|  |
| --- |
| **Software Design Specification Ultimaker E2 Communication Interface** |
| Ultimaker E2 |
| |  | | --- | | Company | Ultimaker  Project | Ultimaker E2 Communication Interface  Project Nr | 53141    Manager | H. Dijkstra  Engineer | B. Thomas | |  | |
|  |
|  |

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Revision History** | | | | |
| **Version** | **Status** | **Date** | **Author** | **Remark** |
| 00 | Draft | 10-01-2019 | BT | Initial creation document |
| 01 | Draft | 23-01-2019 | BT | Add websocket protocol information |
| 0222 | Draft | 06-02-2019 | BT | Add TFT display hardware support package |
| … |  |  |  |  |
| 0.5 | Draft | 09-09-2019 | BT | Added system install and recovery |
| 0.6 | Draft | 09-09-2019 | HD | Add UFP file validation |
| 0.7 | Draft | 12-09-2019 | BT | Add ‘menu’ documentation (not finished) |
| 0.8 | Draft | 29-10-2019 | BT | Added chapter about converting from copied libraries to git submodules. |

Table of Contents

[1 Introduction 6](#_Toc536009657)

[2 Architecture overview 7](#_Toc536009658)

[2.1 Hardware 7](#_Toc536009659)

[2.1.1 Linux board: Onion Omega2+ 7](#_Toc536009660)

[2.1.2 Baseboard 7](#_Toc536009661)

[2.1.3 Power supply and power consumption 7](#_Toc536009662)

[2.2 Software 8](#_Toc536009663)

[3 System software 9](#_Toc536009664)

[3.1 Boot sequence 9](#_Toc536009665)

[3.2 Used device drivers 9](#_Toc536009666)

[3.3 Used system daemons 9](#_Toc536009667)

[3.4 Filesystem overview 10](#_Toc536009668)

[3.5 System configuration files 11](#_Toc536009669)

[3.6 Shell 11](#_Toc536009670)

[3.7 Internationalization and localization 11](#_Toc536009671)

[3.7.1 Example Application 12](#_Toc536009672)

[3.7.2 Generate raw translatable messages 13](#_Toc536009673)

[3.7.3 Switching Locale 13](#_Toc536009674)

[4 Application specific software 14](#_Toc536009675)

[4.1 Job executor 15](#_Toc536009676)

[4.2 Printer driver 15](#_Toc536009677)

[4.3 Display driver 19](#_Toc536009678)

[4.4 Touch driver 20](#_Toc536009679)

[4.5 Configurator 20](#_Toc536009680)

[4.6 Websocket API task 20](#_Toc536009681)

[4.7 HTTP API task 21](#_Toc536009682)

[5 Evaluation of requirements 22](#_Toc536009683)

[6 Implementing system jobs 25](#_Toc536009684)

[6.1 Main task 25](#_Toc536009685)

[6.2 Homing 25](#_Toc536009686)

[6.3 Clean filament path 25](#_Toc536009687)

[6.4 Configure for first-time use 26](#_Toc536009688)

[6.5 Human readable verification token handling 26](#_Toc536009689)

[6.6 Heating buildplate 26](#_Toc536009690)

[6.7 Heating nozzle 26](#_Toc536009691)

[6.8 Loading filament 26](#_Toc536009692)

[6.9 Unloading filament 26](#_Toc536009693)

[6.10 Browse USB filesystem 26](#_Toc536009694)

[6.11 Screen saver 26](#_Toc536009695)

[6.12 Lock down mode 27](#_Toc536009696)

[6.13 Autoexec job 27](#_Toc536009697)

[6.14 Replace material 27](#_Toc536009698)

[6.15 Bed levelling 27](#_Toc536009699)

[7 Abbreviations 27](#_Toc536009700)

# Introduction

# Architecture overview

Architecture is both the process and product of planning, designing, and constructing structures. In this case the structure of a solution for the Ultimaker E2 connectivity board, both hardware and software.

## Hardware

The hardware consists of a single base board that hosts a small system on module computer that runs Linux. Connections to the outside world are mounted on the base board. Furthermore a display with touch interface is also connected to the base board. This chapter discusses the module and the connections to the outside world provided on the base board.

### Linux board: Onion Omega2+

The Onion Omega2+ system on module is a complete computer running Linux. In short it has the following properties:

* A 580MHz MIPS cpu
* 128MB RAM
* 32MB flash memory
* integrated WiFi and LAN (Ethernet) interfaces,
* USB host interface
* micro SD card slot
* 2 UARTs
* 1 SPI interface
* 1 I2C interface
* 1 I2S interface
* Various GPIO

Onion provides a pre-configured OpenWRT open source Linux system.

### Baseboard

The baseboard holds the Onion system-on-module and converts its tiny fragile connection pins to reliable industry-standard connectors. It also provides the module with power. The board has the following connections:

* RJ45 for LAN
* A serial interface for the console
* A serial interface that connects to the real-time controller (Marlin)
* USB host (A-type connector)
* Buzzer
* TFT display
* Touch screen

### Power supply and power consumption

Power is supplied by the real-time controller board. There are no special requirements for low power consumption as the system is powered from the mains net.

## Software

The software consists of a bootloader and the Linux kernel with drivers supporting the hardware. On top of this run a couple of applications that implement the functional requirements. These are discussed in more detail in the following chapters.

The main task of the software is forwarding G-code instructions from memory to the Marlin real-time controller over the serial port. These G-code instructions are extracted from a job. A job is basically a zip file containing:

* A program to execute
* A G-code file
* Zero or more bitmaps
* A certificate
* Any other optional file, used by the program

The certificate is used by the system to verify the job is legitimate. Only jobs that are generated by Ultimaker (or, more precise, are signed by Ultimaker) are allowed to be run on the system.

The program instructs the system what to do, for example: configure the system with specified settings, select a material, set operation parameters such as temperatures and speeds or send G-code data to the Marlin real-time controller.

Jobs can be provided to the system in several ways: they can be stored on the system, they can be read from an USB flash drive and they may be delivered to the system over a network connection.

As discussed in the previous chapter the system is provided with a TFT display and touch panel. A job can be used to drive the display and respond to touch events. High-level functions are provided by the job executor to display a bitmap on the screen, draw a text on the screen in a given font, select an appropriate text from the text catalog for the configured language, send a G-code file to the real-time controller, or change some system configuration setting (for example: Wi-Fi configuration settings).

A Web API task implements functionality to send events to the executing job and implement storage of new jobs and retrieve status information.

An USB mass storage device (flash drive) can be connected to the USB host interface of the system. The system will try to mount the filesystem automatically as read-only. Jobs stored on the storage device can be selected and executed.

Figure 1 shows a drawing of the architecture. Everything encapsulated in the large box runs in kernel mode[[1]](#footnote-1). The boxes above that are applications that run in user mode[[2]](#footnote-2). ‘Printer driver’ refers to the software that communicates with the Marlin real-time controller and is provided by Ultimaker.

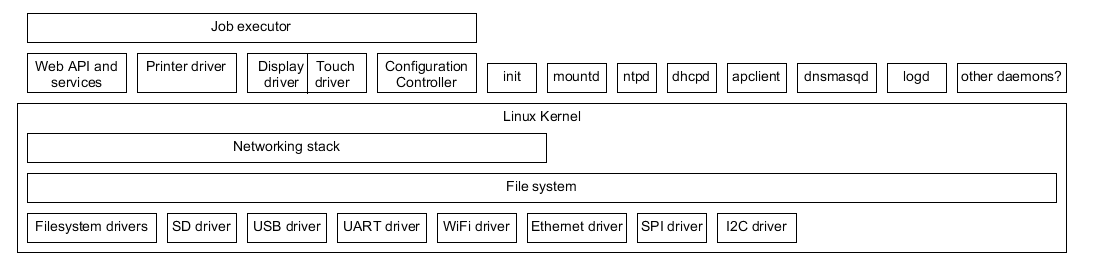


Figure 1

# System software

## Boot sequence

When switched on, the module starts to execute das U-Boot loader. This loader loads the kernel image from on-board flash into memory and jumps to the entry point of the kernel. If the reset button is pressed during power on, the loader can be used to retrieve a new kernel image from a connected USB drive and replace the existing one.

The kernel then mounts a root filesystem read-only and loads the device drivers and allows these to initialize. When the kernel is done initializing, it starts the init program, that mounts all required filesystems and starts up all background daemons and applications. The job executer is also started by the init script at an as early as possible moment, but at least before network configuration and starting other background tasks. This allows the job executor to execute the ‘autoexec’ job that can be used to update the display with the progress of starting up. Once the minimum required functionality is loaded the job executer can be used to invoke jobs. These can be preloaded jobs, for example to level the bed or load the filament or jobs from the mass storage device connected to the USB port. Once networking is up and running the web API tasks can provide jobs as well.

## Used device drivers

\*Needs more detailed information\*

* Wifi and Ethernet drivers
* Serial port driver
* SPI driver
* I2C driver
* GPIO driver
* Framebuffer driver

Note: currently the communication with touch panel will be handled in userspace.

## Used system daemons

* The odhcpd is used to configure IP address using the dhcp protocol
* ap-client handles management of the wifi interface
* dnsmasqd handles domain name resolving
* ntpd synchronizes system time
* mountd
* logd handles logging from both system tasks and applications

A number of daemons are configured by default but it is not clear if these can be disabled or must be retained:

* uhttpd
* onion-helper
* device-client
* rpcd (remote procedure calls daemon? Needed by nfs?)
* dropbear (ssh daemon, I’d rather keep it but violates ‘no listening ports’)

## Filesystem overview

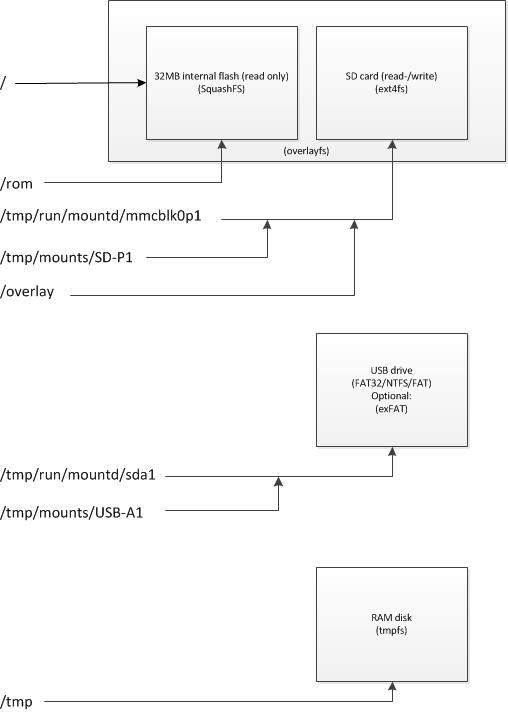


Figure 2

Figure 2 illustrates the structure of the filesystem. The root filesystem is mounted using the squ

## System configuration files

System configuration files are generated by UCI. UCI is intended to centralize the whole configuration of a device. It stores all configuration data in a central database that is manipulated by system utilities or from a program using an uci library. Refer to chapter Configurator on page 18

## Shell

The used shell is busybox.

## Internationalization and localization

Programs within the E2 environment that interface with the user will need to be made aware of multiple languages. Gnu gettext will be used to add Internationalization (i18n) and localization (i10n) to this software. The examples and tools used are based on python, but the usage is also applicable for other programming languages. A specific file structure is used by gettext we suggest to keep this directory structure in the Application root e.g. locales. Within this folder we need to create subfolders for each language and each folder contains a another folder named LC\_MESSAGES with one or multiple \*.po files.

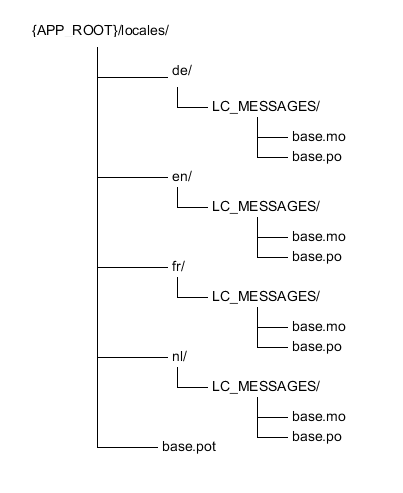


Figure 3

These \*.po files are plain text files contain text segments in the default language (english) and the translation to the specific language. The PO files themselves are also called message catalogs. Apart from the PO files there also might be a .mo (machine object) file that is a compiled binary version of the .po file.

The locales can also contain a .pot file. This is a template file for the .po file. It has all translation strings left empty. In practice the .pot file will be generated with a tool from the source file and should not be modified directly.

