1. Významná rozhraní

1.1 Select smyčka

Pomocí následujících funkcí lze objednat čekání na nějakou událost, vždy je třeba specifikovat funkci, která se zavolá, až daná událost nastane, a pointer, který se jí předá:

- `void set_handlers(int fd, void (*read_handler)(void *), void (*write_handler)(void *), void (*error_handler)(void *), void *data)`
  Registruje handler na čtení, zápis a chybu příslušného file descriptoru. Až bude z `fd` možno číst, zavolá se funkce `read_handler`, až bude možno zapisovat, zavolá se `write_handler`, při chybě se zavolá `error_handler`. Funkce se zavolají s parametrem `data`. Funkce můžou být NULL - v tom případě se na danou událost přestane čekat.
- `int install_timer(ttime t, void (*fn)(void *), void *data)`
  `t` je doba v milisekundách. Za danou dobu zavolá funkci `fn` a předá jí parametr `data`. Funkce vrátí handle timeru, který je možno použít k předčasnému ukončení. Pokud vrátí -1, došlo k neúspěchu.
- `void kill_timer(int timer_handle)`
  Předčasně ukončí timer s příslušným handlem. Pokud handle není platný timer, vyvolá „INTERNAL ERROR“ a dumpne core. Proto, pokud nainstalujete timer a uložíte si handle, je potřeba ve funkci timeru handle zneplatnit, aby nemohlo dojít k zavolání `kill_timer()` s již vykonaným timerem.
- `void install_signal_handler(int signal, void (*fn)(void *), void *data, int immediate)`
  Čeká na signál. Při přijmutí signálu zavolá funkci `fn` s parametrem `data`. Pokud je `immediate` 0, funkce je zavolána až se kód vrátí do `select_loop`. Pokud je parametr `immediate` nastaven na 1, funkce je zavolána ihned — v takovém případě můžou být všechny struktury v nekonzistentním stavu, proto na ně není dobré sahat ani volat funkce, co na ně sahají (nesmí se třeba ani alokovat pamět). Když `fn` je NULL, znamená to odinstalace handleru.
- `ttime get_time()`
  Vrátí čas v milisekundách. Vrácená hodnota se plynule zvětšuje, ale nemůžeme z ní usuzovat, jaký je reálný čas. Funkce je použitelná pro měření, jak dlouho nějaká akce trvala.
- `int register_bottom_half(void (*)(void *), void *)`
  Způsobí, že daná funkce bude zavolána okamžitě, až se program vrátí do select smyčky. Jakmile byl bottom half registrován, nejde ho zrušit (pozor na to, když je objekt, který byl předán funkci jako parametr, mezitím odalokován). Když je registrována stejná funkce se stejným parametrem vícekrát, zavolá se pouze jednou. Bottom halfy se typicky používají pro odalokování ruzných struktur (jako třeba terminál).

  Podobného efektu lze docílit registrováním timeru s časem nula. Rozdíl je v tom, že když registrujeme funkci se stejným parametrem vícekrát, taky bude vícekrát zavolána. Timer jde taky předčasně zrušit, proto je ve většině případů lepší, než bottom halfy.

1.2 Object requester

Object requester poskytuje následující funkce pro vyžádání souboru:

- `void request_object(struct terminal *term, unsigned char *url, int pri, int cache, void (*upcall)(struct object_request *, void *),`

`void *data, struct object_request **rqp)`

Stáhne objekt ze sítě. `term` je terminál, na který se budou vypisovat otázky ohledně stahování. `url` je url, které se má stáhnout. `pri` je priorita (konstanta `PRI_xxx`). cache je úroveň, s jakou se má používat cache. Je definována makry `NC_xxx`. Typicky se používá `NC_CACHE`. `upcall` je funkce, která se zavolá s parametrem `data`. Upcall se volá periodicky, každou 0.1s až 1s po dobu, po kterou je spojení živé. Na místo `rqp` se uloží pointer na request. Pointer na request je platný, dokud se nezavolá `release_object`. V requestu se postupně objevují data.

- `void release_object(struct object_request **rqq)`
  Tato funkce uvolní object request. Pokud se ještě něco stahuje, tak se to zruší, případně odloží na pozadí, a cachová položka je odemčena, takže může být uvolněna. Zavoláním této funkce, se nelze na data objektu nijak dostat.
- `void clone_object(struct object_request *src, struct object_request **dst)`
  Vytvoří kopii requestu, `src` je zdroj, na místo `dst` se uloží kopie. Oba requesty obsahují stejná data a stejné spojení.

Zde je seznam jednotlivých priorit, priority jsou seřazeny od nejvyšší k nejnižší.

- `PRI_MAIN`
- `PRI_DOWNLOAD`
- `PRI_FRAME`
- `PRI_NEED_IMG`
- `PRI_IMG`
- `PRI_PRELOAD`
- `PRI_CANCEL`
- `N_PRI`

Priorita `PRI_CANCEL` znamená, že se spojení stahuje na pozadí, například v případě, kdy uživatel zobrazí stránku, stránka se ještě nestahne celá a uživatel jde na jinou stránku. Stránka se v tom případě stále stahuje na pozadí a když uživatel jde zpět, stránka se mu zobrazí (třeba) již stažená celá.

A zde je popis jednotlivých konstant cacheování:

- `NC_ALWAYS_CACHE`: vždy vrátit položku cache, i když je již vypršelá.
- `NC_CACHE`: vrátit položku z cache.
- `NC_IF_MOD`: poslat „if-modified-since“ požadavek.
- `NC_RELOAD`: nahrát znova.
- `NC_PR_NO_CACHE`: nahrát znova, ale poslat ještě „pragma:no-cache“.

Čtením `object_request->state` lze zjišťovat stav natahování:

1. **Živé spojení**
   - `O_WAITING`: ještě se nezačalo nic stahovat.
   - `O_LOADING`: už se stahuje, je stažena část dokumentu.
2. **Spojení se již ukončilo:**
   - `O_FAILED`: nepodařilo se nic stáhnout.

- `O_INCOMPLETE`: stáhla se část a pak spojení spadlo.
- `O_OK`: stáhl se celý soubor.

Přesnější popis chyby je možno získat z `object_request->stat.state`. To je některá z konstant `S_xxx`, která určuje přímo typ chyby, ke které došlo, nebo stav, ve kterém se spojení právě nachází. `object_request->ce` ukazuje na položku v cachi, kam se stahuje. Může být NULL, pokud se zatím nic nestáhlo. Tato položka je platná pouze ve stavech `O_LOADING`, `O_INCOMPLETE` a `O_OK`. V struct `cache_entry` je uložena HTTP hlavička a seznam fragmentu. Pomocí funkce `defrag_entry` lze položku kdykoli defragmentovat a fragmenty spojit do jednoho, nebo to lze nechat tak, a rozebírat jednotlivé fragmenty.

Zde je přehled jednotlivých chybových stavů. Čísla v závorkách říkají číselnou hodnotu makra. Kódování stavu je takové, že číslo nezáporné znamená, že se ještě přenášejí data, číslo záporné znamená, že se data již nepřenášejí a spojení je ukončeno (ve většině případů chybový stav).

