LCDproc Developer's Guide

Table of Contents

Table of Contents

LCDproc Developer's Guide

The Hitchhiker's Guide to LCDproc 0.5

Markus Dolze

Peter Marschall

Guillaume Filion

0.5.x

Abstract

This document is a guide to LCDproc written for developers. It covers LCDproc 0.5.x. Users should read the user guide.

Table of Contents

List of Examples

ChapterÂ 1.Â Introduction

Table of Contents

About this Document

This document is meant to be a reference for LCDproc developers. It tries to indicate you where to find the relevant information about LCDproc's inner workings.

Note

Please note that this document is still "under construction". If you run into any trouble feel free to write to the LCDproc mailing list. See https://lists.lcdproc.org/wws/info/lcdproc for details on how to subscribe to the list.

Therefore you might want to have a look at https://github.com/lcdproc/lcdproc/tree/master/docs, to get the latest version of this document, unless you want to generate it yourself from the docbook files in GIT).

This document was written for LCDproc 0.5.

In several other places e-mails and other documents have been included in this document. The authors of those are listed below every such document.

ChapterÂ 2.Â The LCDproc client language

Table of Contents

Introduction

The LCDproc clients, for example lcdproc, connect over the network to LCDd. In their communication they use a protocol, often referred to as the "widget language". In this chapter the widget language will be discussed.

Opening a session

The essence of talking to LCDd is quite simple. First you will need to connect to the LCDproc port (usually 13666) on the correct IP address (by default localhost). Once you have established the connection you should say "hello", to let LCDd know you are a good guy. It will respond by telling some LCDproc data, like version and screen width and height. Now your session is open and you can start sending 'real' commands.

LCDd can send a number of strings itself. As a response to your commands, it will usually send a "success" string, or a string starting with "huh" in case of any error. See further below for other strings sent by LCDd.

You can test all these commands by opening a TCP/IP connection manually, like with:

```
telnet localhost 13666
```

This way, you can check how the various commands work. It's in this case best to have no other clients. If you do have other clients, you will receive "listen" and "ignore" messages that will disturb your typing.

Command reference

In this section all commands and their parameters are listed, along with the responses you can expect. If you need a space or a special char in a string, you should quote the string with double quotes. If you need to use a double quote, escape it with a backslash. Escaping also works for "\n\r\t", however not all widget types or display drivers will handle these characters well. The listing is divided into subsections for

1. Basic stuff
2. Screens and widgets
3. Menu stuff
4. Miscellaneous

Basic stuff

hello

Opens the session with the LCDd server program. This command is required before other commands can be issued.

The response will be a string in the format:

`connect` *`parameter`*`...`

The client should read all parameters it needs and store their values. The following parameters are in use:

`LCDproc` *`version`*
: Indicates the version number of LCDd.

`protocol` *`version`*
: Indicates the widget language version number. This number is only changed when the language of a newer version has become incompatible with the previous version.

Note

Each part of the version number shall be treated as an independent numeric value. This means that 0.9 is followed by 0.10.

`lcd`
: This word introduces the next key / value pairs that describe the display's properties.

`wid` *`int`*
: Tells the client the width of the attached display device in characters.

`hgt` *`int`*
: Tells the client the height of the attached display device in characters.

`cellwid` *`int`*
: How many pixels is a character wide (space between character cells not included)

`cellhgt` *`int`*
: How many pixels is a character high (space between character cells not included)

client_set `-name` ***`name`***

Sets attributes for the current client. The current client is the one from the connection that you send this command on, in other words: yourself.

`name` is the client's name as visible to a user.

Screens and widgets

screen_add ***`new_screen_id`***

Adds a screen to be displayed. The screen will be identified by the string *`new_screen_id`*, which is used later when manipulating on the screen.

screen_del ***`screen_id`***

Removes the screen identified by *`screen_id`* from the client's screens.

screen_set ***`screen_id`*** ***`attributes`*`...`**

Sets attributes for the given screen. The following attributes exist:

`-name` *`name`*

Sets the screen's name as visible to a user.

`-wid` *`int`* `-hgt` *`int`*

Sets the size of the screen in characters. If unset, the full display size is assumed.

`-priority` *`pri_class`*

Sets the screen's priority. The following priority classes exist:

`hidden`
: The screen will never be visible

`background`
: The screen is only visible when no normal info screens exists

`info`
: normal info screen, default priority

`foreground`
: an active client

`alert`
: The screen has an important message for the user.

`input`
: The client is doing interactive input.

`int`
: a positive integer that maps to priority classes above according to the mapping given in the table below.

range	priority
1 - 64	foreground
65 - 192	info
193 - â	background

LCDd will only show screens with the highest priority at that moment. So when there are three `info` screens and one `foreground` screen, only the `foreground` screen will be visible. Only `background`, `info` and `foreground` screens will rotate; higher classes do not rotate because their purpose is not suitable for rotation.

`-heartbeat { on | off | open }`

Changes the heartbeat setting for this screen. If set to `open`, the default, the client's heartbeat setting will be used.

`-backlight { on | off | toggle | open | blink | flash }`

Changes the screen's backlight setting. If set to the default value `open`, the state will be determined by the client's setting. `blink` is a moderately striking backlight variation, `flash` is *very* striking.

`-duration` *`value`*

A screen will be visible for this amount of time every rotation. The *`value`* is in eights of a second.

`-timeout` *`value`*

After the screen has been visible for a total of this amount of time, it will be deleted. The *`value`* is in eights of a second. Currently the client will not be informed of the deletion (TODO?).

`-cursor { on | off | under | block}`

Determines the visibility of a cursor. If `on`, a cursor will be visible. Depending on your hardware, this will be a hardware or software cursor. The specified cursor shape (`block` or `under`) might not be available in which case an other cursor shape will be used instead. Default is `off`.

`-cursor_x` *`int`* `-cursor_y` *`int`*
: Set the cursor's x and y coordinates respectively. If not given, the cursor will be set to the leftmost (`-cursor_x`) resp. topmost (`-cursor_y`) position. Coordinates are always 1-based. So the default top-left corner is denoted by (1,1).

key_add ***`screen_id`* *`key`*...** since protocol version 0.4
: Reserves key(s) for interaction of the user with the screen. Reservations with this command always take precedence over reservations with **client_add_key**, but are only in effect while the screen is visible on the display. The server message on key events is appended with *`screen_id`* to distinguish it from client level key events.

  Only one success message is generated for all keys together.

key_del ***`screen_id`* *`key`*...** since protocol version 0.4
: Lift reservation of key(s) by **key_add**. Arguments are processed sequentially, and for each one either a success message is generated, or an error message if the key was not found in the list of reserved keys.

widget_add ***`screen_id`* *`new_widget_id`* *`widgettype`* [`-in` *`frame_id`*]**
: Adds a widget to the given screen. The *`new_widget_id`* sets the identifier for this widget. The optional `-in` *`frame_id`* places the widget into the given frame. The following widget types exist:

  `string`
  : A simple text.

  `title`
  : A title bar on top of the screen.

  `hbar`
  : A horizontal bar.

  `vbar`
  : A vertical bar.

  `pbar`
  : A percentage / progress bar. This widget-type is only available on servers which report a widget language version of 0.4 or higher in their hello response.

  `icon`
  : A predefined icon. For a list of valid names consult `server/widget.c.`

  `scroller`
  : A variation of the string type that scrolls the text horizontally or vertically.

  `frame`
  : A frame with that can contain widgets itself. In fact a frame displays an other screen in it.

  `num`
  : A big number. They have a size of 3x4 characters. The special number 10 is a colon, that you can use for a clock. This character is 1x4.

widget_del ***`screen_id`* *`widget_id`***
: Deletes the given widget from the screen.

widget_set ***`screen_id`* *`widget_id`* *`widgettype_specific_parameters`***
: Sets parameters for a widget. Because not all widgets are created equal, the various widget types require different parameters.

  `string`
  : *`x`* *`y`* *`text`*

    Displays *`text`* at position (*`x`*,*`y`*).

  `title`
  : *`text`*

Uses *`text`* as the title to display.

`hbar`, `vbar`
: *`x`* *`y`* *`length`*

  Displays a horizontal (`hbar`) resp. vertical (`vbar`) starting at position (*`x`*,*`y`*) that is *`length`* pixels wide resp. high.

`pbar`
: *`x`* *`y`* *`width`* *`promille`* [*`begin-label`*] [*`end-label`*]

  Displays a horizontal (`pbar`) and its optional (`begin-label`) and (`end-label`) labels covering *`width`* characters starting at position (*`x`*,*`y`*) where a fraction of (*`promille`* / 1000) is filled.