The .pot file should be copied to the language directories and the translation should be added using an editor. However also an external tool: e.g. poedit (https://poedit.net/) can be used to create the .po and .mo files from the .pot file (with this tool it is also possible to maintain the translations if later new messages are added.

### Example Application

To understand the process and tools that can be used to apply the localization we start with a function that prints some strings:

To understand the process and tools that can be used to apply the localization we start with a function that prints some strings:

**# demo.py**

**def update\_screen()**

**sz = draw.textsize(u'Ready'.decode('utf-8'), font = font)**

**draw.text((315-sz[0],10),u'Ready'.decode('utf-8'),font=font, fill(102,224,255))**

**if \_\_name\_\_ == '\_\_main\_\_':**

**print("Hello World")**

**update\_screen()**

The first step is to mark all translatable strings in the program. To do this we wrap the strings inside \_()

**# demo.py**

**import gettext**

**\_ = gettext.gettext**

**def update\_screen()**

**sz = draw.textsize(\_(u'Ready').decode('utf-8'), font = font)**

**draw.text((315-sz[0],10),(\_(u'Ready').decode('utf-8'),font=font, fill(102,224,255))**

**if \_\_name\_\_ == '\_\_main\_\_':**

**print(\_("Hello World"))**

**update\_screen()**

If you would run this program nothing appears to have changed. However now we are able to extract the translatable strings into a POT file.

### Generate raw translatable messages

The gettext project provides a set of tools to help to parse the source to extract the messages into a general message catalog. The original gettext only supported C or C++ source files, but xgettext can scan a number of programming languages (including python). We will use two specific python tools called pygettext.py and msgformat.py that can only be used for python source code.

**$ /home/tools/pygettext.py -d base -o /home/locales/base.pot demo.py**

That will generate a base.pot file in the /home/locales folder taken from the python source file. (Remember: don't edit this POT template file).

Finally copy this POT file to the language directory destinations as PO files.

**/home/locales/${language}/LC\_MESSAGES/${domain}.po**

Finally edit the PO files to add the proper translations.

From these PO files the catalog can be build using the tool named msgformat.py. This tool parses the .po file and generates an equivalent binary .mo file that is parsed by the gettext module in order to be used in the E2 applications.

**$ cd /home/locales/${language}/LC\_MESSAGE**

**$ /home/tools/msgformat.py -o base.mo base**

### Switching Locale

To have the ability to switch locales in a program we can use the gettext.translation() method. This will read the MO file for the specified domain and language. If the MO file is missing an exception is generated. Add the following to to switch languages:

**# demo.py**

**import gettext**

gettext.install('base', 'locales')

**def update\_screen()**

**sz = draw.textsize(\_(u'Ready').decode('utf-8'), font = font)**

**draw.text((315-sz[0],10),(\_(u'Ready').decode('utf-8'),font=font, fill(102,224,255))**

def test():

print(\_("Hello World"))

**update\_screen()**

if \_\_name\_\_ == "\_\_main\_\_":

nl = gettext.translation('base', localedir= 'locales', languages=['nl'])

nl.install()

test()

de = gettext.translation('base', localedir= 'locales', languages=['de'])

de.install()

test()

fr = gettext.translation('base', localedir= 'locales', languages=['fr'])

fr.install()

test()

en = gettext.translation('base', localedir= 'locales', languages=['en'])

en.install()

test()

# Application specific software

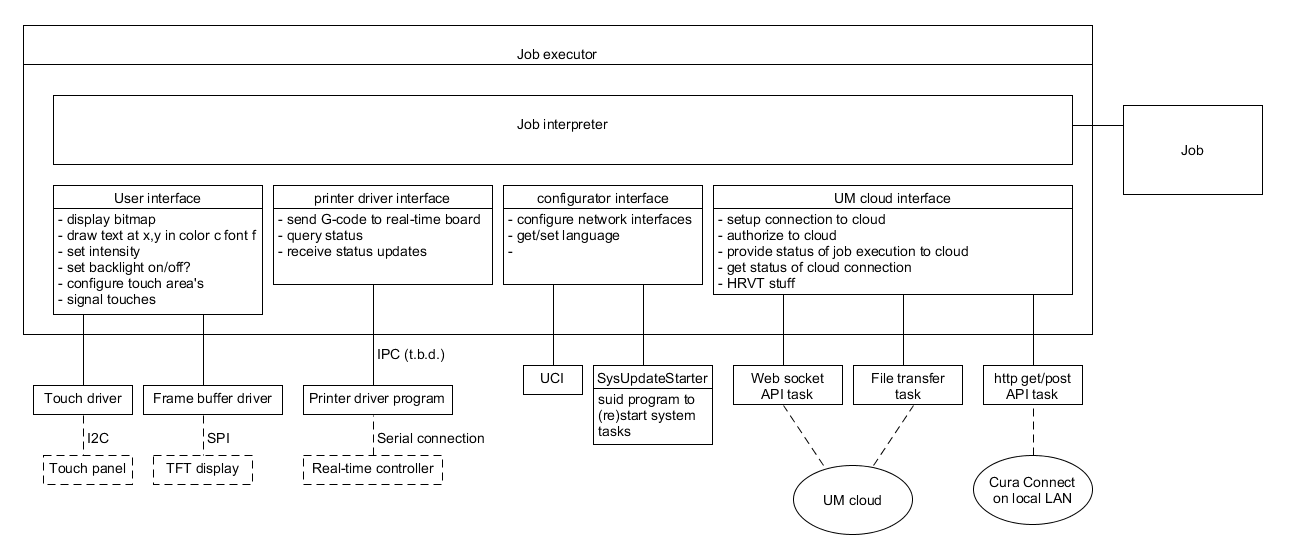


Figure 4

Figure 4 shows all application components of the system. Central component is the job executor. Design philosophy is ‘everything is a job’. Jobs are zipfiles that are used as container for multiple files and provide all required information to execute the actions described by the job. As an example, consider a job that is intended to print an object. For this it needs the G-code file generated by Cura, a bitmap generated by Cura and a very simple script that instructs the executor to show the bitmap on screen and send the G-code to the printer driver. More complex scripts can be embedded in the job to handle special cases such as modifying system configuration.

## Job executor

When a job starts executing, the job executor unpacks the files in the job file to a temporary directory and then starts interpreting the script it contains. The script can contain basically any kind of work, as long as it can be done with the limited privileges the job executor has. For most jobs a set of “System” APIs can be used by the job script. These are implemented as system library interfaces as show in Figure 4. In turn these libraries communicate with external software components (i.e. printer driver program, web API socket API task).

If a job does not contain an script to execute, the job executor considers the job a print job and will execute a build-in script to pass the G-code file to the printer driver.

## Printer driver

The printer driver is a piece of software that acts as a software interface between the job executer and the Marlin real-time controller board. The driver accepts a stream of G-code commands as input. It is important to feed the real-time controller board fast enough with new G-code commands as it needs some G-code instructions in advance to be executed in the future to plan smooth movements of the printer head.

The real-time controller board acknowledges the reception of each G-code instruction and the printer driver is responsible for re-sending an instruction in case of negative acknowledgement.

The real-time controller board has a queue of n G-code instructions where n is an integer number >1. It uses the queued G-code instructions to calculate smooth movements along the instructed coordinates. The printer driver is responsible for keeping this queue filled as underruns might cause jerky movements of the printer head. However, in some cases G-code instructions must not be queued by the real-time controller board but instead executed immediately. For example in the case of retrieving the current coordinates of the printer head it would be rather useless to wait for the completion of the buffered G-codes as this would always result in the coordinates of the last executed G-code instruction. The same is true for retrieving the current nozzle temperature: the controller board should not wait for executing this until all stored G-codes are executed as this would delay the operation significantly, resulting in a temperature at a much later point in time.

The printer driver on one side keeps the queue filled with G-code instructions to move the printer head and extruder, and on the other sides keeps track of the current state of the printer such as current head coordinates and nozzle and bed temperatures. Therefore the printer driver needs to mix the G-code instructions from the job with instructions to retrieve the current state from the machine. The same mechanism can be used to send high-priority instructions to the printer, for example to abort the current operation.

*The printer driver is provided by UM. A strict software interface is to be defined.*

At least the following functions are required:

* Send stream of G-code instructions
* Wait for completion of sent instructions
* Get machine status (x,y,z,e,nozzle temp, bed temp, etc)

**Note**

We have looked at the Marlin protocol drivercode code that was shared by Peter Brier d.d. 16-8-2018:

.\E2 Document share Betronic\Document exchange\2018082018 - UM3 mainboard + UM3 SW

.\griffin.tar\usr\share\griffin\griffin\printer\drivers\marlin\\*

We tried to install this python based driver on the Onion Omega2 module and tried to initialize the driver class. We found the marlinDriver.py includes two problematic drivers:

* from griffin import: dataLogger (This import needs numpy, which doesn’t install correctly on openwrt)
* from griffin.printer.faultHandler import FaultHandler, FaultCode (needs dbus wich is problematic on openwrt)

By simulating a container class, it is possible to initialize (at least partially) the driver:

**from griffin.printer.drivers.marlin.marlinDriver import MarlinDriver**

**from griffin.printer.properties.propertyContainer import PropertyContainer**

**class Controller:**

**def \_\_init\_\_(self):**

**self.\_\_property\_container = PropertyContainer()**

**self.\_driver = MarlinDriver(self, "/dev/ttyS1")**

**def \_hardwareInitializationFinished(self):**

**self.\_driver.start()**

**def getDriver(self):**

**return self.\_driver**

**def debugGetDriverInfo(self):**

**return self.\_driver.debugGetInfo()**

**def debugGetDriverErrorCounts(self):**

**return self.\_driver.GetErrorCounts()**

**def debugGetDriverSuccessCounts(self):**

**return self.\_driver.GetSuccessCounts()**

**def getPropertyContainer(self):**

**return self.\_\_property\_container**

**def getPropertyValue(self, key):**

**if key == "hotend\_count":**

**return 1**

**return key**

**tst = Controller()**

**tst.\_hardwareInitializationFinished()** fr = gettext.translation('base', localedir= 'locales', languages=['fr'])

fr.install()

test()

en = gettext.translation('base', localedir= 'locales', languages=['en'])

en.install()

test()

However during initialisation other modules are called/needed, which we have not studied in detail to understand their functionality and how to simulate/skip its behavior.:

Traceback (most recent call last):

File "test\_driver.py", line 38, in <module>

tst.\_hardwareInitializationFinished()

File "test\_driver.py", line 13, in \_hardwareInitializationFinished

self.\_driver.start()

File "/home/serial/griffin/griffin/printer/drivers/marlin/marlinDriver.py", line 91, in start

PropertyLink(self, PropertyLink.DIRECT\_QUEUE, pc.get("movement\_volume"), "M12000 X%g", lambda v: v["max"]["x"]),

File "/home/serial/griffin/griffin/printer/drivers/marlin/propertyLink.py", line 25, in \_\_init\_\_

assert property is not None

AssertionError

Based on the results above, we extended the sample code we used to periodically get the bed temperature from the printer.

import time

import crcmod

import serial

import sys

sys.path.insert(0, '/home/ctypes')

import baud

def remove\_comments(line, sep):

for s in sep:

i = line.find(s)

if i >= 0:

line = line[:i]

return line.strip()

class Comm:

def \_\_init\_\_(self):

self.SEQ = 0

self.ser = serial.Serial(

port='/dev/ttyS1',

timeout=10,

parity=serial.PARITY\_NONE,

stopbits=serial.STOPBITS\_ONE,

bytesize=serial.EIGHTBITS

)

if self.ser.isOpen():

self.ser.close()

self.ser.open()

#self.ser.isOpen()

baud.set\_baud("/dev/ttyS1", 250000) # set baudrate (using c ioctl call)

def mk\_frame(self,seq, msg):

crc16 = crcmod.mkCrcFun(0x11021, rev=False, initCrc=0xFFFF, xorOut=0x0000)

frame = bytearray((255,255))

frame.append(seq)

frame.append(len(msg))

for c in msg:

frame.append(ord(c))

frame.append(0)

frame.append(0)

crc = crc16(frame[2:-2])

frame[-2] = int(crc/256)

frame[-1] = int(crc%256)

return(frame)

def print\_gcode(self, fname):

with open(fname) as f:

for gcode in f:

if ';End of Gcode' in gcode:

return

stripped\_gcode = remove\_comments(gcode, ';')

if stripped\_gcode != "":

res = self.mk\_frame(self.SEQ, stripped\_gcode)

self.ser.write(res)

print(self.SEQ,repr(res))

rep = self.ser.readline().strip()

if rep[0:2] == b'ok': # message OK

seq=int(rep.decode('utf8').split(' ')[1][1:])

self.SEQ = seq + 1

if rep == b'': # Ultimaker sequence reset

self.SEQ = 0

if rep[0:3] == b'rej': # Out of sequence

seq=int(rep.decode('utf8').split(' ')[1][1:])

self.SEQ = seq

self.ser.reset\_input\_buffer() # Flush error meesages

def main():

um\_com = Comm()

t = um\_com.print\_gcode('HeDi-080820118.gcode')

if \_\_name\_\_ == "\_\_main\_\_":

main()

While this code worked fine for getting the bed temperature, it doesn’t work for sending G-codes to the printer, it seems some commands produce multi-line responses. The solution for that was to read the whole serial response buffer. (which could contain a variable number of lines).