- `S_WAIT (0)`: spojení čeká ve frontě.
- `S_DNS (1)`: čeká se na DNS lookup.
- `S_CONN (2)`: navazuje se spojení.
- `S_SSL_NEG (3)`: vyjednávání SSL protokolu.
- `S_SENT (4)`: byla vyslána žádost.
- `S_LOGIN (5)`: čeká se na FTP login.
- `S_GETH (6)`: v případě HTTP znamená, že se stahuje hlavička, v případě FTP, že se leze do daného adresáře.
- `S_PROC (7)`: server vrátil „HTTP 100“ — server zpracovává žádost.
- `S_TRANS (8)`: přenášejí se data.
- `S_WAIT_REDIR (-999)`: čekání na informace o redirectu.
- `S_OK (-1000)`: vše proběhlo v pořádku.
- `S_INTERRUPTED (-1001)`: spojení bylo přerušeno.
- `S_EXCEPT (-1002)`: exception na socketu.
- `S_INTERNAL (-1003)`: internal error.
- `S_OUT_OF_MEM (-1004)`: nedostatek paměti.
- `S_NO_DNS (-1005)`: DNS nenalezlo server.
- `S_CANT_WRITE (-1006)`: do socketu nelze zapisovat.
- `S_CANT_READ (-1007)`: ze socketu nelze číst.
- `S_MODIFIED (-1008)`: soubor byl modifikován, spojení se restartuje.
- `S_BAD_URL (-1009)`: špatné URL.
- `S_TIMEOUT (-1010)`: timeout na čtení, navázání spojení nebo DNS lookup.
- `S_RESTART (-1011)`: spojení se restartuje z důvodu, že server má chybně implementován protokol HTTP/1.1, nová žádost se pošle v HTTP/1.0
- `S_STATE (-1012)`: zfailovala funkce getsockopt na vrácení chyby.

U chybových stavů ještě mohou být standartní konstanty v rozsahu -1 až -998, které odpovídají systémovým chybám. Tedy například „connection refused“ je `-ECONNREFUSED`.

1.3 Grafika

1.3.1 Rozhraní grafických driverů

Aby program mohl pracovat s rozličnými grafickými zařízeními, bylo navrženo rozhraní pro grafické drivery. Každý grafický driver obsluhuje jeden grafický subsystém (SVGAlib, OS/2, X). Na grafickém driveru může být vytvořeno několik grafických zařízení. V okenních systémech je každé grafické zařízení jedno okno; v neokenních systémech jsou použita virtuální zařízení — třeba na SVGAlib je možno virtuální zařízení přepínat pomocí `Alt-1` — `Alt-0`. Za běhu programu může být aktivní pouze jeden driver, na který ukazuje proměnná `drv`. Každý driver je popsán strukturou struct graphics_driver, která má následující položky:

- `unsigned char *name:`
  Jméno driveru.
- `unsigned char *(*init_driver)(unsigned char *param)`
  Inicializuje driver. Vrátí NULL při úspěchu nebo alokovaný řetězec obsahující popis chyby při neúspěchu. Na začátku programu se může zavolat `init_driver` a na konci se pak musí `shutdown_driver`. Po `shutdown_driver` může následovat další cyklus. Při přechodu do dalšího cyklu musí být všechny devicy seshutdownované.
- `struct graphics_device *(*init_device)(void)`
  Vytvoří nové zařízení. Může se zavolat víckrat `init_device`, ale pak se pro každé device musí zavolat `shutdown_device` dříve, než se zavolá `shutdown_driver`.
- `void (*shutdown_device)(struct graphics_device *dev)`
  Ukončí zařízení.
- `void (*shutdown_driver)(void)`
  Ukončí činnost celého driveru. Při volání této funkce se předpokládá, že všechna zařízení jsou ukončena a v driveru nejsou registrovány žádné bitmapy ani barvy.
- `unsigned char *(*get_driver_param)(void)`
  Vrací poitner na řetězec s parametrem funkce `init_driver` nebo NULL (funkce `init_driver` by se měla příště zavolat s tímto parametrem, pokud uživatel neřekne jinak). Používá se k zapamatování grafického módu, velikosti okna a podobně.
- `int (*get_empty_bitmap)(struct bitmap *dest)`

  Před voláním této funkce jsou vyplněny položky `x` a `y` v `struct bitmap`. V ostatních jsou odpadky. Funkce alokuje místo na bitmapu, nastaví hodnoty `skip` a `data`, ponechá hodnotu `user`, může nastavit `flags`. Můžeme předpokládat, že po volání `get_empty_bitmap` bude zavoláno `register_bitmap`. Po `get_empty_bitmap` uživatel nesmí sahat na `x`, `y`, `skip`, `data`, `flags`. Do `user` může uživatel hrabat jak chce — je to jeho.

  Návratová hodnota je 0 pokud je bitmapa naalokovaná na heapu, 1 pokud je ve videopaměti a 2 pokud je v X serveru.
- `int (*get_filled_bitmap)(struct bitmap *dest, unsigned char *pattern, int n_bytes)`
  Před voláním této funkce jsou vyplněny položky `x` a `y` v `struct bitmap`. Obě musí být `>=0`. V ostatních jsou odpadky. Funkce alokuje místo na bitmapu, nastaví hodnoty `skip` a `data`, ponechá hodnotu `user`, může nastavit `flags`. Bitmapa bude od výroby už zaregistrovaná a bude vyplněna vzorkem `pattern`. `Patern` ukazuje na pole o velikosti `n_bytes`, bitmapa bude těmito byty vyplněna. Může se (snad) předpokládat, že `n_bytes` je stejná hodnota, jako nastaví grafický driver pro počet bitů v pixelu. Bitmapa bude už zaregistrová, takže dovolené operace na ní jsou: `prepare_strip`, `unregister_bitmap`, `draw_bitmap`, `draw_bitmaps`.

  Návratová hodnota je 0 pokud je bitmapa naalokovaná na heapu, 1 pokud je ve videopaměti a 2 pokud je v X serveru.

- `void (*register_bitmap)(struct bitmap *bmp)`

  Registruje vyplněnou bitmapu. Může (ale nemusí) přenést data bitmapy do video-ram a odalokovat je. Čili po `register_bitmap` je už pointer ve `struct bitmap` neplatný!

- `void *(*prepare_strip)(struct bitmap *bmp, int top, int lines)`

  Připraví bitmapu na zápis vodorovného pruhu co je přes celou šířku bitmapy. `bmp` musí být zaregistrovaná bitmapa. `top` je první řádek, který se bude měnit. `length` je počet měněných řádků, musí být `>=0`, jinak program spadne na segfault. Pruh nesmí být mimo bitmapu ani trčet nahoře ani dole z bitmapy jinak to má nárok spadnout na segfault. Pointer co tato funkce vrátí je pointer na který se má začít zapisovat data a při zápisu celého řádku se skočí o `bmp->skip`. Po zavolání `prepare_strip()` musí být zavolán právě jednou odpovídající `commit_strip()`.

- `void (*commit_strip)(struct bitmap *bmp, int top, int lines)`

  Commitne změny do bitmapy. `bmp` musí být připravena pomocí `prepare_strip()` a `top` a `lines` musí být stejné jako v `prepare_strip()` jinak to má nárok spadnout. Po commitnutí se už do bitmapy zase nesmí sahat.

- `void (*unregister_bitmap)(struct bitmap *bmp)`

  Uvolní bitmapu. Nesahá na `bmp->x` a `bmp->y`.

- `void (*draw_bitmap)(struct graphics_device *dev, struct bitmap *hndl, int x, int y)`
  Nakreslí bitmapu na dané zařízení na danou pozici.

- `void (*draw_bitmaps)(struct graphics_device *dev, struct bitmap ** hndls, int n, int x, int y)`

  Nakreslí několik bitmap za sebe. Všechny bitmapy musejí mít stejnou výšku. Funkce je tu proto, aby se snížil overhead při volání `draw_bitmap`.

- `long (*get_color)(int rgb)`

  Alokuje barvu. Parametr je ve tvaru `R*65536+G*256+B`, kde `R, G, B` jsou čísla 0 až 255. Číslo 0 reprezentuje 0 (žádné elektrony do monitoru), číslo 255 reprezentuje maximum elektronů do monitoru, co se dá vytřískat z videokarty. Vrátí handle barvy. Handle je specifický pro daný driver. Handle může být předáván funkcím pro kreslení čar a plnění nebo uvolněn pomocí následující funkce. Každý handle musí být uvolněn před ukončením driveru pomocí `free_color`.

- `void (*free_color)(long color)`
  Uvolní barvu.

