (i.e. value) starts
to	2 → the index at which the info (i.e. value) ends
dfl	%s Since a preprocess function is defined for this item, this could have been NULL, however, if we want — like here — we can use it in our preprocess function.
qxflags	QX_FLAG_SEMI_STATIC → this item changes — and will therefore be updated — only when we send a command/setvar to the UPS QX_FLAG_ENUM → this r/w variable is of the enumerated type and the enumerated values are listed in the info_rw structure (i.e. voltronic_e_batt_type)
preprocess_command	NULL
preprocess_answer	NULL
preprocess	voltronic_p31b This function will be called **after** the command has been sent to the UPS and we got back the answer and stored the value in order to process it to NUT standards: in this case we will check if the value is in the range and then publish the human readable form of it (i.e. Li, Flooded or AGM).

We also know that we can change battery type with the PBTnn\r command; we are expecting either (ACK\r if the command succeeded or (NAK\r if the command is rejected.

```
> [PBTnn\r]             nn = 00/01/02
< [(ACK\r]
   01234
   0
```

Here's the `item_t`:

```
{ "battery.type", 0, voltronic_e_batt_type, "PBT%02.0f\r", "", 5, '(', "", 1, 4, NULL,
  QX_FLAG_SETVAR | QX_FLAG_ENUM, NULL, NULL, voltronic_p31b_set },
```

`info_type`	`battery.type`
`info_flags`	`0`
`info_rw`	`voltronic_e_batt_type` The value provided by the user will be automagically checked by the core nutdrv_qx driver against the enumerated values already set by the non setvar item (i.e. `Li`, `Flooded` or `AGM`), so this could have been `NULL`, however if we want — like here — we can use it in our `preprocess` function.
`command`	`PBT%02.0f\r`
`answer`	Filled at runtime
`answer_len`	`5` ← either `(NAK\r` or `(ACK\r`
`leading`	`(`
`value`	Filled at runtime
`from`	`1` → the index at which the info (i.e. `value`) starts
`to`	`3` → the index at which the info (i.e. `value`) ends
`dfl`	Not used for `QX_FLAG_SETVAR`
`qxflags`	`QX_FLAG_SETVAR` → this item is used to set the variable `info_type` (i.e. `battery.type`) `QX_FLAG_ENUM` → this r/w variable is of the enumerated type and the enumerated values are listed in the `info_rw` structure (i.e. `voltronic_e_batt_type`)
`preprocess_command`	`NULL`
`preprocess_answer`	`NULL`
`preprocess`	`voltronic_p31b_set` This function will be called **before** the `command` is sent to the UPS so that we can fill `command` with the value provided by the user: in this case the function will simply translate the human readable form of battery type (i.e. `Li`, `Flooded` or `AGM`) to the UPS compliant type (i.e. `00`, `01` and `02`) and then fill `value` (the second argument passed to the `preprocess` function).

Instant commands

We know that we have to send to the UPS `Tnn\r` or `T.n\r` in order to start a battery test lasting `nn` minutes or `.n` minutes: we are expecting either `(ACK\r` on success or `(NAK\r` if the command is rejected.

```
> [Tnn\r]
< [(ACK\r]
   01234
   0
```

Here's the `item_t`:

```
{ "test.battery.start", 0, NULL, "T%s\r", "", 5, '(', "", 1, 4, NULL, QX_FLAG_CMD, NULL,  ↩
    NULL, voltronic_process_command },
```

`info_type`	`test.battery.start`
`info_flags`	`0`
`info_rw`	`NULL`
`command`	`T%s\r`
`answer`	Filled at runtime
`answer_len`	`5` ← either `(NAK\r` or `(ACK\r`

`leading`	`(`
`value`	Filled at runtime
`from`	`1` → the index at which the info (i.e. `value`) starts
`to`	`3` → the index at which the info (i.e. `value`) ends
`dfl`	Not used for `QX_FLAG_CMD`
`qxflags`	`QX_FLAG_CMD` → this item is an instant command that will be fired when `info_type` (i.e. `test.battery.start`) is called
`preprocess_command`	`NULL`
`preprocess_answer`	`NULL`
`preprocess`	`voltronic_process_command` This function will be called **before** the `command` is sent to the UPS so that we can fill `command` with the value provided by the user: in this case the function will check if the value is in the accepted range and then fill `value` (the second argument passed to the `preprocess` function) with `command` and the given value.

Information absent in the device

In order to set the server-side var `ups.delay.start`, that will be then used by the driver, we have to provide the following `item_t`:

```
{ "ups.delay.start", ST_FLAG_RW, voltronic_r_ondelay, NULL, "", 0, 0, "", 0, 0, "180",
  QX_FLAG_ABSENT | QX_FLAG_SETVAR | QX_FLAG_RANGE, NULL, NULL, voltronic_process_setvar },
```

`info_type`	`ups.delay.start`
`info_flags`	`ST_FLAG_RW` → this is a r/w var
`info_rw`	`voltronic_r_ondelay` The values stored here will be added to the NUT variable, setting its boundaries: in this case `0` and `599940` will be set as the minimum and maximum value of the variable's range. Those values will then be used by the driver to check the user provided value.
`command`	Not used for `QX_FLAG_ABSENT`
`answer`	Not used for `QX_FLAG_ABSENT`
`answer_len`	Not used for `QX_FLAG_ABSENT`
`leading`	Not used for `QX_FLAG_ABSENT`
`value`	Not used for `QX_FLAG_ABSENT`
`from`	Not used for `QX_FLAG_ABSENT`
`to`	Not used for `QX_FLAG_ABSENT`
`dfl`	`180` ← the default value that will be set for this variable
`qxflags`	`QX_FLAG_ABSENT` → this item isn't available in the device `QX_FLAG_SETVAR` → this item is used to set the variable `info_type` (i.e. `ups.delay.start`) `QX_FLAG_RANGE` → this r/w variable has a settable range and its boundaries are listed in the `info_rw` structure (i.e. `voltronic_r_ondelay`)
`preprocess_command`	`NULL`
`preprocess_answer`	`NULL`
`preprocess`	`voltronic_process_setvar` This function will be called, in setvar, before the driver stores the value in the NUT var: here it's used to truncate the user-provided value to the nearest settable interval.

Information not yet available in NUT

If your UPS reports some data items that are not yet available as NUT variables and you need to process them, you can add them in `item_t` data structure adding the `QX_FLAG_NONUT` flag to its `qxflags`: the info will then be printed to the logs.

So we know that the UPS reports actual input/output phase angles when queried for `QPD\r`:

```
> [QPD\r]
< [(000 120\r]  <- Input Phase Angle -- Output Phase Angle
```

```
   012345678
   0
```

Here's the `item_t` for input phase angle:

```
{ "input_phase_angle", 0, NULL, "QPD\r", "", 9, '(', "", 1, 3, "%03.0f",
  QX_FLAG_STATIC | QX_FLAG_NONUT, NULL, NULL, voltronic_phase },
```

`info_type`	`input_phase_angle` This information will be used to print the value we got back from the UPS in the logs.
`info_flags`	`0`
`info_rw`	`NULL`
`command`	`QPD\r`
`answer`	Filled at runtime
`answer_len`	`9`
`leading`	`(`
`value`	Filled at runtime
`from`	`1` → the index at which the info (i.e. `value`) starts
`to`	`3` → the index at which the info (i.e. `value`) ends
`dfl`	`%03.0f` If there's no `preprocess` function, the format is used to print the value to the logs. Here instead it's used by the `preprocess` function.
`qxflags`	`QX_FLAG_STATIC` → this item doesn't change `QX_FLAG_NONUT` → this item doesn't have yet a NUT variable
`preprocess_command`	`NULL`
`preprocess_answer`	`NULL`
`preprocess`	`voltronic_phase` This function will be called **after** the `command` has been sent to the UPS so that we can parse the value we got back and check it.

Here's the `item_t` for output phase angle:

```
{ "output_phase_angle", ST_FLAG_RW, voltronic_e_phase, "QPD\r", "", 9, '(', "", 5, 7,  ↩
    "%03.0f",
  QX_FLAG_SEMI_STATIC | QX_FLAG_ENUM | QX_FLAG_NONUT, NULL, NULL, voltronic_phase },
```

`info_type`	`output_phase_angle` This information will be used to print the value we got back from the UPS in the logs.
`info_flags`	`ST_FLAG_RW` This could also be `0` (it's not really used by the driver), but it's set to `ST_FLAG_RW` for cohesion with other rw vars — also, if ever a NUT variable would become available for this item, it'll be easier to change this item and its `QX_FLAG_SETVAR` counterpart to use it.
`info_rw`	`voltronic_e_phase` Enumerated list of available value (here: `000`, `120`, `240` and `360`). Since `QX_FLAG_NONUT` is set the driver will print those values to the logs, plus you could use it in the `preprocess` function to check the value we got back from the UPS (as done here).
`command`	`QPD\r`
`answer`	Filled at runtime
`answer_len`	`9`
`leading`	`(`
`value`	Filled at runtime
`from`	`5` → the index at which the info (i.e. `value`) starts
`to`	`7` → the index at which the info (i.e. `value`) ends
`dfl`	`%03.0f` If there's no `preprocess` function, the format is used to print the value to the logs. Here instead it's used by the `preprocess` function.

qxflags	QX_FLAG_SEMI_STATIC → this item changes — and will therefore be updated — only when we send a command/setvar to the UPS QX_FLAG_ENUM → this r/w variable is of the enumerated type and the enumerated values are listed in the info_rw structure (i.e. voltronic_e_phase). QX_FLAG_NONUT → this item doesn't have yet a NUT variable
preprocess_command	NULL
preprocess_answer	NULL
preprocess	voltronic_phase This function will be called **after** the command has been sent to the UPS so that we can parse the value we got back and check it. Here it's used also to store a var that will then be used to check the value in setvar's preprocess function.

If you need also to change some values in the UPS you can add a ups.conf var/flag in the subdriver's own makevartable and then process it adding to its qxflags both QX_FLAG_NONUT and QX_FLAG_SETVAR: this item will be processed only once in QX_WALKMODE_INIT.

The driver will check if the var/flag is defined in ups.conf: if so, it'll then call setvar passing to this item the defined value, if any, and then it'll print the results in the logs.

We know we can set output phase angle sending PPDnnn\r to the UPS:

```
> [PPDn\r]              n = (000, 120, 180 or 240)
< [(ACK\r]
   01234
   0
```

Here's the item_t

```
{ "output_phase_angle", 0, voltronic_e_phase, "PPD%03.0f\r", "", 5, '(', "", 1, 4, NULL,
  QX_FLAG_SETVAR | QX_FLAG_ENUM | QX_FLAG_NONUT, NULL, NULL, voltronic_phase_set },
```

info_type	output_phase_angle This information will be used to print the value we got back from the UPS in the logs and to retrieve the user-provided value in ups.conf. So, name it after the variable you created to use in ups.conf in the subdriver's own makevartable.
info_flags	0
info_rw	voltronic_e_phase Enumerated list of available values (here: 000, 120, 240 and 360). The value provided by the user will be automagically checked by the core nutdrv_qx driver against the enumerated values stored here.
command	PPD%03.0f\r
answer	Filled at runtime
answer_len	5 ← either (NAK\r or (ACK\r
leading	(
value	Filled at runtime
from	1 → the index at which the info (i.e. value) starts
to	3 → the index at which the info (i.e. value) ends
dfl	Not used for QX_FLAG_SETVAR
qxflags	QX_FLAG_SETVAR → this item is used to set the variable info_type (i.e. output_phase_angle) QX_FLAG_ENUM → this r/w variable is of the enumerated type and the enumerated values are listed in the info_rw structure (i.e. voltronic_e_phase). QX_FLAG_NONUT → this item doesn't have yet a NUT variable
preprocess_command	NULL
preprocess_answer	NULL

`preprocess` `voltronic_phase_set`
This function will be called **before** the `command` is sent to the UPS so that we can check user-provided value and fill `command` with it and then fill `value` (the second argument passed to the `preprocess` function).

4.25.6 Support functions

You are already given the following functions:

`int instcmd(const char *cmdname, const char *extradata)`
Execute an instant command. In detail:

- look up the given `cmdname` in the qx2nut data structure (if not found, try to fallback to commonly known commands);
- if `cmdname` is found, call its preprocess function, passing to it `extradata`, if any, otherwise its `dfl` value, if any;
- send the command to the device and check the reply.

Return `STAT_INSTCMD_INVALID` if the command is invalid, `STAT_INSTCMD_FAILED` if it failed, `STAT_INSTCMD_HAND` on success.

`int setvar(const char *varname, const char *val)`
Set r/w variable to a value after it has been checked against its `info_rw` structure. Return `STAT_SET_HANDLED` on success, otherwise `STAT_SET_UNKNOWN`.

`item_t *find_nut_info(const char *varname, const unsigned long flag, const unsigned long n`

Find an item of `item_t` type in `qx2nut` data structure by its `info_type`, optionally filtered by its `qxflags`, and return it if found, otherwise return `NULL`.

- `flag`: flags that have to be set in the item, i.e. if one of the flags is absent in the item it won't be returned.
- `noflag`: flags that have to be absent in the item, i.e. if at least one of the flags is set in the item it won't be returned.

`int qx_process(item_t *item, const char *command)`
Send `command` (a null-terminated byte string) or, if it is `NULL`, send the command stored in the `item` to the UPS and process the reply, saving it in `item`'s `answer`. Return `-1` on errors, `0` on success.

`int ups_infoval_set(item_t *item)`
Process the value we got back from the UPS (set status bits and set the value of other parameters), calling the `item`-specific `preprocess` function, if any, otherwise executing the standard preprocessing (including trimming if `QX_FLAG_TRIM` is set). Return `-1` on failure, `0` for a status update and `1` in all other cases.

`int qx_status(void)`
Return the currently processed status so that it can be checked with one of the `status_bit_t` passed to the `STATUS()` macro (see `nutdrv_qx.h`).

`void update_status(const char *nutvalue)`
If you need to edit the current status call this function with one of the NUT status (all but `OB` are supported, simply set it as not `OL`); prefix them with an exclamation mark if you want to clear them from the status (e.g. `!OL`).

4.25.7 Armac Subdriver

Armac subdriver is based on reverse engineering of Power Manager II software by Richcomm Technologies written in 2005 that is still (as of 2023) being distributed as a valid software for freshly sold UPS of various manufacturers. It uses commands as defined for Megatec protocol - but has a different communication mechanism.

