VisualOS

Programer Reference

Manuel Estrada Sainz

ranty@atdot.org
ranty@soon.com

VisualOS: Programer Reference
by Manuel Estrada Sainz

Table of Contents

Chapter 1. Introduction

This is a programers reference, so you will find lots of C code and tecnical explanations all of it, as you can see, writen in english, as I spect to share all this with the rest of the world via The Internet so if you don’t what to modify the code or find out how the program works internaly, then the user manual will provably benefit you most.

Chapter 2. SubSystems

The program is divided into Subsystems, which communicate with each other via a messaging facility.

2.1. Procesor

This is the subsystem responsable of executing the processes and, usualy, is the one which generates all activity in the other subsystems as their client.

Procesor Interface

Name

`Procesor Interface` — CPU interface to the other subsystems.

Synopsis

```
void        cpu_register_proc_creat         (proc_creat_callback_t func);
void        cpu_register_proc_finish        (proc_finish_callback_t func);
void        (*proc_creat_callback_t)        (gint pid);
void        (*proc_finish_callback_t)       (gint pid);
void        CPU_terminate_proc              (gint pid);
```

Description

This functions allow the other subsystems to interact, througth the messaging service, with the CPU.

Details

cpu_register_proc_creat ()

```
void        cpu_register_proc_creat         (proc_creat_callback_t func);
```

Registers *func* to be called when ever a new process gets created.

func : function to be called.

cpu_register_proc_finish ()

```
void        cpu_register_proc_finish        (proc_finish_callback_t func);
```

Registers *func* to be called when ever a process is terminated.

func : function to be called.

proc_creat_callback_t ()

```
void        (*proc_creat_callback_t)        (gint pid);
```

Function pointer type for the callback used on *cpu_register_proc_creat*.

pid : Process ID of the created process.

proc_finish_callback_t ()

```
void        (*proc_finish_callback_t)       (gint pid);
```

Function pointer type for the callback used on *`cpu_register_proc_finish`*.

`pid :` Process ID of the terminated process.

CPU_terminate_proc ()

```
void        CPU_terminate_proc              (gint pid);
```

Terminate process *`pid`*.

`pid :` Process to terminates.

Procesor Configuration

Name

`Procesor Configuration` — Internal configuration.

Synopsis

```
struct      cpu_config_t;
extern      const cpu_config_t *CPU_config;
cpu_config_t* get_CPU_config                (void);
```

Description

This is the way to find out CPU internal configuration.

Details

struct cpu_config_t

```
typedef struct {
gboolean stop_clock; /* clock should be stoped when something
   interesting happens */
gboolean auto_fill_procs; /* process properties should be
   filled automaticly without any
   user interaction */
prop_io_params_t prop_io_params;   /* current parame-
ters for processes
      IO parameters autofilling */
prop_mem_params_t prop_mem_params; /* current param-
etes for processes
      Memory parametes autofilling */
struct { /* parameters related with the
   graphical representation of the
   subsystem */
gint max_graph_history; /* Maximun pixmap width for the
   different representations */
gint pix_size_step;
} drawing;
} cpu_config_t;
```

CPU_config

```
extern const cpu_config_t *CPU_config;
```

This is a pointer to the configuration data, but should be used only for reading.

get_CPU_config ()

```
cpu_config_t* get_CPU_config                (void);
```

This is the right way to modify the configuration.

Returns : a writable pointer to the configuration data.

Procesor Simulation

Name

`Procesor Simulation` — Simulation of the processor

Synopsis

```
struct      simul_data_t;
struct      simul_io_event_t;
```

```
struct      simul_mem_t;
struct      event_data_t;
void        init_CPU_simulation             (void);
void        init_CPU_simulation_in_proc     (proc_t *proc);
void        end_CPU_simulation_in_proc      (proc_t *proc);
void        cpy_CPU_simulation_data         (simul_data_t *dest,
                                             simul_data_t *src);
simul_data_t* dup_CPU_simulation_data       (simul_data_t *data);
void        free_CPU_simulation_data        (simul_data_t *data);
void        free_CPU_proc_simulation_data   (proc_t *proc);
void        next_CPU_simulation_in_proc     (proc_t *proc);
gint        CPU_proc_current_page           (proc_t *proc);
gint        CPU_proc_next_page              (proc_t *proc);
gboolean    CPU_proc_current_page_is_write  (proc_t *proc);
void        fix_simulation_in_proc          (proc_t *proc);
#define     IO_BLOCK                        (event)
```

Description

This functions serve to manage the CPU's simulation and it's data structures.

Details

struct simul_data_t

```
typedef struct { /* simulation data for a process */
gint start_time; /* time of creation */
gint end_time; /* length of the process */
/* IO properties */
/* this list is never modified until process destruction */
simul_io_event_t *io_events; /* list of all io events */
simul_io_event_t *next_io_event; /* pointer to the next event */
simul_io_event_t *last_io_event; /* pointer to the last event */
```

```
/* MEM properties */
simul_mem_t *pages; /* list of memory accesses */
gint n_pages; /* number of memory accesses */
gint cur_access; /* index to the current access */
} simul_data_t;
```

struct simul_io_event_t

```
typedef struct { /* data for an IO event */
gint block; /* block to read */
gint time; /* time to read @block */
} simul_io_event_t;
```

struct simul_mem_t

```
typedef struct { /* data for a page access */
gint8 page; /* page to access */
gint8 write; /* is it a write access? */
} simul_mem_t;
```

struct event_data_t

```
typedef struct { /* io event data known by all the code */
gint io_block; /* block to access */
} event_data_t;
```

init_CPU_simulation ()

```
void        init_CPU_simulation            (void);
```

Initializes the simulation code.

init_CPU_simulation_in_proc ()

```
void        init_CPU_simulation_in_proc     (proc_t *proc);
```

prepares the data structures for simulation in process *proc*.

proc : process involved.

end_CPU_simulation_in_proc ()

```
void        end_CPU_simulation_in_proc      (proc_t *proc);
```

cleans up simulation data in *proc* to prepare it for termination.