- `void(*fill_area)(struct graphics_device*dev, int x1, int y1, int x2, int y2, long color)`

  Vyplní daný obdélník danou barvou. Budou vyplněny všechny pixely o souřadnicích `x,y`, které leží uvnitř ořezávací oblasti a splňují podmínku: `(x>=x1) && (x<x2) && (y>=y1) && (y<y2)`. Tedy v případě, že `x1<x2` a `y1<y2`, `x1,y1` v obdélníku bude a `x2,y2` tam už nebude.

- `void (*draw_hline)(struct graphics_device *dev, int left, int y, int right, long color)`

  Nakreslí horizontální čáru. Bod `left,y` na ní leží, bod `right,y` na ní neleží. Pokud `left<=right`, pak je čára prázdná.

- `void (*draw_vline)(struct graphics_device *dev, int x, int top, int bottom, long color)`

  Nakreslí vertikální čáru. Bod `x,top` na ní leží, bod `x,bottom` na ní neleží. Pokud je `top>=bottom`, pak je čára prázdná.

- `int (*hscroll)(struct graphics_device *dev, struct rect_set **set, int sc)`
  Scroll. Posune aktuální ořezávanou oblast o `sc` pixelu doprava (eventuálně doleva, pokud je `sc` záporné). Oblast odkryta scrollováním je nedefinovaná. Návratová hodnota:

  - 0 — program nemusí překreslovat odkrytou oblast, bude zavolána funkce `redraw` (viz níže).
  - 1 — program by měl překreslit odkrytou oblast.

  Typicky by se hodnota 1 měla vracet na ne-okenních systémech (svgalib, framebuffer, dos), kde máme jistotu, že po překreslení odkrytého obdélníku bude obrazovka konzistentní. Na druhou stranu v okenních systémech (X, OS/2) může být okno, které scrollujeme, překryto jiným oknem a program neví, které části má překreslit. Musí mu být tedy grafickým driverem zaslán požadavek redraw.

  Pokud set není rovna NULL, obsahuje obdélníky, které je potřeba překreslit (oblast, která byla přikrytá jiným oknem). Program musí tuto oblast překreslit sám, grafický driver to neumí.

  Kdyby nebylo jasno, co je posunout oblast doprava, tak to je načíst klipovací obdélník, nakreslit ho na místo které je více vpravo než kde obdélník původně byl. A při kreslení se samozřejmě klipuje na nastavenou oblast, takže část dat se při tom kreslení zahodí. A část plochy zůstane původní pro libovolný obsah obdélníku. Toto je odkrytá oblast.
- `int (*vscroll)(struct graphics_device *dev, struct rect_set **set, int sc)`
  Totéž jako `hscroll`, ale scrolluje oblast nahoru (když je sc záporné) nebo dolů.
- `void (*set_clip_area)(struct graphics_device *dev, struct rect *r)`
  Nastaví oblast pro ořezávání. Veškeré funkce budou pracovat pouze na této oblasti. `struct rect` má položky `x1, x2, y1, y2`, což jsou souřadnice oblasti. `x1,y1` patří do `clip_area`, `x2,y2` tam již nepatří.
- `int (*block)(struct graphics_device *dev)`
  Vrátí původní textový videomód, vrátí handlování myši a klávesnice a zajistí, že kreslící funkce nebudou už nic kreslit (to jde zajistit celkem jednoduše — je už napsané makro `TEST_INACTIVITY` a existuje proměnná `current_virtual_device`).

  Pokud se zavolá `block` a driver je už zablokovaný (dříve již bylo zavoláno `block` a ještě nebylo zavoláno `unblock`), tak se nic neudělá a vrátí se 1. Jinak se vrátí 0.

  Pro drivery, kde není potřeba blokovat terminál při pouštění externích programů (X, Pmshell) tahle funkce vrací 0 a nedělá nic.
- `void (*unblock)(struct graphics_device *dev)`
  Obnoví zpět grafický mód, klávesnici a myš, a nakonec zavolá `redraw_handler` — čili překreslení celé obrazovky. Musí se brát v potaz, že `graphics_device`, na kterém se ten externí program pustil (a tedy to, co funkce dostane jako parametr), může být jiný, než aktuální `graphics_device`. Takže je lepší překreslovat `current_virtual_device`, pokud je nenulový.
- `int depth`
  Barevná hloubka:
  bity 0–2 — počet bytů na pixel. Pixely chodí do bitmapového zobrazovače tak, že každý pixel okupuje celý počet bytů, a žádný byte není okupován dvěma pixely. Proto např. 1 bit na pixel se nekóduje jako 8 pixelů na byte, ale každý byte obsahuje jeden pixel a tudíž 7 bitů v byte je pak nevyužitých. V případě, že v obrazové paměti jsou pixely v bytech nějak úchylně nasekány, musí je zobrazovač překódovávat.

bity 3–7 — počet bitů na pixel — 4, 8, 15, 16, 24

bit 8 — misordered — to je flag ve SVGAlib, který říká, že karta místo layoutu `RGBRGBRGB` má `BGRBGRBGR`, a některé ostatní layouty mohou být také podobně otočeny. Interní informace grafického driveru, kterou používá pro správnou konstrukci barevných dat z R, G, B.

bit 9 — misordered2 — pro 4 byty na pixel jsou 3 různé možné paměťové organizace. Tento bit je zde proto, že jen bit 8 by to nemohl rozlišit.

Zde je definice paměťové organizace pro jednotlivé podporované `depth`. `depth`, které nejsou v tabulce, jsou nepodoporované. Všechny `depth` v tabulce jsou podporované `dither.c`. V tabulce pokud například červená má barevnou hloubku 5, pak R0 je nejméně významný bit a R4 nejvíce významný bit červeného kanálu.

Bity v bytu jsou číslovány: 0 má váhu 1, 7 má váhu 128.

depth		**Barevná**			**Paměťová organisace: offset bytu v paměti**																											
		hloubka			**+0**								**+1**								**+2**								**+3**							
dec.	**hex.**	**R**	**G**	**B**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
33	0x21	1	2	1					R0	G1	G0	B0																								
65	0x41	3	3	2	R2	R1	R0	G2	G1	G0	B1	B0																								
122	0x7A	5	5	5	G2	G1	G0	B4	B3	B2	B1	B0		R4	R3	R2	R1	R0	G4	G3																
130	0x82	5	6	5	G2	G1	G0	B4	B3	B2	B1	B0	R4	R3	R2	R1	R0	G5	G4	G3																
195	0xC3	8	8	8	B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0								
196	0xC4	8	8	8	B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	0	0	0	0	0	0	0	0
451	0x1C3	8	8	8	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0								
452	0x1C4	8	8	8	0	0	0	0	0	0	0	0	B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0
708	0x2C4	8	8	8	0	0	0	0	0	0	0	0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

Tabulka možných bitových hloubek

- `int x, y`
  Velikost obrazovky — pouze pro drivery, které používají virtuální zařízení. Protože se u okenního systému může stát, že každé okno bude jinak veliké, neboť uživatel může okénka resizovat.
- `flags`
  Flagy, které nastaví grafický driver v `init_driver`. Vzniknou zORováním některých z následujících konstant:
  - `GD_DONT_USE_SCROLL` — scroll je pomalý a vyplatí se volat kompletní překreslení místo scrollu. Program by tedy pokud možno neměl používat scroll a měl by překreslovat celou situaci.
  - `GD_NEED_CODEPAGE` — kódování klávesnice nemůže být zjištěno ze systému, kódování se tedy bere podle proměnné `codepage`, kterou uživatel může nastavit v menu.
- `codepage`
  Kódová stránka klávesnice.