`icon`
: *`x`* *`y`* *`iconname`*

  Displays the icon *`iconname`* at position (*`x`*,*`y`*).

`scroller`
: *`left`* *`top`* *`right`* *`bottom`* *`direction`* *`speed`* *`text`*

  Displays a scroller spanning from position (*`left`*,*`top`*) to (*`right`*,*`bottom`*) scrolling *`text`* in horizontal (`h`), vertical (`v`) or marquee (`m`) direction at a speed of *`speed`*, which is the number of movements per rendering stroke (8 times/second).

`frame`
: *`left`* *`top`* *`right`* *`bottom`* *`width`* *`height`* *`direction`* *`speed`*

  Sets up a frame spanning from (*`left`*,*`top`*) to (*`right`*,*`bottom`*) that is *`width`* columns wide and *`height`* rows high. It scrolls in either horizontal (`h`) or vertical (`v`) direction at a speed of *`speed`*, which is the number of movements per rendering stroke (8 times/second).

  ### Note

  In the current implementation frames can only scroll vertically and only string and hbar widgets work inside frames.

`num`
: *`x`* *`int`*

  Displays decimal digit *`int`* at the horizontal position *`x`*, which is a normal character x coordinate on the display. The special value 10 for *`int`* displays a colon.

Menu stuff

In this section all commands for creation, modification of menus and for interaction with them are described. Although keys may be used for other tasks they are listed here too.

TODO: example for normal (static) menu structure.

Menus may be even be used for wizards (the user is automatically guided through a number of configuration options) by virtue of the options -next and -prev. Here a complete example:

```
client_set name Parenttest
# to be entered on escape from test_menu (but overwritten
# for test_{checkbox,ring})
menu_add_item "" ask menu "Leave menus?" -is_hidden true
  menu_add_item "ask" ask_yes action "Yes" -next _quit_
  menu_add_item "ask" ask_no action "No" -next _close_

menu_add_item "" test menu "Test"
  menu_add_item "test" test_action action "Action"
  menu_add_item "test" test_checkbox checkbox "Checkbox"
  menu_add_item "test" test_ring ring "Ring" -strings "one\ttwo\tthree"
  menu_add_item "test" test_slider slider "Slider" -mintext "" -maxtext "" -value "50"
  menu_add_item "test" test_numeric numeric "Numeric" -value "42"
  menu_add_item "test" test_alpha alpha "Alpha" -value "abc"
  menu_add_item "test" test_ip ip "IP" -v6 false -value "192.168.1.1"
  menu_add_item "test" test_menu menu "Menu"
  menu_add_item "test_menu" test_menu_action action "Submenu's action"

# no successor for menus. Since test_checkbox and test_ring have their
# own predecessors defined the "ask" rule will not work for them
menu_set_item "" test -prev "ask"

menu_set_item "" test_action -next "test_checkbox"
menu_set_item "" test_checkbox -next "test_ring" -prev "test_action"
menu_set_item "" test_ring -next "test_slider" -prev "test_checkbox"
menu_set_item "" test_slider -next "test_numeric" -prev "test_ring"
menu_set_item "" test_numeric -next "test_alpha" -prev "test_slider"
menu_set_item "" test_alpha -next "test_ip" -prev "test_numeric"
menu_set_item "" test_ip -next "test_menu" -prev "test_alpha"
menu_set_item "" test_menu_action -next "_close_"

# replace the main menu with the client's menu as created above
menu_set_main ""
```

client_add_key [`-exclusively` | `-shared`] *`key`*...

Tells the server that the current client wants to make use of the given key(s). If you reserve the key(s) in shared mode, other clients can still reserve these keys too. If you reserve the key(s) in exclusive mode no other client can reserve them again. Shared mode is the default.

Key(s) reserved in shared mode will only be returned when a screen of the current client is active. These keys can be used for interaction with a visible screen. Key(s) reserved in exclusive mode will be returned regardless of which screen is active. They can be used to trigger a special feature or to make a screen come to foreground.

Note that you cannot reserve a key in exclusive mode when an other client has reserved it in shared mode. However a key can be reserved in exclusive mode and on a per screen basis (with **key_add**) just fine. In this case the per screen reservation wins.

Starting with protocol version 0.4 for each *`key`* argument the server answers with either a success message or an error message if there is a reservation conflict.

client_del_key *`key...`*

Ends the reservation of the given key(s). Always returns exactly one success message.

menu_add_item *`menu_id`* *`new_item_id`* *`type`* [*`text`*] [*`item_specific_options`*]

Adds a new menu item to a menu. The main menu of a client, will be created automatically as soon as the client adds an item. This main menu has an empty id ("") and the name is identical to the name of the client. The item specific options are described under menu_set_item below. Use of *`text`* is optional and is a shortcut for "-text *`text`*" option.

Note:

`menu_ids` need to be *unique* (at least within a clients menu hierarchy).

If you want to use a text label that starts with a '-' (minus) character, you have to use the "-text `text`" option.

menu item types

`action`
: This item should trigger an action. It consists of simple text.

`checkbox`
: Consists of a text and a status indicator. The status can be on (Y), off (N) or gray (o).

`ring`
: Consists of a text and a status indicator. The status can be one of the strings specified for the item.

`slider`
: Is visible as a text. When selected, a screen comes up that shows a slider. You can set the slider using the cursor keys. When Enter is pressed, the menu returns.

`numeric`
: Allows the user to input an integer value. Is visible as a text. When selected, a screen comes up that shows the current numeric value, that you can edit with the cursor keys and Enter. The number is ended by selecting a 'null' input digit. After that the menu returns.

`alpha`
: Is visible as a text. When selected, a screen comes up that shows the current string value, that you can edit with the cursor keys and Enter. The string is ended by selecting a 'null' input character. After that the menu returns.

`ip`
: Allows the user to input an IP number (v4 or v6). When selected, a screen comes up that shows an IP number that can be edited - digit by digit - via left/right (switch digit) and up/down keys (increase/decrease).

`menu`
: This is a submenu. It is visible as a text, with an appended >. When selected, the submenu becomes the active menu.

menu_del_item ***`menu_id`*** ***`item_id`*** (version 0.3) , **menu_del_item [*`ignored`*]** ***`item_id`*** (version 0.4)
: Removes a menu item `item_id` from menu `menu_id`. The menu with the special id "" (i.e. the empty string) is the client's main menu.

menu_set_item ***`menu_id`*** ***`item_id`*** ***`item_specific_options`*** (version 0.3) , **menu_set_item** ***`""`*** ***`item_id`*** ***`item_specific_options`*** (version 0.4)
: Sets parameters for the menu item. Each item type knows different parameters. Older versions of LCDd checked that `menu_id` is valid, but otherwise ignored the parameter. Newer version dropped this unnecessary check. It is recommended to pass an empty string, which is compatible with all versions.

options for the various menu items

for all item types

`-text` *`string`*
: The visible text of the item.

`-is_hidden { true | false}` (false)
: If the item currently should not appear in a menu.

`-next` *`successor_id`*
: Sets the menu item to show after hitting the ENTER key when this item is active. This works for *all* menu item types *except menus* i.e. also for menu item types without an own screen e.g., checkbox, ring and action.

  Special values

  `_close_`
  : Equivalent to `-menu_result close`: Close the menu.

  `_quit_`
  : Equivalent to `-menu_result quit`: Quit the menu system.

  `_none_`
  : Equivalent to `-menu_result none`: Keep the item open.

`-prev` *`predecessor_id`*
: Sets the menu item to show after hitting the ESCAPE key when this Item is active. This works for *all* menu item types i.e. also for menu item types without an own screen e.g., checkbox, ring and action.

  Note:

  If you define a predecessor for e.g., a checkbox and its parent menu too, the menu's predecessor is ignored in favor of the checkboxes one.

  This option accepts the same special values as the `-next` option.

`action`

`-menu_result { none | close | quit}` (none)
: Sets what to do with the menu when this action is selected: none: the menu stays as it is; close: the menu closes and returns to a higher level; quit: quits the menu completely so you can foreground your app.

`checkbox`

`-value { off | on | gray }`
: Set the value of the item.

`-allow_gray { false | true}` (false)
: Sets if a grayed checkbox is allowed.

`ring`

`-value` *`int`* (0)
: Sets the index in the stringlist that is currently selected.

`-strings` *`string`* (empty)
: This single string should contain the strings that can be selected. They should be tab-separated (\t).

`slider`

`-value` *`int`* (0)
: Sets its current value.

`-mintext` *`string`*`("")`, `-maxtext` *`string`*`("")`

The texts at the left and right side of the slider.

`-minvalue` *`int`* (0) , `-maxvalue` *`int`* (100)
: The minimum and maximum values of the slider.