Using this mechanism the SEQ numbers were in sync with the real-time engine responses. However just sending a gcode file this way did not result in the printer executing the commands send. (only the FAN ON/OFF commands seemed to work). So further study is needed to find out what is exactly needed for the printer to responds properly.

## Display driver

The display driver consists of a user-mode library that handles interfacing with the TFT display. From the application perspective, the driver needs to provide functions to:

* draw a provided bitmap at location (x,y) on screen
* draw a provided text with a specified font at location (x,y) in specified color
* set display intensity level

The driver needs to be fast enough to update the contents of a full screen in less then 100ms, preferably faster.

The display is connected to the processor using SPI. The kernel provides a framebuffer and handles updating the display with the framebuffer contents over SPI. The framebuffer is accessed as a block device. (/dev/fb0)

## Touch driver

The display is fitted with a touchscreen. The touchscreen’s controller is connected to the CPU over I2C. The touchscreen driver needs the provide the following functionality to the application software:

* Configure non-overlapping area’s in screen coordinates to respond to
* Send an event to the application in case one of the area’s is touched, identifying the area

The driver is implemented as either a daemon with a specified API for use by the job executor or as a hardware device driver.

## Configurator

When connected to a LAN using Ethernet the system will use DHCP to obtain network configuration settings such as IP address, subnet mask, default gateway and nameserver settings. However, in certain cases it may be necessary to configure the LAN interface with a static IP address. This can be accomplished by changing the default configuration of the LAN interface.

OpenWRT uses UCI (unified configuration interface) to maintain all system settings, including network interface configuration. It does this by generating a network configuration file from its own configuration repository. This repository can be modified using the UCI API interface that is available for shell scripts, C programs and more. After changing the configuration the related system component must be restarted to make the change effective. In the case of our network configuration example this is done by invoking /etc/init.d/network restart.

*Does the UCI api require root privileges? If so, how do we solve this? Using a setuid application that can only set allowed settings?*

## UFP file validation

The printer will need to be able to use ufp files to print objects. These ufp files contain also metadata about the object that needs to be retrieved. To be able to cope with various file formats and tags that are used (or might be used in the future) it is decided to use the library that Ultimaker specifically created for this purpose: libCharon. However the full version needs an environment that is not available on the module used in the printer. So a simplified version of libCharon is used. Content extraction is implemented in a validator. This validator is created in check\_ufp\_format.py in the function readUFPFile(filename: str, extract\_images: bool, extract\_gcode: bool) -> str

Information is extracted following this mechanism:

vf = VirtualFile()

vf.open(fname)

for path in vf.listPaths():

v = vf.getData(‘/metadata’ + path)

\*\*\*\* check Paths for key/value pairs of interest

\*\*\*\* extract thumbnails if requested

\*\*\*\* extract toolpath (gcodes) if requested

vf.close()

The actual paths that are extracted as key/value pairs are:

|  |
| --- |
| /metadata/3D/model.gcode/build\_plate/initial\_temperature |
| /metadata/3D/model.gcode/print/time |
| /metadata/3D/model.gcode/print/min\_size/[x/y/z] |
| /metadata/3D/model.gcode/print/max\_size/[x/y/z] |
| /metadata/3D/model.gcode/machine\_type |
| /metadata/3D/model.gcode/extruders/0/initial\_temperature |
| /metadata/3D/model.gcode/extruders/0/guid |
| /3D/model.gcode (extracted as: /toolpath) |
| /Metadata/thumbnail.png (extracted as: /preview/default) |
| /Metadata/thumbnail.png/\*\*\*/240x240 (if no preview/default) |

In the various menu’s two different image scalings are needed: a (max) 50x50 and a (max) 240x130 image. It was decided to include these images size default in the ufp archive. However currently this is not the case. So we implemented a Lite thumbnail resize function, using PIL-Lite to create these images with maintained aspect ration, from a default thumbnail (which can be in any size).

If /preview/default image extraction fails, the Path:

/Metadata/thumbail.png/……/nnnxnnn is searched for a match, if either x or y size is equal/larger than 240 that thumbnail is selected as preview image.

Other metadata is extracted to be able to properly configure the printer (bed temperature and nozzle temperature), or the check if the object can be printed on this printer (min\_size[X/Y/Z], max\_size[X/Y/Z] and machine\_type for wich this object was sliced) or show information about the object (material used and print time)

The material used is stored as GUID in the ufp file, this guid is translated into a displayable string by looking up the guid in a materials.json file.

The check on build volume and printer type is implemented by comparing the information in the ufp file with a configuration json file on the printer (e2\_config.json)

## Websocket API task

The Websocket API task provides the connection between the job executor and the Ultimaker cloud. When a network connection becomes available, the Websocket API task will connect to a predefined UM webserver. After connecting it will upgrade the current http(s) connection to a websocket. First the printer must identify itself using the “connection\_request”. If all goes well the UM cloud will respond with a successful response. Once in this state, messages to send jobs and query printer status can be sent from the Ultimaker cloud to the task. The task will respond with a response, but may also send messages asynchronously from the request/response from/to the Ultimaker cloud.

A message is either a request or a response. Requests can originate from both sides: either the UM cloud or printer. The receiver of a request sends a response to the originator.

A message consists of a JSON object, having two fields: “type” and “payload”. Field “type” indicates the messages’ purpose and whether it is a request or a response. Depending on the purpose the field “payload” contains additional information. For responses, the payload always contains the field “status” that indicates whether or not the command succeeded.

Example request:

{

"type": "connection\_request",

"payload": {

"printer\_id": "b5ff603a-a228-42bf-9ca7-63fef184966d",

"printer\_name": "OfficePrinterTwo",

1. "firmware\_version": "1.0.0",

"mac\_address": "82:A7:6D:09:12:64",

}

}

Example response:

{

"type": "connection\_response",

"payload": {

"status": "SUCCESS",

"printer\_id": "b5ff603a-a228-42bf-9ca7-63fef184966d",

"human\_code": "A6RY3Y",

"secure\_id": "71cMpHAu2sCZM0QdV6\_KJVDFQBG1bYndd2mitC6DP7b5"

}

}

### Connecting to the UM cloud

This request is sent when the printer had not received a successful response of this request before in its lifetime. Beware that firmware updates might require a restart of the lifetime (for example in case of system recovery or when new protocol versions dictate more data in the payload that is required to successfully issue other requests.

#### Request

|  |  |  |
| --- | --- | --- |
| Field “type” | | “connection\_request” |
| Field “payload” | | |
|  | “printer\_id” | UUID of printer, stored in uboot environment  (can be passed on to the linux kernel and read from /proc/cmdline) |
|  | “printer\_name” | ***Where does this come from? It is first time use, so user did not have any opportunity to configure this.*** |
|  | “firmware\_version” | Firmware version in major.minor.build format. ***However, the version of what component? Uboot? (probably not, user cannot upgrade uboot so this verwion is publicly known) Linux kernel? Application? Other components?*** |
|  | “mac\_address” | MAC address of network adapter? ***What network adapter? WIFI or Ethernet? It looks like mac address wifi = mac address Ethernet + 1*** |
|  | “secure\_id” | String of characters consisting of A-Z, a-z and 0-9 characters. Only present if the printer |

#### Response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “connection\_response” |
| Field “payload” | | |
|  | “status” | “SUCCESS” when request was successfully processed  “ERR\_ALREADY\_CONNECTED” in case of failure during processing of the request. |
|  | “printer\_id” | UUID of printer, as received by the UM cloud in the request. Only present when status = “SUCCESS” |
|  | “human\_code” | 6-character code consisting of only A-Z and 0-9 characters. This code is presented on the display of the printer when the printer receives this response. The user reads this code from the display and enters it in the cloud so that the UM cloud can connect an user to a printer. Only present when status = “SUCCESS” and “secure\_id” was absent in request. |
|  | “secure\_id” | String of characters consisting of A-Z, a-z and 0-9 characters. Tbd  Only present when status = “SUCCESS” |

### Disconnecting from the UM cloud

This request from printer to cloud will deregister the printer from the cloud. This is a rather unlikely event to happen as in most cases the printer will be switched off, leaving the printer no chance to deregister.

#### Request

|  |  |
| --- | --- |
| Field “type” | “disconnection\_request” |
| Field “payload” | Absent |

#### Response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “disconnection\_response” |
| Field “payload” | | |
|  | “status” | “SUCCESS” when successfully registered from UM cloud  “ERR\_NOT\_CONNECTED” when the request is received by the UM cloud for a websocket connection that did not sent “connection\_request” during its lifetime. |

### Push job

This is a message initiated by the UM cloud to request the printer to download a printer job. A printer job is a single file. It should be retrieved using https on a separate connection. The URL from where to download the file is provided in the request message.

#### Request

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_request” |
| Field “payload” | | |
|  | “job\_id” | UUID for this job, used to identify it in communication between UM cloud and the printer. |
|  | “job\_url” | URL of the job file. Only supported protocol is https |
|  | “job\_name” | Human readable identification for this job, not unique. |
|  | “username” | Name of the user logged into the UM cloud that commanded this printer job. |

#### Response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_response” |
| Field “payload” | | |
|  | “status” | “SUCCESS” when the job was successfully downloaded on the printer  “ERR\_CANNOT\_DOWNLOAD” if the job was not successfully downloaded within the maximum allowed time. |

### Start/stop/pause/resume a job

These requests can be sent from the UM cloud to the printer to control the processing of the job. A “start” request will execute the specified job from the beginnen. A “stop” request will abort the current current job. The pause request will temporarily stop processing the job whereas “resume” will resume the paused job.

#### Start request

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_start\_request” |
| Field “payload” | | |
|  | “job\_id” | Unique identification of the job as specified with the “push job” request. |

#### Start response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_start\_response” |
| Field “payload” | | |
|  | “status” | “SUCCESS” when the specified job is successfully started  “ERR\_BUSY” when another job is already being executed  “ERR\_INVALID\_JOB” when job cannot be started due to restrictions by the printer (job contains illegal instructions, job is not suitable for this type of printer, not enough local storage available to execute job” |
|  | “job\_id” | The unique identification of the job as specified in the request. |

#### Stop request

|  |  |
| --- | --- |
| Field “type” | “job\_stop\_request” |
| Field “payload” | absent |

#### Stop response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_stop\_response” |
| Field “payload” | | |
|  | Status | “SUCCESS” when the printer successfully aborted the job  “ERR\_ALREADY\_IDLE” if no job was running at the moment the request was received by the printer. |
|  | “job\_id” | The unique identification of the job as specified in the request. |

#### Pause request

|  |  |
| --- | --- |
| Field “type” | “job\_pause\_request” |
| Field “payload” | Absent |

#### Pause response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_pause\_response” |
| Field “payload” | | |
|  | Status | “SUCCESS” when the printer successfully aborted the job  “ERR\_ALREADY\_PAUSED” if the running job was already paused  “ERR\_ALREADY\_IDLE” if no job was running at the moment the request was received by the printer. |
|  | “job\_id” | The unique identification of the job as specified in the request. |

#### Resume request

|  |  |
| --- | --- |
| Field “type” | “job\_resume\_request” |
| Field “payload” | Absent |

#### Resume response

|  |  |  |
| --- | --- | --- |
| Field “type” | | “job\_resume\_response” |
| Field “payload” | | |
|  | Status | “SUCCESS” when the printer successfully aborted the job  “ERR\_BUSY” when the job was not in paused  “ERR\_ALREADY\_IDLE” if no job was running at the moment the request was received by the printer. |
|  | “job\_id” | The unique identification of the job as specified in the request. |

### Request status

This request is used by the UM cloud to query the status of the printer and the job it is printing. Additional fields may be returned by the printer as part of the payload.