`struct bitmap` slouží k reprezentaci bitmap v grafickém rozhraní, obsahuje tyto položky:

- `int x,y`
  Rozměry bitmapy.
- `int skip`
  Vzdálenost v bytech mezi začátky dvou pixelů v bitmapě.
- `void *data`
  Pointer na data bitmapy, na místě, kam ukazuje `data` začíná pixel v levém horním rohu bitmapy.
- `void *user`
  Místo pro data uživatele — vyšší vrstvy nad grafickým driverem. Grafický driver na tuto hodnotu nesmí sahat.
- `void *flags`
  Alokační flagy pro grafický driver.

Bitmapa se smí předávat **POUZE** grafickému driveru, od kterého byla obdržena. Je zakázáno získat bitmapu například od svga driveru a předávat ji X-ovému driveru. Při registrování bitmapy není možné dostat error.

1.3.2 Grafické zařízení

Struktura `struct graphics_device` má následující položky:

- `struct rect size`
  Velikost zařízení. `size.x1 == 0`, `size.y1 == 0`, `size.x2` a `size.y2` obsahují aktuální velikost okna.
- `struct rect clip`
  Aktuální ořezávací oblast. Program může tuto položku číst, ale nesmí tam zapisovat. Změna ořezávací oblasti se dělá přes `set_clip_area`.
- `struct graphics_driver *drv`
  Driver, ke kterému zařízení náleží.
- `void *driver_data`
  Data soukromá pro grafický driver. Program by na ně neměl sahat.
- `void *user_data`
  Data soukromá pro program. Grafický driver by na to neměl sahat.

- `void (*redraw_handler)(struct graphics_device *dev, struct rect *r)`
  Sem si program uloží pointer na funkci, kterou driver zavolá, když bude potřeba překreslit část obrazovky. `redraw` se nevolá při inicializaci okna. Volá se při přepnutí grafické konzole, přesněji ve funkci `unblock` (viz výše).
- `void (*resize_handler)(struct graphics_device *dev)`
  Sem program uloží funkci, kterou driver zavolá, když je změněna velikost zařízení.
- `void (*keyboard_handler)(struct graphics_device *dev, int key, int flags)`
  Sem program uloží funkci, kterou driver zavolá, když je zmáčknuta klávesa. Parametr key je konstanta typu `KBD_xxx` nebo **UNICODE** kód klávesy. `KBD_CTRL_C` se posílá při stisknutí ctrl+c v programu, ne však při zavření okna v okenním systému (windowmanagerem). Při `KBD_CTRL_C` se může objevit dialog, jestli chce uživatel opravdu končit. `KBD_CLOSE` je poslán při požadavku o zavření okna (z windowmanageru), program se ukončí bez dalšího ptaní lusera. `flags` obsahuje zORované následující konstanty:
  - `KBD_SHIFT`
  - `KBD_CTRL`
  - `KBD_ALT`

  `KBD_SHIFT` se posilá pouze pro speciální klávesy (enter, šipky ...), ne pro písmena nebo ascii znaky.
- `void (*mouse_handler)(struct graphics_device *dev, int x, int y, int buttons)`
  Sem program uloží funkci, kterou driver zavolá, když je pohnuto s myší. `buttons` obsahuje zakódovaná tlačítka. Tlačítka se kódují ORováním jedné konstanty z každé následující skupiny:
  Tlačítka:
  - `B_LEFT` — levé tlačítko.
  - `B_MIDDLE` — prostřední tlačítko.
  - `B_RIGHT` — pravé tlačítko.
  - `B_WHEELUP` — kolečko nahoru.
  - `B_WHEELDOWN` — kolečko dolů.
  - `B_WHEELUP1` — posunutí kolečkem o 1 řádek nahoru.
  - `B_WHEELDOWN1` — posunutí kolečkem o 1 řádek dolů.
  - `B_WHEELRIGHT` — kolečko doprava (2. kolečko na myši).
  - `B_WHEELLEFT` — kolečko doleva (2. kolečko na myši).
  - `B_WHEELRIGHT1` — posunutí 2. kolečkem doprava o 1 znak.
  - `B_WHEELLEFT1` — posunutí 2. kolečkem doleva o 1 znak.

  U kolečka se posílá vždy akce `B_MOVE`. `B_WHEELUP1` a `B_WHEELDOWN1` znamená posunutí o 1 řádek (16 pixelů), používá se na OS/2, kde lze nastavit počet eventů na otočení kolečka. Podobně to platí pro `B_WHEELRIGHT1` a `B_WHEELLEFT1`. `B_WHEELUP` a `B_WHEELDOWN` znamená scroll kolečkem o více řádků (64 pixelů), používá se v X a SVGAlib.
  Akce:
  - `B_DOWN` — tlačítko bylo zmáčknuto.
  - `B_UP` — tlačítko bylo puštěno.

- `B_DRAG` — tlačítko je zmáčklé a pohla se myš.
- `B_MOVE` — myš se pohybuje (v tomto případě se kód tlačítka ignoruje).

Pro přístup k jednotlivým skupinám slouží masky `BM_BUTT` a `BM_ACT`. Jestliže je zmáčknuto více tlačítek, posílají se sekvenčně v pořadí v jakém přišly driveru eventy, v případě `B_DRAG` platí vždy poslední zmáčklé tlačítko.

Poslední 4 funkce volá driver, když došlo k nějaké události. Aby se předešlo race-conditionům, musí být funkce volány z bezpečného místa - t.j. z handlerů vytvořených v modulu `select.c`. Grafický driver nesmí tyto funkce volat ze svých funkcí volaných z programu. Například, když program zavolá `hscroll`, grafický driver nesmí volat `redraw` rovnou z kontextu `hscroll`, ale musí si na to registrovat buď bottom half nebo timer s časem 0. Kdyby to volal rovnou, tak by v programu vzniklo rekurzivní volání, a to by nevedlo k ničemu dobrému.

Speciální (neznakové klávesy) jsou reprezentovány těmito konstantami:

- `KBD_ENTER` — enter.
- `KBD_BS` — backspace.
- `KBD_TAB` — tabulátor.
- `KBD_ESC` — escape.
- `KBD_LEFT`, `KBD_RIGHT`, `KBD_UP`, `KBD_DOWN` — kurzorové šipky.
- `KBD_INS`, `KBD_DEL` — insert a delete.
- `KBD_HOME`, `KBD_END` — home, end.
- `KBD_PAGE_UP`, `KBD_PAGE_DOWN` — stránka nahoru, stránka dolů
- `KBD_F1`–`KBD_F12` — funkční klávesy.
- `KBD_CTRL_C` — stisknutí ctrl+c.
- `KBD_CLOSE` — zavření okna windowmanagerem.

1.3.3 Rozhraní virtuálních deviců

Virtuální devicy jsou metoda, jak na jednu obrazovku fysickou umístit několik obrazovek virtuálních (oken prohlížeče). Obrazovky se přepínají pomocí klávesnice — klávesami `Alt-1` až `Alt-0`. Používají se u neokenních systémů (svgalib a framebuffer), protože uživatel typicky chce zobrazovat více stránek současně — v každém okně jednu. Tvůrce grafického driveru nemusí virtuální devicy používat, ale pak si bude muset funkce pro správu deviců napsat sám.

Pro práci s virtuálními devicy slouží tyto funkce:

- `int init_virtual_devices(struct graphics_driver *drv, int n)`
  Zavolá se typicky z `init_driver`. Naalokuje `n` virtuálních zařízení. Návratová hodnota: 0 znamená OK.
- `void shutdown_virtual_devices()`
  Zavolá se z `shutdown_driver`. Odalokuje paměť pro virtuálni devicy.
- `struct graphics_device *init_virtual_device()`
  Dá se jako funkce `init_device` do struktury `graphics_driver`.
- `void shutdown_virtual_device(struct graphics_device *dev)`
  Dá se jako funkce `shutdown_device` do struktury `graphics_driver`.
- `void switch_virtual_device(int i)`
  Přepne virtuální device na device s číslem `i`. Nastaví `current_virtual_device` a pak tomuto devicu pošle redraw. Typicky se tato funkce volá z handleru klávesnice

grafického driveru. Pokud driver zjistí, že stisknutá klávesa má přepnout virtual device, klávesu nepředává dál programu, ale zavolá `switch_virtual_device`.