`-stepsize` *`int`* (1)
: The stepsize of the slider. If you use 0, you can control the movement completely from your client.

`numeric`

`-value` *`int`* (0)
: Sets its current value.

`-minvalue` *`int`* (0) , `-maxvalue` *`int`* (100)
: The minimum and maximum values that are allowed. If one of them is negative, the user will be able to enter negative numbers too.

  TODO: floats!

`alpha`

`-value` *`string`* ("")
: Sets its current value.

`-password_char` *`string`* ("")
: If used, instead of the typed characters, this character will be visible.

`-minlength` *`int`* (0) , `-maxlength` *`int`* (10)
: Sets the minimum and maximum allowed lengths.

`-allow_caps { false | true }` (true) , `-allow_noncaps { false | true }` (false) , `-allow_numbers { false | true }` (false)
: (Dis)allow these groups of characters.

`-allowed_extra` *`string`* ("")
: The chars in this string are also allowed.

`ip`

`-value` *`string`* ("192.168.1.245")
: Set the value of the item, e.g. "192.168.1.245" (v4) or ":::ffff:ffff:ffff:ffff:ffff" (v6).

`-v6 { false | true }` (false)
: Changes IP version from default v4.

`menu`
: This is a submenu. It is visible as a text, with an appended '>'. When selected, the submenu becomes the active menu.

`-parent` *`parentid`*
: (Re)sets the parent of this menu. Parentid has to be of type menu. This function does not change any menu (neither the old nor the new parent) since this option is normally used with hidden menus. Otherwise use menu_add/del_item. Applying this option is equivalent to second argument of the menu_goto command.

menu_goto *`menu_id`* [*`parent_id`*]

Changes current menu to *`menu_id`*. Depending on the configure option `--enable-permissive-menu-goto` the client may switch to any (if enabled) or his menus only (if not enabled).

`menu_id`
: The menu item to go to (any menu type e.g. an action or a menu). Special values:

`_quit_`
: Quits the client menu. This is only available on servers which report a widget language version of 0.4 or higher in their hello response.

`parent_id`
: Resets the parent of *`menu_id`*. This optional parameter can be used to reuse a menu from different places (for wizards etc.). Use it with caution: This may lead to a messy menu structure in particular due to the fact that the menus are not changed !

menu_set_main *`menu_id`*
: Sets the entry point into the menu system. Use this to make the server menu invisible. Note that you may only set the menu to your own clients menus unless the configure option `--enable-permissive-menu-goto` is used. (See `menuscreens.c` for the menu ids of the server menus.)

`menu_id`
: The new main menu, restricted to the client's own menus. Special values:

"" (i.e. the empty string)
: The client's main menu.

`_main_`
: Resets main to the "real" main menu.

Miscellaneous

backlight { `on` | `off` | `toggle` | `blink` | `flash` }
: Sets the client's backlight state.

output { `on` | `off` | *`int`* }
: Sets the general purpose output on some display modules to this value. Use `on` to set all outputs to high state, and `off` to set all to low state. The meaning of the integer value depends on your specific device, usually it is a bit pattern describing the state of each output line.

info
: This command provides information about the driver.

noop
: This command does nothing and is always successful. Can be useful to be sent at regular intervals to make sure your connection is still alive.

sleep *`int`*
: Sleep for the given number of seconds. *`int`* must be a positive integer in the range from 1 to 60.

Note:

This command is currently ignored on the server side.

LCDd messages

LCDd can send messages back to the client. These messages can be directly related to the last command, or generated for some other reason. Because messages can be generated at any moment, the client should read from the connection at regular intervals. A very simple client could simply ignore all received messages. Not reading the messages will cause trouble !

`success`
: This is the response to a command in case everything went OK.

`huh? error_description`
: This is the response to a command in case something has gone wrong. The description is not meant to be parsed, it's only meant for the programmer of the client. It might be that your command has only been partially executed, for example if you try to reserve 3 keys, and one fails. Your client might need to undo its actions completely.

`listen` *`screen_id`*, `ignore` *`screen_id`*
: The screen with the *`screen_id`* given is now visible on the display (`listen`) or it is not visible anymore on the display (`ignore`).

`key` *`key`*
: This message will be sent if there was a keypress that should be delivered to the current client.

`menuevent` *`event_type id`* [*`value`*]
: The user did something with a client supplied menu. The type of event can be:

  `select` (action)
  : The item was activated.

  `update` (checkbox, ring, numeric, alpha)
  : The item was modified by the user, so LCDd sends an updated *`value`*.

  `plus` (slider) , `minus` (slider)
  : The slider was moved to left (`minus`) or right (`plus`), so LCDd sends an updated *`value`*.

  `enter`
  : This item has been entered, which means it is currently active on the screen. The client could now for example update the value of the item. If it is a menu, it may be needed to update the values of the items in it too, because they may be visible too.

  `leave`
  : This item has been left, so it is currently not the (main) active item anymore.

  Multiple messages may be generated by one action of the user.

ChapterÂ 3.Â Programming for LCDproc

Table of Contents

Get the source

If you want to start programming for LCDproc you will need the have the most current source code available. You can get it several ways:

1. Download GIT version of as a tarball (preferred).
2. Download the latest version from GIT.
3. Download the last stable release from Github. (This is not recommended as stable release may be months behind the current version.)

Download GIT Version of LCDproc as a Tarball

The source is available as tarball. You can download them from https://github.com/lcdproc/lcdproc/archive/master.tar.gz. For development we recommended to use the 'master' branch.

To extract the files run

```
$ tar xvfz lcdproc-master.tar.gz
```

Download The Latest Version of LCDproc from GitHub

Of course you can download the latest stuff from GitHub. For more information on how to use GitHub see https://guides.github.com/.

```
$ git clone https://github.com/lcdproc/lcdproc.git
```

Once you've done that and want to update the downloaded files to the latest stuff you can use the "pull" command of git (make sure to be in the lcdproc directory!):

```
$ git pull
```

Now that once you have downloaded the files you can prepare them for compiling, but first you should (you don't have to) copy them to another place on your machine.

Code style guideline

LCDproc has been developed by many contributors over many years. You may find different programming styles (naming, indention, etc) in the source code.

When modifying an existing file, please take a careful look at its style and program continuing that style instead of mixing it up with another one even if it does not comply with the guidelines written below.

For newly added files the following guideline describes how source code should look like.

Note

All new submitted files will be passed through GNU **indent** to enforce the style described below.

File format and indention

- *Language:* The programming language used for LCDd (server core), drivers and the lcdproc client is Standard `C`, currently `C99`. No other programming language will be accepted.
- *File encoding:* Files shall either encoded as UTF-8 or ISO-8859-1 and line endings shall be Unix type.
- *Line length:* Lines of source code should be wrapped at column 100. Comment lines should wrap at column 79.
- *Indention:* Tab indention shall be used (with tab width set to 8 characters).
- *License:* LCDproc is released under GNU General Public License version 2 (GPL v2) and every file shall have a standard copyright notice.

Naming conventions

- *Function names:* Function names shall be lowercase. We do not use CamelCase (some historical exceptions may exist). Multiple words are separated by underscore.
- *Variable names:* We do not use Hungarian Notation. CamelCase may be used, but names shall begin with a lowercase letter.
- *Constants:* Constants shall be written in uppercase using underscore to separate multiple words.

ExampleÂ 3.1.Â Names of constants, variables and functions

```
/* Constants */
#define KEYPAD_AUTOREPEAT_DELAY 500
#define KEYPAD_AUTOREPEAT_FREQ 15

/* Variable names */
MODULE_EXPORT char * api_version = API_VERSION;
MODULE_EXPORT int stay_in_foreground = 0;
MODULE_EXPORT int supports_multiple = 1;

/* Function names */
void HD44780_position(Driver *drvthis, int x, int y);
static void uPause(PrivateData *p, int usecs);
unsigned char HD44780_scankeypad(PrivateData *p);
```

Comments

- All code comments shall be `C`-style comments (/* */). Comments spanning multiple lines shall have a star at the beginning of each line.
- `C++`-style comments (//) may be used to comment out single lines of code to disable these lines. Larger blocks of code which shall be disabled should be wrapped within `C`-style comments or using pre-processor directives (#if ... #endif).

  `C++`-style comments should not be used in general.
- We use Doxygen to document our source code. Functions shall be documented using Doxygen-style comments (/** *). See Doxygen Manual for more information and how to use it.
- If you carefully formatted a comment, you may use the special comment /*- */ (comment start is â star minusâ) to prevent automatic reformatting. This usually applies to the standard copyright notice.