#### Request

|  |  |
| --- | --- |
| Field “type” | “status\_request” |
| Field "payload” | absent |

#### Response

|  |  |  |  |
| --- | --- | --- | --- |
| Field “type” | | | “status\_response” |
| Field “payload” | | | |
|  | “status” | | “SUCCESS” |
|  | “printer” | | |
|  | | “status” | “PRINTING” when a job is being executed  “PAUSED” when a job was being executed but is temporarily paused  “IDLE” when no job is being executed |
|  | | “network\_name” | SSID? |
|  | | “material\_guid” | UUID of material |
|  | | “material\_name” | Name of material as specified by material job file |
|  | “job” | | |
|  | | “job\_id” | UUID of job currently being executed |
|  | | “job\_name” | Name of the job as provided by the job itself |
|  | | “estimated\_time\_left” | Number of seconds |

## File transfer task

The file transfer task is triggered by the websocket API task to retrieve a job. As the job is just a single file and the used protocol is HTTPS this implies that the task is really simple. It could be implemented using curl.

## HTTP API task

The http API task consists of a webserver that implements APIs using simple http GET and POST methods. As in general the webserver cannot be reached from outside the local network the use of this API is limited to machines connected to the same local network. If the router/firewall that connect the local network to the internet is properly configured the APIs can also be used from any machine on the internet.

The following APIs need to be implemented:

|  |  |  |
| --- | --- | --- |
| **Type** | **Endpoint** | **Description** |
| POST | /auth/request | Request authorization token to access printer |
| GET | /auth/check/{id} | Check for available id (only successful after user accepted on screen) |
| GET | /auth/verify | “This API call always does authentication checking for digest authentication” |
| POST | /printer/beep | Generate beep for specified duration[[3]](#footnote-3) |
| POST | /printer/led/blink | Blink specified led for specified duration3 |
| GET | /printer/heads | Returns all heads of the printer |
| GET | /printer/heads/{head\_id} | Return head information |
| GET | /printer/heads/{head\_id}/extruders | Return all extruders for this head |
| GET | /printer/heads/{head\_id}/extruders/{extruder\_id} | Return extruder information |
| GET | /printer/heads/{head\_id}/extruders/{extruder\_id}/active\_material | Return active material of the extruder |
| GET | /printer/heads/{head\_id}/extruders/{extruder\_id}/active\_material/length\_remaining | Length of material remaining on spool in mm |
| GET | /printer/heads/{head\_id}/extruders/{extruder\_id}/hotend | Returns/sets hotend of extruder |
| GET  PUT | /printer/heads/{head\_id}/extruders/{extruder\_id}/hotend/temperature | Returns temperature of extruder |
| GET | /printer/heads/{head\_id}/extruders/{extruder\_id}/active\_material/guid | Returns GUID of the active material |
| /printer/heads/{head\_id}/extruders/{extruder\_id}/active\_material/GUID |
| GET | /print\_job | Returns information of current printjob |
| POST | /print\_job | Print a job (contains file in payload) |
| GET | /print\_job/name (?) | Returns name of the printjob |
| GET | /print\_job/datetime\_started (?) | Returns timestamp of start of printjob in ISO8601 |
| GET | /print\_job/datetime\_finished (?) | Returns timestamp of last finished printjob in ISO8601 format |
| GET | /print\_job/datetime\_cleaned (?) | Returns timestamp of last print job clean build plate (clean=remove?) |
| GET | /print\_job/source (?) | Returns source: USB, WEB\_API or CALIBRATION\_MENU |
| GET | /print\_job/gcode | Returns G-code of of active printjob |
| GET | /print\_job/container | Returns file of active printjob, G-code, g-zipped, or UFP |
| GET | /print\_job/state | Get printjob state information |
| PUT | /print\_job/state | Continue, abort or pause the printjob |
| GET | /history/print\_jobs | Get historic info of printjobs |
| GET | /history/print\_jobs/{uuid} | Get detailed info of specified job |
| GET | /history/events | Get event history |
| POST | /history/events | Used to register the completion of maintenance tasks |
| GET | /system | Returns entire system object WTMB |
| GET | /system/platform | Returns a string identifying the underlying platform in human readable form |
| GET | /system/hostname | Returns the hostname of this machine |
| GET | /system/firmware | Returns the version of the firmware currently running |
| PUT | Trigger firmware update; firmware update are downloaded by the printer itself |
| GET | /system/firmware/status | Returns the status of the firmware update (cannot work? During update printer does not respond!) |
| GET | /system/firmware/stable | Returns firmware version of the most recent stable version |
| GET | /system/firmware/testing | Returns firmware version of the most recent ‘test’ version |
| GET | /system/memory | Returns current memory usage |
| GET | /system/time | Returns current UTC time |
| GET | /system/name | Returns the name of the printer |
| PUT | Sets name of the printer |
| GET | /system/uptime | Returns the uptime in seconds |
| GET | /materials | Returns all known materials |
| POST | Write material files in XML?! |
| GET | /materials/{material\_guid}/ | Returns XML of specified material |
| PUT | Updates XML of specified material |
| DELETE | Deletes the specified material |

# Evaluation of requirements

* FR-UI001 – FR-UI007 are hardware requirements, not subject to this architecture
* FR-UI008: display should be responsive (lag < .1s)
  + This is partly determined by hardware and partly by software
  + Display driver must by fast and efficient
  + Touch driver must be fast and efficient
  + Job executor must be fast
    - It may be necessary to execute all jobs within same process to prevent creation of a new process and loading the necessary software for that process. Solution: Job executor only executes 1 job at a time and consists of single process.
* FR-UI010: Implemented using timeout event and screensaver job in job executor
* FR-UI011: Implemented in display driver
* FR-UI012: Implemented in touch driver
* FR-UI013: As long as UM can provide bitmaps for display that meet this requirement
* FR-UI014: As long as provided functionality for detecting touches and displaying bitmaps and text overlays is fast enough all other properties of UX are determined by UM
* FR-UI015: Determined by provided bitmaps and overlay texts
* FR-UI016: Text can be drawn by configured and provided font (NOTO sans regular)
* FR-UI017: Implemented by specific job
* FR-EL001 – FR-EL004: outside scope of this document
* FR-EL005: Wifi and LAN interfaces are implemented in software as Linux network interfaces that use the Linux networking software (network stack, routing, name resolution, firewall functionality, etc)
* FR-EL006 – FR-EL007: outside scope
* FR-EL008: determined by inserted SD card
* FR-EL009 – FR-EL012: outside scope
* FR-NS001: Use Linux networking stack; WebSocket is implemented at application level but implementations are available
* FR-NS002: standard Linux
* FR-NS003: standard Linux, ‘proxy’ ?, ‘Eduroam’ ?
* FR-NS004: transfer of configuration implemented by job; translate to UCI config
* FR-NS005: ignoring this meets this requirement
* FR-NS006: implemented by events in the job executor, handled by jobs
* FR-NS007 – FR-NS010: standard Linux
* FR-SG001: requirement met
* FR-SG002: Can be met, but needs attention during development
* FR-SG003: todo after development
* FR-CC001: Implemented by the Web API interface
* FR-CC002: requirement met
* FR-CC003: partly implemented by Websocket API task, partly by (system) jobs
* FR-CC004 – FR-CC005: outside scope
* FR-CC006 – FR-CC007: determined by hardware and system software
* FR-CC008: Implemented by job executor
* FR-CC009: Updates are implemented as job and thus require valid certificate.
* FR-SC001: handled by job executor; note that open wifi must be selected by user
* FR-SC002: implemented by job executor
* FR-SC003: hardcoded into job executor (is this a good idea?)
* FR-SC004: handled by UCI and configuration jobs and the job executor
* FR-SC005: can be implemented by job and job executor
* FR-SC006: can be implemented as job
* FR-SC007: n.a.
* FR-SU001: Implemented by job executor and system jobs
* FR-SU002: auto mount? Configure Linux. If mounted by job executor then job executor implements this
* FR-SU003: handled by provided Linux file system drivers
* FR-SU004: basic Linux functionality
* FR-SU005: Implemented by 2-mode system API for job executor: privileged or restricted
* FR-SU006: handled by job executor
* FR-SU007: material types are jobs that set configuration settings to material related values
* FR-SU008: hardware requirement
* PR-P7: Implemented by splash screen of uboot
* PR-P8: Implemented by starting job executor directly at init. Job executor monitors startup progress and visualizes it by ‘autoexec’ job
* FR-PR002 – FR-PR003: See PR-P8
* FR-UM001: implemented by printer driver
* FR-UM002: see printer driver
* FR-UM003: see printer driver
* FR-UM004: n.a. (note that there seems to be simulator available!)
* FR-UM005: hardware requirement, Linux supported, related to printer driver
* FR-UM006: 100 G-code lines / sec implies that 1 line must be transmitted in at most 10 ms. 250kbps is 25kBps so 1 byte takes 40us. In 10ms 250 bytes can be transmitted, so this should be easily possible.
* FR-FU001: Requirement met by uboot
* FR-FU002: Requirement met by uboot
* FR-FU003: Implemented as job and supported by job executer; needs special attention as the job executor itself can be upgraded by this as well.
* FR-FU004 – FR-FU005: Any source for a job is suitable
* FR-FU006: job executer can only handle 1 job at a time
* FR-BMxxx: mechanical requirements, outside scope
* FR-P001: USB jobs are supported
* FR-P002: Implemented by Web API and job executor
* FR-P003: Implemented by Web API and job executor
* FR-P004: In order to implement this the job needs a bitmap picture of the object; progress can be shown as percentage of the file, drawn on the screen using text overlay. If the total length of the required filament is provided, a percentage of the processed filament can also be used.
* FR-P005: G-code is retrieved from job and spooled to printer driver
* FR-P006: Mechanical restriction; only one extruder is available to this requirement is met
* FR-P007: Feedback from the marlin print can provide the trigger to execute a predefined job
* FR-P008: see FR-P007
* FR-P009: implemented as job
* FR-P010: Select material? Implemented as job
* FR-S001: implemented as job
* FR-S002: firmware version can be shown on request by drawing text as overlay on the screen using a job
* FR-S004: factory reset is implemented by restoring system image using uboot, when no USB drive is available?
* FR-S006: implemented as jobs and executed by job executor
* FR-S007: implemented as job
* FR-S008: n.a.
* FR-S009: n.a.
* FR-S010: implemented as configuration setting that is used to retrieve the proper text using gnu gettext
* FR-S011: implemented as job
* FR-M001 – FR-M004: implemented as jobs
* FR-M005: can be implemented as job; LED is connected to marlin real-time controller
* FR-M006: implemented as job
* FR-M007: stats could be collected by job executor. Job could provide functionality to show these
* FR-M008: Must be implemented by printjob as the job manages the entire display contents
* FR-M009: n.a.
* FR-M010: n.a.
* FR-M011: implemented at linux level
* FR-M012: implemented by uboot
* FR-M013: won’t
* FR-M014: won’t, but could be implemented using jobs
* FR-M016: won’t
* FR-M017: direct API – as listening? Implemented in web API
* FR-M018: implemented in web API; this is a process that handles communication with UM cloud and converts commands to triggering events and storing jobs
* FR-M019: won’t

# Implementing system jobs

## Main task

Various menus, allows browsing USB

## Print Job

The printer is able to print a PLA print job from the internal SD card

A print job can be started in different ways:

1. The user selects the “Print from USB” from the main menu
2. The user selects the “Print from USB” from load/change material menu
3. A print job is received from a host printer (CURA connect)
4. A print job is received from CURA of Cloud

If “Print from USB” is selected, an USB drive needs to be inserted and is automounted en scanned for .ufp files.

If one or more .ufp files are found they are displayed in a menu, otherwise an error message is shown. These .ufp files are checked for proper formatting and thumbnails (format 50x50 and 240x120) are extracted. if the user selects the object to print it is first copied to the internal SDcard, additional information is shown and it is checked additionally (using libcharon) for:

1. Correct ufp format
2. Correct buildsize (max X/Y/Z)
3. Correct material guid (only PLA is allowed)

## Abort Print

The user is able to abort the current print job

## Pause / Resume Print

The user is able to pause and resume the current print job

## Update firmware / Factory default / Emergency recovery

The user can perform a firmware update or factory reset or emergency recovery

## Homing

Homing can be implemented as as follows:

* Load the bitmap for ‘homing’ from the job
* Load the G-code script for homing (most likely 1 instruction) from the job
* Wait till either the action is aborted or the G-code script finished executing

Optionally:

* Specific handling of either finished executing script or aborted.

## Clean filament path

It is unclear what this action actually is supposed to do. Assuming it is just another series of G-code commands it can be implemented identical to ‘homing’.