Note: currently does nothing.

proc : the process involved.

cpy_CPU_simulation_data ()

```
void        cpy_CPU_simulation_data         (simul_data_t *dest,
                                             simul_data_t *src);
```

Copies the contents of *src* into *dest*, allocating dynamic memory when needed.

dest : the target of the copy

src : the source for the copy

dup_CPU_simulation_data ()

```
simul_data_t* dup_CPU_simulation_data       (simul_data_t *data);
```

Duplicates a simul_data_t structure.

`data :` a pointer to the data to be copied

Returns : a pointer to newly allocated memory with the same content of *`data`*

free_CPU_simulation_data ()

```
void        free_CPU_simulation_data        (simul_data_t *data);
```

Frees all dynamic memory asociated to *`data`*, including *`data`* itself.

`data :` the data to be freed

free_CPU_proc_simulation_data ()

```
void        free_CPU_proc_simulation_data   (proc_t *proc);
```

Free all simulation related dynamic memory from the *`proc`* structure.

`proc :` a pointer to a proc_t structure

next_CPU_simulation_in_proc ()

```
void        next_CPU_simulation_in_proc     (proc_t *proc);
```

Once the process had an event it prepares the process for its next event, making it a termination event if necesary.

proc : the process involved.

CPU_proc_current_page ()

```
gint        CPU_proc_current_page           (proc_t *proc);
```

proc : the process involved.

Returns : the page which *proc is using on this very moment.*

CPU_proc_next_page ()

```
gint        CPU_proc_next_page              (proc_t *proc);
```

Makes the process move to its next memory page and should be called once for each "clock tick" that *proc* is running.

proc : the process involved.

Returns : the new page which the process is using.

CPU_proc_current_page_is_write ()

```
gboolean    CPU_proc_current_page_is_write  (proc_t *proc);
```

Checks if *proc* is writing to memory or only reading.

proc : the process involved.

Returns : TRUE when *proc is writing to its current page.*

fix_simulation_in_proc ()

```
void        fix_simulation_in_proc          (proc_t *proc);
```

Makes the simulation parameters of a process coherent, prevents: multiple events at the same time, events after process termination, determines the next event ...

proc : process whose simulation data should be fixed.

IO_BLOCK()

```
#define IO_BLOCK(event) (((event_data_t *)(event.data))-
>io_block)
```

Extracts the block from a IO event.

event : process event.

Processes

Name

`Processes` — Process Handling.

Synopsis

```
proc_t*     create_process                (void);
proc_t*     new_process                   (void);
void        insert_process                (proc_t *proc);
void        free_process                  (proc_t *proc);
gint        destroy_process               (proc_t *proc);
proc_queue_t get_proc_list                (void);
proc_t*     get_proc_by_pid               (gint pid);
void        select_process                (proc_t *proc);
proc_t*     get_CPU_selected_proc         (void);
void        save_processes_to_file        (void);
void        load_processes_from_file      (void);
#define     burst                         (proc)
```

Description

This functions provide general process handling. Creation, insertion into the system, destruction, finding a certain process...

Details

create_process ()

```
proc_t*     create_process                  (void);
```

Do everything necesary to have a new process in the system.

Returns : the newly created process.

new_process ()

```
proc_t*     new_process                     (void);
```

Allocate data for new process.

Note: the process will have to be inserted to have any efect.

Returns : the newly allocate process data.

insert_process ()

```
void        insert_process                  (proc_t *proc);
```

Inserts process *`proc`* in the system.

Note: *`proc`* can be obtained with *`new_process`*.

`proc :` the process involved.

free_process ()

```
void        free_process                    (proc_t *proc);
```

Definitely free all data related to *proc*.

Note: *proc* will be gone for good.

proc : process involved.

destroy_process ()

```
gint        destroy_process                 (proc_t *proc);
```

Start considering *proc* a terminated process and remove it from the system.

Note: the data of *proc* is not freed, and *proc* will be saved by *save_processes_to_file*.

proc : process involved.

Returns : nothing important.

get_proc_list ()

```
proc_queue_t get_proc_list                  (void);
```

Returns : the list of all currently running processes.

get_proc_by_pid ()

```
proc_t*     get_proc_by_pid                 (gint pid);
```

pid : process ID

Returns : the data of process *pid or NULL if there is no process with PID pid.*

select_process ()

```
void        select_process                  (proc_t *proc);
```

Makes *proc* the selected process.

Note: some code will do things to the selected process.

proc : process involved.

get_CPU_selected_proc ()

```
proc_t*     get_CPU_selected_proc           (void);
```

Find out which is the currently selected process

Returns : the currently selected process.

save_processes_to_file ()

```
void        save_processes_to_file          (void);
```

Asks the user for a filename and saves all processes on the current session to a file, ready to be loaded in a new session.

Note: Both terminated and not yet running processes will be written.

load_processes_from_file ()

```
void        load_processes_from_file        (void);
```

Asks the user for a filename and loads all processes it can find it it.

Note: Not all processes will be visible at once, they will be inserted at the right time.

burst()

```
#define burst(proc) (proc->next_event.time - proc->time)
```

Calculates the current burst (time until next voluntary event) for *proc*.

proc : process involved.

Processor Status

Name

`Processor Status` — Status of the processor.

Synopsis

```
struct      proc_queues_t;
const proc_queues_t* get_CPU_queues         (void);
proc_t*     get_CPU_current_proc            (void);
proc_queue_t get_CPU_queue                  (gint nqueue);
proc_queue_t get_CPU_wait_queue             (void);
gint        request_nqueues                 (gint nqueues);
void        move_proc_to_queue              (proc_t *proc,
                                             gint new_queue);
void        move_proc_to_CPU                (proc_t *proc);
gint        suspend_proc                    (proc_t *proc);
gint        wakeup_proc                     (proc_t *proc);
```

Description

Here is descrived show to find out the status of the processor, the currently running process and all the ready queues.