  ExampleÂ 3.2.Â Standard copyright notice

  ```
  /*-
   * Copyright (C) 2010 Your Name <your_email_address>
   *
   * This file is released under the GNU General Public License.
   * Refer to the COPYING file distributed with this package.
   */
  ```

Statement style

- *Function declarations:*

  Function declarations have their declaration and opening brace split across two lines.

  Function names start in column one. The return type is placed on the previous line.

  There is no space between the function name and '('.

  ExampleÂ 3.3.Â A function declaration

  ```
  /**
   * This is a Doxygen function description.
   *
   * \param y      The number of years
   * \param str    Pointer to a string containing X
   * \return       0 on success; -1 on error
   */
  int
  this_is_a_function(int y, char *str)
  {
          code
  }
  ```

- *Operators:*

  There shall be a space characters before/after an operator or assignment, except for increment (â ++â) or decrement (â --â) operators.

  ExampleÂ 3.4.Â Space around operators

```
if (p->dispSizes[dispID - 1] == 1 && p->width == 16) {
        if (x >= 8) {
                x -= 8;
                relY = 1;
        }
}

x--;                    /* Convert 1-based coords to 0-based */
y--;
```

- *Function calls:*

  There shall be no space between the function call and the opening brace '(' of the parameter list. Within the parameter list a space shall be after each parameter.

  ExampleÂ 3.5.Â Function call

```
lib_vbar_static(drvthis, x, y, len, promille, options, p->cellheight, 0);
```

- *Compound statements:*

  Opening braces occur on the same line as the statement.

  Else statements: Else statements are placed on a line of their own, even is there is a previous closing brace.

  Opening and closing braces may be omitted on single line compound statements. However, if one part of an if-else-statement requires braces the other part shall have braces as well.

  ExampleÂ 3.6.Â If-else with braces

```
if (...) {
        code
}
else {
        code
}
```

  ExampleÂ 3.7.Â If-else with single statements

```
        if (...)
                print();
        else
                err = 1;
```

ExampleÂ 3.8.Â Other compound statements

```
        while (...) {
                code
        }

        for (a = 0; a < max; a++) {
                code
        }

        /* case labels are not indented */
        switch (icon) {
        case ICON_BLOCK_FILLED:
                HD44780_set_char(drvthis, 6, block_filled);
                break;
        case ICON_HEART_FILLED:
                HD44780_set_char(drvthis, 0, heart_filled);
                break;
        case ICON_HEART_OPEN:
                HD44780_set_char(drvthis, 0, heart_open);
                break;
        default:
                return -1;      /* Let the core do other icons */
        }
```

Example indent profile

The following example shows an indent profile that checks and corrects the coding style guideline described above. Copy the following text into file `.indent.pro` in your home directory.

ExampleÂ 3.9.Â indent.pro example

```
-TPrivateData
-TDriver

-l100
-lc79
-i8
-ip8
-di1
-cd33
-ci8
-cli0
-ts8

-ut
-nsob
-sc
```

```
        -psl
        -npcs
        -lp
        -fc1
        -nbc
        -nce
        -brs
        -br
        -sbi0
        -saw
        -sai
        -saf
        -nprs
        -hnl
        -fca
        -cdw
        -nbfda
        -nbbo
        -ncdb
        -nbad
        -cs
```

Submitting code

When you have finished modifying the code you may decide to submit it to the LCDproc project as a pull request. Read the documentation to learn how to make one.

ChapterÂ 4.Â Shared files

Table of Contents

Introduction

Here we provide functions that should be used by all parts of the program.

report.h : Debugging and reporting

To enable the debug() function on all of the software, just type:

```
./configure --enable-debug and recompile with 'make'.
```

ProcedureÂ 4.1.Â Enabling the debug() function only in specific files:

1. Configure without enabling debug (that is without --enable-debug).
2. Edit the source file that you want to debug and put the following line at the top, before the #include "report.h" line: `#define DEBUG`.
3. Then recompile with 'make'.

This way, the global DEBUG macro is off but is locally enabled in certain parts of the software.

The reporting levels have the following meaning.

Reporting Levels

0 RPT_CRIT
: Critical conditions: the program stops right after this. Only use this if the program is actually exited from the current function.

1 RPT_ERR
: Error conditions: serious problem, program continues. Use this just before you return -1 from a function.

2 RPT_WARNING

Warning conditions: Something that the user should fix, but the program can continue without a real problem. Ex: Protocol errors from a client.

3 RPT_NOTICE
: Major event in the program: (un)loading of driver, client (dis)connect.

4 RPT_INFO
: Minor event in the program: the activation of a setting, details of a loaded driver, a key reservation, a keypress, a screen switch.

5 RPT_DEBUG
: Insignificant event: What function has been called, what subpart of a function is being executed, what was received and sent over the socket, etc.

Levels 4 (maybe) and 5 (certainly) should be reported using the debug function. The code that this function generates will not be in the executable when compiled without debugging. This way memory and CPU cycles are saved.

report.h file defines 3 functions for debugging and reporting:

Sets reporting level and message destination

```
int set_reporting( application_name, Â
Â                  new_level,        Â
Â                  new_dest);        Â
char * application_name;
int new_level;
int new_dest;
Â
```
Returns -1 on error or 0 on success.

Report the message to the selected destination if important enough

```
void report( level,  Â
Â            format, Â
Â            );      Â
const int level;
const char *format;
...;
Â
```
Returns nothing (void).

The format parameter is the same as the one used by printf.

Send debugging information if important enough

Consider the debug function to be exactly the same as the report function. The only difference is that it is only compiled in if DEBUG is defined.

LL.h : Linked Lists (Doubly-Linked Lists)

Creating a list

To create a list, do the following:

```
LinkedList *list;
list = LL_new();
if(!list) handle_an_error();
```

The list can hold any type of data. You will need to typecast your datatype to a "void *", though. So, to add something to the list, the following would be a good way to start:

```
typedef struct my_data {
  char string[16];
  int number;
} my_data;

my_data *thingie;

for(something to something else) {
  thingie = malloc(sizeof(my_data));
  LL_AddNode(list, (void *)thingie);  // typecast it to a "void *"
}
```

For errors, the general convention is that "0" means success, and a negative number means failure. Check LL.c to be sure, though.

Changing data

To change the data, try this:

```
thingie = (my_data *)LL_Get(list);  // typecast it back to "my_data"
thingie->number = another_number;
```

You don't need to "Put" the data back, but it doesn't hurt anything.

```
LL_Put(list, (void *)thingie);
```

However, if you want to point the node's data somewhere else, you'll need to get the current data first, keep track of it, then set the data to a new location:

```
my_data * old_thingie, new_thingie;

old_thingie = (my_data *)LL_Get(list);
LL_Put(list, (void *)new_thingie);

// Now, do something with old_thingie.  (maybe, free it?)
```

Or, you could just delete the node entirely and then add a new one:

```
my_data * thingie;

thingie = (my_data *)LL_DeleteNode(list, NEXT);
free(thingie);
```

```
thingie->number = 666;

LL_InsertNode(list, (void *)thingie);
```

Iterations throught the list

To iterate on each list item, try this:

```
LL_Rewind(list);
do {
  my_data = (my_data *)LL_Get(list);
  /* ... do something to it ... */
} while(LL_Next(list) == 0);
```

Using the list as a stack or a queue

You can also treat the list like a stack, or a queue. Just use the following functions:

```
LL_Push()      // Regular stack stuff: add, remove, peek, rotate
LL_Pop()
LL_Top()
LL_Roll()

LL_Shift()     // Other end of the stack (like in perl)
LL_Unshift()
LL_Look()
LL_UnRoll()

LL_Enqueue()   // Standard queue operations
LL_Dequeue()
```

There are also other goodies, like sorting and searching.

Future

Array-like operations will come later, to allow numerical indexing:

```
LL_nGet(list, 3);
LL_nSwap(list, 6, 13);
LL_nPut(list, -4, data);   // Puts item at 4th place from the end..
```

More ideas for later:

```
LL_MoveNode(list, amount);  // Slides a node to another spot in the list
-- LL_MoveNode(list, -1); // moves a node back one toward the head
```

That's about it, for now... Be sure to free the list when you're done!

See LL.c for more detailed descriptions of these functions.

Shared files specific for drivers

Driving an LCD display is not easy; you need to address ports, to send bytes in a certain order, to respect timing, and unfortunately no two operating system let you do this in the same way. But don't despair! There's hope! Someone in a galaxy far far away, has Already done the dirty job for you! This dirty job has been put in shared files. These shared files are full cross platform and are auto-magically configured by the configure script. You only need to include them and use their functions to benefit from them.