## Configure for first-time use

The autoexec job verifies the existence of the printer configuration file. If not found, the ‘first time use’ job is started. This job shows for each configuration setting the first value for this setting from the list of possible values. Using the touch panel the next and previous values can be selected or the current value can be confirmed. When confirmed, the value shown on the display is saved in the configuration file and the next setting is shown, again with the first from a list of possible values for that setting. When all configuration settings have been set this job is done.

## Human readable verification token handling

Once the network software is up after boot, the Web API interface program will connect to the Ultimaker cloud using http(s). Once connected, the connection is upgraded to a websocket connection. The websocket api program then sends a request to the Ultimaker cloud to identify itself. If the Ultimaker cloud has no record of this machine the request is denied. In that case a HRVT is displayed on screen. The user logs in onto the Ultimaker cloud and registers the printer using the HRVT. The Ultimaker cloud now links the HRVT to the authentication attempt before and sends a message to the printer to acknowledge the authentication. The printer can continue, now connected to the Ultimaker cloud.

## Heating buildplate

* To what temperature? Set by user or by configured material?

## Set extruder (nozzle) temperature

The user is able to regulate the nozzle temperature by hand via the internal display

## Material Handling

The user can change, load, unload material with the help of the UI

### Loading material

Simply fast forward the filament for xx mm until fully inserted in the extruder

### Unloading material

Simple fast reverse the filament for xx mm so it can be removed from the extruder

### Replace material

First unload than load new material

## Networking

### Connect printer via open lan

The user can connect the printer to an open LAN network (needing no further authentication)

There are three separated flows:

1. Connect via LAN
2. Connect via Wifi
3. Advanced configuration

### Disconnect printer from lan

The user can disconnect the printer from the network

### Show connection info

The system displays the connection information to the user via the internal display

## Browse USB filesystem

~~If no job is running, a job is started at the event of mounting the USB filesystem. “Current dir” is set to the filesystem’s root. The job identifies the existence of all jobs in the ‘current directory’ and displays these. All available subdirectories are presented as well. If a job is selected by the user, it is executed. If a directory is selected, the current directory is changed to that directory and procedure repeats.~~

Handled by main task

## Screen saver

If no job was run for x seconds and no user input was received for y seconds the screensaver job is started. This job can contain the screensaver image and a script to load it.

Optionally the intensity can be lowered and/or the backlight can be switched off.

## Lock down mode

A system API is added to switch the system into lock down (restricted) mode. In this mode only a limited set of system APIs can be executed. System APIs to access and/or change configuration settings are disallowed. The job that is executed to enter lock down mode invokes the mentioned system API. Todo: procedure to exit restricted mode

## Autoexec job

* Show boot progress
* Execute first-time settings job
* Execute

## Level buildplate

The user is able to level the buildplate using the procedure as described by ultimaker.

## Move buildplate

The user is able to move the buildplate up and down

# Use case scenario’s

* First use registration
* Non-first use registration

# Board support for TFT display

## Hardware

At the time of writing no carrier board for the onion was yet available. Therefore tests were done with an experimental setup based on standard onion modules ‘expansion dock’ and ‘ethernet expansion’. Basic connections are:

* Power
* SPI
* Backlight
* Data/control?

The final display was also not available, so tests were done with another display that uses the same controller as the final display. However, touch could not be tested as the experimental display contains a different (and non-working) touch controller.

## Software

This section describes the result of work done in sprint 3. Goals are:

* Get something on screen within 3 seconds after power on
* Get the application driving the display up as fast as possible but at least the first display modification must be done before 13 seconds after startup

When the system is switched on, first the uboot boot loader starts running. This software reads the linux kernel image and root filesystem image into ram and boots the kernel. If the kernel takes a long time to boot, to meet requirement PR-P7 (UM) the only option is to modify the bootloader so that it can initialize the display and write an image on it.

On the other side, if the kernel would boot very quickly displaying the first image on the screen could be done after loading the drivers for the display.

A management decision was made to opt for the version to not modify the boot loader. So final goal became to have the linux kernel loading the drivers and display the splash screen within 3 seconds after power on.

After the first splash the display must be updated at least once every 10 seconds (PR-P8 (UM)). The easiest way would be to start the application as quick as possible and have that update the display. This implies that the application must have updated the display before 13 seconds after power on. If this requirement is met, requirement FR-PR002 that requires the system to be operation within 35 seconds after power on is implicitly also met.

The next sections describe the modifications made and the results obtained.

### Find the source

Before we can modify anything we must first obtain the sources of the current firmware image and be able to rebuild it. In this stage the stock omega image was used, obtained by running git:

git clone <https://github.com/OnionIoT/source.git>

### Modify to reduce unneeded drivers and packages

In order to have the kernel boot as fast as possible all unnecessary drivers and system components must be removed to save time. Loading drivers and initializing them takes time, so why waste that on stuff not being used anyway?

Enter the source directory of the omega system software repository. From the directory where git was invoked, enter

cd source

and then enter

make menuconfig

This command starts a user interface where all packages can be configured. The Linux kernel is also just another package but only limited configuration options can be set so in the next section the kernel will be configured as well. Now first remove unneeded packages and drivers.

Target profile (multiple devices) 🡪

select Onion Omega+

Image configuration 🡪 Preinit configuration options 🡪

* disable failsafe
* set failsafe/debug timeout to 0

Image configuration 🡪 Version configuration 🡪

* set manufacturer name to ‘betronic’
* set manufacturer URL to ‘www.betronic.nl’

(These settings appear in /etc/device\_info)

Network 🡪 WWAN 🡪

In order to be able to disable ppp software, disable the following:

* comgt
* comgt-directip
* comgt-ncm
* uqmi

Network 🡪

Disable:

* ppp
* wwan (at the bottom of the list)

Kernel modules 🡪 sound support 🡪

Disable: sound support (everything)

Kernel modules 🡪 USB support 🡪

Disable:

* kmod-usb-acm
* kmod-usb-hid
* kmod-usb-net
* kmod-usb-printer
* kmod-usb-serial
* kmod-usb-wdw

Kernel modules 🡪 video support 🡪

Disable kmod-video-core

Kernel modules 🡪 W1 support 🡪

Disable kmod-w1 (1-wire)

Kernel modules 🡪 network support 🡪

Disable kmod-ppp

Kernel modules 🡪 other modules 🡪

Disable:

* kmod-bluetooth
* kmod-pps
* kmod-gpio
* kmod-ldisc

Now exit menuconfig and save the configuration.

menuconfig only supports a limited set of kernel building configuration options. Therefore the kernel must be configured separately. Start the kernel configuration tool by entering

make kernel\_menuconfig

Bus options 🡪

Disable support for PCI controllers

Device drivers 🡪

Disable sound card support

### Modify to add static kernel modules for TFT support

menuconfig only supports a limited set of kernel building configuration options. Therefore the kernel must be configured separately. Start the kernel configuration tool by entering

make kernel\_menuconfig

To be able to access the TFT display as soon as possible the necessary drivers are compiled as static part of the kernel instead of loadable modules. Therefore select the following with a ‘\*’ and not a ‘m’. The later configures the driver as loadable module, the former as static part of the kernel.

Device drivers 🡪 Graphics support 🡪 Frame buffer devices 🡪

Select support for frame buffer devices

Device drivers 🡪 Graphics support 🡪 Frame buffer devices 🡪

Select enable firmware EDID

Device drivers 🡪 Graphics support 🡪 Frame buffer devices 🡪

Select Framebuffer foreign endianness support and keep the default setting ‘Support for big- and little-endian framebuffers’

Device drivers 🡪 Graphics support 🡪 Frame buffer devices 🡪

Select:

* Enable video mode handling helpers
* Enable tile blitting support

Device drivers 🡪 Graphics support 🡪

Select backlight and LCD device support

Device drivers 🡪 Staging drivers 🡪

Enable:

* Staging board support
* Support for small TFT LCD display modules

Device drivers 🡪 Staging drivers 🡪 Support for small TFT LCD display modules 🡪

Enable:

* FB driver for the ILI9341 LCD Controller
* Generic FB driver for TFT LCD displays
* Module to for adding FBTFT devices

When device drivers are loaded dynamically they can receive arguments by specifying these with insmod. Since the static modules are, well, static and don’t need insmod, the arguments have to be specified on the kernel command line. Each setting requires the name of the device driver and a . in front of it. Now select Kernel hacking and below Built-in kernel command line (that is selected by default) add to the kernel parameters the following, all on a single line separated by a space:

* fbtft\_device.name="fb\_ili9341"
* fbtft\_device.busnum=32766
* fbtft\_device.cs=1
* fbtft\_device.speed=96666000
* fbtft\_device.mode=3
* fbtft\_device.fps=100
* fbtft\_device.txbuflen=153600
* fbtft\_device.custom
* fbtft\_device.buswidth=8
* fbtft\_device.bgr=1
* fbtft\_device.gpios=reset:2,dc:3
* fbtft\_device.width=240
* fbtft\_device.height=320
* fbtft\_device.rotate=90
* fbtft\_device.debug=2

Now if you’d build a new firmware image by entering

make

and then install this image on the omega, a /dev/fb0 device should be available and if you’d write to it, the data should appear on the display.

### Add splash screen

As discussed in the beginning of this chapter something needs to be displayed on the display as soon as possible to meet the requirements. The approach taken to accomplish this is to modify the driver that handles /dev/fb0 and have it write something to the display as soon as all initialization is done.

Now ‘something’ should be data that represents an image. This data needs to be stored somewhere. To prevent problems due to not yet initialized file systems and the like the, the image is compiled in the kernel as static data. The xxd program is used to convert the image into a statically declared C array.

Only a couple of lines are added to the device driver to write the image to the display after initialization. For this a patch file was created that adds the required lines to the device driver.

A small ‘package’ was created to utilize these, named bootsplash.tar.gz. The package contains the following files:

* framebuffer\_splash.patch, a patch file to add the required code to the device driver
* gensplash.sh, a script to convert a bitmap to an C source file that can be included by the driver
* umlogo.bmp, an example image that can be used as starting point for other pictures by opening it in paint and changing it.

As only the raw binary contents of the bitmap are needed in the fixed 320x240 pixels in 5:6:5 (16-bit) color the header of the bitmap needs to be stripped. This is taken care of by the gensplash.sh utility. To make sure the bitmap meets these requirements it is best to modify the existing sample image as that insures the correct resolution and color coding.

Assuming the current directory is still the source tree root, enter the following command to unpack the package:

tar xzf <path-to>bootsplash.tar.gz

Now patch the driver by entering the following on a single line

patch build\_dir/target-mipsel\_24kc\_musl-1.1.16/linux-ramips\_mt7688/linux-4.4.74/drivers/video/fbdev/core/fbmem.c < bootsplash/framebuffer\_splash.patch

Next, enter the bootsplash directory and generate the C include file:

cd bootsplash/

./gensplash.sh

And then recompile the firmware image

cd ..

make

Install the new image on the omega. After the reboot the splash screen should be shown immediately after initializing the drivers and the display.

According to the timestamp shown in the kernel console output the image is drawn within a second after starting the kernel. Add to this the three to four seconds the bootloader spends on loading and decompressing the kernel and the nett result is that the image is displayed within 5 seconds after startup.

### Demonstrate earliest display modification after startup

The preceding sections demonstrate the solution that meets requirement PR-P7 (UM). To meet requirement PR-P7 (UM) the application should be started within 10 seconds after displaying the splash screen. This section documents the test done to verify the technical feasibility of this.

The LEDE configuration used for the Omega starts the /etc/preinit script as first process of the kernel. This script in turn starts procd to handle startup of other tasks but also handles loading the dynamic device drivers. The first opportunity to start a program is from within this script.

A solution that might come to mind is to change the /etc/preinit script on the omega after installing the firmware image. However, at the early boot stage that /etc/preinit is run not all filesystems are mounted and as such neither is the overlay filesystem that would be required to read the ‘modified’ /etc/preinit. The only proper solution is to modify it directly on the build system tree.

The file is stored in <source root>/package/base-files/files/etc. Open it in your favourite editor. As an example an existing file is sent to the display. The file in this example contains binary data that does not represent a decent image but it demonstrates the concept.

Find the line that reads [ -z "$PREINIT" ] && exec /sbin/init. Insert a new line after this line that reads

/bin/cat /bin/busybox > /dev/fb0 &

Save the file and rebuild the firmware image. Then install the new image on the omega. After reboot the display should show a weird image that resembles noise on an old television set. With our tests the image is shown about 4 seconds after the splash screen image is displayed.

The program /bin/cat reads the file /bin/busybox and sends it to device /dev/fb0. The & is required to continue the boot process while this program is running.