There are also functions to move processes around and change the number of ready queues.

Details

struct proc_queues_t

```
typedef struct { /* Processor status */
gint nqueues; /* Number of queues */
proc_t *current; /* Currently running process */
proc_queue_t *queue; /* Ready process queues */
proc_queue_t wait; /* Blocked processes */
} proc_queues_t;
```

get_CPU_queues ()

```
const proc_queues_t* get_CPU_queues         (void);
```

Retrive the processor's status.

Returns : a pointer to the proc_queues_t structure.

get_CPU_current_proc ()

```
proc_t*     get_CPU_current_proc            (void);
```

Returns : The currently running process, which may be NULL if the processor is idle.

get_CPU_queue ()

```
proc_queue_t get_CPU_queue                    (gint nqueue);
```

nqueue : requested queue.

Returns : ready queue number *nqueue.*

get_CPU_wait_queue ()

```
proc_queue_t get_CPU_wait_queue               (void);
```

Returns : the blocked process queue.

request_nqueues ()

```
gint        request_nqueues                   (gint nqueues);
```

Sets the number of queues for handling ready processes.

NOTE: when shrinking the lower queues, those which will be deleted, must be empty or otherwise they will be concatenated to the first queue.

nqueues : requested number of queues.

Returns : nothing important.

move_proc_to_queue ()

```
void        move_proc_to_queue              (proc_t *proc,
                                             gint new_queue);
```

Move *proc* to queue number *new_queue*.

NOTE: The *proc* should not be blocked.

proc : process to be moved.
new_queue : target queue for *proc.*

move_proc_to_CPU ()

```
void        move_proc_to_CPU                (proc_t *proc);
```

Starts running process *proc*.

NOTE: the processor should be idle.

proc : process to run.

suspend_proc ()

```
gint        suspend_proc                    (proc_t *proc);
```

Move *proc* out of the way when it blocks.

proc : process involved.
Returns : nothing important.

wakeup_proc ()

```
gint        wakeup_proc                         (proc_t *proc);
```

Move a process back when it becomes ready again letting the current algorithm decide were to put it.

`proc :` process involved.
Returns : nothing important.

See Also

Process Queues

How to inspect process queues.

Process Queues

Name

`Process Queues` — Process Queue Handling.

Synopsis

```
typedef     proc_queue_t;
#define     DECLARE_PROC_QUEUE              (queue)
#define     proc_queue_empty                (queue)
#define     proc_data                       (element)
#define     proc_queue_next                 (element)
#define     proc_queue_find                 (queue, proc)
#define     proc_queue_len                  (queue)
#define     proc_queue_init                 (queue)
#define     proc_queue_foreach              (queue, func, data)
#define     proc_queue_concat               (dest, orig1, orig2)
#define     proc_queue_remove               (queue, proc)
#define     proc_queue_append               (queue, proc)
#define     proc_queue_nth                  (queue, n)
#define     proc_queue_end                  (element)
```

Description

This are the functions to use when working with any queue of processes in the CPU.

Details

proc_queue_t

DECLARE_PROC_QUEUE()

```
#define     DECLARE_PROC_QUEUE(queue)
```

Declares a new and empty process queue with name *queue*.

queue : name for the queue.

proc_queue_empty()

```
#define     proc_queue_empty(queue)
```

Is *queue* empty?

queue : process queue involved.

proc_data()

```
#define     proc_data(element)
```

Retrives the process data from the queue *element*.

element : process queue element involved.

proc_queue_next()

```
#define     proc_queue_next(element)
```

Gets the next element on the queue starting at *element*.

element : process queue element involved.

proc_queue_find()

```
#define     proc_queue_find(queue, proc)
```

Find the queue element for process *proc*.

queue : process queue involved.
proc : process involved.

proc_queue_len()

```
#define     proc_queue_len(queue)
```

Calculate the number of elements on *queue*.

queue : process queue involved.

proc_queue_init()

```
#define     proc_queue_init(queue)
```

Initializes process queue *queue*.

Note: applyed to a non empty queue will loose all its elements.

queue : process queue involved.

proc_queue_foreach()

```
#define     proc_queue_foreach(queue, func, data)
```

Will call *func* for every process on *queue* using the process pointer as the first arguemt and *data* as the second.

queue : process queue involved.
func : function to be called.
data : second argument to *func.*

proc_queue_concat()

```
#define     proc_queue_concat(dest, orig1, orig2)
```

Concatenates *orig1* and *orig2* into *dest*.

dest : Target queue.
orig1 : First source queue.
orig2 : Second source queue.

proc_queue_remove()

```
#define     proc_queue_remove(queue, proc)
```

Remove *proc* from *queue*.

queue : process queue involved.
proc : process to be removed.

proc_queue_append()

```
#define     proc_queue_append(queue, proc)
```

Append *proc* to *queue*.

queue : process queue involved.
proc : process to be appended.

proc_queue_nth()

```
#define     proc_queue_nth(queue, n)
```

Get element number *n* from *queue*.

queue : queue involved.
n : element index.

proc_queue_end()

```
#define proc_queue_end(element) proc_queue_empty(element)
```

Is this element the end of the queue?

element : process queue element involved.

Processor Algorithms

Name

`Processor Algorithms` — Interface for Algorithms.