- `struct graphics_device *current_virtual_device`
  Proměnná ukazuje na virtual device, který je aktuálně zvolený (může být i NULL). V každé funkci grafického driveru, která něco kreslí, by se mělo zjistit, zda device, na který se má kreslit == `current_virtual_device`, a pouze v takovém případě kreslení provést. Jinak se budou do obrazu prolínat kusy grafiky z neviditelných deviců, což může, ale nemusí působit esteticky.

1.3.4 Obdélníky

Pro reprezentaci obdélníků se používá `struct rect`. Obsahuje souřadnice `x1, x2, y1, y2`, kde `x1<x2` a `y1<y2`. Obdélníky se předávají funkcím na vyplnění plochy a na nastavení ořezávací oblasti.

Pro reprezentaci množiny obdélníků slouží `struct rect_set`. Tato struktura reprezentuje sjednocení všech jejích obdélníků. S ní se pracuje pomocí těchto funkcí:

- `struct rect_set *init_rect_set(void)`
  Naalokuje a nainicialisuje strukturu, vrátí na ni ukazatel. Při chybě vrací NULL.
- `void add_to_rect_set(struct rect_set **set, struct rect *rect)`
  Přidá do množiny obdélník `rect` (sjednotí obdélník s obdélníky v množině).
- `void exclude_rect_from_set(struct rect_set **set, struct rect *rect)`
  Vyjme obdélník `rect` z množiny `set` (od množiny odečte obdélník).
- `static inline void exclude_from_set(struct rect_set **s, int x1, int y1, int x2, int y2)`
  Jako `exclude_rect_from_set()`, ale obdélník se zadává přímo pomocí souřadnic.

Dále existují tyto pomocné funkce pro práci s obdélníky:

- `int do_rects_intersect(struct rect *r1, struct rect *r2)`
  Vrátí 1, pokud obdélníky `r1` a `r2` mají neprázdný průnik, jinak vrátí 0.
- `int is_rect_valid(struct rect *rect)`
  Otestuje, zda se jedná o platný obdélník, pokud ano, vrátí 1, jinak vrací 0.
- `void intersect_rect(struct rect*out, struct rect*r1, struct rect*r2)`

  Spočítá průnik dvou obdélníků `r1` a `r2`, výsledek uloží do `out`. Tato funkce se nesmí volat na obdélníky, které mají prázdný průnik, v takovém případě je v `out` nesmysl.
- `void unite_rect(struct rect *out, struct rect *r1, struct rect *r2)`
  Spočítá nejmenší obdélník, do kterého se vejde sjednocení obdélníků `r1` a `r2`, a uloží ho do `out`.

1.4 Rozhraní javascriptu

1.4.1 Typy objektů

Rozhraní javascriptu zná tyto typy objektů:

- `JS_OBJ_T_DOCUMENT`: Dokument.
- `JS_OBJ_T_FRAME`: Rám.
- `JS_OBJ_T_LINK`: Odkaz.
- `JS_OBJ_T_FORM`: Formulář.
- `JS_OBJ_T_ANCHOR`: Kotva na stránce.

- `JS_OBJ_T_IMAGE`: Obrázek.
- `JS_OBJ_T_TEXT`: Textový řádek ve formuláři.
- `JS_OBJ_T_PASSWORD`: Řádek pro zadávání hesla ve formuláři.
- `JS_OBJ_T_TEXTAREA`: Textová plocha ve formuláři.
- `JS_OBJ_T_CHECKBOX`: Zaškrtávací políčko ve formuláři.
- `JS_OBJ_T_RADIO`: Radio tlačítko ve formuláři.
- `JS_OBJ_T_SELECT`: Vybírací políčko ve formuláři.
- `JS_OBJ_T_SUBMIT`: Odesílací tlačítko formuláře.
- `JS_OBJ_T_RESET`: Resetovací tlačítko formuláře.
- `JS_OBJ_T_HIDDEN`: Skrytá položka formuláře.
- `JS_OBJ_T_BUTTON`: Tlačítko ve formuláři.

Všechny objekty jsou jednoznačně identifikovány v rámci dokumentu. K identifikaci tedy stačí dvojice: ID dokumentu, ID objektu.

Typ objektu je uložen ve spodních `JS_OBJ_MASK_SIZE` bitech identifikace a získá se zANDováním id s maskou `JS_OBJ_MASK`, nebo zavoláním funkce `jsint_object_type()`, která tuto operaci provede.

1.4.2 Pomocné funkce

Rozhraní jsint obsahuje tyto pomocné funkce, které usnadní programování upcallů:

- `struct f_data_c *jsint_find_document(long doc_id)`
  Najde dokument s identifikátorem `doc_id`. Když dokument neexistuje, vrátí NULL.
- `void *jsint_find_object(struct f_data_c *document, long obj_id)`
  Najde v rámci dokumentu `document` objekt s identifikací `obj_id` a vrátí na něj ukazatel. Pokud objekt neexistuje, vrátí NULL. Volající musí vědět o jaký typ objektu jde a void pointer interpretovat správně.
- `int jsint_can_access(struct f_data_c *first, struct f_data_c *second)`
  Otestuje přístupová práva, jestli skript běžící ve `first` může přistupovat k dokumentu `second`. Vrací: 1=může, 0=nemůže. **Přístupová práva se musí otestovat v každém upcallu, který sahá na „cizí“ objekty/dokumenty!**
- `int jsint_object_type(long id)`
  Jak již bylo řečeno, identifikace objektu se skládá z typu a vlastní identifikace. Tato funkce dostane identifikátor objektu a vrátí typ objektu.

1.5 Obrázky

1.5.1 Přiblížení principu rozhraní

Obrázky jsou v `img.c` a `imgcache.c`, `gif.c`, `xbm.c`, `png.c`, `jpeg.c` a `tiff.c`. Rozhraní je následující:

- **Funkce:** Rozhraní sestává z následujících funkcí: `insert_image`, `img_draw_image`, `img_change_image`, `img_destruct_image`. Všechny funkce krom `change_image` se volají ze sazeče HTML který je volá jak je mu libo. `change_image` je volána javascriptem (upcallem `js_upcall_set_image_src`, když javascript mění zdrojové URL obrázku. `insert_image` vytvoří objekt `g_object_image` a vloží ho do vysázeného dokumentu. `destruct_image` to `g_object_image` zničí. `destruct_image` se nesmí volat víckrát na ten samý objekt. `destruct_image` se smí volat jen na objekt vytvořený pomocí `insert_image`. `draw_image` má za argument `g_object_image`

vzniklý z `insert_image`, na který nebyl zavolán `destruct_image`. Nakreslí obrázek. `change_image` změní grafická data v `g_object_image`, nesmí se volat na `g_object_image`, na který byl zavolán `destruct_image`.

- **Struktury:** `struct g_object_image`, `cached_image` `g_object_image` je rozšířením `struct g_object`. `g_object image` má položky `id`, `name`, `border`, `src`, které může číst Javascript. Položky které nejsou ani v `g_object` ani nejsou pro Javascript jsou pouze pro interní potřebu obrázků. `cached_image` je struktura zcela interní pro obrázky.
- **Globální proměnné:** žádné.
- **Funkce používané obrázky:** funkce pro všeobecné použití.