These files are provided only for drivers, others are provided for all of LCDproc. These files are located in the shared directory, they have a dedicated chapter in this book.

port.h : Parallel port I/O

The file `port.h`, located in the `server/drivers/` directory provide Input/Output and port permissions for the PC compatible parallel port, also known as the LPT port.

Of course, these functions will only work if the computer where LCDproc runs has parallel port! In these situations, the configure script will see this and disable drivers that need a parallel port.

The functions in `port.h` are defined and as 'static inline'. Therefore each driver including this header file gets its own copy of the functions and they are inlined into the driver's code. As a result calls to port_in() and port_out() are directly translated to inb() or outb() or assembly code by the compiler. There is less to no overhead in using them.

`port.h` file defines 6 static inline functions for port I/O:

Read a byte from port

```
 static inline int port_in( port); Â
unsigned short int port;
Â
```

Returns the content of the byte.

Write a char(byte) 'val' to port

```
 static inline void port_out( port, Â
 Â                              val); Â
unsigned short int port;
unsigned char val;
Â
```

Returns nothing (void).

Get access to a specific port

```
 static inline int port_access( port); Â
unsigned short int port;
Â
```

Returns `0` if successful, `-1` if failed.

Close access to a specific port

`static inline int` **`port_deny`**`(` *port*`);` Â
`unsigned short int` *port*;
Â
Returns `0` if successful, `-1` if failed.

Get access to multiple sequential ports

`static inline int` **`port_access_full`**`(` *port*, Â
Â *count*`);` Â
`unsigned short int` *port*;
`unsigned short int` *count*;
Â
Returns `0` if successful, `-1` if failed.

Close access to multiple sequential ports

`static inline int` **`port_deny_full`**`(` *port*, Â
Â *count*`);` Â
`unsigned short int` *port*;
`unsigned short int` *count*;
Â
Returns `0` if successful, `-1` if failed.

Example use

```
#include "port.h"

/*
 * Get access to these 3 ports:
 *  0x378 (CONTROL),
 *  0x379 (STATUS) and
 *  0x37A (DATA)
 */
if (port_access_multiple(0x378, 3) == -1) {
        /* Access denied, do something */
}

/* Write a 'A' to the control port */
port_out(0x378, 'A');

/* Read from the status port */
char status = port_in(0x379);

/* Close the 3 ports */
port_deny_multiple(0x378, 3);
```

adv_bignum.h : Write Big-Numbers

`adv_bignum.h` is the headerfile for `libbignum.a` (made from `adv_bignum.c`) which contains everything needed to show big-numbers, including the fonts for the different displays. (All files are located in the `server/drivers/` directory.)

There are only a few requirements to the calling driver:

- The following functions have to be implemented by the driver:

  `height()`
  : to determine the display's height and thus the maximal height of the big numbers to be displayed.

  `get_free_chars()`
  : to determine the number of user-definable characters that can be used in the generation of big numbers.

  `set_char()`
  : to define a character necessary to write a big number. Of course this is only necessary if there really are user-definable characters, i.e. only if `get_free_chars()` returns a value greater `0`.

  `chr()`
  : to actually write the characters the big numbers consist of.
- The display's *`cellwidth`* has to be `5` (`6` works also in some cases) and the *`cellheight`* `7` or `8`.
- The custom-characters (if any) have to be at character positions *`offset`*+`0`, *`offset`*+`1`, *`offset`*+`2`, ... *`offset`*+ `get_free_chars()-1`,
- *`offset`*+ `get_free_chars()-1` must be less than `32`,

The library determines the correct font, depending on the display size and the number of user-defined characters itself. So it is easy to integrate into the driver.

Provided Functions

```
void lib_adv_bignum( drvthis,  Â
Â                    x,        Â
Â                    num,      Â
Â                    offset,   Â
Â                    do_init); Â
Driver *drvthis;
int x;
int num;
int offset;
int do_init;
Â
```

The main thing the driver has to do is to call this function from its `num()` function with the parameters described below.

`drvthis`
: the pointer pointing to the Driver structure passed to the driver's `num()` function.

`x`
: the horizontal position of the top-left corner of the big-number (the big-numbers don't have a y position). The placing of the characters is done by the client, so the driver only has to forward the position to the lib. The bignumlib has no influence on the placing of the characters.

`num`
: the number (legal: `0` - `9`, and `:`) to be written.

`offset`

the character position where the user-definable characters start (usually `0`). The user-definable characters (if any) are then expected to be at the character positions *`offset`*`+0`, *`offset`*`+1`, *`offset`*`+2`, ... *`offset`*`+ get_free_chars()-1` and *`offset`*`+ get_free_chars()-1` is required to be less than `32`.

`do_init`

if not `0`, lib_adv_bignum will set the custom characters of the display for the big-numbers.

The driver has to check if the custom-characters have to be set or if it is already done and tell it to the lib (using the *`do_init`* parameter). The common way is to use variable called *`p->ccmode`* or similar. In the different drivers there are some differences in the naming and handling of this variable. So the responsibility of checking and setting is left to the driver.

ExampleÂ 4.1.Â Calling `lib_adv_bignum()`

```
#include "adv_bignum.h"

MODULE_EXPORT void
myDriver_num( Driver * drvthis, int x, int num )
{
  PrivateData *p = drvthis->private_data;
  int do_init = 0;

  if (p->ccmode != CCMODE_BIGNUM){      // Are the custom-characters set up correctly? If n
    do_init = 1;                        // Lib_adv_bignum has to set the custom-characters.
    p->ccmode = CCMODE_BIGNUM;          // Switch custom-charactermode to bignum.
  }

  // Lib_adv_bignum does everything needed to show the big-numbers.
  lib_adv_bignum(drvthis, x, num, 0, do_init);
}
```

All that's left to do is to add `libbignum.a` to the libs and `adv_bignum.h` sources of your driver in the `Makefile` (or the file that generates the `Makefile`).

ExampleÂ 4.2.Â Enabling adv_bignum support in `Makefile.am`

```
myDriver_LDADD: libLCD.a libbignum.a

myDriver_SOURCES: lcd.h lcd_lib.h myDriver.c myDriver.h adv_bignum.h
```

Internal Structure and Functions

The only purpose of `lib_adv_bignum()` is to determine the best display-dependent big-number function, based upon the values of the driver's `height()` and `get_free_chars()` functions, and call it.

The display-dependent functions are named `adv_bignum_num_`*`N`*`_`*`M`*`()`, where *`N`* is the display's height in lines and *`M`* the number of used user-definable characters. The bits of the user-characters are stored in *`static char bignum`* (take a look at the source and you will see what I mean). (On a display with a *`cellheight`* of 7 the lowest line stored is not shown.) While *`static char num_map`* defines the placing in the big number. (A big number is always 3 characters wide and 4 characters high. On a big number for 2 line displays the 2 lower lines are not in use.)

If user-definable characters have to be set, the driver's `set_char()` function will be called once for every user-definable character.

Now `adv_bignum_write_num()` is called. This function places the 6 or 12 characters the big-number consists of in the framebuffer using the drivers `chr()` function.

ChapterÂ 5.Â The LCDproc driver API

Table of Contents

This chapter describes the driver API of v0.5 of LCDproc.

Overview of Operation

The API defines functions which drivers may implement to provide certain services. If a driver implements a function, the function will be detected by the server. For some optional functions the server core provides default implementations.

The API consists of functions to:

- print data to the display,
- set display properties (backlight, brightness, contrast, extra LEDs),
- read input from an input device (may be the display), and
- inform the server of the driver/display capabilities.

The API is best described by starting with the struct lcd_logical_driver which is defined in server/drivers/lcd.h.