It uses two types of USB interrupt transfers: - 4 bytes to send a command (usually single transfer). - 6 byte chunk to read a reply (multiple transfers).

Transfers are similar to those of the richcomm nut driver, but the transferred data is not short binary commands. Instead, serial text data is overlaid in these transfers in a way that creates a badly made USB serial interface. UPS reply looks similar to this:

```
 0  1  2  3  4  5
HL 00 00 00 00 00
```

HL is a control byte. Its high nibble meaning is unknown. It changes between two possible values during transmission. Low nibble encodes number of bytes that have a meaning in the transaction. For example there are 5 bytes that might contain ASCII serial data, but only some might be valid, and other might be random, stale buffer data, etc.

What follows is set of observed transmissions by various UPSes gathered from Github issues.

Transfer dumps

Vultech V2000

```
419.987514     [D4] armac command Q1
419.988307     [D4] armac cleanup ret i=0 ret=6 ctrl=c0
420.119402     [D4] read: ret 6 buf 81: 28 30 31 30 30  >(0100<
420.130383     [D4] read: ret 6 buf c1: 32 30 31 30 30  >20100<
420.141408     [D4] read: ret 6 buf 82: 33 33 31 30 30  >33100<
420.152201     [D4] read: ret 6 buf c3: 2e 30 20 30 30  >.0 00<
420.153237     [D4] read: ret 6 buf 82: 30 30 20 30 30  >00 00<
420.164299     [D4] read: ret 6 buf c1: 30 30 20 30 30  >00 00<
420.175293     [D4] read: ret 6 buf 82: 2e 30 20 30 30  >.0 00<
420.186358     [D4] read: ret 6 buf c3: 20 32 33 30 30  > 2300<
420.190322     [D4] read: ret 6 buf 83: 33 2e 30 30 30  >3.000<
420.194323     [D4] read: ret 6 buf c1: 20 2e 30 30 30  > .000<
420.205358     [D4] read: ret 6 buf 81: 30 2e 30 30 30  >0.000<
420.216318     [D4] read: ret 6 buf c2: 31 34 30 30 30  >14000<
420.227445     [D4] read: ret 6 buf 83: 20 34 39 30 30  > 4900<
420.228334     [D4] read: ret 6 buf c2: 2e 30 39 30 30  >.0900<
420.239461     [D4] read: ret 6 buf 81: 20 30 39 30 30  > 0900<
420.250411     [D4] read: ret 6 buf c2: 32 37 39 30 30  >27900<
420.261405     [D4] read: ret 6 buf 83: 2e 30 20 30 30  >.0 00<
420.265468     [D4] read: ret 6 buf c3: 32 30 2e 30 30  >20.00<
420.269465     [D4] read: ret 6 buf 81: 38 30 2e 30 30  >80.00<
420.280322     [D4] read: ret 6 buf c1: 20 30 2e 30 30  > 0.00<
420.291469     [D4] read: ret 6 buf 82: 30 30 2e 30 30  >00.00<
420.302465     [D4] read: ret 6 buf c3: 30 30 31 30 30  >00100<
420.303511     [D4] read: ret 6 buf 82: 00 30 31 30 30  >          <- This has 0x00 and  ↩
    '0', will be read as "00"
420.303515     [D3] found null byte in status bits at 43 byte, assuming 0.
420.314425     [D4] read: ret 6 buf c1: 31 30 31 30 30  >10100<    <- this has '1'
420.325432     [D4] read: ret 6 buf 81: 0d 30 31 30 30  >.0100<    <- and this finishes  ↩
    with `\r`.
420.325442     [D3] armac command Q1 response read: '(233.0 000.0 233.0 014 49.0 27.0 20.8  ↩
    00001001'
```

```
1.185164     [D4] armac command ID
1.316257     [D4] read: ret 6 buf c1: 23 31 00 30 30  >#1
1.327309     [D4] read: ret 6 buf 81: 20 31 00 30 30  > 1
1.338264     [D4] read: ret 6 buf c2: 20 20 00 30 30  >
1.349151     [D4] read: ret 6 buf 83: 20 20 20 30 30  >   00<
1.360277     [D4] read: ret 6 buf c2: 20 20 20 30 30  >   00<
1.371322     [D4] read: ret 6 buf 83: 20 20 20 30 30  >   00<
1.382265     [D4] read: ret 6 buf c3: 20 20 20 30 30  >   00<
1.393156     [D4] read: ret 6 buf 82: 20 20 20 30 30  >   00<
1.404324     [D4] read: ret 6 buf c3: 20 20 20 30 30  >   00<
1.415342     [D4] read: ret 6 buf 83: 20 20 20 30 30  >   00<
1.426292     [D4] read: ret 6 buf c2: 20 20 20 30 30  >   00<
1.437203     [D4] read: ret 6 buf 83: 20 20 20 30 30  >   00<
1.448328     [D4] read: ret 6 buf c3: 56 34 2e 30 30  >V4.00<
```

```
1.459293     [D4] read: ret 6 buf 82: 31 30 2e 30 30  >10.00<
1.470274     [D4] read: ret 6 buf c3: 20 20 20 30 30  >   00<
1.481208     [D4] read: ret 6 buf 82: 20 20 20 30 30  >   00<
1.492261     [D4] read: ret 6 buf c1: 0d 20 20 30 30  >
1.492270     [D3] armac command ID response read: '#                          V4.10     '
```

```
4.749667     [D4] armac command F
4.876638     [D4] read: ret 6 buf 81: 23 31 00 30 30  >#1
4.887614     [D4] read: ret 6 buf c1: 32 31 00 30 30  >21
4.898644     [D4] read: ret 6 buf 82: 32 30 00 30 30  >20
4.909595     [D4] read: ret 6 buf c3: 2e 30 20 30 30  >.0 00<
4.920648     [D4] read: ret 6 buf 82: 30 30 20 30 30  >00 00<
4.931629     [D4] read: ret 6 buf c3: 35 20 32 30 30  >5 200<
4.942601     [D4] read: ret 6 buf 83: 34 2e 30 30 30  >4.000<
4.953666     [D4] read: ret 6 buf c2: 30 20 30 30 30  >0 000<
4.964535     [D4] read: ret 6 buf 83: 35 30 2e 30 30  >50.00<
4.975540     [D4] read: ret 6 buf c2: 30 0d 2e 30 30  >0
4.975546     [D3] armac command F response read: '#220.0 005 24.00 50.0'
```

Armac R/2000I/PSW

```
112.966856     [D4] armac command Q1
112.968197     [D4] armac cleanup ret i=0 ret=6 ctrl=c0         <- Cleanups required.
113.091193     [D4] read: ret 6 buf 81: 28 30 0d 2e 30  >(0     <- Usually 1-3 bytes  ↩
    available in transfer.
113.103211     [D4] read: ret 6 buf c1: 30 30 0d 2e 30  >00
113.115180     [D4] read: ret 6 buf 82: 30 30 0d 2e 30  >00
113.117144     [D4] read: ret 6 buf c3: 2e 30 20 2e 30  >.0 .0<
113.120150     [D4] read: ret 6 buf 81: 31 30 20 2e 30  >10 .0<
113.132178     [D4] read: ret 6 buf c1: 34 30 20 2e 30  >40 .0<
113.144159     [D4] read: ret 6 buf 82: 30 2e 20 2e 30  >0. .0<
113.146149     [D4] read: ret 6 buf c3: 30 20 32 2e 30  >0 2.0<
113.149173     [D4] read: ret 6 buf 81: 32 20 32 2e 30  >2 2.0<
113.161167     [D4] read: ret 6 buf c1: 37 20 32 2e 30  >7 2.0<
113.173159     [D4] read: ret 6 buf 82: 2e 30 32 2e 30  >.02.0<
113.175157     [D4] read: ret 6 buf c3: 20 30 30 2e 30  > 00.0<
113.178158     [D4] read: ret 6 buf 81: 32 30 30 2e 30  >200.0<
113.190157     [D4] read: ret 6 buf c1: 20 30 30 2e 30  > 00.0<
113.202161     [D4] read: ret 6 buf 82: 30 30 30 2e 30  >000.0<
113.204154     [D4] read: ret 6 buf c3: 2e 30 20 2e 30  >.0 .0<
113.207150     [D4] read: ret 6 buf 81: 34 30 20 2e 30  >40 .0<
113.219174     [D4] read: ret 6 buf c1: 36 30 20 2e 30  >60 .0<
113.231165     [D4] read: ret 6 buf 82: 2e 38 20 2e 30  >.8 .0<
113.233157     [D4] read: ret 6 buf c3: 20 35 36 2e 30  > 56.0<
113.237149     [D4] read: ret 6 buf 81: 2e 35 36 2e 30  >.56.0<
113.249168     [D4] read: ret 6 buf c1: 30 35 36 2e 30  >056.0<
113.261155     [D4] read: ret 6 buf 83: 20 31 30 2e 30  > 10.0<
113.263151     [D4] read: ret 6 buf c2: 30 30 30 2e 30  >000.0<
113.266152     [D4] read: ret 6 buf 81: 31 30 30 2e 30  >100.0<
113.278161     [D4] read: ret 6 buf c1: 30 30 30 2e 30  >000.0<  <- No Null bytes.
113.290155     [D4] read: ret 6 buf 82: 30 30 30 2e 30  >000.0<
113.292159     [D4] read: ret 6 buf c1: 0d 30 30 2e 30  >
113.292169     [D3] armac command Q1 response read: '(000.0 140.0 227.0 002 00.0 46.8 56.0  ↩
    10001000'
```

Next query would return 0x80 control byte - 0 available bytes. This used to terminate transmission, but some UPS don't work like that.

Armac R/3000I/PF1

```
0.083301     [D4] armac command Q1
0.164847     [D4] read: ret 6 buf a6: 28 32 34 31 2e  >(241.<
```

```
0.184839     [D4] read: ret 6 buf 86: 35 20 30 30 30  >5 000<
0.205851     [D4] read: ret 6 buf a6: 2e 30 20 32 33  >.0 23<
0.226849     [D4] read: ret 6 buf 86: 30 2e 33 20 30  >0.3 0<
0.247859     [D4] read: ret 6 buf a6: 30 30 20 34 39  >00 49<
0.268862     [D4] read: ret 6 buf 86: 2e 39 20 32 2e  >.9 2.<
0.289857     [D4] read: ret 6 buf a6: 32 35 20 34 38  >25 48<
0.309866     [D4] read: ret 6 buf 86: 2e 30 20 30 30  >.0 00<
0.330863     [D4] read: ret 6 buf a6: 30 30 30 30 30  >00000<
0.827913     [D4] read: ret 6 buf 83: 31 0d 30 30 30  >1 000<
0.827927     [D3] armac command Q1 response read: '(241.5 000.0 230.3 000 49.9 2.25 48.0  ↩
    00000001'
0.827954     [D4] armac command ID
1.394985     [D4] read: ret 6 buf a5: 4e 41 4b 0d 30  >NAK  <
1.395001     [D3] armac command ID response read: 'NAK'
```

This UPS sends higher nibble set to 6 often, which exceeds available bytes. Maybe means that more are available. Its serial-USB bridge is probably faster. We read 5 bytes in case 6 nibble is sent. End of transmission is marked by `\r`, no 0 nibble is sent.

4.25.8 Notes

You must put the generated files into the `drivers/` subdirectory, with the name of your subdriver preceded by `nutdrv_qx_`, and update `nutdrv_qx.c` by adding the appropriate `#include` line and by updating the definition of `subdriver_list`.

Please, make sure to add your driver in that list in a smart way: if your device supports also the basic commands used by the other subdrivers to claim a device, add something that is unique (i.e. not supported by the other subdrivers) to your device in your claim function and then add it on top of the slightly supported ones in that list.

You must also add the subdriver to `NUTDRV_QX_SUBDRIVERS` list variable in the `drivers/Makefile.am` and call "`autoreconf` and/or "`./configure`" from the top level NUT directory.

You can then recompile `nutdrv_qx`, and start experimenting with the new subdriver.

For more details, have a look at the currently available subdrivers:

- `nutdrv_qx_bestups.{c,h}`
- `nutdrv_qx_masterguard.{c,h}`
- `nutdrv_qx_mecer.{c,h}`
- `nutdrv_qx_megatec.{c,h}`
- `nutdrv_qx_megatec-old.{c,h}`
- `nutdrv_qx_mustek.{c,h}`
- `nutdrv_qx_q1.{c,h}`
- `nutdrv_qx_voltronic.{c,h}`
- `nutdrv_qx_voltronic-qs.{c,h}`
- `nutdrv_qx_voltronic-qs-hex.{c,h}`
- `nutdrv_qx_zinto.{c,h}`
- `nutdrv_qx_ablerex.{c,h}`

5 Driver/server socket protocol

Here's a brief explanation of the text-based protocol which is used between the drivers and server.

The drivers may send things on the socket at any time. They will send out changes to their local storage immediately, without any sort of prompting from the server. As a result, the server must always check on any driver sockets for activity.

In terms of communications, each driver is a server on the Unix socket (or Windows named pipe) which it creates, and the data server `upsd` is a client which knows where to find such sockets, how they are named, and connects to all of them to send commands and receive data updates.

During development, it is possible to use tools like `socat` to connect to the socket (you may want to enable `NOBROADCAST` mode soon), e.g.

```
socat - UNIX-CONNECT:/var/state/ups/dummy-ups-UPS1
```

For more insight, NUT provides an optional tool of its own (not built by default): the `sockdebug` which is built when `configure --with-dev` is in effect, or can be requested from the root directory of the build workspace:

```
make sockdebug && \
./server/sockdebug dummy-ups-UPS1
```

5.1 Formatting

All parsing on either side of the socket is done by parseconf, so the same rules about escaping characters and "quoting multi-word elements" apply here. Values which may contain odd characters are typically sent through pconf_encode to apply \ characters where necessary.

The "" construct is used throughout to force a multi-word value to stay together on its way to the other end.

5.2 Commands used by the drivers

These commands (or semantically responses to server commands in some cases) can be sent by drivers to the data server over the socket protocol.

5.2.1 SETINFO

```
SETINFO <varname> "<value>"
```

```
SETINFO ups.status "OB LB"
```

There is no "ADDINFO" — if a given variable does not exist, it is created upon receiving the first SETINFO command.