1.5.2 Přesný popis rozhraní

- `struct g_object_image *insert_image(struct g_part *p, struct image_description *im)`
  Dostane `g_part` a `image_description`. Její povinností je vyrobit `struct g_object_image` a vyplnit do položky `mouse_event` `g_text_mouse`, do draw `img_draw_image`, do `destruct` `img_destruct_image`, do `get_list` NULL. `link_num` a `link_order` musí zkopírovat z `image_description`. `alt`, `name` a `orig_src` musí zkopírovat z `g_object_image` (`orig_src` kopíruje z proměnné `src` v `g_object_image`). Do `map` musí dát NULL. Do `xw` a `yw` rozměr obrázku (takový jaký plyne z obrázku samotného, určení v HTML kódu nebo spojením obou těchto údajů, případně, pokud něco z toho není známo, může být rozměr jakýkoliv si `insert_image` dočasně usmyslí) patřičně zvětšený podle momentálního nastavení uživatelem. `xw` i `yw` musí být ≥ 0. `id, border, vspace` a `hspace` musí opsat ze struktury `image_description` (tyto údaje jsou pro Javascript). Musí vždy provést

  ```
  image->name=stracpy(im->name);
  image->src=stracpy(im->url);
  ```

  Pokud je `insert_flag` v `image_description` nenulový musí ještě vložit obrázek do seznamu všech obrázků ve `f_data`:

  ```
  add_to_list(current_f_data->images,&image->image_list);
  ```

  Jinak musí provést:

  ```
  image->image_list.prev=NULL;
  image->image_list.next=NULL;
  ```

  `insert_image` může zavolat `af=request_additional_file()` pro stažení obrázku ze sítě. Pokud se dá zjistit rozměr obrázku z již stažených dat ze sítě, musí ho zjistit a podle toho nastavit `xw` a `yw`. Pokud nedá (a hrozí že se později rozměr změní), musí nastavit `af->need_reparse` na 1. `need_reparse` se smí jen nastavit na 1 a nesmí se nulovat. V tom případě se při zavolání `img_draw_image` obrázek nemusí nakreslit korektně - může být velký a uřízlý nebo malý a doplněný pozadím. Stejně se to má chovat, když se obrázek reloadne a bude mít pak jiné rozměry. `img_draw` nesmí měnit `xw` a `yw`. `xw` a `yw` nastavuje `insert_image` na začátku při vytvoření struct `g_object_image`.
- `void img_draw_decoded_image(struct graphics_device *dev, struct decoded_image *img, int x, int y, int xw, int yw, int xo, int yo)`
  Dostane struct `g_object_image` a souřadnice a musí zobrazit obrázek na dané souřadnice, přičemž musí právě počmárat obdélník s velikostí `xw` a `yw`. Nesmí ani přetahovat ani nechávat prázdná místa. Když je obrázek už stažen a přesto nejsou známy jeho oba rozměry, pak se musí nakreslit rozbitý rámeček s rozměry, jako kdyby místo obrázku se stahla ikonka s lebkou. Když ještě obrázek není stažen a

nejsou známy jeho rozměry, tak se musí zobrazit to samé jako s lebkou, jen místo lebky je tam ta červená ikona. Když je obrázek stažen a jsou známy jeho rozměry tak se zobrazí obrázek a tam kde to ještě není dotažené tak pozadí.

Jestli se stáhlo něco nového tak se to musí zpracovat před nakreslením obrázku.

- `void change_image (struct g_object_image *goi, unsigned char *url, unsigned char *src, struct f_data *fdata)`
  Dostane pointer na `g_object_image`, string s URL, string se src atributem a pointer na `f_data`. Funkce zajistí nahrazení grafických dat v `g_object_image` daty z obrázku na daném URL. Jestliže obrázek s daným URL bude již v cachi, použije obrázek z cache. Jestliže dané URL bude stejné jako URL v `g_object_image`, zahájí se stahování obrázku ze sítě bez ohledu na cache (to je z důvodu, aby se dal měnit obrázek, když se změní a jméno zůstane).

  Funkce nebude nikterak modifikovat rozměry `g_object_image`. Pokud obrázek na daném URL bude menší, doplní se v `g_object_image` pozadím. Pokud bude větší, v `g_object_image` bude se zobrazovat výřez příslušné velikosti od levého horního rohu. Funkce bude volána ze select smyčky. Z funkce je nutno zavolat `request_additional_file` a `refresh_image`, které zajistí, že se obrázek bude automaticky periodicky překreslovat během natahování.

  Funkce nahradí `orig_src` v `g_object_image` argumentem `src`.

- `void img_destruct_cached_image(struct cached_image *img)`
  Musí kromě vlastních operací obrázků vždy zavolat:

```
if (goi->name)mem_free(goi->name);
if (goi->src)mem_free(goi->src);
if (goi->alt)mem_free(goi->alt);
if (goi->name)mem_free(goi->name);
if (goi->orig_src)mem_free(goi->orig_src);
release_image_map(goi->map);
del_from_list(&goi->image_list);
```

  a musí zcela zlikvidovat strukturu `struct g_object_image *goi` a poté zavolat `mem_free(goi)`.

1.5.3 Popis struct cached_image

Struct `cached_image` může být v rozličných roztodivných stavech. Hlavní stavová proměnná je `state`. Proměnná state může mít hodnotu 0, 1, 2, 3, 4, 5, 6, 7, 12, 13, 14, nebo 15. Představuje stavy ve kterých se nacacheovaný obrázek nachází. Na následujících stránkách jsou tabulky s významem jednotlivých stavů.

Stav	Ví se typ souboru?	Hlavička obrázku (určující width a height) byla přečtena	wanted_xw i wanted_yw určeno obojí současně	Obrazový proud skončil	img_draw_image nakreslí a uživatel uvidí	bitmapa	Obrazová data (buffer, rows_added)	Obrazové informace (width, height, buffer_bytes_per_pixel, red_gamma, green_gamma, blue_gamma)	dregs	Odkud se bere xww a yww	decoder, last_length	image_type	gamma_stamp	need_reparse	gamma_table
0	ne	ne	ne	ne	rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw<0?scale(32):wanted_xw; yww=wanted_yw<0?scale(32):wanted_yw`	neplatí	neplatí	neplatí	ano	neplatí
1	ne	ne	ne	ano	rozbitý rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw<0?scale(32):wanted_xw; yww=wanted_yw<0?scale(32):wanted_yw`	neplatí	neplatí	neplatí	ne	neplatí
2	ne	ne	ano	ne	rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw, yww=wanted_yw`	neplatí	neplatí	neplatí	ne	neplatí
3	ne	ne	ano	ano	rozbitý rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw, yww=wanted_yw`	neplatí	neplatí	neplatí	ne	neplatí
4	ne	ano	ne	ne	Toto jsou nesmyslné stavy, které nikdy nesmí nastat										
5	ne	ano	ne	ano											
6	ne	ano	ano	ne											
7	ne	ano	ano	ano											

Stav	Ví se typ souboru?	Hlavička obrázku (určující width a height) byla přečtena	wanted_xw i wanted_yw určeno obojí současně	Obrazový proud skončil	img_draw_image nakreslí a uživatel uvidí	bitmapa	Obrazová data (buffer, rows_added)	Obrazové informace (width, height, buffer_bytes_per_pixel, red_gamma, green_gamma, blue_gamma)	dregs	Odkud se bere xww a yww	decoder, last_length	image_type	gamma_stamp	need_reparse	gamma_table
8	ano	ne	ne	ne	rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw<0?scale(32):wanted_xw; yww=wanted_yw<0?scale(32):wanted_yw`	běží	platí	neplatí	ano	neplatí
9	ano	ne	ne	ano	rozbitý rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw<0?scale(32):wanted_xw; yww=wanted_yw<0?scale(32):wanted_yw`	neplatí	neplatí	neplatí	ne	neplatí
10	ano	ne	ano	ne	rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw, yww=wanted_yw`	běží	platí	neplatí	ne	neplatí
11	ano	ne	ano	ano	rozbitý rám	neplatí	neplatí	neplatí	neplatí	`xww = wanted_xw, yww=wanted_yw`	neplatí	neplatí	neplatí	ne	neplatí
12	ano	ano	ne	ne	obrázek	`bmp->user?obrázek:nic`	`strip_optimized?neplatí:platí`	platí	`strip_optimized?platí nebo NULL:neplatí`	`xww = wanted_xw<0?scale(32):wanted_xw; yww=wanted_yw<0?scale(32):wanted_yw`	běží	platí	platí	ne	NULL nebo naalokována