Below is a commented version of lcd_logical_driver. Each function is described more detailed in the section called â   Functions in Detailâ   .

```
typedef struct lcd_logical_driver {

        //////// Variables to be provided by the driver module
        // The driver loader will look for symbols with these names !

        // pointer to a string describing the API version
        char *api_version;

        // Does this driver require to be in foreground ?
        int *stay_in_foreground;

        / Does this driver support multiple instances ?
        int *supports_multiple;

        // What should alternatively be prepended to the function names ?
        char **symbol_prefix;

        //////// Functions to be provided by the driver module

        //// Mandatory functions (necessary for all drivers)

        // initialize driver: returns >= 0 on success
        int (*init)             (Driver *drvthis);
```

```
// close driver
void (*close)           (Driver *drvthis);


//// Essential output functions (necessary for output drivers)

// get display width / height (in characters; 1-based)
int (*width)            (Driver *drvthis);
int (*height)           (Driver *drvthis);

// clear screen
void (*clear)           (Driver *drvthis);

// flush screen contents to LCD
void (*flush)           (Driver *drvthis);

// write string s at position (x,y)
void (*string)          (Driver *drvthis, int x, int y, const char *str);

// write char c at position (x,y)
void (*chr)             (Driver *drvthis, int x, int y, char c);


//// essential input functions (necessary for input drivers)

// get key from driver: returns a string denoting the key pressed
const char *(*get_key)  (Driver *drvthis);


//// Extended output functions (optional; core provides alternatives)

// draw a bar from pos (x,y) upward / to the right filling promille of len chars
void (*vbar)            (Driver *drvthis, int x, int y, int len, int promille, int
void (*hbar)            (Driver *drvthis, int x, int y, int len, int promille, int

// draw a progressbar covering lenght chars at pos (x,y) showing promille% progres
void (*pbar)            (Driver *drvthis, int x, int y, int len, int promille);

// display (big) number num at horizontal position x
void (*num)             (Driver *drvthis, int x, int num);

// set heartbeat state; animate heartbeat
void (*heartbeat)       (Driver *drvthis, int state);

// draw named icon at position (x,y)
int (*icon)             (Driver *drvthis, int x, int y, int icon);

// set cursor type and move it to position (x,y)
void (*cursor)          (Driver *drvthis, int x, int y, int type);


//// User-defined character functions

// set special character / get free characters
// - It is currently unclear how this system should work exactly
// - The set_char function expects a simple block of data with 1 byte for each pix
//   (So that is 8 bytes for a 5x8 char)
void (*set_char)        (Driver *drvthis, int n, unsigned char *dat);
int (*get_free_chars)   (Driver *drvthis);

// get width / height of a character cell (in pixels)
// - necessary to provide info about cell size to clients
// - if not defined, the core will provide alternatives returning default values
int (*cellwidth)        (Driver *drvthis);
int (*cellheight)       (Driver *drvthis);


//// Hardware functions

// get / set the display's contrast
int (*get_contrast)     (Driver *drvthis);
void (*set_contrast)    (Driver *drvthis, int promille);

// get / set brightness for given backlight state
int (*get_brightness)   (Driver *drvthis, int state);
void (*set_brightness)  (Driver *drvthis, int state, int promille);

// set backlight state
void (*backlight)       (Driver *drvthis, int state);

// set output
void (*output)          (Driver *drvthis, int state);


//// Informational functions
// get a string describing the driver and it's features
const char * (*get_info) (Driver *drvthis);



//////// Variables in server core, available for drivers

// name of the driver instance (name of the config file section)
// - do not change from the driver; consider it read-only
// - to be used to access the driver's own section in the config file
char * name;

// pointer to the driver instance's private data
// - filled by the server by calling store_private_ptr()
// - the driver should cast this to it's own private structure pointer
void * private_data;


//////// Functions in server core, available for drivers

// store a pointer to the driver instance's private data
int (*store_private_ptr) (struct lcd_logical_driver * driver, void * private_data)

// Config file functions, cwprovided by the server
// - see configfile.h on how to use these functions
// - as sectionname, always use the driver name: drvthis->name
short (*config_get_bool) (char * sectionname, char * keyname,
                        int skip, short default_value);
long int (*config_get_int) (char * sectionname, char * keyname,
                        int skip, long int default_value);
double (*config_get_float) (char * sectionname, char * keyname,
                        int skip, double default_value);
const char *(*config_get_string) (char * sectionname, char * keyname,
                        int skip, const char * default_value);
                        // Returns a string in server memory space.
                        // Copy this string.
int config_has_section  (const char *sectionname);
int config_has_key      (const char *sectionname, const char *keyname);
```

```
        // Display properties functions (for drivers that adapt to other loaded drivers)
        // - the return the size of another already loaded driver
        // - if no driver is loaded yet, the return values will be 0
        int (*get_display_width) ();
        int (*get_display_height) ();
} Driver;
```

Private Data

To support multiple instances it is necessary to stop using global variables to store a driver's data in. Instead, you should store it in a structure, that you allocate and store on driver's init.

In the driver's private structure will probably at least be something like:

```
typedef struct MyDriver_private_data {
        int fd;                         // file descriptor for the LCD device
        int width, height;              // dimension of the LCD (in characters, 1-based
        int cellwidth, cellheight;      // Size of each LCD cell, in pixels
        unsigned char *framebuf;        // Frame buffer...
} PrivateData;
```

You allocate and store this structure like this:

```
        PrivateData *p;

        // Allocate and store private data
        p = (PrivateData *) malloc(sizeof(PrivateData));
        if (p == NULL)
                return -1;
        if (drvthis->store_private_ptr( drvthis, p ) < 0)
                return -1;

        // initialize private data
        p->fd = -1;
        p->cellheight = 8;
        p->cellwidth = 6;

        (... continue with the rest of your init routine)
```

You retrieve this private data pointer by adding the following code to the beginning of your functions:

```
        PrivateData *p = (PrivateData *) drvthis->private_data;
```

Then you can access your data like:

```
        p->framebuf
```

Functions in Detail

`int` **`(*init)`**`(` *drvthis*`);` Â
`Driver *`*drvthis*;
Â
The init() function. It starts up the LCD, initializes all variables, allocates private data space and stores the pointer by calling store_private_ptr(); Returns <0 on error.

```
void (*close)( drvthis); Â
Driver *drvthis;
Â
```

Shut down the connection with the LCD. Called just before unloading the driver.

```
int (*width)( drvthis); Â
Driver *drvthis;
Â
```

Get the screen width in characters. The result is 1-based.

```
int (*height)( drvthis); Â
Driver *drvthis;
Â
```

Get the screen height in character lines. The result is 1-based.

```
void (*clear)( drvthis); Â
Driver *drvthis;
Â
```

Clear the framebuffer.

```
void (*flush)( drvthis); Â
Driver *drvthis;
Â
```

Flush the framebuffer to the LCD.

```
void (*string)( drvthis, Â
Â               x,       Â
Â               y,       Â
Â               str);    Â
Driver *drvthis;
int x;
int y;
const char *str;
Â
```

Place string `str` into position (*x*,*y*) in the framebuffer. All coordinates are 1-based, i.e. (1,1) is top left. The driver should check for overflows, i.e. that the positional parameters are within the screen's boundaries and cut off the part of the string that is out of bounds.

```
void (*chr)( drvthis, Â
Â            x,       Â
Â            y,       Â
Â            c);      Â
Driver *drvthis;
int x;
int y;
char c;
Â
```

Place a single character *c* into position (*x*,*y*) in the framebuffer. The driver should check for overflows, i.e. that the positional parameters are within the screen's boundaries and ignore the request if the character is out of bounds.

```
void (*vbar)( drvthis,   Â
Â             x,         Â
Â             y,         Â
Â             len,       Â
Â             promille,  Â
Â             options);  Â
Driver *drvthis;
int x;
int y;
int len;
int promille;
int options;
Â
```

Draw a vertical bar at position (*x*,*y*) that has maximal length `len`, where a fraction of (`promille` / 1000) is filled.

```
void (*hbar)( drvthis,   Â
Â             x,         Â
Â             y,         Â
Â             len,       Â
Â             promille,  Â
Â             options);  Â
Driver *drvthis;
int x;
int y;
int len;
int promille;
int options;
Â
```

Draw a solid horizontal bar at position (*x*,*y*) that has maximal length `len`, where a fraction of (`promille` / 1000) is filled. The non filled part of the maximum length should not be drawn / touched by this function.

```
void (*pbar)( drvthis,   Â
Â             x,         Â
Â             y,         Â
Â             len,       Â
Â             promille); Â
Driver *drvthis;
int x;
int y;
int len;
int promille;
Â
```

Draw a progress bar covering `len` characters at position (*x*,*y*), where a fraction of (`promille` / 1000) is filled. Unlike a hbar, this function must draw a background for the the non-filled part of the bar. For example this function may draw an outline in the form of a (non-filled) rectangle covering the entire length and then fill part of that rectangle to indicate the current progress.

```
void (*num)( drvthis, Â
Â            x,       Â
Â            num);    Â
Driver *drvthis;
int x;
int num;
Â
```

Display big number `num` at horizontal position `x`.

```
void (*heartbeat)( drvthis, Â
Â                  state);  Â
Driver *drvthis;
int state;
Â
```

Sets the heartbeat to the indicated state: 0=off, 1=on. Use HEARTBEAT_ON to say that we want to display/refresh the heartbeat. The driver choose how to do it.

```
int (*icon)( drvthis, Â
Â            x,       Â
Â            y,       Â
Â            icon);   Â
Driver *drvthis;
int x;
int y;
int icon;
Â
```

Draw named icon `icon` at position (*x*,*y*). If the driver returns -1 the server core will draw an appropriate replacement character.

```
void (*cursor)( drvthis, Â
Â               x,       Â
Â               y,       Â
Â               type);   Â
Driver *drvthis;
int x;
int y;
int type;
Â
```

Move cursor to position (*x*,*y*), setting its type to `type`.

```
void (*set_char)( drvthis, Â
Â                 ch,      Â
```

Â *dat*); Â

`Driver *`*drvthis*;
`char` *ch*;
`unsigned char *`*dat*;
Â
The set_char function expects a simple block of data with 1 byte for each pixel-line. (So that is 8 bytes for a 5x8 char)

`int` **`(*get_free_chars)`**`(` *drvthis*`);` Â
`Driver *`*drvthis*;
Â
Get total number of custom characters available.