If the /bin/busybox part of the line is replaced by an image in the proper binary format that image will be shown instead of the weird mess that busybox represents.

Obviously this is just a silly example demonstrating the concept. In the final application this command is replaced by the application that handles interaction with the user. That allows user interaction within 10 seconds after power on while remaining parts of the system still are being brought up.

# Board support for USB emergency recovery

To meet the field upgrade requirements it must be possible to install a new firmware image from USB MSD. As the firmware image contains kernel, root filesystem and applications this method allows recovery from any thinkable situation as long as the hardware and bootloader are intact.

The default U-Boot loader provided on the Omega allows installing firmware from USB MSD but requires user interaction on the console. As in the final product there is no console available this is not a valid option.

The default bootloader behaves as follows: when system is powerd on, it checks the ‘reset’ button. If it is pressed, it shows a menu on the console. One option on the menu is recovery from USB MSD. When no selection is made within 40 seconds it defaults to starting up the normal kernel. In case of a damaged firmware image and no user interaction on the console this would lead to an inoperable system.

A simple solution was implemented consisting of only two minor changes to the sources. The first change is that the default option in the menu, the option chosen if no selection was made by the user, is option ‘2’, that is: flash firmware from USB storage. With only this change the following scenario is possible:

The user keeps reset pressed during power on. The bootloader will not load and start the firmware but displays the menu on the console. The user has no console so does not see this menu. If no selection was made within 40 seconds the bootloader attempts to mount the USB stick and copy the firmware image into RAM. When this is completed successfully the image is copied from RAM to internal flash. After this the bootloader simply loads and runs the image from flash as it would do under normal circumstances.

As the user won’t see anything happen ‘till the reboot is finished it could be confusing that nothing happens within a large period of time. To reduce this period the timeout of 40 seconds is reduced to 5 seconds. This allows developers to still choose other options from the menu and still reduces the recovery boot time by 35 seconds.

## Preparing the build environment

To be able to obtain and build the bootloader a number of software packages must be installed on the development machine. In our case a minimal CentOS installation was used as starting point. The following commands install the required packages:

* yum groupinstall 'Development Tools'
* yum install libgcc.i686
* yum update
* yum install zlib-devel
* yum install wget
* yum install zlib-static
* yum install java
* yum install glibc.i686

## Obtaining the bootloader

The original omega bootloader can be obtained by executing the following command:

git clone <https://github.com/OnionIoT/omega2-bootloader.git>

The package contains cross compilers used to build the bootloader in binary form. Run the script setup\_env.sh to install these into /opt/buildroot-gcc342 and make other preparations to the environment.

## Modify bootloader to change default recovery option and menu selection timeout

Open the file lib\_mips/board.c in your favourite editor. Find the line that reads ‘BootType = 'b'’. Change ‘b’ into ‘2’. This changes the default boot option if no user input was received by the recovery boot menu.

Changing the timeout is a bit more complicated, as the bootloader source seems poorly modified by the Onion people. Immediately following the line above a while loop is entered that should be running for the duration of the timeout. The variable timer1 counts down from CONFIG\_BOOTDELAY to 0 where CONFIG\_BOOTDELAY is set to 5. The original author intended this to be in seconds, where the while loop iterates once every second and decrements the value of timer1. The one second spend in the loop must be used to poll the console for user input. To implement this the original author appearantly wrote a for-loop that runs 100 times, polling for console input and sleeping 10 milliseconds. However, the onion people modified this to have a blinking led, delaying two times for 30 ms, so 60 ms per for-loop iteration. 100 \* 60ms = 6 seconds, so instead of once per second the while loop iterates once per six seconds, causing the total timeout to be in the range of 5 \* 6 = 30 seconds. For some reason the printf line just above the while loop states that the timeout is 8 times the value of timer1, which is 5 seconds, for a total of 40 seconds.

This rather weird mess is ‘improved’ a bit by changing the for-loop to run 16 times instead of 100. 16 times 60 ms is about 1 second. This modification causes one iteration of the while-loop to take about 1 second instead of 6, resulting in the intended behaviour by the original author. Now the printf statement still prints that the timeout is 40 seconds, but this is easily fixed by changing timer1 \* 8 to simply timer1, removing \* 8.

## Building and installing the new bootloader

Save the modified file and then from the source root (omega2-bootloader) enter

make

This generates a new uboot.bin file that must be installed in place of the current bootloader on the omega. This is a bit risky, as, when this goes wrong, the omega module won’t boot and cannot recover from this. The only solution would be to access the internal flash memory chip but the costs of the required efforts will exceed the price of a new module.

After this scary statement now take the following steps to install the bootloader: switch power on while keeping the ‘reset’ button pressed. On the console a menu is shown that looks like this:

Please select option:

[ Enter ]: Boot Omega2.

[ 0 ]: Start Web recovery mode.

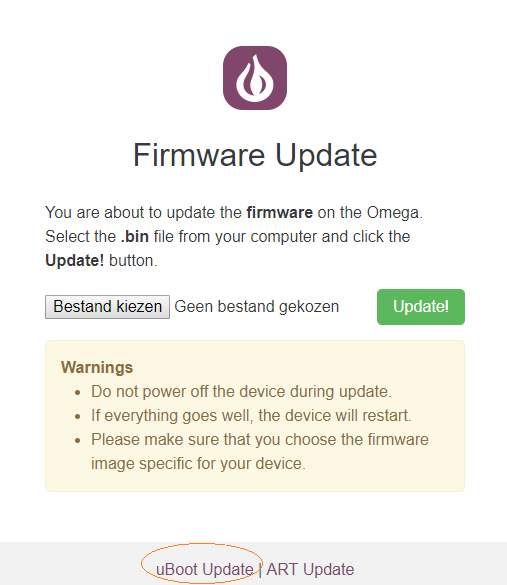
[ 1 ]: Start command line mode.

[ 2 ]: Flash firmware from USB storage.

Enter 0 to start web recovery mode. The bootloader will now start a tiny http server running on a small TCP/IP stack. The IP-address is fixed to 192.168.8.8. If this subnet is used on the network attached to the development computer, detach the computer from the network, connect the development computer to the omega using a cross cable and assign the computer a static IP address in the same subnet range, i.e. 192.168.8.1

If the subnet is not used on the network attached to the development computer the computer can be configured as ‘multi-homed’, having two IP addresses: one for the regular network and one for the “private” network with the omega module. In this case the omega is simply connected to the network.

In both cases, on the development computer, start a browser and browse to 192.168.8.8. A page appears that looks like this:



On this page click the link ‘uBoot update’ as highlighted on the picture. Basically the same page appears but now referring to the bootloader. Click on the grey button and in the file dialog select the uboot.bin file. Then click on the green button to start the update.

Uploading and replacing the firmware only takes a few seconds. After this the device attempts to reboot. Now verify the correct operation of the modified bootloader by powering the module on while keeping reset pressed. The text on the console should now represent 5 seconds instead of 40 seconds timeout and the default option should be to attempt to mount the USB MSD.

# Little VGL and pylvgl

## Introduction

Little VGL is a small C library for graphical user interfaces. It supports a large list of widgets such as label, button, (progress) bar, check box, drop down list and also some basic primitives such as line and arc. To interface with the display hardware it needs a driver. Drivers for various displays are available in the supplemental driver package.

Little VGL has its own documentation. It is far from complete but very to the point so it is a smart idea to study it. See https://docs.littlevgl.com/

## Using framebuffer driver

The UM2 hardware consists of an Onion SoC module that has a TFT display with touch support attached to it. The software interface to this is using a standard Linux framebuffer device. The kernel contains device drivers to map the frame buffer to the physical display.

The Little VGL drivers package contains a "generic" driver to output the graphical data to the framebuffer. Once LVGL is initialized the library's user does not have to deal with the framebuffer device.

## Using a pointing device

LVGL also contains built-in support for pointing devices (such as mouse or touch interface). The link between LVGL and this hardware is also taken care of in a standard driver for LVGL contained in the LVGL driver package.

Although using a mouse works just fine, supporting the (capacitive) touch panel is more difficult. For one thing it sends different events that are by default not handled by LVGL. The events that LVGL responds to by default are EV\_REL and EV\_ABS. EV\_ABS events are sent by the capacitive touch driver so up to here everything works just fine. However, whereas the LVGL library expects codes ABS\_X and ABS\_Y the drivers sends codes ABS\_MT\_POSITION\_X and ABS\_MT\_POSITION\_Y. In order to handle these events the evdev.c file was modified to support these codes as well. Then there is yet another problem: the X coordinates sent by the touch driver are mirrored. That basically means that the top left corner of the TFT display corresponds to the to top right corner of the touch panel. To solve this, the X (and optionally Y) coordinates are mirrored again. That is implemented by subtracting the coordinates received from the touchpanel driver from the maximum pixel coordinate values. In our case the display is 320x240 so X can reach a value between 0 and 319. The lv\_evdev driver converts the received x by setting x to 319-(old x). So the resolution of the display is hardcoded in the evdev.c file.

## Implemented objects

A full list of widgets supported by LVGL can be found at the homepage (documentation).

## pylvgl

### Introduction

pylvgl Is the LVGL library for Python. It consists of the following components:

* The Little VGL library itself
* The Little VGL driver for framebuffer and input device
* A C source file with support and helper functions
* A Python module to wrap the previous.

The first three compile and link to a shared object file. The last uses Python's ctypes module to import and link to the shared object file.

A pylvgl library already existed but was not used as it must be linked to the Python interpreter itself. In our case this must be cross compiled for the target system and to avoid problems with this the ctypes library solution was built.

### Deviations from LVGL

For the most part the user of pylvgl can refer to the original LVGL documentation, but at some points there are differences. This section discusses these differences and how to take care of them

#### Styles

A style is a C struct that contains all needed information for drawing widgets in a certain style (color, font, radius, etc). A couple of styles are predefined by LVGL and supplied as global variable. To avoid problems with structure alignment and sharing global variables the pylvgl library provides a "wrapper" around the C struct, treating it as an object. In C-code one can just declare the correct C struct, fill it and use it as style for various widgets. To achieve the same from Python, first a style object must be created using lvh\_style\_create(). Note the prefix lvh instead of lv. The additional h stands for 'helper' and indicates that the function is not implemented in LVGL itself but rather in the pylvgl wrapper library.

lvh\_style\_create() expects a single parameter that is a reference to another style, either a predefined one or one that was created earlier. If NULL is used as value the style will remain uninitialized and may contain random values. The function returns a reference to the newly created style.

To set a property of a style, in C simply the corresponding member of the struct is assigned a value. From Python, lvh\_style\_set\_attribute() is used instead. It expects 3 arguments: a reference to the style to manipulate, a character string describing the name of the property to change and a value to assign to the property. The name of the property is equal to its name in the C struct. Currently the following properties are supported:

|  |
| --- |
| "glass" |
| "body.empty" |
| "body.main\_color" |
| "body.main\_color.red" |
| "body.main\_color.green" |
| "body.main\_color.blue" |
| "body.grad\_color" |
| "body.grad\_color.red" |
| "body.grad\_color.green" |
| "body.grad\_color.blue" |
| "body.radius" |
| "body.opa" |
| "body.border.color" |
| "body.border.color.red" |
| "body.border.color.green" |
| "body.border.color.blue" |
| "body.border.width" |
| "body.border.part" |
| "body.border.opa" |
| "body.shadow.color" |
| "body.shadow.color.red" |
| "body.shadow.color.green" |
| "body.shadow.color.blue" |
| "body.shadow.width" |
| "body.shadow.type" |
| "body.padding.hor" |
| "body.padding.ver" |
| "body.padding.inner" |
| "text.font" |
| "text.color" |
| "text.color.red" |
| "text.color.green" |
| "text.color.blue" |
| "text.opa" |
| "text.letter\_space" |
| "text.line\_space" |
| "image.color" |
| "image.color.red" |
| "image.color.green" |
| "image.color.blue" |
| "image.opa" |
| "image.intense" |
| "line.color" |
| "line.color.red" |
| "line.color.green" |
| "line.color.blue" |
| "line.opa" |
| "line.width" |

The value parameter that lvh\_style\_set\_attribute() receives is always an unsigned long. Depending on the datatype of the member of the C struct the value is copied as either an unsigned long or an unsigned short (unsigned 32-bit or 16-bit integers).

Colors are stored as unsigned long values. The relation between the values and the corresponding colors depend on the number of colors and the number of bits per primary color supported by the display. In the originating application a display with 5:6:5 bits RGB values is used.

The function lvh\_mkcolor() can be used to convert a 3\*8 bits RGB value to a 5:6:5 bits RGB value. It expects 3 arguments: The level of red, the level of green and the level of blue. Each level is a value between 0 and 255.