Synopsis

```
struct      cpu_algorithm_t;
cpu_algorithm_t* get_CPU_current_algorithm  (void);
GSList*     init_CPU_algorithms             (void);
gint        regis-
ter_CPU_algorithm          (cpu_algorithm_t *algorithm);
gint        deallocate_algorithm_private_data
                                            (proc_queue_t proc_list);
gint        set_CPU_heart_beat              (gint freq);
gint        reset_CPU_timer                 (void);
```

Description

Great effort has been devoted to making the addicion of new algorithms as easy as posible, here is documented what there is to know to be able to write your own algorithms.

Details

struct cpu_algorithm_t

```
typedef struct { /*This struct is all that we know about each al-
gorithm*/
gchar * name;
gint (*select) (void);
gint (*unselect) (void); /* These two functions will be
                                                    called be-
fore and after the
                                                    use of an al-
gorithm to let
                                                    it keep a low mem-
ory usage
                                                    when not in use.*/
gint (*clock) (void); /* timer interrupt. */
gint (*select_proc) (proc_t *proc);
/* notifies the algorithm of a
   process selection by the
   user */
GtkWidget * process_properties;
GtkWidget * properties; /* Each algorithm will maintain
   it’s own properties widgets.
   NULL means "no properties".
   They should be set to NULL
   when destroyed. If not
   destroyed in "unselect" the
   system will destroy them.*/
gint (*init_proc) (proc_t *proc);
/* This function should allocate
   and initialice algorith data
   and anything else to get
   a new process going, like
   sticking it into a queue. */
```

```
gint (*end_proc) (proc_t *proc);
/* This function should free the
   algorithm specific data of
   proc but should not take it
   out of its queue */
gint (*event) (proc_t *proc); /* This function is called when
   ever a process gets waked up
   by an event and we have to
   put it in some queue. */
gint (*next) (proc_t *proc); /* This function is called when
   ever the current process gets
   suspended waiting for some
   event and we have to choose
   another one to run.
   It receives the suspended
   process as argument just in
   case its needed. */
} cpu_algorithm_t;
```

get_CPU_current_algorithm ()

```
cpu_algorithm_t* get_CPU_current_algorithm  (void);
```

Find out which is the current algorithm.

Returns : the struct which descrives the current algorithm.

init_CPU_algorithms ()

```
GSList*     init_CPU_algorithms             (void);
```

Initializes the CPU algorithms code.

Mainly will call init functions for each algorithm.

Returns : a pointer to the algorithm structs linked list

register_CPU_algorithm ()

```
gint        regis-
ter_CPU_algorithm          (cpu_algorithm_t *algorithm);
```

Each algorithm should call this function in it's initialization function to register its algorithm struct.

algorithm : algorithm struct to register.
Returns : nothing important.

deallocate_algorithm_private_data ()

```
gint        deallocate_algorithm_private_data
                                              (proc_queue_t proc_list);
```

This function uses the algorithm's private data of each process as argument to g_free.

This is for convinience of algorithm writers.

proc_list : queue of processes.
Returns : nothing important.

set_CPU_heart_beat ()

```
gint        set_CPU_heart_beat              (gint freq);
```

Set timer interrupt frequency. Which means, the calling frequency of algorithm function *clock*.

Zero means that the timer interupt is not desired.

freq : new frequency in "time units".
Returns : nothing important.

reset_CPU_timer ()

```
gint        reset_CPU_timer                 (void);
```

Resets the "time unit" counter so we will have a full timeslice until the next interupt.

Returns : nothing important.

See Also

Process Queues

How to inspect process queues.

Processor Simulation

How simulation works.

Property Widget Facility

How to hadle numerical algorithm properties with the user without learning GTK+.

2.2. I/O

This subsystem is responsable of accepting block disk accesses, simulating them and reporting its client when done.

IO Interface

Name

`IO Interface` — IO interface to the other subsystems.

Synopsis

```
void        (*block_ready_callback_t)       (gint block);
gint        io_register_block_ready         (block_ready_callback_t func);
void        io_request_block                (gint block);
void        io_request_swap_block           (gint block);
```

Description

This functions allow the other subsystems to interact, througth the messaging service, with the I/O subsystem.

Details

block_ready_callback_t ()

```
void        (*block_ready_callback_t)       (gint block);
```

Function pointer type for the callback used on *`io_register_block_ready`*.

`block :` block number of the fulfilled IO access.

io_register_block_ready ()

```
gint        io_register_block_ready         (block_ready_callback_t func);
```

Instructs the IO subsystem to call *`func`* when a block access is finished.

`func :` function to be called when a requested block access is finished.

Returns : nothing important.

io_request_block ()

```
void        io_request_block                (gint block);
```

Instructs the IO subsystem to accesses *`block`* from the data area.

block : data block to be accessed.

io_request_swap_block ()

```
void        io_request_swap_block           (gint block);
```

Instructs the IO subsystem to accesses *block* from the swap area.

block : swap block to be accessed.

Geometry

Name

Geometry — Geometry of the device.

Synopsis

```
gint        get_IO_blocks_per_track         (void);
gint        get_IO_max_data_block           (void);
gint        get_IO_max_swap_block           (void);
gint        get_IO_ntracks                  (void);
```

```
gint        get_IO_last_data_track          (void);
gint        IO_request_track                (io_request_t *request);
void        init_IO_geometry                (void);
```

Description

This functions serve to find out the geometry of the disk and do some translations based on it.

Details

get_IO_blocks_per_track ()

```
gint        get_IO_blocks_per_track         (void);
```

Returns : the number of blocks per track.

get_IO_max_data_block ()

```
gint        get_IO_max_data_block           (void);
```

Returns : the maximun data block number.

get_IO_max_swap_block ()

```
gint        get_IO_max_swap_block           (void);
```

Returns : the maximun swap block number.

get_IO_ntracks ()

```
gint        get_IO_ntracks                  (void);
```

Returns : the number of tracks on the disk.

get_IO_last_data_track ()

```
gint        get_IO_last_data_track          (void);
```

Returns : the last data track number.