`int` **`(*cellwidth)`**`(` *drvthis*`);` Â
`Driver *`*drvthis*;
Â
Return the width of a character cell in pixels. The result is 1-based.

`int` **`(*cellheight)`**`(` *drvthis*`);` Â
`Driver *`*drvthis*;
Â
Return the height of a character cell in pixels. The result is 1-based.

`int` **`(*get_contrast)`**`(` *drvthis*`);` Â
`Driver *`*drvthis*;
Â
Get the contrast value from the driver. The return value is an integer in the range from 0 to 1000. Many displays do not support getting or setting contrast using software.

`int` **`(*set_contrast)`**`(` *drvthis*, Â
Â *promille*`);` Â
`Driver *`*drvthis*;
`int` *promille*;
Â
Sets the contrast to the given value, which is an integer in the range from 0 to 1000. It is up to the driver to map the logical interval [0, 1000] into the interval that the hardware supports. Many displays do not support software setting of contrast.

`int` **`(*get_brightness)`**`(` *drvthis*, Â
Â *state*`);` Â
`Driver *`*drvthis*;
`int` *state*;
Â
Get the brightness value from the driver for the given backlight state. The parameter `state` determines which one is returned. The return value is an integer in the range from 0 to 1000. Many displays do not support getting or setting brightness using software.

`int` **`(*set_brightness)`**`(` *drvthis*, Â

```
Â                                 state,     Â
Â                                 promille); Â
Driver *drvthis;
int state;
int promille;
Â
```

Set the brightness for the given backlight state to the value given. Value must be an integer in the range from 0 to 1000. It is up to the driver to map the logical interval [0, 1000] into the interval that the hardware supports. Many displays do not support software setting of brightness.

```
void (*backlight)( drvthis, Â
Â                  state); Â
Driver *drvthis;
int state;
Â
```

Sets the backlight to the given brightness state. Often hardware can only support two values for the backlight: on and off. In that case any value of state > 0 will switch the backlight on.

```
void (*output)( drvthis, Â
Â               state); Â
Driver *drvthis;
int state;
Â
```

Sets the output value. Some displays/wirings have a general purpose output, which can be controlled by calling this function. See the 'output' command in the 'widget language'.

```
const char *(*get_key)( drvthis); Â
Driver *drvthis;
Â
```

Checks if a key has been pressed on the device. Returns NULL for "no key pressed", or a string describing the pressed key. These characters should match the keypad-layout.

```
const char *(*get_info)( drvthis); Â
Driver *drvthis;
Â
```

Returns a string describing the driver and its features.

```
short (*config_get_bool)( sectionname,    Â
Â                         keyname,        Â
Â                         skip,           Â
Â                         default_value); Â
char *sectionname;
char *keyname;
int skip;
short default_value;
Â
```

Call to server. Retrieve a bool from the config file. Sectionname should be the name of the driver (as in the struct). If the key cannot be found, the default value will be returned. skip should be 0 usually, but if you want to retrieve multiple identical keys, then increase skip to get every next value.

```
long int (*config_get_int)( sectionname,    Â
Â                           keyname,        Â
Â                           skip,           Â
Â                           default_value); Â
char *sectionname;
char *keyname;
int skip;
long int default_value;
Â
```

Call to server. Retrieve an integer from the config file.

```
double (*config_get_float)( sectionname,    Â
Â                           keyname,        Â
Â                           skip,           Â
Â                           default_value); Â
char *sectionname;
char *keyname;
int skip;
double default_value;
Â
```

Call to server. Retrieve a float from the config file.

```
const char *(*config_get_string)( sectionname, Â
Â                                 keyname,     Â
Â                                 skip,        Â
Â                                 default);    Â
char *sectionname;
char *keyname;
int skip;
const char *default;
Â
```

Call to server. Retrieve a string from the config file. Fill result with a pointer to some available space. You can fill it with a default value. If the key is found, it will be overwritten with the value from the key. Note that you should always first copy the the returned string. It is in the address space of the server, and will be freed at the next call.

```
int config_has_section( sectionname); Â
const char *sectionname;
Â
```

Returns whether a section exists. Does not need to be called prior to a call to a config_get_* function.

```
int config_has_key( sectionname, Â
Â                   keyname);    Â
const char *sectionname;
const char *keyname;
```

Â

Returns the number of times a key exists. Does not need to be called prior to a call to a config_get_* function.

```
First version, Joris Robijn, 20011016
Corrected and expanded, Peter Marschall 20060411
Sync'd with lcd.h, Markus Dolze, 20090322
```

ChapterÂ 6.Â Adding your driver to LCDproc

Table of Contents

Introduction

LCDproc is meant to be modular, it is relatively easy to add new input and output drivers to LCDproc.

This chapter will explain you the major steps and few gotchas of adding your own driver to LCDproc. Enjoy!

Be sure to read ChapterÂ 3, *Programming for LCDproc* and ChapterÂ 5, *The LCDproc driver API* as well.

As a starting point you may take a look at the debug driver. It is available as `server/drivers/debug.c`.

Rules for accepting new drivers

LCDproc is open source software. Anyone is free to take LCDproc's code, write his own driver and publish the modified sources somewhere again. If you want your driver to be included in LCDproc's code some conditions have to be met:

1. The hardware (display or enclosing product) is publicly sold *OR* the schematics and firmware (if required) are publicly available. [1]
2. The driver is released under (L)GPL and has an appropriate copyright notice.
3. The submitter is or is acting on behalf of the original driver developer. [2]
4. The driver description contains a valid email address for contacting the submitter or developer.
5. The code is commented *AND* includes appropriate Doxygen comments, especially for internal / non-API functions.
6. End user documentation (updates to man pages *AND* User's Guide in Docbook format) is available.
7. Driver options are described in the end user documentation *AND* `LCDd.conf`.
8. The driver adheres to the style guide as described in the section called â Code style guidelineâ .

Autoconf, automake, and Everything!

How I Learned to Stop Worrying and Love the Configure Script

It was decided pretty early in LCDproc's life to use GNU autoconf and GNU automake. This allows LCDproc to be ported to several platforms with much less effort. It can be quite daunting to understand how autoconf & automake interact with each others and with your code, but don't be discouraged. We have taken great care in making this as simple as possible for programmers to add their own driver to LCDproc. Hopefully, you'll only have to modify two files, one for autoconf and one for automake.

The first thing you need to do is to find a name for your driver, it should be as descriptive as possible; most drivers are named after their respective chipset, for example hd44780, mtc_s16209x, sed1330 and stv5730, others are named after the company that makes that particular LCD display, for example CFontz and MtxOrb. Remember that these names are case sensitive. In this chapter, we'll use myDriver (which is an absolute non-descriptive name).

Autoconf and its friend, acinclude.m4

You need to add your driver to function LCD_DRIVERS_SELECT of file acinclude.m4. This can be done in three steps.

Step 1

First you need to add your driver name to the list of possible choices in the help screen.

This:

```
AC_ARG_ENABLE(drivers,
        [  --enable-drivers=<list> compile driver for LCDs in <list>.]
        [                  drivers may be separated with commas.]
        [                  Possible choices are:]
        [                    bayrad,CFontz,CFontz633,CFontzPacket,curses,CwLnx,]
        [                    glcdlib,glk,hd44780,icp_a106,imon,IOWarrior,irman,]
        [                    joy,lb216,lcdm001,lcterm,lirc,ms6931,mtc_s16209x,]
        [                    MtxOrb,NoritakeVFD,pyramid,sed1330,sed1520,serialVFD,]
        [                    sli,stv5730,svga,t6963,text,tyan,ula200,xosd]
        [                  'all' compiles all drivers;]
        [                  'all,!xxx,!yyy' de-selects previously selected drivers],
        drivers="$enableval",
```

becomes:

```
AC_ARG_ENABLE(drivers,
        [  --enable-drivers=<list> compile driver for LCDs in <list>.]
        [                  drivers may be separated with commas.]
        [                  Possible choices are:]
        [                    bayrad,CFontz,CFontz633,CFontzPacket,curses,CwLnx,]
        [                    glcdlib,glk,hd44780,icp_a106,imon,IOWarrior,irman,]
        [                    joy,lb216,lcdm001,lcterm,lirc,ms6931,mtc_s16209x,]
        [                    MtxOrb,NoritakeVFD,pyramid,sed1330,sed1520,serialVFD,]
        [                    sli,stv5730,svga,t6963,text,tyan,ula200,xosd,myDriver]
        [                  'all' compiles all drivers;]
        [                  'all,!xxx,!yyy' de-selects previously selected drivers],
        drivers="$enableval",
```

Step 2

Second, you need to add your driver to the list of all drivers.