5.2.2 DELINFO

```
DELINFO <varname>
```

```
DELINFO ups.temperature
```

5.2.3 ADDENUM

```
ADDENUM <varname> "<value>"
```

```
ADDENUM input.transfer.low "95"
```

5.2.4 DELENUM

```
DELENUM <varname> "<value>"

DELENUM input.transfer.low "98"
```

5.2.5 ADDRANGE

```
ADDRANGE <varname> <minvalue> <maxvalue>

ADDRANGE input.transfer.low 95 100
```

5.2.6 DELRANGE

```
DELRANGE <varname> <minvalue> <maxvalue>

DELRANGE input.transfer.low 95 100
```

5.2.7 SETAUX

```
SETAUX <varname> <numeric value>

SETAUX ups.id 8
```

This overrides any previous value. The auxiliary value is presently used as a length byte for read-write variables that are strings.

5.2.8 SETFLAGS

```
SETFLAGS <varname> <flag>...

SETFLAGS ups.id RW STRING
```

Note that this command takes a variable number of arguments, as multiple flags are supported. Also note that they are not crammed together in "", since "RW STRING" would mean something completely different.

This also replaces any previous flags for a given variable.

5.2.9 ADDCMD

```
ADDCMD <cmdname>

ADDCMD load.off
```

5.2.10 DELCMD

```
DELCMD <cmdname>

DELCMD load.on
```

5.2.11 DUMPDONE

```
DUMPDONE
```

This is only used to tell the server that every possible item has been transmitted in response to its DUMPALL request. Once this has been received by the server, it can be sure that it knows everything that the driver does.

5.2.12 PONG

```
PONG
```

This is sent in response to a PING from the server. It is only used as a sanity check to make sure that the driver has not gotten stuck somewhere.

5.2.13 DATAOK

```
DATAOK
```

This means that the driver is able to communicate with the UPS, and the data should be treated as usable. It is always sent at the end of the dump if the data is not stale. It may also be sent at other times.

5.2.14 DATASTALE

```
DATASTALE
```

This is sent by the driver to inform any listeners that the data is no longer usable. This usually means that the driver is unable to get any sort of meaningful response from the UPS. You must not rely on any status information once this has been sent.

This will be sent in the beginning of a dump if the data is stale, and may be repeated. It is cleared by DATAOK.

5.2.15 TRACKING

```
TRACKING <id> <value>
```

This is sent in response to an INSTCMD or SET VAR that includes a TRACKING, upon completion of request execution by the driver. <value> is the integer return value from the driver handlers instcmd and setvar (see drivers/upshandler.h). The server is in charge of translating these codes into strings, as per docs/net-protocol.txt GET TRACKING.

5.3 Commands sent by the server

The data server `upsd` (or technically any client that connects to a Unix socket or Windows named pipe provided by each NUT driver) can send the following commands to the driver:

5.3.1 PING

```
PING
```

This is sent to check on the health of a driver. The server should only send this when it hasn't heard anything valid from a driver recently. Some drivers have very little to say in terms of updates, and this may be the only communications they have with the server on a normal basis.

If a driver does not respond with the PONG within a few seconds at the most, it should be treated as dead/unavailable. Data stored in the server must not be passed on to the clients when this happens.

Note
For the `upsd` data server, the MAXAGE setting in upsd.conf controls how long since the last message from the driver it is considered stale. At 1/3 of this time the server sends a `PING` command to the driver, so there is some time for a `PONG` to arrive and reset the timer (any other message would serve that goal as well).

5.3.2 INSTCMD

```
INSTCMD <cmdname> [<cmdparam>] [TRACKING <id>]

INSTCMD panel.test.start
INSTCMD load.off 10
INSTCMD load.on 10 TRACKING 1bd31808-cb49-4aec-9d75-d056e6f018d2
```

NOTE:

- <cmdparam> is an additional and optional parameter for the command,
- "TRACKING <id>" can be provided to track commands execution status, if TRACKING was set to ON on upsd. In this case, driver will later return the execution status, using TRACKING.

5.3.3 SET

```
SET <varname> "<value>" [TRACKING <id>]

SET ups.id "Data room"
SET ups.id "Data room" TRACKING 2dedb58a-3b91-4fab-831f-c8af4b90760a
```

NOTE:

- "TRACKING <id>" can be provided to track commands execution status, if TRACKING was set to ON on upsd. In this case, driver will later return the execution status, using TRACKING.

5.3.4 DUMPALL

```
DUMPALL
```

The server uses this to request a complete copy of everything the driver knows. This is returned in the form of the same commands (SETINFO, etc.) that would be used if they were being updated normally. As a result, the same parsing happens either way.

The server can tell when it has a full copy of the data by waiting for DUMPDONE. That special response from the driver is sent once the entire set has been transmitted.

5.3.5 NOBROADCAST

This connection does not want to receive broadcast messages (implemented by `send_to_all()` method in `dstate.c`). Default is to receive everything.

5.3.6 BROADCAST (NUM)

This connection specified whether it wants to receive broadcast messages (implemented by `send_to_all()` method in `dstate.c`), and by default enables that — unless disabled by providing an optional zero or negative numeric argument. Note that initial default is to receive everything, so this command may be useful for connections that disabled broadcasts at some point.

5.4 Design notes

5.4.1 Requests

There is no way to request just one variable. This was done on purpose to limit the complexity of the drivers. Their job is to send out updates and handle a few simple requests. DUMPALL is provided to give the server a known foundation.

To track a limited set of variables, a server just needs to do DUMPALL, then only have handlers that remember values for the variables that matter. Anything else should be ignored.

5.4.2 Access/Security

There are no access controls in the drivers. Anything that can connect to their sockets can make requests, including SET and INSTCMD if supported by the driver and hardware. These sockets must be kept secure. If your operating system does not honor permissions or modes on sockets, then you must store them in a directory with suitable permissions to limit access.

5.4.3 Command limitations

As parseconf is used to handle decoding and chunking of the data, there are some limits on what may be used. These default to 32 arguments of 512 characters each, which should be more than enough for everything which is currently needed by the software.

These limits are strictly for sanity purposes, and may be raised if necessary. parseconf itself can handle vast numbers of arguments and characters, with some speed penalty as things get really big.

5.4.4 Re-establishing communications

If the server loses its connection to the driver and later reconnects, it must flush any local storage and start again with DUMPALL. The driver may have changed the internal state considerably during that time, and any other approach could leave old elements behind.

6 NUT configuration management with Augeas

6.1 Introduction

Configuration has long been one of the two main NUT weaknesses. This is mostly due to the framework nature of NUT, and its many components and features, which make NUT configuration a very complex task.

In order to address this point, NUT now provides configuration tools and manipulation abstraction, to anybody who want to manipulate NUT configuration, through Augeas lenses and modules.

From Augeas homepage:

"Augeas is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native config files."

In other words, Augeas is the dreamed Registry, with all the advantages (such as a uniform interface and tools), and the added bonus of being free/libre open source software and letting liberty on configuration file format.

6.2 Requirements

To be able to use Augeas with NUT, you will need to install Augeas, and also the NUT provided lenses, which describe NUT configuration files format.

6.2.1 Augeas

Having Augeas installed. You will need at least version 0.5.1 (prior versions may work too, reports are welcome).

As an example, on Debian and derivatives, do the following:

```
$ apt-get install augeas-lenses augeas-tools
```

And optionally:

```
$ apt-get install libaugeas0 libaugeas-dev python-augeas
```

On RedHat and derivatives, you have to install the packages *augeas* and *augeas-libs*.

6.2.2 NUT lenses and modules for Augeas

These are the *.aug files in the present directory.

You can either install the files to the right location on your system, generally in */usr/share/augeas/lenses/*, or use these from NUT source directory (*nut/scripts/augeas*). The latter is to be preferred for the time being.

6.3 Create a test sandbox

Note
for now, it's easier to include an existing /etc/nut/ directory.

```
$ export AUGEAS_ROOT=./augeas-sandbox
$ mkdir $AUGEAS_ROOT
$ sudo cp -pr /etc/nut $AUGEAS_ROOT
$ sudo chown -R $(id -nu):$(id -ng) $AUGEAS_ROOT
```

6.4 Start testing and using

Augeas provides many tools and languages bindings (Python, Perl, Java, PHP, Ruby, ...), still with the same simple logic.

This chapter will only illustrate some of these. Refer to the language binding's help and Augeas documentation for more information.

6.4.1 Shell

Start an augeas shell using:

```
$ augtool -b
```

Note
if you have not installed NUT lenses, add *-I/path/to/nut/scripts/augeas*.

From there, you can perform different actions like:

- list existing nut related files:

```
augtool> ls /files/etc/nut/
nut.conf/ = (none)
upsd.users/ = (none)
upsmon.conf = (none)
ups.conf/ = (none)
upsd.conf/ = (none
```

or using:

```
augtool> match /files/etc/nut/*
/files/etc/nut/nut.conf = (none)
/files/etc/nut/upsd.users = (none)
/files/etc/nut/upsmon.conf = (none)
/files/etc/nut/ups.conf = (none)
/files/etc/nut/upsd.conf = (none)
```

Note
if you don't see anything, you may search for error messages by using:

+ augtool> ls /augeas/files/etc/nut/*/errors and augtool> get /augeas/files/etc/nut/ups.conf/error/message /augeas/files/etc/nut/ups.conf/er
= Permission denied

- create a new device entry (in ups.conf), called *augtest*:

```
augtool> set /files/etc/nut/ups.conf/augtest/driver dummy-ups
augtool> set /files/etc/nut/ups.conf/augtest/port auto
augtool> save
```

- list the devices using the *usbhid-ups* driver:

```
augtool> match /files/etc/nut/ups.conf/*/driver dummy-ups
```

C ~

A library is available for C programs, along with pkg-config support.

You can get the compilation and link flags using the following code in your configure script or Makefile:

```
CFLAGS="`pkg-config --silence-errors --cflags augeas`"
LDFLAGS="`pkg-config --silence-errors --libs augeas`"
```

Here is an code sample using this library for NUT configuration:

```
augeas *a = aug_init(NULL, NULL, AUG_NONE);
ret = aug_match(a, "/files/etc/nut/*", &matches_p);
ret = aug_set(a, "/files/etc/nut/ups.conf/augtest/driver", "dummy-ups");
ret = aug_set(a, "/files/etc/nut/ups.conf/augtest/port", "auto");
ret = aug_save(a);
```

6.4.2 Python

The `augeas` class abstracts access to the configuration files.

```
$ python
Python 2.5.1 (r251:54863, Apr  8 2008, 01:19:33)
[GCC 4.3.0 20080404 (Red Hat 4.3.0-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import augeas
>>> a = augeas.augeas()
>>> a.match("/files/etc/nut/*")
['/files/etc/nut/upsd.users', '/files/etc/nut/upsmon.conf', '/files/etc/nut/ups. ↩
   conf', '/files/etc/nut/upsd.conf']
>>> a.set("/files/etc/nut/ups.conf/augtest/driver", "dummy-ups")
True
>>> a.set("/files/etc/nut/ups.conf/augtest/port", "auto")
True
>>> a.save()
True
>>>

$ grep -A 2 augtest /etc/nut/ups.conf
[augtest]
driver=dummy-ups
port=auto
```

6.4.3 Perl

The Perl binding is available through CPAN and packages.

```
use Config::Augeas;

my $aug = Config::Augeas->new( root => $aug_root ) ;

my @a = $aug->match("/files/etc/nut/*") ;
my $nb = $aug->count_match("/files/etc/nut/*") ;

$aug->set("/files/etc/nut/ups.conf/augtest/driver", "dummy-ups") ;
$aug->set("/files/etc/nut/ups.conf/augtest/port", "auto") ;

$aug->save ;
```

6.4.4 Test the conformity testing module

Existing configuration files can be tested for conformity. To do so, use:

```
$ augparse -I ./ ./test_nut.aug
```

6.5 Complete configuration wizard example

Here is a Python example that generate a complete and usable standalone configuration:

```
import augeas

device_name="dev1"
driver_name="usbhid-ups"
```

```
port_name="auto"

a = augeas.augeas()

# Generate nut.conf
a.set("/files/etc/nut/nut.conf/MODE", "standalone")

# Generate ups.conf
# FIXME: chroot, driverpath?
a.set(("/files/etc/nut/ups.conf/%s/driver" % device_name), driver_name)
a.set(("/files/etc/nut/ups.conf/%s/port" % device_name), port_name)

# Generate upsd.conf
a.set("/files/etc/nut/upsd.conf/#comment[1]", "just to touch the file!")

# Generate upsd.users
user = "admin"
a.set(("/files/etc/nut/upsd.users/%s/password" % user), "dummypass")
a.set(("/files/etc/nut/upsd.users/%s/actions/SET" % user), "")
# FIXME: instcmds lens should be fixed, as per the above rule
a.set(("/files/etc/nut/upsd.users/%s/instcmds" % user), "ALL")

monuser = "monuser"
monpasswd = "******"
a.set(("/files/etc/nut/upsd.users/%s/password" % monuser), monpasswd)
a.set(("/files/etc/nut/upsd.users/%s/upsmon" % monuser), "primary")

# Generate upsmon.conf
a.set("/files/etc/nut/upsmon.conf/MONITOR/system/upsname", device_name)
# Note: we prefer to omit localhost, not to be bound to a specific
# entry in /etc/hosts, and thus be more generic
#a.set("/files/etc/nut/upsmon.conf/MONITOR/system/hostname", "localhost")
a.set("/files/etc/nut/upsmon.conf/MONITOR/powervalue", "1")
a.set("/files/etc/nut/upsmon.conf/MONITOR/username", monuser)
a.set("/files/etc/nut/upsmon.conf/MONITOR/password", monpasswd)
a.set("/files/etc/nut/upsmon.conf/MONITOR/type", "primary")

# FIXME: glitch on the generated content
a.set("/files/etc/nut/upsmon.conf/SHUTDOWNCMD", "/sbin/shutdown -h +0")

# save config
a.save()
a.close()
```

7 NUT device discovery

7.1 Introduction

nut-scanner(8) is available to discover supported NUT devices (USB, SNMP, Eaton XML/HTTP and IPMI) and NUT servers (using Avahi or the classic connection method).

This tool actually use a library, called **libnutscan**, to perform actual processing.

7.1.1 Client access library

The nutscan library can be linked into other programs to give access to NUT discovery. Both static and shared versions are provided.

nut-scanner(8) is provided as an example of how to use the nutscan functions.