IO_request_track ()

```
gint        IO_request_track                (io_request_t *re-
quest);
```

Calculates the track of a certain *request*.

request : IO request involved.

Returns : the track number corespoinding to *request.*

init_IO_geometry ()

```
void        init_IO_geometry                (void);
```

Initialices the geometry calculation code.

IO Simulation

Name

`IO Simulation` — Simulation of the device

Synopsis

```
gint        init_IO_simulation              (void);
gint        IO_algorithm_event              (io_request_t *request);
gint        get_IO_head_pos                 (void);
io_queue_t  get_IO_reading_queue            (void);
void        set_IO_reading_queue            (io_queue_t new_reading);
io_queue_t  get_IO_requested_queue          (void);
```

Description

This functions serve to manage the disk’s simulation and it’s data structures.

Details

init_IO_simulation ()

```
gint        init_IO_simulation              (void);
```

initialices the IO simulation code.

Returns : nothing important.

IO_algorithm_event ()

```
gint        IO_algorithm_event              (io_request_t *re-
quest);
```

Insert a new request in the IO subsystem using the current algorith.

Mainly passes *`request`* over to the current algorithm and puts it the the queue of requested blocks.

`request :` request to be inserted.

Returns : nothing important.

get_IO_head_pos ()

```
gint        get_IO_head_pos                 (void);
```

Returns : the track number over which the head is currently flying.

get_IO_reading_queue ()

```
io_queue_t  get_IO_reading_queue            (void);
```

Get all pending requests as ordered by the current algoritym.

Returns : the request’s "reading" queue (ordered by the current algorithm).

set_IO_reading_queue ()

```
void        set_IO_reading_queue            (io_queue_t new_reading);
```

Sets the request’s "reading" queue.

This function should be called when ever the reading queue is modified by external means, even if the pointer to the queue is not changed.

`new_reading` : newly ordered "reading" queue.

get_IO_requested_queue ()

```
io_queue_t  get_IO_requested_queue          (void);
```

Get all pending requests in order of arrival.

Returns : the request’s "requested" queue (in chronological ordered).

Request Queues

Name

`Request Queues` — IO Request Queue Handling.

Synopsis

```
typedef     io_queue_t;
#define     DECLARE_IO_QUEUE                (queue)
#define     io_queue_empty                  (queue)
#define     io_request_data                 (element)
#define     io_queue_next                   (element)
#define     io_queue_len                    (queue)
#define     io_queue_init                   (queue)
#define     io_queue_foreach                (queue, func, data)
#define     io_queue_concat                 (dest, orig1, orig2)
#define     io_queue_remove                 (queue, request)
#define     io_queue_append                 (queue, request)
#define     io_queue_end                    (element)
```

Description

This are the functions to use when working with any queue of requests in the IO subsystem.

Details

io_queue_t

DECLARE_IO_QUEUE()

```
#define     DECLARE_IO_QUEUE(queue)
```

Declares a new and empty request queue with name *queue*.

queue : name for the queue.

io_queue_empty()

```
#define     io_queue_empty(queue)
```

Is *queue* empty?

queue : request queue involved.

io_request_data()

```
#define     io_request_data(element)
```

Retrives the request data from the queue *element*.

element : request queue element involved.

io_queue_next()

```
#define     io_queue_next(element)
```

Gets the next element on the queue starting at *element*.

element : request queue element involved.

io_queue_len()

```
#define     io_queue_len(queue)
```

Calculate the number of elements on *queue*.

queue : request queue involved.

io_queue_init()

```
#define     io_queue_init(queue)
```

Initializes request queue *queue*.

Note: applyed to a non empty queue will loose all its elements.

queue : request queue involved.

io_queue_foreach()

```
#define     io_queue_foreach(queue, func, data)
```

Will call *func* for every request on *queue* using the request pointer as the first arguemt and *data* as the second.

queue : request queue involved.
func : function to be called.
data : second argument to *func.*

io_queue_concat()

```
#define     io_queue_concat(dest, orig1, orig2)
```

Concatenates *orig1* and *orig2* into *dest*.

dest : Target queue.
orig1 : First source queue.
orig2 : Second source queue.

io_queue_remove()

```
#define     io_queue_remove(queue, request)
```

Remove *request* from *queue*.

queue : request queue involved.
request : request to be removed.

io_queue_append()

```
#define     io_queue_append(queue, request)
```

Append *request* to *queue*.

queue : request queue involved.
request : request to be appended.

io_queue_end()

```
#define io_queue_end(element) io_queue_empty(element)
```

Is this element the end of the queue?

element : request queue element involved.

IO Algorithms

Name

IO Algorithms — Interface for Algorithms.

Synopsis

```
struct      io_algorithm_t;
io_algorithm_t* get_IO_current_algorithm    (void);
GSList*     init_IO_algorithms              (void);
gint        register_IO_algorithm           (io_algorithm_t *al-
gorithm);
```

Description

Great effort has been devoted to making the addicion of new algorithms as easy as posible, here is documented what there is to know to be able to write your own algorithms.

Details

struct io_algorithm_t

```
typedef struct { /* This struct is all we know
   about each algorithm */
gchar * name;
gint (*select) (void);
gint (*unselect) (void); /* These two functions will be
                                                    called be-
fore and after the
                                                    use of an al-
gorithm to let
                                                    it keep a low mem-
ory usage
                                                    when not in use.*/
GtkWidget * properties; /* Each algorithm will maintain
```

```
                                                    it's own prop-
erties widget.
                                                    NULL means "no properti
                                                    It should be set to NUL
                                                    when de-
stroyed. If not
                                                    de-
stroyed in "unselect" the
                                                    sys-
tem will destroy it.*/
gint (*event) (io_request_t *request);
/* This is called to inform the
   algorithm of block request*/
gint (*done_reading) (void); /* This ia called to inform the
   algorithm about all requested
   reads been done */
} io_algorithm_t;
```

get_IO_current_algorithm ()

```
io_algorithm_t* get_IO_current_algorithm    (void);
```

Find out which is the current algorithm.