The following "getter" functions are defined to obtain references to the predefined styles in the C environment:

|  |
| --- |
| lvh\_get\_style\_plain |
| lvh\_get\_style\_plain\_color |
| lvh\_get\_style\_pretty\_color |
| lvh\_get\_style\_pretty |
| lvh\_get\_style\_btn\_pr |
| lvh\_get\_style\_btn\_rel |
| lvh\_get\_style\_transp\_tight |

Each of these returns a reference to the corresponding global variable defined in LVGL. lvh\_style\_set\_attribute() can also be used to alter these predefined styles.

The function lvh\_print\_style() can be used to print the values of all the C struct members to stdout. It expects a reference to a style as argument.

#### Fonts

In LVGL a font is a struct containing all needed data for that font. A number of fonts are predefined and are available as global variables. Additional fonts can be added by converting a range of characters to glyphs using a true-type font. This data can be generated on the LVGL website and results in a C source file that must be linked to the LVGL application.

In Python fonts are specified as references to objects allocated by the C code. The function lvh\_style\_get\_font() returns a reference to the font and expects the name of that font as ASCIIZ character string. Currently the following fonts are included in pylvgl:

|  |
| --- |
| "lv\_font\_dejavu\_40" |
| "lv\_notosans\_regular\_10" |
| "lv\_notosans\_regular\_15" |
| "lv\_notosans\_regular\_25" |
| "lv\_notosans\_regular\_30" |
| "lv\_notosans\_regular\_35" |

#### Build-in symbols

TODO

#### lodepng

LVGL itself cannot read and interpret bitmap images, although it can display bitmap images. To have LVGL read bitmapped images at runtime the *lodepgn* library was added to the pylvgl library. To use this from Python only the *lodepng* library needs to be initialized using png\_decoder\_init(). Once an image is created its source can be set to a png file using LVGLs own lv\_img\_set\_src(). LVGL itself detects that it’s an PNG image and converts it to its internal format using the *lodepng* library.

#### Helpers

A lot of "simple" functions in LVGL are defined as "inline". This results in the problem that they are not accessible from Python. Therefore very simple 1:1 wrappers are defined in the pylvgl library. These functions are named exactly as their LVGL counterparts but have a prefix lvh\_ instead of lv\_. In the Python import library these functions are renamed to their original names, so to the user of the library won't notice this.

As Python probably won't run on embedded systems without OS the use of pylvgl is probably limited to the Linux environment only. Within the Linux environment probably the only display driver to be used is the framebuffer display driver. Therefore, instead of using lv\_disp\_drv\_init() and lv\_disp\_drv\_register() the Python code can suffice by invoking lvh\_init\_display\_driver(). The same is true for the pointing device: it will probably only ever use Linux' input system and therefore invoking lvh\_init\_pointing\_device() replaces lv\_indev\_drv\_init() and lv\_indev\_drv\_register().

#### Constants

Many constants are defined by #define or enum in LVGL's C code. These "constants" have been redefined with the same name in the pylvgl import library. Python does not support real constants, so these variables are only constant as long as they are not manipulated.

#### Callbacks

The LVGL C-code makes heavily use of callbacks. The ctypes interface does support callbacks, but it is somewhat complicated. Callbacks are used for example when a button is pressed.

To be able to implement callbacks the ctypes library declares its own callback to the LVGL code. This callback in turn invokes the Python callback function. To be able for ctypes to know what Python code to call and in what context, such a Python callback function needs to be declared in a particular way.

LVGL basically support 3 "types" of callbacks:

lv\_action\_t

lv\_btnm\_action\_t

lv\_task\_prototype

The lv\_action\_t has the following C prototype:

unsigned long func(unsigned long)

To define a corresponding Python function use the following syntax:

def callbackfunction(refobj):

return 0

To be able to pass this function as a callback to the LVGL library, it must first be converted to the correct prototype. These prototypes are predefined in the import library for the callback types used by LVGL. In this example, the callback function is passed as parameter to the registering function using its prototype:

lv\_register\_callback(lv\_action\_t(callbackfunction))

(lv\_register\_callback is not a real function but is only used as an example here)

In most cases LVGL uses the lv\_action\_t prototype. In case of a button matrix the callback is specified as lv\_btnm\_action\_t as this function receives an additional parameter with the text off the pressed button, so that the callback implementation can identify what button was pressed.

For tasks use lv\_task\_prototype. Tasks are callbacks that are periodically invoked by LVGL in a context where it is safe to access the existing LVGL objects.

# System installation and recovery options

## Introduction

This chapter discusses the process of automatic initial install and recovery from various situations. A single script is responsible for initializing virgin systems, upgrading and recovery from fatal faults.

### Starting points

Various reasons exists for installation or recovery of the system. First, most obvious reason is a brand new onion module that is used for the first time. This can be due to a first time boot of a brand new system or due to the onion module being replaced on a previously working system. Brand new modules lack the modified bootloader and have a default linux system configured. This system will run but lacks device drivers for various peripherals such as TFT display and touch screen. Also obviously the UM2 application is missing from this default image.

A similar situation exists if no OS image is installed in the internal flash or if the OS image is damaged. In general this situation can only exist if a user damaged or erased the flash contents in a previous session or when the flash contents are damaged by external causes, perhaps caused by ESD or lightning. Obviously the system won’t run in any way. If the flash contents where the bootloader is stored are damaged then basically the onion module needs to be replaced as there is no means to rewrite the bootloader into flash.

The system uses a SD card as root partition. If the SD card gets damaged for some reason or is replaced then obviously the system also won’t work anymore. The root partition is one of three partitions on the SD card, the other two being partition 2 that contains everything required to initialize the root partition and partition 3 that is used for swap.

Finally, although there are more efficient ways to accomplish the same, the recovery procedure can be used on a perfectly fine working system to upgrade all software, including the operating system.

### Goals

The primary goal is to have a single procedure to “recover” from every starting point discussed in the previous section. Easiest way to do so is probably to write a script that executes all required tasks to setup a system. The script is named “recoverybootFase1.sh”.

The first thing the script must be able to handle is to initialize the mass storage device with the required partitions. As mass storage device a SD card is used. As the SD card may be new, may be unpartitioned, contain the incorrect amount of partitions or contain partitions of the wrong size, the script must identify the situation and if needed remove any existing partitions, create new partitions and install filesystems on them.

After this step, or when the existing partitions do not contain the correct contents, the required system files should be stored on the various partitions. The main source for these system files is an inserted USB stick. When no USB stick is installed the script will attempt to download required data from the internet but this requires server infrastructure that is not necessarily available.

Once the mass storage device contains the required data it needs to be mounted as “upper” layer of the overlay filesystem. By default a partition of internal flash is used for this. To be able to use the mass storage device for this the system configuration must be adapted to support this. This mainly involves changing fstab. However, fstab is provided by uci on openwrt based systems, so uci must be used to configure the mass storage device as upper layer.

Once the correct data is available on the mass storage device and it is mounted as root partition the applications can be run. The applications must be automatically started at startup as there is no means for users to do this when only the TFT display and touchscreen are accessible.

Recreating partitions and mounting the SD card as root partition may require reboots.

In a real world situation the only interface to the user is the TFT display. For the user to have any idea of what is going on during the execution of the procedure that may take 5 minutes or more the script must update the TFT display to reflect the current status of the process.

## Running

The system uses an SD card that is inserted in the onion module as root filesystem and swap storage. The SD card is partitioned in three partitions. The first partition contains the root filesystem. The second partition is mounted read only and can be used to recover when data on the first partition gets damaged. It only contains factory provided files. The third partition contains swap memory.

Partition 2 needs a fixed size of 1.5GB. Partition 3 needs a fixed size of 0.5GB. The remaining storage on the SD card can be assigned to partition 1. The minimum required size for a SD card is 4GB, thus partition 1 is at least 2GB.

### Starting for the first time after module installation

As discussed in the previous section the install and recovery script can be started from various starting points. In the case of a brand new onion module the user needs to prepare an USB stick that contains a FAT filesystem that contains at least 2 files in the root directory: OMEGA2.BIN and a file named E2\_yyyymmdd.IMG where yyyymmdd is the date of generation of the image. To activate the installation process from USB stick the system must be switched on while holding the button in pressed state. This will cause the bootloader to not attempt to load a linux kernel image into memory but instead try to access OMEGA2.BIN on the USB stick. If this file is found it is written to internal flash. This may take a very long time (~13 minutes) and during this stage the TFT display remains black.

The OMEGA2.BIN file contains both a linux kernel image and a compressed root filesystem. Once its contents are written to flash the system will try to boot in the normal way. The kernel is loaded from internal flash into RAM. Next, the bootloader will jump to the kernel entry point that causes the kernel to boot. In this stage it uses the compressed root filesystem in internal flash as root filesystem.

The new kernel contains a driver for the TFT display and will show the splash screen about 3 seconds after the bootloader jumped to the kernel entry point.

As in this case the kernel was started for the first time the startup process will first mount a ramfs as root fs and start initializing the internal flash partition for the overlay. The compressed root filesystem that is part of the kernel image is always read only. The internal flash partition that is used for the overlay is used to store any modifications done once the overlay filesystem is mounted as root.

The compressed root filesystem image contains the recoverybootFase1.sh script that is started by /etc/rc.local. /etc/rc.local is a script that is executed by init basically after all other startup scripts have run.

### Starting with no OS or default OS installed

This is basically the same as starting with a brand new onion module, as discussed in the previous section.

### Starting with damaged or replaced SD card

The init process will mount the SD card as root partition, or more accurately, as overlay of the compressed root filesystem in internal flash. If something is wrong with the SD card, either it was replaced or damaged, the startup process won’t be able to mount the SD card as overlay. The system will fall back to using the read/write internal flash partition. This will cause the original /etc/rc.local to be executed, the one stored in the compressed readonly image. Again this script will run recoverybootFase1.sh.

### Upgrading

From the application menu it is possible to install upgrades. For upgrading only applications a separate script exists that is executed from the menu. When upgrading to a new kernel the sysupgrade utility is used to install a new kernel and compressed root filesystem image. Installing this new image causes the original /etc/rc.local script to be executed that executes recoverybootFase1.sh

## Implementation

This chapter discusses the implementation of the recoverybootFase1.sh script.

The scripts contains a “main” section that attempts to find the starting point for recovery actions by identifying what requirements are already met and what requirements need to be configured. To configure the various requirements the script contains various functions that will be discussed in more detail in the following sections.

### “main” section

The main section will first identify if an SD card is available. If not, the script can’t continue and will therefore show a message on the TFT display and then exit. To detect the availability of the SD card it checks the existence of /dev/mmcblk0.

Next, it will try to find the number of partitions on the SD card. If the number of partitions on the SD card doesn’t match the required amount of 3 the SD card will be repartitioned.

The number of existing partitions on the SD card is determined in two ways. First, it counts the number of partitions reported by fdisk. Next it counts the number of /dev/mmcblk0p\* files that exist. If either one is not 3 it will perform a “full init”.

If 3 partitions are found on the SD card, the script will continue to check if partition 1 of the SD card is mounted as overlay by checking for the word “overlay” in /etc/config/fstab. If this is not found the script will perform an “init\_partitioned”.

Checking the existence of the word “overlay” in fstab alone is not enough, so next the script checks if the uuid of partition 1 on the SD card appears in the current /etc/config/fstab. If this is not found the script will also perform an “init\_partitioned”.

Another, final, check is made to make sure that the overlay is not mounted on the internal flash. If it is then basically a “full init” is performed by first recreating the partitions and then invoke “init\_partitioned”.

Regardless if init\_partition was executed or not the script will check for the existence of a valid filesystem on partition 2 on the SD card. If this is not the case it will again invoke “init\_partitioned”.

Next the script will check for a valid filesystem on partition 1 on the SD card. If this does not exist the script will recreate the filesystem on partition 1 and reinitialize swap and then execute “init\_partitioned”.

If up to this point no problem was found the contents of /etc/rc.local are compared to /home/scripts/etc\_rc.local. If they differ /etc/rc.local is replaced by /home/scripts/etc\_rc.local and the system will reboot. The new contents of /etc/rc.local will start the application at the end of the initialization by init.

### “full\_init”

In short, the function full\_init in the recovery script will completely erase the contents of the SD card and provide it with the new contents from USB stick. The function is very short and consists of sequentially executing create\_partitions and init\_partitioned.

### create\_partitions

The end result of executing this function is that the SD card is correctly partitioned, filesystems are created on them, swap is activated and partition 2 is populated with the “factory contents”, everything required to run the applications.