Here is a simple example that scans for USB devices, and use its own iteration function to display results:

Scanning and reporting example

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Only enable USB scan */
#define HAVE_USB_H

#include "nut-scan.h"

int main()
{
        nutscan_options_t * opt;
        nutscan_device_t *device;

        nutscan-init();

        if ((device = nutscan_scan_usb()) == NULL) {
                printf("No device found\n");
                exit(EXIT_FAILURE);
        }

        /* Rewind the list */
        while(device->prev != NULL) {
                device = device->prev;
        }

        /* Print results */
        do {
                printf("USB device found\n\tdriver: \"%s\"\n\tport: \"%s\"\n",
                        device->driver, device->port);

                /* process options (serial number, bus, ...) */
                opt = &(device->opt);
                do {
                        if( opt->option != NULL ) {
                                printf("\t%s",opt->option);
                                if( opt->value != NULL ) {
                                        printf(": \"%s\"", opt->value);
                                }
                                printf("\n");
                        }
                        opt = opt->next;
                } while( opt != NULL );

                device = device->next;
        }
        while( device != NULL );

        exit(EXIT_SUCCESS);
}
```

This library file and the associated header files are not installed by default. You must `./configure --with-dev` to enable building and installing these files. The libraries can then be built and installed with `make` and `make install` as usual. This must be done before building other (non-NUT) programs which depend on them.

For more information, refer to the nutscan(3), manual page and the various nutscan_*(3) functions documentation referenced in the same file.

7.1.2 Configuration helpers

NUT provides helper scripts to ease the configuration step of your program, by detecting the right compilation and link flags.

For more information, refer to a Appendix B: NUT libraries complementary information.

7.2 Python

Python support for NUT discovery features is not yet available.

7.3 Perl

Perl support for NUT discovery features is not yet available.

7.4 Java

Java support for NUT discovery features is not yet available.

8 Creating new client

NUT provides bindings for several common languages that are presented below. All these are released under the same license as NUT (the GNU General Public License).

If none of these suits you for technical or legal reasons, you can implement one easily using the Network protocol information.

The latter approach has been used to create the Python *PyNUT* module, the Nagios *check_ups* plugin (and probably others), which can serve as a reference.

8.1 C / C++

8.1.1 Client access library

`libupsclient` and `libnutclient` libraries can be linked into other programs to give access to upsd and UPS status information. Both static and shared versions are provided.

These library files and associated header files are not installed by default. You must `./configure --with-dev` to enable building and installing these files. The libraries can then be built and installed with `make` and `make install` as usual. This must be done before building other (non-NUT) programs which depend on them.

Low-level library: libupsclient

`libupsclient` provides a low-level interface to directly dialog with upsd. It is a wrapper around the NUT network protocol.

For more information, refer to the upsclient(3), manual page and the various upscli_*(3) functions documentation referenced in the same file.

Clients like upsc are provided as examples of how to retrieve data using the upsclient functions. Other programs not included in this package may also use this library, such as wmnut.

High level library: libnutclient

`libnutclient` provides a high-level interface representing devices, variables and commands with an object-oriented API in C++ and C.

For more information, refer to the libnutclient(3) manual page.

```
#include <iostream>
#include <unistd.h>
#include <stdlib.h>

#include <nutclient.h>

using namespace nut;
using namespace std;

/* argv[1] is the mandatory NUT device name (@localhost),
 *    used to list variables from
 * argv[2] is an optional command. When provided, it will be
 *    executed and possibly with status tracking enabled
 */
int main(int argc, char **argv)
{
  Client *client;
  try
  {
    // Connection
    client = new TcpClient("localhost", 3493);

    if (argc >= 2)
    {
      // Reading data from device
      Device mydev = client->getDevice(argv[1]);
      cout << "Description: " << mydev.getDescription() << endl;
      Variable var = mydev.getVariable("device.model");
      cout << "Model: " <<  var.getValue()[0] << endl;

      if (argc >= 3)
      {
        // Authenticate to NUT server
        const char *user = getenv("NUT_USER");
        const char *password = getenv("NUT_PASSWD");
        client->authenticate(user ? user : "", password ? password : "");

        // Enable command tracking, if available
        if (client->hasFeature(Client::TRACKING))
        {
          cout << "Server can do command tracking" << std::endl;
          client->setFeature(Client::TRACKING, true);
        }
        else
        {
          std::cout << "Server can't do command tracking" << std::endl;
        }

        // Perform an asynchronous command
        TrackingID id = mydev.executeCommand(argv[2]);
        TrackingResult result;
        do
        {
          sleep(1);
          result = client->getTrackingResult(id);
        }
```

```
          while (result == PENDING);

          // Display result of command
          const char *output = "<UNRECOGNIZED>";
          switch (result)
          {
            case SUCCESS: output = "SUCCESS"; break;
            case FAILURE: output = "FAILURE"; break;
            case UNKNOWN: output = "UNKNOWN"; break;
          }
          cout << "Command sent, result=" << output << endl;
        }
      }
    }
    catch (NutException &ex)
    {
      cerr << "Unexpected problem : " << ex.str() << endl;
    }
    delete client;
    return 0;
  }
```

8.1.2 Configuration helpers

NUT provides helper scripts to ease the configuration step of your program, by detecting the right compilation and link flags.

For more information, refer to a Appendix B: NUT libraries complementary information.

8.2 Python

The PyNUT module, contributed by David Goncalves, can be used for connecting a Python script to `upsd`. Note that this code (and the accompanying NUT-Monitor application, later separated into NUT-Monitor-py2gtk2 and NUT-Monitor-py3qt5, suitable for two generations of Python ecosystem) is licensed under the GPL v3.

The `PyNUTClient` class abstracts the connection to the server. In order to list the status variables for `ups1` on the local `upsd`, the following commands could be used:

```
$ cd scripts/python/module
$ python
...
>>> import PyNUT
>>> from pprint import pprint
>>> client = PyNUT.PyNUTClient()
>>> vars = client.GetUPSVars('ups1')
>>> pprint(vars)
{'battery.charge': '90',
 'battery.charge.low': '30',
 'battery.runtime': '3690',
 'battery.voltage': '230.0',
...
```

Further examples are given in the `test_nutclient.py` file. To see the entire API, you can run `pydoc` from the `module` directory.

If you wish to make the module available to everyone on the system, you will probably want to install it in the `site-packages` directory for your Python interpreter. (This is usually one of the last items in `sys.path`.)

8.3 Perl

The old Perl bindings from CPAN have recently been updated and merged into the NUT source code. These operate in a similar fashion to the Python bindings, with the addition of access to single variables, and additional interpretation of the results. The Perl class instance encapsulates a single UPS, where the Python class instance represents a connection to the server (which may service multiple UPS units).

```
use UPS::Nut;

$ups = new UPS::Nut( NAME => "myups",
                     HOST => "somemachine.somewhere.com",
                     PORT => "3493",
                     USERNAME => "upsuser",
                     PASSWORD => "upspasswd",
                     TIMEOUT => 30,
                     DEBUG => 1,
                     DEBUGOUT => "/some/file/somewhere",
                   );
if ($ups->Status() =~ /OB/) {
   print "Oh, no!  Power failure!\n";
}

tie %other_ups, 'UPS::Nut',
    NAME => "myups",
    HOST => "somemachine.somewhere.com",
    ... # same options as new();
;

print $other_ups{MFR}, " ", $other_ups{MODEL}, "\n";
```

8.4 Java

The NUT Java support has been externalized. It is available at https://github.com/networkupstools/jnut

9 Network protocol information

Since May 2002, this protocol has an official port number from IANA, which is **3493**. The old number (3305) was a relic of the original code's ancestry, and conflicted with other services. Version 0.50.0 and up use 3493 by default.

This protocol runs over TCP. UDP support was dropped in July 2003. It had been deprecated for some time and was only capable of the simplest query commands as authentication is impossible over a UDP socket.

A library, named libupsclient, that implement this protocol is provided in both static and shared version to help the client application development.

9.1 Old command removal notice

Before version 1.5.0, a number of old commands were supported. These have been removed from the specification. For more information, consult an older version of the software.

9.2 Command reference

Multi-word elements are contained within "quotes" for easier parsing. Embedded quotes are escaped with backslashes. Embedded backslashes are also escaped by representing them as \\. This protocol is intended to be interpreted with parseconf (NUT parser) or something similar.

9.3 Revision history

Here's a table to present the various changes that happened to the NUT network protocol, over the time:

Protocol version	NUT version	Description
1.0	< 1.5.0	Original protocol (legacy version)
1.1	>= 1.5.0	Original protocol (without old commands)
1.2	>= 2.6.4	Add "LIST CLIENT" and "NETVER" commands
		Add ranges of values for writable variables
1.3	>= 2.8.0	Add "cmdparam" to "INSTCMD"
		Add "TRACKING" commands (GET, SET)
		Add "PRIMARY" as alias to older "MASTER" (implementation tested to be backwards compatible in `upsd` and `upsmon`)
		Add "PROTVER" as alias to older "NETVER"

Note
Any new version of the protocol implies an update of `NUT_NETVERSION` in *configure.ac* file.

ERRATA: Earlier revisions of this table mistakenly mentioned `LIST CLIENTS` as added since 2.6.4. The actual added command was `LIST CLIENT` (no `S`) as documented in its section below.

9.4 GET

Retrieve a single response from the server.

Possible sub-commands:

9.4.1 NUMLOGINS

Form:

```
GET NUMLOGINS <upsname>
GET NUMLOGINS su700
```

Response:

```
NUMLOGINS <upsname> <value>
NUMLOGINS su700 1
```

<*value*> is the number of clients which have done LOGIN for this UPS. This is used by the upsmon in primary mode to determine how many clients are still connected when starting the shutdown process.

This replaces the old "REQ NUMLOGINS" command.

9.4.2 UPSDESC

Form:

```
GET UPSDESC <upsname>
GET UPSDESC su700
```

Response:

```
UPSDESC <upsname> "<description>"
UPSDESC su700 "Development box"
```

<*description*> is the value of "desc=" from ups.conf for this UPS. If it is not set, upsd will return "Unavailable".

This can be used to provide human-readable descriptions instead of a cryptic "upsname@hostname" string.

9.4.3 VAR

Form:

```
GET VAR <upsname> <varname>
GET VAR su700 ups.status
```

Response:

```
VAR <upsname> <varname> "<value>"
VAR su700 ups.status "OL"
```

This replaces the old "REQ" command.

9.4.4 TYPE

Form:

```
GET TYPE <upsname> <varname>
GET TYPE su700 input.transfer.low
```

Response:

```
TYPE <upsname> <varname> <type>...
TYPE su700 input.transfer.low ENUM
```

<*type*> can be several values, and multiple words may be returned:

- *RW*: this variable may be set to another value with SET
- *ENUM*: an enumerated type, which supports a few specific values
- *STRING:n*: this is a string of maximum length n
- *RANGE*: this is an numeric, either integer or float, comprised in the range (see LIST RANGE)
- *NUMBER*: this is a simple numeric value, either integer or float

ENUM, STRING and RANGE are usually associated with RW, but not always. The default <type>, when omitted, is numeric, so either integer or float. Each driver is then responsible for handling values as either integer or float.

Note that float values are expressed using decimal (base 10) english-based representation, so using a dot, in non-scientific notation. So hexadecimal, exponents, and comma for thousands separator are forbidden. For example: "1200.20" is valid, while "1,200.20" and "1200,20" are invalid.

This replaces the old "VARTYPE" command.

9.4.5 DESC

Form:

```
GET DESC <upsname> <varname>
GET DESC su700 ups.status
```

Response:

```
DESC <upsname> <varname> "<description>"
DESC su700 ups.status "UPS status"
```

<*description*> is a string that gives a brief explanation of the named variable. upsd may return "Unavailable" if the file which provides this description is not installed.

Different versions of this file may be used in some situations to provide for localization and internationalization.

This replaces the old "VARDESC" command.

9.4.6 CMDDESC

Form:

```
GET CMDDESC <upsname> <cmdname>
GET CMDDESC su700 load.on
```

Response:

```
CMDDESC <upsname> <cmdname> "<description>"
CMDDESC su700 load.on "Turn on the load immediately"
```

This is like DESC above, but it applies to the instant commands.

This replaces the old "INSTCMDDESC" command.

9.4.7 TRACKING

Form:

```
GET TRACKING      (activation status of TRACKING)
GET TRACKING <id> (execution status of a command / setvar)
GET TRACKING 1bd31808-cb49-4aec-9d75-d056e6f018d2
```

Response:

```
ON                    (TRACKING feature is enabled)
OFF                   (TRACKING feature is disabled)
PENDING               (command execution is pending)
SUCCESS               (command was successfully executed)
ERR UNKNOWN           (command execution failed with unknown error)
ERR INVALID-ARGUMENT (command execution failed due to missing or invalid argument)
ERR FAILED            (command execution failed)
```

9.5 LIST

The LIST functions all share a common container format. They will return "BEGIN LIST" and then repeat the initial query. The list then follows, with as many lines are necessary to convey it. "END LIST" with the initial query attached then follows.

The formatting may seem a bit redundant, but it makes a different form of client possible. You can send a LIST query and then go off and wait for it to get back to you. When it arrives, you don't need complicated state machines to remember which list is which.

9.5.1 UPS

Form:

```
LIST UPS
```

Response:

```
BEGIN LIST UPS
UPS <upsname> "<description>"
...
END LIST UPS

BEGIN LIST UPS
UPS su700 "Development box"
END LIST UPS
```

<*upsname*> is a name from ups.conf, and <description> is the value of desc= from ups.conf, if available. It will be set to "Unavailable" otherwise.

This can be used to determine what values of <upsname> are valid before calling other functions on the server. This is also a good way to handle situations where a single upsd supports multiple drivers.

Clients which perform a UPS discovery process may find this useful.

9.5.2 VAR

Form:

```
LIST VAR <upsname>
LIST VAR su700
```

Response:

```
BEGIN LIST VAR <upsname>
VAR <upsname> <varname> "<value>"
...
END LIST VAR <upsname>

BEGIN LIST VAR su700
VAR su700 ups.mfr "APC"
VAR su700 ups.mfr.date "10/17/96"
...
END LIST VAR su700
```

This replaces the old "LISTVARS" command.

9.5.3 RW

Form:

```
LIST RW <upsname>
LIST RW su700
```

Response:

```
BEGIN LIST RW <upsname>
RW <upsname> <varname> "<value>"
...
END LIST RW <upsname>

BEGIN LIST RW su700
RW su700 output.voltage.nominal "115"
RW su700 ups.delay.shutdown "020"
...
END LIST RW su700
```

This replaces the old "LISTRW" command.

9.5.4 CMD

Form:

```
LIST CMD <upsname>
LIST CMD su700
```

Response:

```
BEGIN LIST CMD <upsname>
CMD <upsname> <cmdname>
...
END LIST CMD <cmdname>

BEGIN LIST CMD su700
CMD su700 load.on
CMD su700 test.panel.start
...
END LIST CMD su700
```

This replaces the old "LISTINSTCMD" command.

9.5.5 ENUM

Form:

```
LIST ENUM <upsname> <varname>
LIST ENUM su700 input.transfer.low
```

Response:

```
BEGIN LIST ENUM <upsname> <varname>
ENUM <upsname> <varname> "<value>"
...
END LIST ENUM <upsname> <varname>

BEGIN LIST ENUM su700 input.transfer.low
ENUM su700 input.transfer.low "103"
ENUM su700 input.transfer.low "100"
...
END LIST ENUM su700 input.transfer.low
```

This replaces the old "ENUM" command.

Note
this does not support the old "SELECTED" notation. You must request the current value separately.

9.5.6 RANGE

Form:

```
LIST RANGE <upsname> <varname>
LIST RANGE su700 input.transfer.low
```

Response:

```
BEGIN LIST RANGE <upsname> <varname>
RANGE <upsname> <varname> "<min>" "<max>"
...
END LIST RANGE <upsname> <varname>

BEGIN LIST RANGE su700 input.transfer.low
RANGE su700 input.transfer.low "90" "100"
RANGE su700 input.transfer.low "102" "105"
...
END LIST RANGE su700 input.transfer.low
```

9.5.7 CLIENT

Form:

```
LIST CLIENT <device_name>
LIST CLIENT ups1
```

Response:

```
BEGIN LIST CLIENT <device_name>
CLIENT <device name> <client IP address>
...
END LIST CLIENT <device_name>

BEGIN LIST CLIENT ups1
CLIENT ups1 ::1
CLIENT ups1 192.168.1.2
END LIST CLIENT ups1
```

9.6 SET

9.6.1 VAR

Form:

```
SET VAR <upsname> <varname> "<value>"
SET VAR su700 ups.id "My UPS"
```

Response:

```
OK                          (if TRACKING is not enabled)
OK TRACKING <id>            (if TRACKING is enabled)
ERR <message> [<extra>...] (see Error responses)
```

9.6.2 TRACKING

Form:

```
SET TRACKING <value>
SET TRACKING ON
SET TRACKING OFF
```

Response:

```
OK
ERR INVALID-ARGUMENT  (if <value> is not "ON" or "OFF")
ERR USERNAME-REQUIRED (if not yet authenticated)
ERR PASSWORD-REQUIRED (if not yet authenticated)
```

9.7 INSTCMD

Form:

```
INSTCMD <upsname> <cmdname> [<cmdparam>]
INSTCMD su700 test.panel.start
INSTCMD su700 load.off.delay 120
```

Note
<cmdparam> is an additional and optional parameter for the command.

Response:

```
OK                          (if TRACKING is not enabled)
OK TRACKING <id>            (if TRACKING is enabled)
ERR <message> [<extra>...] (see Error responses)
```

9.8 LOGOUT

Form:

```
LOGOUT
```

Response:

```
OK Goodbye      (recent versions)
Goodbye...      (older versions)
```

Used to disconnect gracefully from the server.

9.9 LOGIN

Form:

```
LOGIN <upsname>
```

Response:

```
OK       (upon success)
```

or various errors

Note
This requires "upsmon secondary" or "upsmon primary" in upsd.users

Use this to log the fact that a system is drawing power from this UPS. The upsmon primary will wait until the count of attached systems reaches 1 — itself. This allows the secondaries to shut down first.

Note
You probably shouldn't send this command unless you are upsmon, or a upsmon replacement.

9.10 PRIMARY (since NUT 2.8.0) or MASTER (deprecated)

Note
This command was renamed in NUT 2.8.0 to "PRIMARY" with the older name "MASTER" kept as deprecated alias for compatibility.

Form:

```
MASTER <upsname>
```

Response:

```
OK MASTER-GRANTED        (upon success)
```

Form:

```
PRIMARY <upsname>
```

Response:

```
OK PRIMARY-GRANTED       (upon success)
```

or various errors

Note
This requires "upsmon primary" in upsd.users

Note
Previously documented response was just `OK` — clients checking that server reply **starts with** that keyword would handle all cases.

This function doesn't do much by itself. It is used by upsmon to make sure that primary-mode functions like FSD are available if necessary.

9.11 FSD

Form:

```
FSD <upsname>
```

Response:

```
OK FSD-SET      (success)
```

or various errors

Note
This requires "upsmon primary" in upsd.users, or "FSD" action granted in upsd.users

---

upsmon in primary mode is the primary user of this function. It sets this "forced shutdown" flag on any UPS when it plans to power it off. This is done so that secondary systems will know about it and shut down before the power disappears.

Setting this flag makes "FSD" appear in a STATUS request for this UPS. Finding "FSD" in a status request should be treated just like a "OB LB".

It should be noted that FSD is currently a latch — once set, there is no way to clear it short of restarting upsd or dropping then re-adding it in the ups.conf. This may cause issues when upsd is running on a system that is not shut down due to the UPS event.

Note also that certain drivers can propagate the "FSD" state declared by the smarter UPSes themselves, e.g. when an UPS is charging after an outage and its battery level is below the "safe for load" threshold configured on the device itself. In this case the device usually does not power up its outlets automatically, but it can be forced by the systems administrator. The rationale behind such FSD during charging allows enough power to be guaranteed for systems to both boot and shut down safely, if the wall power disappears again, trading off prolonged unavailability of the shut down servers for the safety of their data. In such cases, administrators should be ready to disarm their upsmon clients until the batteries are charged, to avoid quick shutdowns of quickly restored servers — but only if they are sure about the wall power being restored for good (e.g. outage was due to maintenance).

9.12 PASSWORD

Form:

```
PASSWORD <password>
```

Response:

```
OK      (upon success)
```

or various errors

Sets the password associated with a connection. Used for later authentication for commands that require it.

9.13 USERNAME

Form:

```
USERNAME <username>
```

Response:

```
OK      (upon success)
```

or various errors

Sets the username associated with a connection. This is also used for authentication, specifically in conjunction with the upsd.users file.

9.14 STARTTLS

Form:

```
STARTTLS
```

Response:

```
OK STARTTLS
```

or various errors

This tells upsd to switch to TLS mode internally, so all future communications will be encrypted. You must also change to TLS mode in the client after receiving the OK, or the connection will be useless.

9.15 Other commands

- HELP: lists the commands supported by this server
- VER: shows the version of the server currently in use
- NETVER: shows the version of the network protocol currently in use (aliased as PROTVER since NUT v2.8.0, or formal protocol version 1.3)

These three are not intended to be used directly by programs. Humans can make use of this program by using telnet or netcat. If you use telnet, make sure you don't have it set to negotiate extra options. upsd doesn't speak telnet and will probably misunderstand your first request due to the extra junk in the buffer.

9.16 Error responses

An error response has the following format:

```
ERR <message> [<extra>...]
```

<message> is always one element; it never contains spaces. This may be used to allow additional information (<extra>) in the future.

<message> can have the following values:

- *ACCESS-DENIED*

  The client's host and/or authentication details (username, password) are not sufficient to execute the requested command.

- *UNKNOWN-UPS*

  The UPS specified in the request is not known to upsd. This usually means that it didn't match anything in ups.conf.

- *VAR-NOT-SUPPORTED*

  The specified UPS doesn't support the variable in the request.

  This is also sent for unrecognized variables which are in a space which is handled by upsd, such as server.*.

- *CMD-NOT-SUPPORTED*

  The specified UPS doesn't support the instant command in the request.

- *INVALID-ARGUMENT*

  The client sent an argument to a command which is not recognized or is otherwise invalid in this context. This is typically caused by sending a valid command like GET with an invalid subcommand.

- *INSTCMD-FAILED*

  upsd failed to deliver the instant command request to the driver. No further information is available to the client. This typically indicates a dead or broken driver.

- *SET-FAILED*

  upsd failed to deliver the set request to the driver. This is just like INSTCMD-FAILED above.

- *READONLY*

  The requested variable in a SET command is not writable.

- *TOO-LONG*

  The requested value in a SET command is too long.

- *FEATURE-NOT-SUPPORTED*

  This instance of upsd does not support the requested feature. This is only used for TLS/SSL mode (STARTTLS) at the moment.

- *FEATURE-NOT-CONFIGURED*

  This instance of upsd hasn't been configured properly to allow the requested feature to operate. This is also limited to START-TLS for now.

- *ALREADY-SSL-MODE*

  TLS/SSL mode is already enabled on this connection, so upsd can't start it again.

- *DRIVER-NOT-CONNECTED*

  upsd can't perform the requested command, since the driver for that UPS is not connected. This usually means that the driver is not running, or if it is, the ups.conf is misconfigured.

- *DATA-STALE*

  upsd is connected to the driver for the UPS, but that driver isn't providing regular updates or has specifically marked the data as stale. upsd refuses to provide variables on stale units to avoid false readings.

  This generally means that the driver is running, but it has lost communications with the hardware. Check the physical connection to the equipment.

- *ALREADY-LOGGED-IN*

  The client already sent LOGIN for a UPS and can't do it again. There is presently a limit of one LOGIN record per connection.

- *INVALID-PASSWORD*

  The client sent an invalid PASSWORD — perhaps an empty one.

- *ALREADY-SET-PASSWORD*

  The client already set a PASSWORD and can't set another. This also should never happen with normal NUT clients.

- *INVALID-USERNAME*

  The client sent an invalid USERNAME.

- *ALREADY-SET-USERNAME*

  The client has already set a USERNAME, and can't set another. This should never happen with normal NUT clients.

- *USERNAME-REQUIRED*

  The requested command requires a username for authentication, but the client hasn't set one.

- *PASSWORD-REQUIRED*

  The requested command requires a passname for authentication, but the client hasn't set one.

- *UNKNOWN-COMMAND*

  upsd doesn't recognize the requested command.

  This can be useful for backwards compatibility with older versions of upsd. Some NUT clients will try GET and fall back on REQ after receiving this response.

- *INVALID-VALUE*

  The value specified in the request is not valid. This usually applies to a SET of an ENUM type which is using a value which is not in the list of allowed values.

9.17 Future ideas

9.17.1 Dense lists

The LIST commands may be given the ability to handle options some day. For example, "LIST VARS <ups> +DESC" would return the current value like now, but it would also append the description of that variable.

9.17.2 Get collection

Allow to request only a subtree, which can be a collection, or a sub collection.

10 NUT developers tools

NUT provides several tools for clients and core developers, and QA people.

10.1 Device simulation

The dummy-ups driver propose a simulation mode, also known as *Dummy Mode*. This mode allows to simulate any kind of devices, even non existing ones.

Using this method, you can either replay a real life sequence, recorded from an actual device, or directly interact through 'upsrw` or by editing the device file, to modify the variables' values.

Here is an example to setup a device simulation:

- install NUT as usual, if not already done
- get a simulation file (`.dev`) or sequence (`.seq`), or generate one using the device recorder. Sample files are provided in the `data` directory of the NUT source. You can also download these from the development repository, such as the evolution500.seq.
- copy the simulation file to your sysconfig directory, like `/etc/nut` or `/etc/ups`
- configure NUT for simulation (ups.conf(5)):

```
[dummy]
        driver = dummy-ups
        port = evolution500.dev
        desc = "dummy-ups in dummy mode"
```

- now start NUT, at least `dummy-ups` and `upsd`:

```
$ upsdrvctl start dummy
$ upsd
```

- and check the data:

```
$ upsc dummy
...
```

- you can also use `upsrw` to modify the data in memory:

```
$ upsrw -s ups.status="OB LB" -u user -p password dummy
```

- or directly edit your copy of `/etc/nut/evolution500.seq`. In this case, modification will only apply according to the `TIMER` events and the current position in the sequence.

For more information, refer to dummy-ups(8) manual page.

10.2 Device recording

To complete `dummy-ups`, NUT provides a device recorder script called `nut-recorder.sh` and located in the *tools/* directory of the NUT source tree.

This script uses `upsc` to record device information, and stores these in a differential fashion every 5 seconds (by default).

Its usage is the following:

```
Usage: dummy-recorder.sh <device-name> [output-file] [interval]
```

For example, to record information from the device *myups* every 10 seconds:

```
nut-recorder.sh myups@localhost myups.seq 10
```

During the recording, you will want to generate power events, such as power failure and restoration. These will be tracked in the simulation files, and be eventually be replayed by the dummy-ups driver.

11 NUT core development and maintenance

This section is intended to people who want to develop new core features, or to do some maintenance.

11.1 NUT-specific autoconf macros

The following NUT-specific autoconf macros are defined in the m4/ directory.

- NUT_TYPE_SOCKLEN_T
- NUT_TYPE_UINT8_T
- NUT_TYPE_UINT16_T