Returns : the struct which descrives the current algorithm.

init_IO_algorithms ()

```
GSList*     init_IO_algorithms              (void);
```

Initializes the IO algorithms code.

Mainly will call init functions for each algorithm.

Returns : a pointer to the algorithm structs linked list.

register_IO_algorithm ()

```
gint        register_IO_algorithm           (io_algorithm_t *al-
gorithm);
```

Each algorithm should call this function in it's initialization function to register its algorithm struct.

algorithm : algorithm struct to register.
Returns : nothing important.

See Also

IO Simulation

How to inspect and manage requests.

Geometry

How manage geometry.

Property Widget Facility

How to hadle numerical algorithm properties with the user without learning GTK+.

Request Queues

How to hadle request queues.

IO Miscelaneous

Name

`IO Miscelaneous` — Other interesting functions.

Synopsis

```
gint        io_server_init                  (void);
gint        io_block_ready_server           (io_request_t *io_request);
```

Description

This functions wouldn’t make a proper section, but I believe that they are relevant enough to be mencioned.

Details

io_server_init ()

```
gint        io_server_init                  (void);
```

Called from the subsystem's code to initialize the interface code.

Returns : nothing important.

io_block_ready_server ()

```
gint        io_block_ready_server           (io_request_t *io_request);
```

Called from the subsystem's code to tell the client about a fulfilled request.

`io_request :` the fulfilled request.
Returns : nothing important.

2.3. Memory

This subsystem is responsable for accepting memory read and write requests, managing memory for all processes: stealing and swapping pages as necesary.

Memory Interface

Name

`Memory Interface` — MEM interface to the other subsystems.

Synopsis

```
void        (*page_ready_callback_t)        (gint pid,
                                             gint page);
void        mem_register_page_ready         (page_ready_callback_t func);
gboolean    mem_touch_page                  (proc_t *proc,
                                             gint page,
                                             gboolean write);
```

Description

This functions allow the other subsystems to interact, througth the messaging service, with the Memory.

Details

page_ready_callback_t ()

```
void        (*page_ready_callback_t)        (gint pid,
                                             gint page);
```

Function pointer type for the callback used on *mem_register_page_ready*.

pid : the process involved.
page : the process' page which has become ready.

mem_register_page_ready ()

```
void        mem_register_page_ready         (page_ready_callback_t func);
```

Registers *func* to be called whenever a processes page becomes available in physical memory.

func : function to be called.

mem_touch_page ()

```
gboolean    mem_touch_page                  (proc_t *proc,
                                             gint page,
                                             gboolean write);
```

Access the page number *page* of process *proc*. The access will be for writing if *write* is TRUE.

proc : process involved.
page : page we pretend to use.
write : are we writing to the specified page?
Returns : TRUE if the page was available and FALSE if a page fault ocurred and the process has to wait.

Memory Configuration

Name

`Memory Configuration` — Internal configuration.

Synopsis

```
struct      mem_config_t;
extern      const mem_config_t *MEM_config;
mem_config_t* get_MEM_config                (void);
```

Description

This is the way to find out Memory internal configuration.

Details

struct mem_config_t

```
typedef struct {
gboolean stop_clock; /* clock should be stoped when something
   interesting happens */
gboolean disabled; /* this subsystem is disabled */
```

```
} mem_config_t;
```

MEM_config

```
extern const mem_config_t *MEM_config;
```

This is a pointer to the configuration data, but should be used only for reading.

get_MEM_config ()

```
mem_config_t* get_MEM_config                (void);
```

This is the right way to modify the configuration.

Returns : a writable pointer to the configuration data.

Memory Algorithms

Name

`Memory Algorithms` — Interface for Algorithms.

Synopsis

```
struct      mem_algorithm_t;
mem_algorithm_t* get_MEM_current_algorithm  (void);
GSList*     init_MEM_algorithms             (void);
void        regis-
ter_MEM_algorithm          (mem_algorithm_t *algorithm);
```

Description

Great effort has been devoted to making the addicion of new algorithms as easy as posible, here is documented what there is to know to be able to write your own algorithms.

Details

struct mem_algorithm_t

```
typedef struct { /* This struct is all
   that we know about
   each algorithm */
gchar * name;
void (*select) (void);
void (*unselect) (void); /* These two functions
   will be called before
   and after the use of
   an algorithm to let
   it keep a low memory
   usage when not in
   use.*/
GtkWidget * properties; /* Each algorithm will
```

```
   maintain it's own
   properties widget.
   NULL means "no
   properties".  It
   should be set to NULL
   when destroyed. If
   not destroyed in
   "unselect" the system
   will destroy it.*/
void (*page_access) (gint pid, gint page); /* lets the algorithm
   keep track of page
   accesses */
gint (*select_frame) (void); /* it should return a
   frame to be assigned
   when out of memory */
} mem_algorithm_t;
```

get_MEM_current_algorithm ()

```
mem_algorithm_t* get_MEM_current_algorithm  (void);
```

Find out which is the current algorithm.

Returns : the struct which descrives the current algorithm.

init_MEM_algorithms ()

```
GSList*     init_MEM_algorithms             (void);
```

Initializes the MEM algorithms code.

Mainly will call init functions for each algorithm.

Returns : a pointer to the algorithm structs linked list.

register_MEM_algorithm ()

```
void        regis-
ter_MEM_algorithm          (mem_algorithm_t *algorithm);
```

Each algorithm should call this function in it’s initialization function to register its algorithm struct.

algorithm : algorithm struct to register.

Swap

Name

Swap — Swapping system.