Stav	Více typů souboru?	Hlavička obrázku (určující width a height) byla přečtena	wanted_xw i wanted_yw určeno obojí současně	Obrazový proud skončil	img_draw_image nakreslí a uživatel uvidí	bitmapa	Obrazová data (buffer, rows_added)	Obrazové informace (width, height, buffer_bytes_per_pixel, red_gamma, green_gamma, blue_gamma)	dregs	Odkud se bere xww a yww	decoder, last_length	image_type	gamma_stamp	need_reparse	gamma_table
13	ano	ano	ne	ano	obrázek	obrázek	neplatí	neplatí	neplatí	Pokud `(wanted_xw<0&&wanted_yw<0)`, tak `xww=scale(width)` a `yww=scale(height)`. Pokud `(wanted_xw>=0&&wanted_yw<0)`, pak `xww=wanted_xw` a `yww=(xww*height+(width>>1))/width`. Pokud `(wanted_yw>=0&&wanted_xw<0)`, pak `yww=wanted_yw` a `xww=(yww*width+(height>>1))/height`. `width` a `height` jsou rozměry, které měl obrázek při dekódování, nikoli již aktuální stav položky v `cimg` (tam můžou být nesmysly)	neplatí	neplatí	platí	ne	neplatí
14	ano	ano	ano	ne	obrázek	`bmp->user?obrázek:nic`	`strip_optimized?neplatí:platí`	platí	`strip_optimized?platí nebo NULL:neplatí`	`xww=wanted_xw`, `yww=wanted_yw`	běží	platí	platí	ne	NULL nebo naalokována
15	ano	ano	ano	ano	obrázek	obrázek	neplatí	neplatí	neplatí	`xww=wanted_xw`, `yww=wanted_yw`	neplatí	platí	platí	ne	neplatí

Uvedené hodnoty jsou zaručeny pouze mimo funkce obrázků. Funkce obrázků mohou být v různých polovičatých stavech a položky používat na různé pomocné úkony takže tam to zaručeno není. `scale(něco)` může představovat hodnotu odpovídající staršímu nastavení `global_fdatac->ses->ds.image_scale`. Což ovšem nevadí, protože jakmile si jednou obrázek „nadiktuje“ svoje rozměry, tyto rozměry se předají při návratu z `insert_image` sazeči, a tomu je jedno, co v nich je.

	0	1	2	3	8	9	10	11	12	13	14	15
0	nic	soubor je celý stažený			je znám typ souboru							
1	count2 se změnil	nic										
2			nic	soubor skončil			je znám typ souboru					
3			count2 se změnil	nic								
8	count2 se změnil				nic	soubor skončil			přečetla se hlavička			
9	count2 se změnil					nic						
10			count2 se změnil				nic	soubor skončil			přečetla se hlavička	
11			zvětšil se count2					nic				
12	count2 se zvětšil								nic	soubor skončil		
13	count2 se zvětšil									nic		
14			count2 se zvětšil								nic	soubor skončil
15			count2 se zvětšil									nic

Tato tabulka říká, při jakých příležitostech dochází k přechodům mezi jednotlivými stavy. V řádcích jsou stavy před přechodem, ve sloupcích jsou stavy po přechodu.

`wanted_xw`, `wanted_yw` jsou požadované rozměry obrázku vyjádřené v pixelech displaye. Jsou to tedy hodnoty z tagů `width` a `height` naškálované na `img->scale`. Pokud tag nebyl specifikován, je v příslušné proměnné -1. `wanted_xw` ani `wanted_yw` nesmí nabývat hodnoty 0.

`scale` je škálování pro které položka z cache platí. Při hledání položek v cachi se ignoruje v případě, že `wanted_x` a `wanted_y` jsou obě specifikované (aby se mohly ztotožnit obrázky které se neliší ve výsledné velikosti).

`xww` a `yww` jsou výsledné rozměry obrázku podle aktuálních vědomostí, vyjádřený v pixelech displaye. V případě, že se nějaký rozměr nedá nijak určit, dá se tam provizorně `scale(32)`. `xww` i `yww` musí být `>=1`.

`image_type` pokud platí, definuje, o jaký druh obrázku se jedná z hlediska Content-Type. Zjištění typu, pro který není do **Links** vestavěn dekodér, se považuje za chybu v kódovém proudu a interpretuje se, jako kdyby soubor skončil, proto v proměnné `image_type` je v případě platnosti vždy hodnota poukazující na konkrétní typ toku obrazového kódu. Pokud se identifikuje nějaký známý typ, do `image_type` se přiřadí jedna z konstant:

- `IM_PNG`
- `IM_MNG`
- `IM_JPG`
- `IM_PCX`
- `IM_BMP`
- `IM_TIF`
- `IM_GIF`
- `IM_XBM`

`rows_added` když platí a je nula tak to znamená, že bitmapa zobrazuje přesně to samé, co je v bufferu a také to znamená, že bitmapa je platná. Pokud `rows_added` platí a je 1, pak to znamená, že do bufferu bylo zapsáno a nebyla updatována bitmapa, tudíž bitmapu nelze kreslit, neboť by zobrazovala starý stav, a tudíž je nutno před kreslením bitmapy nebo likvidací bufferu nutno buffer předělat do bitmapy. Platné `rows_added` a `rows_added==1` neznamená nutně, že bitmapa je neplatná. Když je `rows_added` platné a bitmapa neplatná, je vždy nastaveno `rows_added` na 1.

`buffer` má paměťovou organizaci pro 8-bitové kanály podle následující tabulky. R, G, B jsou barevné složky, A je alpha. Složky jsou uloženy v charech.

buffer_bytes_per_pixel	**Obsah paměti (po charech)**
3	RGB
4	RGBA

Pro 16-bitové kanály má buffer následující paměťovou organizaci, složky jsou uloženy v unsigned shortech:

buffer_bytes_per_pixel	**Obsah paměti (po charech)**
`3*sizeof(unsigned short)`	RGB
`4*sizeof(unsigned short)`	RGBA

`strip_optimized` signalizuje, že se nepoužívá buffer, a namísto toho se dekódovaný proužek rovnou ditheruje (v restartovatelné ditherovací engině s pomocnou chybovou řádkou `dregs`). V takovém případě `buffer` ani `rows_added` není platné. Ostatní obrazové informace platné jsou protože jsou k ditherování potřeba. Hodnota této proměnné se rozhoduje konkrétní dekódovací funkcí určitého formátu (JPEG, PNG, TIFF, GIF, XBM) před zavoláním `header_dimensions_known()`. Pokud je `strip_optimized` nastaveno když se obrázek bude škálovat a je zavoláno `header_dimensions_known()`, které `strip_optimized` automaticky shodí, protože nelze kreslit po kusech a současně škálovat.

`width` a `height` určují (v případě, že jsou platné, tedy spolu s bufferem) rozměry bufferu. Musí platit: `width>=1, height>=1`.

„`dregs` jsou NULL nebo platné“ znamená, že v okamžiku, kdy se tato položka „vyplňuje“, tak když je nastaveno `dither_images`, tak se naalokuje, jinak se dá na NULL.

`dregs` je chybový řádek z minulého běhu ditherovací enginy. Má smysl jen ve stavech 12 a 14 a to ještě, když je zapnuto `strip_optimized`. Tento způsob ditherování se dá použít jedině když obrázek není prokládaný, takže se kreslí zezhora dolů v jednom kuse, a současně když se obrázek nezvětšuje ani nezmenšuje (na to se musí totiž celý dekódovat a pak naráz zmenšit nebo zvětšit).