```
Check for the corresponding type in the system header files, and
#define a replacement if necessary.
```

- NUT_CHECK_LIBGD
- NUT_CHECK_LIBNEON
- NUT_CHECK_LIBNETSNMP
- NUT_CHECK_LIBPOWERMAN

- NUT_CHECK_LIBOPENSSL
- NUT_CHECK_LIBNSS
- NUT_CHECK_LIBUSB
- NUT_CHECK_LIBWRAP

  Determine the compiler flags for the corresponding library. On success, set nut_have_libxxx="yes" and set LIBXXX_CFLAGS and LIBXXX_LDFLAGS. On failure, set nut_have_libxxx="no". This macro can be run multiple times, but will do the checking only once. Here "xxx" should of course be replaced by the respective library name.

  The checks for each library grow organically to compensate for various bugs in the libraries, pkg-config, etc. This is why we have a separate macro for each library.

- NUT_CHECK_IPV6

  Check for various features required to compile the IPv6 support. dnl Check for various features required for IPv6 support. Define a preprocessor symbol for each individual feature (HAVE_GETADDRINFO, HAVE_FREEADDRINFO, HAVE_STRUCT_ADDRINFO, HAVE_SOCKADDR_STORAGE, SOCKADDR_IN6, IN6_ADDR, HAVE_IN6_IS_ADDR_V4MAPPED, HAVE_AI_ADDRCONFIG). Also set the shell variable nut_have_ipv6=yes if all the required features are present. Set nut_have_ipv6=no otherwise.

- NUT_CHECK_OS

  Check for the exact system name and type.
  This was only used in the past to determine the packaging rule to be used through the OS_NAME variable, but may be useful for other purposes in the future.

- NUT_REPORT_FEATURE(FEATURE, VALUE, VARIABLE, DESCRIPTION)

  Schedule a line for the end-of-configuration feature summary. The FEATURE is a descriptive string such that the sentence "Checking whether to FEATURE" makes sense, and VALUE describes the decision taken (typically yes or no). The feature is also reported to the terminal.

  Also use VARIABLE and DESCRIPTION for defining AM_CONDITIONAL and AC_DEFINE (only if VALUE = "yes"). VARIABLE is of the form 'WITH_<NAME>'.

- NUT_REPORT(FEATURE, VALUE)

  Schedule a line for the end-of-configuration feature summary, without printing anything to the terminal immediately.

- NUT_PRINT_FEATURE_REPORT

  Print out a list of the features that have been reported by previous NUT_REPORT_FEATURE macro calls.

- NUT_ARG_WITH(FEATURE, DESCRIPTION, DEFAULT)

  Declare a simple --with-FEATURE option with the given DESCRIPTION and DEFAULT. Sets the variable nut_with_FEATURE.

11.2 NUT roadmap and ideas for future expansion

Here are some ideas that have come up over the years but haven't been implemented yet. This may be a good place to start if you're looking for a rainy day hacking project.

11.2.1 Roadmap

2.6

This release is focused on the website and documentation rewrite, using the excellent AsciiDoc.

2.8

This branch will focus on configuration and user interface improvements.

3.0

This major transition will mark the final switch to a complete power device broker.

11.2.2 Non-network "upsmon"

Some systems don't want a daemon listening to the network. This can be for security reasons, or perhaps because the system has been squashed down and doesn't have TCP/IP available. For these situations you could run a driver and program that sits on top of the driver socket to do local monitoring.

This also makes monitoring extremely easy to automate - you don't need to worry about usernames, passwords or firewalling. Just start a driver and drop this program on top of it.

- Parse ups.conf and open the state socket for a driver
- Send DUMPALL and enter a select loop
- Parse SETINFOs that change ups.status
- When you get OB LB, shut down

11.2.3 Completely unprivileged upsmon

upsmon currently retains root in a forked process so it can call the shutdown command. The only reason it needs root on most systems is that only privileged users can signal init or send a message on /dev/initctl.

In the case of systems running sysvinit (Slackware, others?), upsmon could just open /dev/initctl while it has root and then drop it completely. When it's time to shut down, fire a control structure at init across the lingering socket and tell it to enter runlevel 0.

This has been shown to work in local tests, but it's not portable. It could only be offered as an option for those systems which run that flavor of init. It also needs to be tested to see what happens to the lingering fd over time, such as when init restarts after an upgrade.

For other systems, there is always the possibility of having a suid program which does nothing but prod init into starting a shutdown. Lock down the group access so only upsmon's unprivileged user can access it, and make that your SHUTDOWNCMD. Then it could drop root completely.

11.2.4 Chrooted upsmon

upsmon could run the network monitoring part in a chroot jail if it had a pipe to another process running outside for NOTIFY dispatches. Such a pipe would have to be constructed extremely carefully so an attacker could not compromise it from the jailed process.

A state machine with a tightly defined sequence could do this safely. All it has to do is dispatch the UPS name and event type.

```
[start] [type] [length] <name> [stop]
```

11.2.5 Monitor program with interpreted language

Once in awhile, I get requests for a way to shut down based on the UPS temperature, or ambient humidity, or at a certain battery charge level, or any number of things other than an "OB LB" status. It should be obvious that adding a way to monitor all of that in upsmon would bloat upsmon for all those people who really don't need anything like that.

A separate program that interprets a list of rules and uses it to monitor the UPS equipment is the way to solve this. If you have a condition that needs to be tested, add a rule.

Some of the tools that such a language would need include simple greater-than/less-than testing (if battery.charge < 20), equivalence testing (if ups.model = "SMART-UPS 700"), and some way to set and clear timers.

Due to the expected size and limited audience for such a program, it might have to be distributed separately.

Note
Python may be a good candidate.

11.2.6 Sandbox

- check to refresh and integrate the tasks list and feature requests list from Alioth
- add "Generic ?Ascii? driver": I've got to think more about that, but the recent solar panel driver, and the powerman internal approach of a generic engine with a scripting interface is a cool idea. Ref http://powerman.svn.sourceforge.net/viewvc/-powerman/trunk/etc/apcpdu.dev?revision=969&view=markup
- integrate the (future) new powerman LUA engine (maybe/must-be used for the driver above?) for native PDU support
- see how we can help and collaborate with DeviceKit-power

A NUT command and variable naming scheme

RFC 9271 Recording Document
This document is defined by RFC 9271 published by IETF at https://www.rfc-editor.org/info/rfc9271 and is referenced as the document of record for the variable names and the instant commands used in the protocol described by the RFC.
On behalf of the RFC, this document records the names of variables describing the abstracted state of an UPS or similar power distribution device, and the instant commands sent to the UPS using command `INSTCMD`, as used in commands and messages between the Attachment Daemon (the `upsd` in case of NUT implementation of the standard) and the clients.

This document defines the standard names of NUT commands and variables (not to be confused with device status data described in the `docs/new-drivers.txt` in NUT source codebase).

Developers should use the names recorded here, with dstate functions and data mappings provided in NUT drivers for interactions with power devices.

If you need to express a state which cannot be described by any existing name, please make a request to the NUT developers' mailing list for definition and assignment of a new name. Clients using unrecorded names risk breaking at a future update. If you wish to experiment with new concepts before obtaining your requested variable name, you should use a name of the form `experimental.x.y` for those states.

Put another way: if you make up a name that is not in this list and it gets into the source code tree, and then the NUT community comes up with a better name later, clients that already use the undocumented variable will break when it is eventually changed. An explicitly "experimental" data point is less surprising in this regard.

Similarly, some source files (`drivers/*-mib.c` and `drivers/*-hid.c`) may mention data point names following the pattern of `unmapped.x.y`. These are generated by helper scripts which walk the reports from SNMP and USB HID devices, respectively `scripts/subdriver/gen-snmp-subdriver.sh` and `scripts/subdriver/gen-usbhid-subdriver.sh` and assign names based on strings in those reports. The `unmapped` entries should not be exposed in "production" builds of the NUT drivers. They are an aid for developers to know that such entries are served by their device, so an existing standard NUT name can be assigned for the concept (or new name negotiated with the community), but are normally hidden with `#if WITH_UNMAPPED_DATA_POINTS` clauses which can be enabled in custom NUT builds by use of `./configure --with-unmapped-data-points` option.

Note
In the descriptions, "opaque" means programs should not attempt to parse the value for that variable as it may vary greatly from one UPS (or similar device) to the next. These strings are best handled directly by the user.

A.1 Structured naming

All standard NUT names of variables and commands are structured, with a certain domain-specific prefix and purpose-specific suffix parts. NUT tools provide and interpret them as dot-separated strings (although third-party tools might restructure them by cutting and pasting at the dot separation location, e.g. to represent as a JSON data tree or as data model classes for specific programming languages).

If you would be making a parser of this information, please do also note that in some **but not all** cases there is a defined data point for some reading or command at the "root level" of what evolved to be a collection of further structured related information (and there are no guarantees for future evolution in this regard), for example:

- an `input.voltage` reports the momentary voltage level value and there is a `input.voltage.maximum` for a certain related detail;
- conversely, there are several items like `input.transfer.reason` but there is no actual `input.transfer` report.

There may be more layers than two (e.g. `input.voltage.low.warning`), and in certain cases detailed below there may be a variable component in the practical values (e.g. the `n` in `ambient.n.temperature.alarm` variable or `outlet.n.load.off` command names).

A.2 Time and Date format

When possible, dates should be expressed in ISO 8601 and RFC 3339 compatible Calendar format, that is to say "YYYY-MM-DD", or otherwise a Combined Date and Time representation (`<date>T<time>`, so "YYYY-MM-DDThh:mm"). Separators for the date (hyphen) and time (colon) components are required to conform to both ISO 8601 "extended" format and RFC 3339 required format.

In the case of Date and Time representation, a timezone can be added as per RFC 3339 and the newer revisions of the ISO 8601 standard (which allow for negative offsets):

- by appending the letter `Z` for UTC (e.g. "YYYY-MM-DDThh:mmZ"), or
- by appending the complete "hours:minutes" positive or negative time offsets from UTC (e.g. "YYYY-MM-DDThh:mm+03:00").

For more details see examples at Wikipedia page on ISO 8601 and the publicly available RFC at RFC 3339.

Other representations from those specifications are not necessarily supported.

A.3 Variables

A.3.1 device: General unit information

Note
some of these data will be redundant with ups.* information during a transition period. The ups.* data will then be removed.

Name	Description	Example value
device.model	Device model	BladeUPS
device.mfr	Device manufacturer	Eaton
device.serial	Device serial number (opaque string)	WS9643050926
device.type	Device type (ups, pdu, scd, psu, ats)	ups
device.description	Device description (opaque string)	Some ups
device.contact	Device administrator name (opaque string)	John Doe
device.location	Device physical location (opaque string)	1st floor
device.part	Device part number (opaque string)	123456789
device.macaddr	Physical network address of the device	68:b5:99:f5:89:27
device.uptime	Device uptime in seconds	1782
device.count	Total number of daisychained devices	1

Note
When present, `device.count` implies daisychain support. For more information, refer to the NUT daisychain support notes chapter of the user manual and developer guide.

A.3.2 ups: General unit information

Name	Description	Example value
ups.status	UPS status	OL
ups.alarm	UPS alarms	OVERHEAT
ups.time	Internal UPS clock time (opaque string)	12:34
ups.date	Internal UPS clock date (opaque string)	01-02-03
ups.model	UPS model	SMART-UPS 700
ups.mfr	UPS manufacturer	APC
ups.mfr.date	UPS manufacturing date (opaque string)	10/17/96
ups.serial	UPS serial number (opaque string)	WS9643050926
ups.vendorid	Vendor ID for USB devices	0463
ups.productid	Product ID for USB devices	0001
ups.firmware	UPS firmware (opaque string)	50.9.D
ups.firmware.aux	Auxiliary device firmware	4Kx
ups.temperature	UPS temperature (degrees C)	042.7
ups.load	Load on UPS (percent)	023.4
ups.load.high	Load when UPS switches to overload condition ("OVER") (percent)	100
ups.id	UPS system identifier (opaque string)	Sierra
ups.delay.start	Interval to wait before restarting the load (seconds)	0

Name	Description	Example value
ups.delay.reboot	Interval to wait before rebooting the UPS (seconds)	60
ups.delay.shutdown	Interval to wait after shutdown with delay command (seconds)	20
ups.timer.start	Time before the load will be started (seconds)	30
ups.timer.reboot	Time before the load will be rebooted (seconds)	10
ups.timer.shutdown	Time before the load will be shutdown (seconds)	20
ups.test.interval	Interval between self tests (seconds)	1209600 (two weeks)
ups.test.result	Results of last self test (opaque string)	Bad battery pack
ups.test.date	Date of last self test (opaque string)	07/17/12
ups.display.language	Language to use on front panel (* opaque)	E
ups.contacts	UPS external contact sensors (* opaque)	F0
ups.efficiency	Efficiency of the UPS (ratio of the output current on the input current) (percent)	95
ups.power	Current value of apparent power (Volt-Amps)	500
ups.power.nominal	Nominal value of apparent power (Volt-Amps)	500
ups.realpower	Current value of real power (Watts)	300
ups.realpower.nominal	Nominal value of real power (Watts)	300
ups.beeper.status	UPS beeper status (enabled, disabled or muted)	enabled
ups.type	UPS type (* opaque)	offline
ups.watchdog.status	UPS watchdog status (enabled or disabled)	disabled
ups.start.auto	UPS starts when mains is (re)applied	yes
ups.start.battery	Allow to start UPS from battery	yes
ups.start.reboot	UPS coldstarts from battery (enabled or disabled)	yes
ups.shutdown	Enable or disable UPS shutdown ability (poweroff)	enabled

Note

When present, the value of `ups.start.auto` has an impact on `shutdown.*` commands. For the sake of coherence, shutdown commands will set `ups.start.auto` to the right value before issuing the command. That is, `shutdown.stayoff` will first set `ups.start.auto` to `no`, while `shutdown.return` will set it to `yes`.

Note

When possible, time-stamps and dates should be expressed as detailed above in the Time and Date format chapter.

A.3.3 input: Incoming line/power information

Name	Description	Example value
input.voltage	Input voltage (V)	121.5
input.voltage.maximum	Maximum incoming voltage seen (V)	130

Name	Description	Example value
input.voltage.minimum	Minimum incoming voltage seen (V)	100
input.voltage.status	Status relative to the thresholds	critical-low
input.voltage.low.warning	Low warning threshold (V)	205
input.voltage.low.critical	Low critical threshold (V)	200
input.voltage.high.warning	High warning threshold (V)	230
input.voltage.high.critical	High critical threshold (V)	240
input.voltage.nominal	Nominal input voltage (V)	120
input.voltage.extended	Extended input voltage range	no
input.transfer.delay	Delay before transfer to mains (seconds)	60
input.transfer.reason	Reason for last transfer to battery (* opaque)	T
input.transfer.low	Low voltage transfer point (V)	91
input.transfer.high	High voltage transfer point (V)	132
input.transfer.low.min	smallest settable low voltage transfer point (V)	85
input.transfer.low.max	greatest settable low voltage transfer point (V)	95
input.transfer.high.min	smallest settable high voltage transfer point (V)	131
input.transfer.high.max	greatest settable high voltage transfer point (V)	136
input.sensitivity	Input power sensitivity	H (high)
input.quality	Input power quality (* opaque)	FF
input.current	Input current (A)	4.25
input.current.nominal	Nominal input current (A)	5.0
input.current.status	Status relative to the thresholds	critical-high
input.current.low.warning	Low warning threshold (A)	4
input.current.low.critical	Low critical threshold (A)	2
input.current.high.warning	High warning threshold (A)	10
input.current.high.critical	High critical threshold (A)	12
input.frequency	Input line frequency (Hz)	60.00
input.frequency.nominal	Nominal input line frequency (Hz)	60
input.frequency.status	Frequency status	out-of-range
input.frequency.low	Input line frequency low (Hz)	47
input.frequency.high	Input line frequency high (Hz)	63
input.frequency.extended	Extended input frequency range	no
input.transfer.boost.low	Low voltage boosting transfer point (V)	190
input.transfer.boost.high	High voltage boosting transfer point (V)	210
input.transfer.trim.low	Low voltage trimming transfer point (V)	230
input.transfer.trim.high	High voltage trimming transfer point (V)	240
input.load	Load on (ePDU) input (percent of full)	25
input.realpower	Current sum value of all (ePDU) phases real power (W)	300
input.realpower.nominal	Nominal sum value of all (ePDU) phases real power (W)	850
input.power	Current sum value of all (ePDU) phases apparent power (VA)	500
input.source	The current input power source	1
input.source.preferred	The preferred power source	1

Name	Description	Example value
input.phase.shift	Voltage dephasing between input sources (degrees)	181

A.3.4 output: Outgoing power/inverter information

Name	Description	Example value
output.voltage	Output voltage (V)	120.9
output.voltage.nominal	Nominal output voltage (V)	120
output.frequency	Output frequency (Hz)	59.9
output.frequency.nominal	Nominal output frequency (Hz)	60
output.current	Output current (A)	4.25
output.current.nominal	Nominal output current (A)	5.0

A.3.5 Three-phase additions

The additions for three-phase measurements would produce a very long table due to all the combinations that are possible, so these additions are broken down to their base components.

Phase Count Determination

`input.phases` (3 for three-phase, absent or 1 for 1phase)

`output.phases` (as for `input.phases`)

DOMAINs

Any input or output is considered a valid DOMAIN.

- input (should really be called input.mains, but keep this for compat)
  - input.bypass
  - input.servicebypass
- output (should really be called output.load, but keep this for compat)
  - output.bypass
  - output.inverter
  - output.servicebypass

Specification (SPEC)

Voltage, current, frequency, etc are considered to be a specification of the measurement.

With this notation, the old 1phase naming scheme becomes DOMAIN.SPEC

Example: `input.current`

CONTEXT

When in three-phase mode, we need some way to specify the target for most measurements in more detail. We call this the CONTEXT.

With this notation, the naming scheme becomes DOMAIN.CONTEXT.SPEC when in three-phase mode.

Example: `input.L1.current`

Valid CONTEXTs