Synopsis

```
void        MEM_swap_init                  (void);
void        MEM_swapout_page               (gint pid,
```

```
                                                gint page);
void        MEM_swapin_page                    (gint pid,
                                                gint page,
                                                guint32 set_flags);
```

Description

This functions serve to get process pages in and out of the swap device.

Details

MEM_swap_init ()

```
void        MEM_swap_init                      (void);
```

Gets things ready to be able to swap memory in and out.

MEM_swapout_page ()

```
void        MEM_swapout_page                   (gint pid,
                                                gint page);
```

Writes *page* from Proces' *pid* virtual memory address space from memory into a free block in the swap device.

pid : The Process Identification number.

page : Which one of the process' pages we what back.

MEM_swapin_page ()

```
void        MEM_swapin_page                 (gint pid,
                                             gint page,
                                             guint32 set_flags);
```

Reads *page* from Proces' *pid* virtual memory address space into memory from the swap device

If the page has never been swapped out we will suppose it is in the first free swap block and request an IO access to it.

If the page is already swaping in then we will take note of *set_flags* and let it be.

pid :	The Process Identification number
page :	Which one of the process' pages we what back
set_flags :	This flags will be set on the frame when when one is assigned

Memory Status

Name

Memory Status — Status of the memory.

Synopsis

```
struct      frame_info_t;
enum        mem_frame_flags_t;
#define     FRAME_LOCKED                    (frame)
#define     FRAME_REFERENCED                (frame)
#define     FRAME_MODIFIED                  (frame)
struct      proc_pages_info_t;
#define     PAGE_VALID                      (proc_pages, page)
#define     NO_FRAME
#define     NO_PAGE
#define     NO_PROC
#define     NO_BLOCK
void        init_page_info                  (void);
frame_info_t* get_free_frame                (void);
void        put_free_frame                  (gint frame);
gboolean    have_free_frame                 (void);
frame_info_t* get_frame_info                (gint frame);
frame_info_t* get_frames_list               (void);
frame_info_t* mem_frames_next               (frame_info_t *frame);
proc_pages_info_t* get_proc_pages           (gint pid,
                                             gboolean creat);
proc_pages_info_t* get_proc_pages_list      (void);
proc_pages_info_t* proc_pages_next          (proc_pages_info_t *pages);
gint        virt_to_phys                    (gint pid,
                                             gint page);
void        mem_page_invalid                (gint pid,
                                             gint page);
gint        mem_assign_frame                (gint pid,
                                             gint page,
                                             gint frame);
void        mem_page_valid                  (gint pid,
                                             gint page);
```

Description

Here is descrived show to find out the status of memory, the frames available, what process they belong to, valid and invalid pages...

Details

struct frame_info_t

```
typedef struct { /* what there is to know about a frame */
gint frame; /* frame number */
gint proc; /* process it belongs to or NO_PROC */
gint page; /* page it belongs to of NO_PAGE */
guint32 flags; /* See mem_frame_flags_t */
guint32 private_flags; /* algorithm dependet flags */
}frame_info_t;
```

enum mem_frame_flags_t

```
typedef enum { /* bit indexes for frame flags */
MEM_FRAME_LOCKED=0, /* frame is locked and should not be
   stolen of assigned */
MEM_FRAME_REFERENCED, /* frame has been referenced recently */
MEM_FRAME_MODIFIED /* frame is modified and should be writen to
   swap if stolen */
} mem_frame_flags_t;
```

FRAME_LOCKED()

```
#define FRAME_LOCKED(frame) (test_bit(MEM_FRAME_LOCKED, &frame-
>flags))
```

Test whether *frame* is locked.

frame : frame involved.

FRAME_REFERENCED()

```
#define FRAME_REFERENCED(frame) (test_bit(MEM_FRAME_REFERENCED, &frame->flags))
```

Test whether *frame* has been referenced.

frame : frame involved.

FRAME_MODIFIED()

```
#define FRAME_MODIFIED(frame) (test_bit(MEM_FRAME_MODIFIED, &frame->flags))
```

Test whether *frame* has been modified.

frame : frame involved.

struct proc_pages_info_t

```
typedef struct { /* memory related information for a
   process */
gint pid; /* process id of the process */
gint n_pages; /* number of pages its using */
guint32 bitmap; /* bitmap of valid pages */
```

```
frame_info_t *frame[MAX_PAGES];
/* frames where the pages are stored*/
gint block[MAX_PAGES]; /* swap blocks assigned to pages */
GSList *node; /* GSList link this struct is hanging
   from */
} proc_pages_info_t;
```

PAGE_VALID()

```
#define PAGE_VALID(proc_pages, page) (proc_pages-
>bitmap & (1<<page))
```

Test whether *page* is valid in *proc_pages*.

proc_pages : process memory information.

page : page involved.

NO_FRAME

```
#define NO_FRAME -1 /* frame number when there is no frame */
```

NO_PAGE

```
#define NO_PAGE -1 /* page number when there is no page */
```

NO_PROC

```
#define NO_PROC -1 /* process number when there is no process */
```

NO_BLOCK

```
#define NO_BLOCK -1 /* block number when there is no block */
```

init_page_info ()

```
void        init_page_info                  (void);
```

Initialize the code which keeps track of pages and frames.

get_free_frame ()

```
frame_info_t* get_free_frame                (void);
```

Returns : a free frame if any, NULL otherwise.

put_free_frame ()

```
void        put_free_frame                  (gint frame);
```

Give back a frame to be returned by *`get_free_frame`* later.

`frame :` frame to return.

have_free_frame ()

```
gboolean    have_free_frame                 (void);
```

Returns : TRUE if we have free memory frames available.

get_frame_info ()

```
frame_info_t* get_frame_info                (gint frame);
```

frame : frame involved.

Returns : *frame's information data.*

get_frames_list ()

```
frame_info_t* get_frames_list               (void);
```

Returns : the first element on the list of frame information structures.

mem_frames_next ()

```
frame_info_t* mem_frames_next               (frame_info_t *frame);
```

frame : a frame information structure.