This:

```
allDrivers=[bayrad,CFontz,CFontz633,...(big list)...,tyan,ula200,xosd]
```

becomes:

```
allDrivers=[bayrad,CFontz,CFontz633,...(big list)...,tyan,ula200,xosd,myDriver]
```

Step 3

Then last, you need to add your driver to be big switch-case in this function, see below.

```
                        myDriver)
                                DRIVERS="$DRIVERS myDriver${SO}"
                                actdrivers=["$actdrivers myDriver"]
                                ;;
```

If your driver only works in some platform or requires a particular library or header, you can add your autoconf test here. You can see how other drivers do it, but if you're not sure on how to do this, just send an email to the mailing list and we'll make it for you.

Automake and its friend, Makefile.am

Already half of the job is done! Not to bad, wasn't it? The rest should be just as easy. In this section, you'll be adding your driver to the file server/drivers/Makefile.am. As you can guess, it's the Makefile for the drivers. This can be done in three (or two) simple steps.

Step 1

First, you need to add your driver to the list of drivers in this file, this list is called EXTRA_PROGRAMS.

This

```
EXTRA_PROGRAMS = bayrad CFontz ...(big list)... ula200 xosd
```

becomes

```
EXTRA_PROGRAMS = bayrad CFontz ...(big list)... ula200 xosd myDriver
```

Step 2

This second step is only needed if your driver needs a particular library. If it doesn't, you can skip to step 3.

You basically need to put you driver name followed by _LDADD and equal this to the name of the library that you need. Usually, these library are substituted by a autoconf variable, if you're not comfortable with this, you send an email to the mailing list and we'll set this up for you.

For example, we would put this for our fictional driver

```
myDriver_LDADD = @SOMESTRANGELIB@
```

Step 3

Last but not least, you need to specify which source files should be associated with your driver. You put your driver name followed by `_SOURCES` and equal this to a space separated list of the source and header files. See below for an example.

```
myDriver_SOURCES =  lcd.h myDriver.c myDriver.h
```

Test your setup

You're almost done! You only need to check out if you didn't made any mistake. Just run sh autogen.sh to regenerate the configure script and Makefiles, then run ./configure --enable-drivers=myDriver and type make. If your driver compiles without error, then congratulations, you've just added your driver to LCDproc! Remember to submit a patch to the mailing list or open a pull request on github so that we can add it to the standard distribution, but do not forget the documentation.

If you had an error, just send us an email describing it to the mailing list and we'll try to help you.

It's all about documentation

Please do not forget to also add the required documentation, so that your driver can be used from others as well.

Within the source code

We use Doxygen to document functions and data types. The doxygen documentation can be created anytime by changing to the `docs/` directory and running **doxygen**.

When documenting your driver's API functions you may use a short hand version and add 'API:' to the beginning of your comment and leave out the parameter and return value description (as we know what the API is doing). If you use some clever algorithm inside a function please add a few words about it.

Note

Always document functions internal to the driver! We do know what the API does (or is expected to do) but we don't know about what your driver does internally.

Read the section called â   Commentsâ   on how for format comments.

The configuration file, LCDd.conf

Extend the LCDproc server's configuration file with a section that holds a standard configuration for your driver together with short descriptions of the options used.

```
â ¦

## MyDriver for MyDevice ##
[MyDriver]

# Select the output device to use [default: /dev/lcd]
Device=/dev/ttyS0
```

```
# Set the display size [default: 20x4]
Size=20x4

â ¦
```

The daemon's manual page, LCDd.8

Append your driver to the list of drivers in `docs/LCDd.8.in`, the manual page of LCD, so that users can find your driver when doing **`man LCDd`**.

```
â ¦
.TP
.B ms6931
MSI-6931 displays in 1U rack servers by MSI
.TP
.B mtc_s16209x
MTC_S16209x LCD displays by Microtips Technology Inc
.TP
.B MtxOrb
Matrix Orbital displays (except Matrix Orbital GLK displays)
.TP
.B MyDriver
displays connected using MyDevice
.TP
.B NoritakeVFD
Noritake VFD Device CU20045SCPB-T28A
.TP
.B pyramid
LCD displays from Pyramid (http://www.pyramid.de)
.TP
.B sed1330
SED1330/SED1335 (aka S1D13300/S1D13305) based graphical displays
â ¦
```

The user guide

Step 1

Please add a file `myDriver.docbook`, that describes the configuration of your driver and the hard/software needed, to the directory `docs/lcdproc-user/drivers/`.

Step 2

Define a Docbook entity for your driver file in `lcdproc-user.docbook`.

```
â ¦
<!ENTITY ms6931 SYSTEM "drivers/ms6931.docbook">
<!ENTITY mtc_s16209x SYSTEM "drivers/mtc_s16209x.docbook">
<!ENTITY MtxOrb SYSTEM "drivers/mtxorb.docbook">
<!ENTITY MyDriver SYSTEM "drivers/MyDriver.docbook">
<!ENTITY NoritakeVFD SYSTEM "drivers/NoritakeVFD.docbook">
<!ENTITY pylcd SYSTEM "drivers/pylcd.docbook">
<!ENTITY sed1330 SYSTEM "drivers/sed1330.docbook">
â ¦
```

Step 3

Add the freshly defined entity to `drivers.docbook` to include the documentation of your driver into the *LCDproc User's Guide*.

```
â ¦
&ms6931;
&mtc_s16209x;
&MtxOrb;
&MyDriver;
&NoritakeVFD;
&pylcd;
&sed1330;
â ¦
```

Step 4

Add the newly defined file `myDriver.docbook` to the Makefile in the directory `docs/lcdproc-user/drivers/`.

```
## Process this file with automake to produce Makefile.in

EXTRA_DIST =    bayrad.docbook \
                CFontz.docbook \
                ...
MyDriver.docbook \
                ...
## EOF
```

---

[1] Therefore I will not commit drivers for displays ripped out from an old telephone for your private hardware project and are not available otherwise.

[2] I will not submit drivers found somewhere on the internet and submitted without the original developer's written acknowledgement.

ChapterÂ 7.Â Making a release

Table of Contents

This chapter describes the steps necessary to create a software release of LCDproc. It is intended to guide the release manager when creating a new release. (This is somewhat outdated now, that lcdproc development happens on github. We should revise this once things are fully in place.)

There is a checklist how to create a release on github in `docs/release-checklist.md`

Creating a documentation release

Any release of LCDproc is accompanied the the user guide and developer guide. Here is how to create these documentation packages.

ProcedureÂ 7.1.Â Steps to create the documentation package

1. Get the release tarball and extract it and change to `docs/lcdproc-user`.
2. Create the documentation package by running: **`xmlto -o lcdproc-0-5-`*`A`*`-user-html xhtml lcdproc-user.docbook`**
3. Create a tarfile of the documentation package: **`tar -czf lcdproc-0-5-`*`A`*`-user-html.tar.gz lcdproc-0-5-`*`A`*`-user-html`**
4. Repeat the above steps for the developers guide, replacing â -userâ with â -devâ where appropriate.
5. Upload the files to the Sourceforge file release system.

The online documentation consists of the user and developer guide, each converted to a single file for viewing online.

ProcedureÂ 7.2.Â Steps to create the online documentation

1. Get the release tarball and extract it and change to `docs/lcdproc-user`.
2. Create the documentation file by running: **`xmlto xhtml-nochunks lcdproc-user.docbook`**
3. Rename the file: **`mv lcdproc-user.html lcdproc-0-5-`*`A`*`-user.html`**
4. Repeat the above steps for the developers guide, replacing â -userâ with â -devâ where appropriate.
5. Upload the files to our Sourceforge web site (not the file release system!) and change `htdocs/docs/index.html` to point to the new files.

AppendixÂ A.Â GNU Free Documentation License

Table of Contents

Version 1.1, March 2000

PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

VERBATIM COPYING

COPYING IN QUANTITY

MODIFICATIONS

the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in

Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.