`gamma_table` se vyskytuje jen když obrázek má dost pixelů (vyplatí se) a když nemá 16 bitů na složku (pak by tabulka byla moc velká). Mohla by se udělat pro 16-bitové obrázky kdyby měl obrázek víc nebo rovno než řekněme 131072 pixelů, ale 16-bitové obrázky jsou poměrně vzácné. Navíc by zabírala v paměti asi 300kB. `gamma_table` má paměťovou organizaci 3*256 unsigned shortů. Prvních 256 je pro červenou, další pro zelenou, a posledních 256 pro modrou složku. Může přítomna jen ve stavech 12 a 14 a v ostatních je odalokovaná. Když není ve stavech 12 a 14 přítomna, je na jejím místě NULL, aby se to poznalo.

Pomocná položka `bmp->user` se zde používá jako indikátor, zda bitmapa tam je nebo není. Pokud je `bmp` platná a `user` je NULL, pak zbytek struktury `bmp` není platný a nic není alokováno a `bmp` se bere jako prázdná. Pokud je `bmp` platná a `user` není NULL, pak zbytek struktury `bmp` je platný a bitmapa je alokována a je v ní obrázek který se může kreslit. Žádný rozměr bitmapy nemůže být nula. To je ošetřeno již na začátku funkcí `insert_image` a `img_draw`, že se hned vrací a nic se nedělá, takže se to sem vůbec nedostane.

`last_length`, pokud platí, říká, kde jsme naposled skončili při lití dat do dekodéru. Z této definice je již jasné, že je platný pouze ve stavech, kdy dekodér běží, a že se nastavuje na nulu při startu dekodéru.

`last_count2` je platný vždy. Slouží ke zjištění, zda se `count2` změnilo (což by naznačovalo reload).

1.5.4 Reakce obrázků na změny od uživatele

Změní-li se **gamma**, zvětší se globální proměnná `gamma_stamp` a zavolá se na všechny obrázky `redraw`. Ten zjistí, že se změnila globální proměnná `gamma_stamp`, a v případě že `cimg->gamma_stamp` je platná, otestuje tyto dvě proměnné proti sobě a zachová se, jako by se změnil `count2` (úplný reload, protože se zcela změní vzhled obrázku).

Změní-li se **škálování** (v `global_fdatac->ses->ds.image_scale`), pak se přeparsuje a přesází celý dokument, takže se změní i příslušné požadované rozměry, a v cachi se začne hledat něco jiného.

`need_reparse` znamená, že kromě případu reloadu se nemůže změnit rozměr místa, který obrázek zabírá.

1.6 Ditherování a barvy

Provádí se v `dither.c` a `dip.c`. `dither.c` je nutno nejdříve nainicializovat voláním `init_dither()`, čímž se vygenerují ditherovací tabulky. Ditherovací tabulky se jmenují `red_table`, `green_table` a `blue_table`. Každá z nich má 65536 položek typu int, protože vstup do ditherovače má 16 bitů na barevný kanál. Celkově tedy tabulky zabírají v paměti prohlížeče 0.75MB při velikosti integeru 4 byty. Teoreticky by se mohly samotné tabulky udělat např. 12-bitové (zabíraly by pak jen 49kB) a ditherování jako takové nechat 16-bitové, ale není to kritické a raději s tím počkáme, než abychom to zabugovali. Jediný problém zde činí fakt, že tabulky se počítají v plovoucí čárce a staré procesory i486 bez matematické emulace se při startu prohlížeče v grafickém režimu skutečně zapotí.

Vstup do ditherovací tabulky je 16-bitové číslo 0–65535 značící lineární hodnotu světla, kterou se snažíme reprezentovat. Tabulka nám pak sdělí, jaký kód máme vyplodit na výstup (do bitmapy grafického driveru) a jaký skutečný světelný tok vyplozenému kódu odpovídá. Skutečnou hodnotu odečteme od požadované a získáme chybu, kterou rozdistribuujeme do pixelů, které budou zpracovávány později. 7/16 půjde do pixelu vpravou, 1/16 do pixelu vpravo dole, 5/16 do pixelu dole a 3/16 do pixelu vlevo dole.

Tato metoda se nazývá jednosměrný Floyd-Steinberg a byla vybrána na základě testovacích obrázků v přednášce Počítačová Grafika I. Josefa Pelikána. Dělá se samozřejmě v prostoru přímo úměrném světlu vycházejícímu z monitoru, protože jedině v takovém prostoru dává smysl a optimální výsledky. Proto je také nutno prohlížeč kalibrovat, aby ditherování a jiné operace pracovaly s fyzikálně a fyziologicky podloženými čísly a nikoliv s brambory a švestkami.

Jiný možný pohled na použitou metodu je jako na sigma-delta modulaci. Ditherovací algoritmus funguje jako sigma-delta modulátor a posouvá kvantizační šum do vyšších frekvencí, na které je optická soustava méně citlivá (díky rozmazanosti monitoru a zobrazovacím vadám lidského oka) čímž se subjektivně zlepšuje dojem z obrazu.

Kromě ditherovacích tabulek prohlížeč používá ještě tabulky pro zaokrouhlování `round_red_table`, `round_green_table` a `round_blue_table`. Tyto mají každá 256 položek typu unsigned short, takže zabírají celkem 1536 bytů. Vstupem je sRGB hodnota 0–255 a výstupem je 16-bitová lineární světelná hodnota odpovídající barvě, která se zobrazí, nastaví-li se tato hodnota jako HTML pozadí. Vzhledem k tomu, že pozadí se nikdy neditheruje, jsou tyto tabulky potřeba také na to, aby se pozadí u transparentních obrázků vyplnilo barvou, která bude korespondovat s pozadím stránky. Pozadí stránky se totiž zaokrouhlí na nejbližší barvu, která je zobrazitelná bez ditherování.

Ditherování obrazu se dělá voláním funkce `dither()` a zaokrouhlování se dělá funkcí `(*round_fn)`. Jedna z těchto dvou možností se provádí podle toho, jestli si uživatel zapnul nebo vypnul ditherování obrázku a písmenek. Barva pozadí se grafickému driveru nastaví podle hodnoty, kterou vrátí `dip_get_color_sRGB` (ten vrací už přímo barvu pro driver). Je to ještě cachované, skutečný výpočet dělá `real_get_color_sRGB`.

Popřeďové barvy písmenek se nezaokrouhlují, pouze pokud jsou příliš podobné pozadí, vygeneruje se kontrastní barva a dosadí se do popředí, aby text byl čitelný. To dělá funkce `separate_fg_bg`. Pozadí písmenek se stejně jako pozadí obrázků zaokrouhluje, aby souhlasilo s pozadím stránky.

Ditherovací tabulky se vypočítají ve funkci `make_16_table`. Buňka ditherovací tabulky obsahuje číslo typu int (předpokládáme, že int podrží aspoň 32 bitů). 16 horních bitů obsahuje kód pro grafický driver, který se pošle do bitmapy, 16 dolních bitů číslo 0–65535 popisující přesně světlo, které vychází z monitoru při zobrazení tohoto kódu.

Pokud paměťová organizace obrazovky má 8 bitů na barevný kanál, pak je těcho 8 bitů uloženo v bitech 16–23 položky ditherovací tabulky. Pokud paměťová organizace má 16 bitů na pixel (a to ať už 15 nebo 16 významných), pak v případě, že existuje datový

typ `t2c` 2-charový datový typ, je v bitech 16–31 uložen obsah pixelu tak, že se vezme pixel (s nastavenými jen těmi barevnými bity, od kterého barevného kanálu je tabulka), načte se do `t2c`, a toto se uloží do bitů 16–31 ditherovací tabulky. Pokud zabírá pixel 2 byty a není definován `t2c`, je v bitech 16–23 uložen obsah bytu paměti s nižší adresou a 24–31 bytu s vyšší adresou. Pokud je 1 byte na pixel, pak je tento byte (s nastavenými jen těmi bity, pro který kanál je ditherovací tabulka) uložen v bitech 16–23 položky ditherovací tabulky.