```
L1-L2  \
L2-L3   \
L3-L1    for voltage measurements
L1-N    /
L2-N   /
L3-N  /

L1  \
L2  for current and power measurements
L3  /
N  - for current measurement
```

Valid SPECs

Note
For cursory readers — the following couple of tables lists just the short `SPEC` component of the larger `DOMAIN.CONTEXT.SPEC` naming scheme for phase-aware values, as discussed in other sections of this chapter.

Valid with/without context (i.e. per phase or aggregated/averaged)

Name	Description
alarm	Alarms for phases, published in ups.alarm
current	Current (A)
current.maximum	Maximum seen current (A)
current.minimum	Minimum seen current (A)
current.status	Status relative to the thresholds
current.low.warning	Low warning threshold (A)
current.low.critical	Low critical threshold (A)
current.high.warning	High warning threshold (A)
current.high.critical	High critical threshold (A)
current.peak	Peak current
voltage	Voltage (V)
voltage.nominal	Nominal voltage (V)
voltage.maximum	Maximum seen voltage (V)
voltage.minimum	Minimum seen voltage (V)
voltage.status	Status relative to the thresholds
voltage.low.warning	Low warning threshold (V)
voltage.low.critical	Low critical threshold (V)
voltage.high.warning	High warning threshold (V)
voltage.high.critical	High critical threshold (V)
power	Apparent power (VA)
power.maximum	Maximum seen apparent power (VA)
power.minimum	Minimum seen apparent power (VA)
power.percent	Percentage of apparent power related to maximum load
power.maximum.percent	Maximum seen percentage of apparent power
power.minimum.percent	Minimum seen percentage of apparent power
realpower	Real power (W)
powerfactor	Power Factor (dimensionless value between 0.00 and 1.00)
crestfactor	Crest Factor (dimensionless value greater or equal to 1)
load	Load on (ePDU) input

Valid without context (i.e. aggregation of all phases):

Name	Description
frequency	Frequency (Hz)
frequency.nominal	Nominal frequency (Hz)
realpower	Current value of real power (Watts)
power	Current value of apparent power (Volt-Amps)

A.3.6 EXAMPLES

Partial Three phase — Three phase example:

```
input.phases: 3
input.frequency: 50.0
input.L1.current: 133.0
input.bypass.L1-L2.voltage: 398.3
output.phases: 3
output.L1.power: 35700
output.powerfactor: 0.82
```

Partial Three phase — One phase example:

```
input.phases: 3
input.L2.current: 48.2
input.N.current: 3.4
input.L3-L1.voltage: 405.4
input.frequency: 50.1
output.phases: 1
output.current: 244.2
output.voltage: 120
output.frequency.nominal: 60.0
```

A.3.7 battery: Any battery details

Name	Description	Example value
battery.charge	Battery charge (percent)	100.0
battery.charge.approx	Rough approximation of battery charge (opaque, percent)	<85
battery.charge.low	Remaining battery level when UPS switches to LB (percent)	20
battery.charge.restart	Minimum battery level for UPS restart after power-off	20
battery.charge.warning	Battery level when UPS switches to "Warning" state (percent)	50
battery.charger.status	Status of the battery charger (see the note below)	charging
battery.voltage	Battery voltage (V)	24.84
battery.voltage.cell.max	Maximum battery voltage seen of the Li-ion cell (V)	3.44
battery.voltage.cell.min	Minimum battery voltage seen of the Li-ion cell (V)	3.41
battery.voltage.nominal	Nominal battery voltage (V)	024

Name	Description	Example value
battery.voltage.low	Minimum battery voltage, that triggers FSD status	21,52
battery.voltage.high	Maximum battery voltage (i.e. battery.charge = 100)	26,9
battery.capacity	Battery capacity (Ah)	7.2
battery.capacity.nominal	Nominal battery capacity (Ah)	8.0
battery.current	Battery current (A)	1.19
battery.current.total	Total battery current (A)	1.19
battery.status	Health status of the battery (opaque string)	ok
battery.temperature	Battery temperature (degrees C)	050.7
battery.temperature.cell.max	Maximum battery temperature seen of the Li-ion cell (degrees C)	25.85
battery.temperature.cell.min	Minimum battery temperature seen of the Li-ion cell (degrees C)	24.85
battery.runtime	Battery runtime (seconds)	1080
battery.runtime.low	Remaining battery runtime when UPS switches to LB (seconds)	180
battery.runtime.restart	Minimum battery runtime for UPS restart after power-off (seconds)	120
battery.alarm.threshold	Battery alarm threshold	0 (immediate)
battery.date	Battery installation or last change date (opaque string)	11/14/20
battery.date.maintenance	Battery next change or maintenance date (opaque string)	11/13/24
battery.mfr.date	Battery manufacturing date (opaque string)	2005/04/02
battery.packs	Number of internal battery packs	1
battery.packs.bad	Number of bad battery packs	0
battery.packs.external	Number of external battery packs	1
battery.type	Battery chemistry (opaque string)	PbAc
battery.protection	Prevent deep discharge of battery	yes
battery.energysave	Switch off when running on battery and no/low load	no
battery.energysave.load	Switch off UPS if on battery and load level lower (percent)	5
battery.energysave.delay	Delay before switch off UPS if on battery and load level low (min)	3
battery.energysave.realpower	Switch off UPS if on battery and load level lower (Watts)	10

NOTE: `battery.charger.status` replaces the historic flags `CHRG` and `DISCHRG` that were exposed through `ups.status`. The `battery.charger.status` can have one of the following values:

- `charging`: battery is charging,
- `discharging`: battery is discharging,
- `floating`: battery has completed its charge cycle, and waiting to go to resting mode,
- `resting`: the battery is fully charged, and not charging nor discharging.

Note
When possible, time-stamps and dates should be expressed as detailed above in the Time and Date format chapter.

A.3.8 ambient: Conditions from external probe equipment

Note
multiple sensors can be exposed using the indexed notation. *ambient.**, without index or using *0*, relates to the embedded sensor. For example: *ambient.temperature* represent the embedded sensor temperature. Other sensors (external, communication card, ...) can use indexes from *1* to *n*. For example: *ambient.1.temperature* for the first external sensor temperature.

Name	Description	Example value
ambient.n.present	Ambient sensor presence	yes
ambient.n.temperature	Ambient temperature (degrees C)	25.40
ambient.n.temperature.alarm	Temperature alarm (enabled/disabled)	enabled
ambient.n.temperature.status	Ambient temperature status relative to the thresholds	warning-low
ambient.n.temperature.high	Temperature threshold high (degrees C)	60
ambient.n.temperature.high.warning	Temperature threshold high warning (degrees C)	40
ambient.n.temperature.high.critical	Temperature threshold high critical (degrees C)	60
ambient.n.temperature.low	Temperature threshold low (degrees C)	5
ambient.n.temperature.low.warning	Temperature threshold low warning (degrees C)	10
ambient.n.temperature.low.critical	Temperature threshold low critical (degrees C)	5
ambient.n.temperature.maximum	Maximum temperature seen (degrees C)	37.6
ambient.n.temperature.minimum	Minimum temperature seen (degrees C)	18.1
ambient.n.humidity	Ambient relative humidity (percent)	038.8
ambient.n.humidity.alarm	Relative humidity alarm (enabled/disabled)	enabled
ambient.n.humidity.status	Ambient humidity status relative to the thresholds	warning-low
ambient.n.humidity.high	Relative humidity threshold high (percent)	80
ambient.n.humidity.high.warning	Relative humidity threshold high warning (percent)	70
ambient.n.humidity.high.critical	Relative humidity threshold high critical (percent)	80
ambient.n.humidity.low	Relative humidity threshold low (percent)	10
ambient.n.humidity.low.warning	Relative humidity threshold low warning (percent)	20
ambient.n.humidity.low.critical	Relative humidity threshold low critical (percent)	10
ambient.n.humidity.maximum	Maximum relative humidity seen (percent)	60
ambient.n.humidity.minimum	Minimum relative humidity seen (percent)	13
ambient.n.contacts.x.status	State of the dry contact sensor x	open

A.3.9 outlet: Smart outlet management

Note
n stands for the outlet index. A special case is "outlet.0" which is equivalent to "outlet" (without index), and represent the whole set of outlets of the device. The most important data is "outlet.count", used to iterate over the whole set of outlets. For more information, refer to the NUT outlets management and PDU notes chapter of the user manual.

Name	Description	Example value
outlet.count	Total number of outlets	12
outlet.switchable	General outlet switch ability of the unit (yes/no)	yes
outlet.n.id	Outlet system identifier (opaque string)	1
outlet.n.name	Outlet name (opaque string)	A1
outlet.n.desc	Outlet description (opaque string)	Main outlet
outlet.n.groupid	Identifier of the group to which the outlet belongs to	1
outlet.n.switch	Outlet switch control (on/off)	on
outlet.n.status	Outlet switch status (on/off)	on
outlet.n.alarm	Alarms for outlets and PDU, published in ups.alarm	outlet 1 low voltage warning
outlet.n.switchable	Outlet switch ability (yes/no)	yes
outlet.n.autoswitch.charge.low	Remaining battery level to power off this outlet (percent)	80
outlet.n.battery.charge.low	Remaining battery level to power off this outlet (percent)	80
outlet.n.delay.shutdown	Interval to wait before shutting down this outlet (seconds)	180
outlet.n.delay.start	Interval to wait before restarting this outlet (seconds)	120
outlet.n.timer.shutdown	Time before the outlet load will be shutdown (seconds)	20
outlet.n.timer.start	Time before the outlet load will be started (seconds)	30
outlet.n.current	Current (A)	0.19
outlet.n.current.maximum	Maximum seen current (A)	0.56
outlet.n.current.status	Current status relative to the thresholds	good
outlet.n.current.low.warning	Low warning threshold (A)	0.10
outlet.n.current.low.critical	Low critical threshold (A)	0.05
outlet.n.current.high.warning	High warning threshold (A)	0.30
outlet.n.current.high.critical	High critical threshold (A)	0.40
outlet.n.realpower	Current value of real power (W)	28
outlet.n.voltage	Voltage (V)	247.0
outlet.n.voltage.status	Voltage status relative to the thresholds	good
outlet.n.voltage.low.warning	Low warning threshold (V)	205
outlet.n.voltage.low.critical	Low critical threshold (V)	200
outlet.n.voltage.high.warning	High warning threshold (V)	230
outlet.n.voltage.high.critical	High critical threshold (V)	240
outlet.n.powerfactor	Power Factor (dimensionless, value between 0 and 1)	0.85
outlet.n.crestfactor	Crest Factor (dimensionless, equal to or greater than 1)	1.41
outlet.n.power	Apparent power (VA)	46

Name	Description	Example value
outlet.n.type	Physical outlet type	french

outlet.group: groups of smart outlets

This is a refinement of the outlet collection, providing grouped management for a set of outlets. The same principles and data than the outlet collection apply to outlet.group, especially for the indexing *n* and "outlet.group.count".

Most of the data published for outlets also apply to outlet.group, including: id, name (similar as outlet "desc"), status, current and voltage (including status, alarm and thresholds).

Some specific data to outlet groups exists:

Name	Description	Example value
outlet.group.n.type	Type of outlet group (OPAQUE)	outlet-section
outlet.group.n.color	Color-coding of the outlets in this group (OPAQUE)	yellow
outlet.group.n.count	Number of outlets in the group	12
outlet.group.n.phase	Electrical phase to which the physical outlet group (Gang) is connected to	L1

Example:

```
outlet.group.1.current: 0.00
outlet.group.1.current.high.critical: 16.00
outlet.group.1.current.high.warning: 12.80
outlet.group.1.current.low.warning: 0.00
outlet.group.1.current.nominal: 16.00
outlet.group.1.current.status: good
outlet.group.1.id: 1
outlet.group.1.name: Branch Circuit A
outlet.group.1.phase: L1
outlet.group.1.status: on
outlet.group.1.voltage: 244.23
outlet.group.1.voltage.high.critical: 265.00
outlet.group.1.voltage.high.warning: 255.00
outlet.group.1.voltage.low.critical: 180.00
outlet.group.1.voltage.low.warning: 190.00
outlet.group.1.voltage.status: good
...
outlet.group.count: 3.00
```

A.3.10 driver: Internal driver information

Name	Description	Example value
driver.name	Driver name	usbhid-ups
driver.version	Driver version (NUT release)	X.Y.Z
driver.version.internal	Internal driver version	1.23.45
driver.version.data	Version of the internal data mapping, for generic drivers	Eaton HID 1.31
driver.version.usb	USB library version	libusb-1.0.21
driver.parameter.xxx	Parameter xxx (ups.conf or cmdline -x) setting	(varies)
driver.flag.xxx	Flag xxx (ups.conf or cmdline -x) status	enabled (or absent)

Name	Description	Example value
driver.state	Current state in driver's lifecycle, primarily to help readers discern long-running init (with full device walk) or cleanup stages from the stable working loop	init.starting, init.quiet, init.device, init.info, init.updateinfo (first walk), reconnect.trying, reconnect.updateinfo, updateinfo, quiet, dumping, cleanup.upsdrv, cleanup.exit

A.3.11 server: Internal server information

Name	Description	Example value
server.info	Server information	Network UPS Tools upsd vX.Y.Z - https://www.networkupstools.org/
server.version	Server version	X.Y.Z

A.4 Instant commands

Name	Description
load.off	Turn off the load immediately
load.on	Turn on the load immediately
load.off.delay	Turn off the load possibly after a delay
load.on.delay	Turn on the load possibly after a delay
shutdown.return	Turn off the load possibly after a delay and return when power is back
shutdown.stayoff	Turn off the load possibly after a delay and remain off even if power returns
shutdown.stop	Stop a shutdown in progress
shutdown.reboot	Shut down the load briefly while rebooting the UPS
shutdown.reboot.graceful	After a delay, shut down the load briefly while rebooting the UPS
test.panel.start	Start testing the UPS panel
test.panel.stop	Stop a UPS panel test
test.failure.start	Start a simulated power failure
test.failure.stop	Stop simulating a power failure
test.battery.start	Start a battery test
test.battery.start.quick	Start a "quick" battery test
test.battery.start.deep	Start a "deep" battery test
test.battery.stop	Stop the battery test
test.system.start	Start a system test
calibrate.start	Start runtime calibration
calibrate.stop	Stop runtime calibration
bypass.start	Put the UPS in bypass mode
bypass.stop	Take the UPS out of bypass mode
reset.input.minmax	Reset minimum and maximum input voltage status
reset.watchdog	Reset watchdog timer (forced reboot of load)
beeper.enable	Enable UPS beeper/buzzer
beeper.disable	Disable UPS beeper/buzzer
beeper.mute	Temporarily mute UPS beeper/buzzer
beeper.toggle	Toggle UPS beeper/buzzer
outlet.n.shutdown.return	Turn off the outlet possibly after a delay and return when power is back
outlet.n.load.off	Turn off the outlet immediately
outlet.n.load.on	Turn on the outlet immediately

Name	Description
outlet.n.load.cycle	Power cycle the outlet immediately
outlet.n.shutdown.return	Turn off the outlet and return when power is back

B NUT daisychain support notes

NUT supports daisychained devices for any kind of device that proposes it. This chapter introduces:

- for developers: how to implement such mechanism,
- for users: how to manage and use daisychained devices in NUT in general, and how to take advantage of the provided features.

B.1 Introduction

It's not unusual to see some daisy-chained PDUs or UPSs, connected together in master-slave mode, to only consume 1 IP address for their communication interface (generally, network card exposing SNMP data) and expose only one point of communication to manage several devices, through the daisy-chain master.

This breaks the historical consideration of NUT that one driver provides data for one unique device. However, there is an actual need, and a smart approach was considered to fulfill this, while not breaking the standard scope (for base compatibility).

B.2 Implementation notes

B.2.1 General specification

The daisychain support uses the device collection to extend the historical NUT scope (1 driver — 1 device), and provides data from the additional devices accessible through a single management interface.

A new variable was introduced to provide the number of devices exposed: the `device.count`, which:

- defaults to 1
- if higher than 1, enables daisychain support and access to data of each individual device through `device.X.{...}`

To ensure backward compatibility in NUT, the data of the various devices are exposed the following way:

- `device.0` is a special case, for the whole set of devices (the whole daisychain). It is equivalent to `device` (without `.X` index) and root collections. The idea is to be able to get visibility and control over the whole daisychain from a single point.
- daisy-chained devices are available from `device.1` (master) to `device.N` (slaves).

That way, client applications that are unaware of the daisychain support, will only see the whole daisychain, as it would normally seem, and not nothing at all.

Moreover, this solution is generic, and not specific to the ePDU use case currently considered. It thus support both the current NUT scope, along with other use cases (parallel / serial UPS setups), and potential evolution and technology change (hybrid chain with UPS and PDU for example).

Devices status handling

To be clarified...

Devices alarms handling

Devices (master and slaves) alarms are published in `device.X.ups.alarm`, which may evolve into `device.X.alarm`. If any of the devices has an alarm, the main `ups.status` will publish an `ALARM` flag. This flag is be cleared once all devices have no alarms anymore.

Note
ups.alarm behavior is not yet defined (all devices alarms vs. list of device(s) that have alarms vs. nothing?)

Example

Here is an example excerpt of three PDUs, connected in daisychain mode, with one master and two slaves:

```
device.count: 3
device.mfr: EATON
device.model: EATON daisychain PDU
device.1.mfr: EATON
device.1.model: EPDU MI 38U-A IN: L6-30P 24A 1P OUT: 36XC13:6XC19
device.2.mfr: EATON
device.2.model: EPDU MI 38U-A IN: L6-30P 24A 1P OUT: 36XC13:6XC19
device.3.mfr: EATON
device.3.model: EPDU MI 38U-A IN: L6-30P 24A 1P OUT: 36XC13:6XC19
...
device.3.ups.alarm: high current critical!
device.3.ups.status: ALARM
...
input.voltage: ??? (proposal: range or list or average?)
device.1.input.voltage: 237.75
device.2.input.voltage: 237.75
device.3.input.voltage: 237.75
...
outlet.1.status: ?? (proposal: "on, off, off)
device.1.outlet.1.status: on
device.2.outlet.1.status: off
device.3.outlet.1.status: off
...
ups.status: ALARM
```

B.2.2 Information for developers

Note
these details are dedicated to the `snmp-ups` driver!

In order to enable daisychain support for a range of devices, developers have to do two things:

- Add a `device.count` entry in a mapping file (see `*-mib.c`)
- Modify mapping entries to include a format string for the daisychain index

Optionally, if there is support for outlets and / or outlet-groups, there is already a template formatting string. So you have to tag such templates with multiple definitions, to point if the daisychain index is the first or second formatting string.

Base support

In order to enable daisychain support on a mapping structure, the following steps have to be done:

- Add a "device.count" entry in the mapping file: snmp-ups will determine if the daisychain support has to be enabled (if more than 1 device). To achieve this, use one of the following type of declarations:

  a) point at an OID which provides the number of devices:

  ```
  { "device.count", 0, 1, ".1.3.6.1.4.1.13742.6.3.1.0", "1",
          SU_FLAG_STATIC, NULL },
  ```

  b) point at a template OID to guesstimate the number of devices, by walking through this template, until it fails:

  ```
  { "device.count", 0, 1, ".1.3.6.1.4.1.534.6.6.7.1.2.1.2.%i", "1",
          SU_FLAG_STATIC, NULL, NULL },
  ```

- Modify all entries so that OIDs include the formatting string for the daisychain index. For example, if you have the following entry:

  ```
  { "device.model", ST_FLAG_STRING, SU_INFOSIZE,
          ".1.3.6.1.4.1.534.6.6.7.1.2.1.2.0", ... },
  ```

  And if the last "0" of the the 4th field represents the index of the device in the daisychain, then you would have to adapt it the following way:

  ```
  { "device.model", ST_FLAG_STRING, SU_INFOSIZE,
          ".1.3.6.1.4.1.534.6.6.7.1.2.1.2.%i", ... },
  ```

Templates with multiple definitions

If there already exist templates in the mapping structure, such as for single outlets and outlet-groups, you also need to specify the position of the daisychain device index in the OID strings for all entries in the mapping table, to indicate where the daisychain insertion point is exactly.

For example, using the following entry:

```
{ "outlet.%i.current", 0, 0.001, ".1.3.6.1.4.1.534.6.6.7.6.4.1.3.0.%i",
        NULL, SU_OUTLET, NULL, NULL },
```

You would have to translate it to:

```
{ "outlet.%i.current", 0, 0.001, ".1.3.6.1.4.1.534.6.6.7.6.4.1.3.%i.%i",
        NULL, SU_OUTLET | SU_TYPE_DAISY_1, NULL, NULL },
```

`SU_TYPE_DAISY_1` flag indicates that the daisychain index is the first `%i` specifier in the OID template string. If it is the second one, use `SU_TYPE_DAISY_2`.

Devices alarms handling

Two functions are available to handle alarms on daisychain devices in your driver:

- `device_alarm_init()`: clear the current alarm buffer
- `device_alarm_commit(const int device_number)`: commit the current alarm buffer to "device.<device_number>.ups.a and increase the count of alarms. If the current alarms buffer is empty, the count of alarm is decreased, and the variable "device.<device_number>.ups.alarm" is removed from publication. Once the alarm count reaches "0", the main (device.0) ups.status will also remove the "ALARM" flag.

Note
When implementing a new driver, the following functions have to be called:

- `alarm_init()` at the beginning of the main update loop, for the whole daisychain. This will set the alarm count to "0", and reinitialize all alarms,
- `device_alarm_init()` at the beginning of the per-device update loop. This will only clear the alarms for the current device,
- `device_alarm_commit()` at the end of the per-device update loop. This will flush the current alarms for the current device,
- also `device_alarm_init()` at the end of the per-device update loop. This will clear the current alarms, and ensure that this buffer will not be considered by other subsequent devices,
- `alarm_commit()` at the end of the main update loop, for the whole daisychain. This will take care of publishing or not the "ALARM" flag in the main ups.status (`device.0`, root collection).

C NUT libraries complementary information

This chapter provides some complementary information about the creation process of NUT libraries, and using these in your program.

C.1 Introduction

NUT provides several libraries, to ease interfacing with 3rd party programs:

- **libupsclient**, to interact with NUT server (upsd),
- **libnutclient**, to interact with NUT server at high level,
- **libnutscan**, to discover NUT supported devices.

External applications, such as asapm-ups, wmnut, and others, currently use it. But it is also used internally (by upsc, upsrw, upscmd, upsmon and dummy-ups) to reduce storage footprint and memory consumption.

The runtime libraries are installed by default. However, to install other development files (header, additional static and shared libraries, and compilation helpers below), you will have to provide the *--with-dev* flag to the *configure* script.

C.2 libupsclient-config

In case pkgconfig is not available on the system, an alternate helper script is provided: *libupsclient-config*.

It will be installed in the same directory as NUT client programs (BINDIR), providing that you have enabled the *--with-dev* flag to the *configure* script.

The usage is about the same as pkg-config and similar tools.

To get CFLAGS, use:

```
$ libupsclient-config --cflags
```

To get LD_FLAGS, use:

```
$ libupsclient-config --libs
```

References: libupsclient-config(1) manual page,

Note
this feature may evolve (name change), or even disappear in the future.

C.3 pkgconfig support

pkgconfig enables a high level of integration with minimal effort. There is no more needs to handle hosts of possible NUT installation path in your configure script !

To check if NUT is available, use:

```
$ pkg-config --exists libupsclient --silence-errors
```

To get CFLAGS, use:

```
$ pkg-config --cflags libupsclient
```

To get LD_FLAGS, use:

```
$ pkg-config --libs libupsclient
```

pkgconfig support (*.pc*) files are provided in the present directory of the source distribution (*nut-X.Y.Z/lib/*), and installed in the suitable system directory if you have enabled *--with-dev*.

The default installation directory ("/usr/lib/pkgconfig/") can be changed with the following command:

```
./configure --with-pkgconfig-dir=PATH
```

You can also use this if you are sure that pkg-config is installed:

```
PKG_CHECK_MODULES(LIBUPSCLI, libupsclient >= 2.4.0)
PKG_CHECK_MODULES(LIBNUTSCAN, libnutscan >= 2.6.2)
```

C.4 Example configure script

To use NUT libraries in your program, use the following code in your configure (.in or .ac) script:

```
AC_MSG_CHECKING(for NUT client library (libupsclient))
pkg-config --exists libupsclient --silence-errors
if test $? -eq 0
then
        AC_MSG_RESULT(found (using pkg-config))
        LDFLAGS="$LDFLAGS `pkg-config --libs libupsclient`"
        NUT_HEADER="`pkg-config --cflags libupsclient`"
else
        libupsclient-config --version
        if test $? -eq 0
        then
                AC_MSG_RESULT(found (using libupsclient-config))
                LDFLAGS="$LDFLAGS `libupsclient-config --libs`"
                NUT_HEADER="`libupsclient-config --cflags`"
        else
                AC_MSG_ERROR("libupsclient not found")
        fi
fi
```

This code will test for pkgconfig support for NUT client library, and fall back to libupsclient-config if not available. It will issue an error if none is found!

The same is also valid for other NUT libraries, such as libnutscan. Simply replace *libupsclient* occurrences in the above example, by the name of the desired library (for example, *libnutscan*).

Note
this is an alternate method. Use of PKG_CHECK_MODULES macro should be preferred.

C.5 Future consideration

We are considering the following items:

- provide libupsclient-config support for libnutscan, and libnutconfig when available. This requires to rename and rewrite the script in a more generic way (libnut-config), with options to address specific libraries.
- provide pkgconfig support for libnutconfig, when available.

C.6 Libtool information

NUT libraries are built using Libtool. This tool is integrated with automake, and can create static and dynamic libraries for a variety of platforms in a transparent way.

References:

- libtool
- David MacKenzie's Autobook (RedHat)
- DebianLinux.Net, The GNU Build System