To recreate the partition table first every existing partition on the SD card that is mounted is unmounted. This is done by using fdisk to scan for existing partitions on the SD card.

Next this function deletes all existing partitions by counting the number of partitions on the SD card and then generate a “script” to delete these. This “script” is then sent to fdisk using I/O redirection.

Depending on the situation fdisk may decide that a reboot is necessary to activate the new partition table. If it does decide so, it prints a message containing “next reboot”. The output of fdisk is redirected to a file on the ramdisk. The contents of this file are then examined to check for the existence of a line containing these words. If found the script will execute “reboot” and exit. (The reboot program simply signals init to shutdown and reboot. As this may take a while the script must exit after running reboot. )

Before creating the new partitions the script needs to calculate the size for partition1. The size of partition 1 is the size of the SD minus the 1.5GB for partition 2 and minus 0.5GB for swap.

Next the script will generate a script for fdisk to be executed to create the partitions. After that, ext4 filesystems are created on partition 1 and 2. If needed partition 3 is initialized as swap partition and then the swap is activated.

Finally the init\_p2 function is executed.

### init\_p2

The main purpose of this function is to populate partition 2 with the required data to be able to populate partition 1 with everything needed to run the application.

The first step is to mount partition 2 on its mountpoint /tmp/mnt/mmcblk0p2. If this mountpoint does not exist already the script will create it.

Next it checks if the USB mass storage device was mounted on its mountpoint. If not, the mountpoint is created if it did not exist already and the USB device is mounted there. If this fails the script outputs a message on the TFT display and continues. If it succeeded the script simply continues.

Next the script will try to obtain an image for the contents of partition 2. It searches the USB device for files with a name that matches E2\_yyyymmdd.img. If multiple are found the youngest is used.

The image file is, despite its name, a gzipped tar file. This file contains mainly the image for partition 2 and a kernel image.

If a file was found the script continues by unpacking it and copy a tar file with the contents intended for partition 2 of the SD card.

If no file was found on the USB stick (or no USB stick was available) the script will try to download an image file from a pre-declared location and extract p2.bin by unpacking it from the image and then copy it to partition 2 on SD card as p2.tgz.

Next, the script creates a /home directory on partition 2 of the SD card and unpacks the contents op p2.tgz in that directory.

### init\_partitioned

This function executes recover\_fstab and create\_overlay sequentially, in between copying raw bitmaps to the TFT framebuffer to update a “progress bar”. After this it executes reboot and exits.

### recover\_fstab

This function configures partition 1 of the SD card as overlay partition for the root filesystem and also configures “auto mount” for the USB stick. However, to prevent a mismatch between a previously configured partition 1 on the SD card and the newly generated configuration it first removes /etc/.extroot-uuid. As this function may be invoked from different places and in different stages the SD card may be mounted at either /tmp/mnt/mmcblk0p1 or at /mnt/mmcblk0p1 this function removes the .extroot-uuid file using both paths. One of them will fail obviously but that effect is simply ignored.

The system uses uci as single point of configuration and therefore fstab is also handled by uci. /etc/config/fstab is generated by ‘block detect’ and then modified to enable the overlay partition, not mount partition 1 but if its mounted as read-only, and auto-mount the USB device. Uci is used to change these configuration settings. After this /etc/init.d/fstab enable is executed to activate the configuration changes.

### create\_overlay

This function prepares partition 1 of the SD card to be used as overlay for the root filesystem. To refresh memory, remember that the root filesystem is mounted as overlay with the compressed root filesystem contents of the kernel image as lower layer and partition 1 of the SD card as upper layer. The system will first attempt to find the file on the upper layer. If it is not found there, it tries to find it on the lower filesystem.

In the original configuration a part of internal flash is used for the upper layer. During system setup before recoverybootFase1.sh is run all local configurations are stored in this upper layer. It is therefore important to copy the contents of this part to partition 1 of the SD card as well as otherwise these configuration files are missing once the SD card is mounted as overlay.

Create\_overlay takes the following steps: first is checks if a directory /overlay exists. If it does not, then the script is probably run on a system that already has partition 1 mounted as overlay. A directory /overlay is created if it did not already exist. Then /dev/mtdblock6, which is the part of internal flash that is used as upper layer when no SD card is available, is mounted on this directory. If the mount fails, /etc/config/fstab is deleted, partition 1 on the SD card is deleted using fdisk, and the overlay directory is unmounted and removed and the function returns value 1.

So the script continues here if the previous step was successful. In this case it creates the mount point for SD card partition 1 and mounts partition 1 to this mount point. Next it copies the contents of /overlay to partition 1 of the SD card.

Next it creates a mount point for partition 2 of the SD card (if it did not already exist) and mounts partition 2 on that mount point. If a directory /home does not exist on partition 2 the script invokes init\_p2. Otherwise it checks if the SD partition 1 contains a directory named /upper. If it does not the script returns from the function; otherwise it creates a home directory in the /upper directory and copies all files from the /home directory of partition 2 to this directory. Finally it copies the etc\_rc.local file in the home directory to /etc/rc.local, causing the application to start up at next boot.

### copy\_recovery\_to\_workpartition

This function unmounts SD card partition 1, creates a new filesystem on this partition and if needed create and activate swap.

## Utility

### text2fb

During execution of scripts, for example at startup, as a user it is useful to have some understanding what is going on to understand the amount of progress and why certain steps take a rather long time to execute. Writing text to the TFT display is not easy from a script, as the framebuffer is pixel oriented. This requires a font to convert the text into a bitmap that can be written to the framebuffer. For this a simple utility was written named text2fb.

text2fb is cross compiled for the target system and copied into the /bin directory. The program is to be used like this:

text2fb [x coordinate] [y coordinate] [foreground color] [background color] [text]

Where

[x coordinate] is the x-coordinate of the top left of the first character

[y coordinate] is the y-coordinate of the top left of the first character

[foreground color] is a numeric value for the color that is used to write pixels generated from the font

[background color] is a numeric value for the color that is used to write all pixels that are not part of the font.

[text] is a string that is shown on the TFT display

It is outside the scope of this section to explain the conversion from color to value. However, some easy ones: 0 = black, 65535 (= all 1s) = white. Experimenting with other values is one way to discover possible colors.

The used font was picked from the internet and is a simple 8x8 pixel font that is enlarged by a value of 2. This causes that only a rather limited amount of characters can be written on a single line. (320/16=20).

Usage example:

text2fb 0 0 0 65535 “The quick brown fox”

### Debugging

The script contains statements that are for debugging purposes only. When the script is run from /etc/rc.local the output is redirected to /home/scripts/boot.log. The option –x causes the bash interpreter to print the line it is executing on stdout. This information can also be found in /home/scripts/boot.log

The system may reboot a couple of times, depending on the starting point. This can make debugging hard, as the environment has changed after the reboot. So consider temporarily making the reboot statements comments.

# User interface application

Directory structure

Globals

The executor

The job baseclass

Error handling

Implementing a menu page

Standard classes

Threebuttonmenujob

Filebrowser

UMMessageBox

Controlling the backlight

## Introduction

The user interface is implemented as python program that runs as normal user process under linux. This chapter explains its philosophy, the architecture and implementation.

Earlier in the project the ‘everything is a job’ paradigm was suggested, where menu options and actions are specified in the UFP file. This lead to the idea of a framework, where a single python script can “execute” other scripts. These scripts then could be supplied as part of a UFP file, allowing enormous flexibility in defining jobs. However, more recently feedback was received that everything-in-a-single-file was preferred. Although the architecture was not radically changed by this information further development is not based on the “everything is a job” paradigm.

The user interface application basically consists of an executor and a job baseclass. Menus are implemented by inheriting from the job baseclass and customize it by overloading the required member functions.

The generic usable files such as the executor and the job baseclass as in /home/menu/bin. Application specific files that use the generic files are in /home/menu.

## Global settings

Every application needs settings. The same is true for the exector. Global settings in this case are implemented as a Python source file that is included in all programs. The file is named globals.py and is stored in /home/menu/bin. Currently the following settings are defined:

|  |  |  |
| --- | --- | --- |
| Setting name | Current value | Description |
| MOUNT\_PATH | /tmp/mnt/sda1 | Mountpoint for USB disk |
| UFP\_WDIR | /tmp/x | Directory where UFP files are processed. |
| SYMBOL\_\* |  | Utf8 sequences for corresponding symbol. |
| DIM\_SPEED | 1 | Number of seconds between each dim step of TFT backlight |
| MAINLOOP\_FAST\_CYCLE\_MS | 5 | Delay in mainloop iteration when lvgl is busy in milliseconds |
| MAINLOOP\_SLOW\_CYCLE\_MS | 2000 | Delay in mainloop iteration when lvgl is NOT busy in milliseconds |
| LAN\_INTERFACE | eth0 | Interface name for wired connection |
| WLAN\_INTERFACE | apcli0 | Interface name for wireless connection |
| CONNECT\_TYPE\_LAN | 0 | Constant |
| CONNECT\_TYPE\_WLAN | 1 | Constant |
| CONNECT\_TYPE\_UNKNOWN | 2 | Constant |
| GCODE | /home/3D/model.gcode | Filename of gcode file |
| THUMBNAIL | /tmp/x/Metadata/thumbnail.png |  |

# Abbreviations

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| I2S |  |  |
| I2C | Inter-IC communication |  |
| MSD | Mass storage device |  |
| SPI | Serial Peripheral Interface |  |
| UM | Ultimaker |  |
| USB | Universal Serial Bus |  |
| UX | User experience |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |



Betronic BV

Zekeringstraat 32-F, 5e verdieping,

1014 BS Amsterdam, The Netherlands

+31 20 30 38 500

# Git and submodules

Working with git is complex as it is, but using submodules in a project makes git even harder to use.

At some point during the project copies of the lvgl and lv\_drivers projects were added to the repository. At a later point it was decided that these projects should have been submodules of the e2-embedded project instead of copies. Changing this is rather complex and therefore the recipe used is documented here.

Simply recursively deleting the libs/lvgl and libs/lv\_drivers directories will not work. The problem is that git will consider this a mutation of the previous commit, resulting in the effect when the new commit is checked out git will try to re-delete the directories. Therefore git must be told to forget about these files completely. This can be done using the following commands:

git rm --cached libs/lv\_drivers -r

git rm --cached libs/lvgl –r

If attempts were made before to add these projects as submodules the following needs to be done. Simply open the file and delete all lines that refer to the components in question.

From .gitmodules remove lines

From .git/config remove lines

Also remove objects from the git repository:

rm -rf .git/modules/libs/lvgl

rm -rf .git/modules/libs/lv\_drivers

Now we can permanently remove all the copies:

rm libs/lv\_drivers -Rf

rm libs/lvgl/ -Rf

Now git does not remember anything of the lvgl and lv\_drivers copied projects so the modules can be added in the regular way:

cd libs

git submodule add <https://github.com/littlevgl/lvgl.git>

git submodule add <https://github.com/littlevgl/lv_drivers.git>

This will add the head commit of the master branch. To change to another commit, simply enter the directory and use git checkout to checkout the desired commit.

The exact steps taken to ‘fix’ the submodule problem from the re-add-submodules branch were:

git rm --cached libs/lv\_drivers -r

git rm --cached libs/lvgl -r

remove lines from .gitmodules

remove lines from .git/config (if needed)

rm -rf .git/modules/libs/lvgl

rm -rf .git/modules/libs/lv\_drivers

rm libs/lv\_drivers -Rf

rm libs/lvgl/ -Rf

cd libs

rm .gitignore

git submodule add <https://github.com/littlevgl/lvgl.git>

git submodule add <https://github.com/littlevgl/lv_drivers.git>

cd lv\_drivers

git checkout a688824dfea92e14001600767b60765950db0136

cd ../lvgl

git checkout 84781c62c2a8291aacdc7abd6ae54e64a2b6b72b

cd ../..

git add -u

git commit

Notes:

* The .gitignore file in the libs directory excluded the lvgl and lv\_drivers directories. As it did not mention any other files, removing the .gitignore file from this directory solves it.
* In this specific situation the modules were refered to in .gitmodules but NOT in .git/config. Therefore only the references in .gitmodules had to be deleted.
* Using version 6 of lvgl results in problems due to the fact that lvgl had changed significantly from version 5 to version 6. In particular, the callback mechanism is made more generic and transparent. However pylvgl was created before version 6 was available and therefore does not support version 6. This requires to change to the specific commit.

1. Software that runs in kernel mode has unlimited access to all resources. [↑](#footnote-ref-1)
2. Software that runs in user mode only has access to its private resources. [↑](#footnote-ref-2)
3. Only used to identify printer (in pool of multiple printers) [↑](#footnote-ref-3)