Returns : the next frame information structure or NULL if *frame is the last element.*

get_proc_pages ()

```
proc_pages_info_t* get_proc_pages           (gint pid,
                                             gboolean creat);
```

Retrives the memory related information for *pid*.

If there is no memory information for *pid* and *creat* is TRUE the information will be created.

pid : process involved.

creat : it TRUE a process information structure will be created.

Returns : memory information for *pid if applyable of NULL otherwise.*

get_proc_pages_list ()

```
proc_pages_info_t* get_proc_pages_list      (void);
```

Returns : the first element of the memory infomation structures.

proc_pages_next ()

```
proc_pages_info_t* proc_pages_next          (proc_pages_info_t *pages);
```

pages : a process' memory information structure.

Returns : then next memory information structure or NULL if *pages is the last element.*

virt_to_phys ()

```
gint        virt_to_phys                    (gint pid,
                                             gint page);
```

pid : process involved.

page : a page in *pid's virtual memory.*

Returns : the frame number coresponding to *page in pid's address space or NO_FRAME if there is non assigned.*

mem_page_invalid ()

```
void        mem_page_invalid                (gint pid,
                                             gint page);
```

Makes *page* of *pid*'s address space invalid so *pid* will incure a page fault if it tryes to use it.

pid : process involved.

page : invalid page.

mem_assign_frame ()

```
gint        mem_assign_frame                (gint pid,
                                             gint page,
                                             gint frame);
```

Assings *frame* to *page* in *pid*'s address space.

pid :	process involved.
page :	a page in *pid's address space.*
frame :	a free frame.
Returns :	0 if all went well -1 otherwise.

mem_page_valid ()

```
void        mem_page_valid                  (gint pid,
                                             gint page);
```

Makes *page* of *pid*'s address space valid for *pid* to use.

pid :	process involved.
page :	valid page.

Memory Miscelaneous

Name

Memory Miscelaneous — Other interesting functions.

Synopsis

```
#define     MAX_FRAMES
#define     MAX_PAGES
gint        mem_server_init                 (GladeXML *xml);
gint        mem_page_ready_server           (gint pid,
                                             gint page);
gint        mem_page_bitmap_update_server   (gint pid,
                                             guint32 new_page_bitmap);
```

Description

This functions wouldn't make a proper section, but I believe that they are relevant enough to be mencioned.

Details

MAX_FRAMES

```
#define MAX_FRAMES 4 /* total number of frames available */
```

MAX_PAGES

```
#define MAX_PAGES 32 /* maximun number of pages per process */
```

mem_server_init ()

```
gint        mem_server_init                 (GladeXML *xml);
```

Called from the subsystem's code to initialize the interface code.

xml : Glade interface object.
Returns : nothing important.

mem_page_ready_server ()

```
gint        mem_page_ready_server           (gint pid,
                                             gint page);
```

Called from the subsystem's code to tell the client about a fulfilled page fault.

pid : process involved.
page : ready page.
Returns : nothing important.

mem_page_bitmap_update_server ()

```
gint        mem_page_bitmap_update_server   (gint pid,
                                             guint32 new_page_bitmap);
```

Called from the subsystem's code to update the valid page bitmap on the client.

pid : process involved.
new_page_bitmap : valid page bitmap.
Returns : nothing important.

2.4. Clock

This subsystem is responsable for generating a common time reference for all other subsystems.

Clock Interface

Name

`Clock Interface` — CLOCK interface to the other subsystems.

Synopsis

```
gint        (*tick_callback_t)              (gint time);
gint        clock_register_tick             (tick_callback_t func);
gint        get_time                        (void);
void        CLOCK_stop                      (void);
```

Description

This functions allow the other subsystems to interact, througth the messaging service, with the CLOCK.

Details

tick_callback_t ()

```
gint        (*tick_callback_t)              (gint time);
```

function pointer to be used with *`clock_reguster_tick`*.

`time :` current time in "time units".
Returns : nothing important.

clock_register_tick ()

```
gint        clock_register_tick             (tick_callback_t func);
```

Instructs the CLOCK subsystem to call func for every "time unit".

`func :` function to be called as time goes by.
Returns : nothing important.

get_time ()

```
gint        get_time                        (void);
```

Returns : the current time in "time units".

CLOCK_stop ()

```
void        CLOCK_stop                      (void);
```

Tells the CLOCK to stop counting "time units".

2.5. Requestor

This subsystem is not important for the general understanding of the program, but is included here for completnes. It is there to allow the user to request I/O data and Memory manualy acting as the client for those subsystems.

Chapter 3. Helpers

Scheduler

Name

`Scheduler` — Scheduling facility

Synopsis

```
gint        sched_init                  (void);
gpointer    sched_event                 (gint sched_time,
                                         sched_callback_t func,
                                         gpointer data,
                                         gint flags);
gpointer    sched_delay                 (gint delay,
                                         sched_callback_t func,
                                         gpointer data,
                                         gint flags);
void        (*sched_callback_t)         (gint time,
                                         gpointer data);

enum        sched_flags_t;
```

Description

This facility allows the code to do things at certain times without having to handle and count all the clock's TICK events.

Details

sched_init ()

```
gint        sched_init                      (void);
```

Initializes the scheduling facility. Mainly registering a callback for the clock subsystem.

Returns : nothing interesting.

sched_event ()

```
gpointer    sched_event                     (gint sched_time,
                                             sched_callback_t func,
                                             gpointer data,
                                             gint flags);
```

Function *`func`* will be called at time *`sched_time`* with arguments *`sched_time`* and *`data`*.

`sched_time :` absolute time as seen by the clock subsystem.

`func :` function to be called when the time comes.

`data :` will be used as the second argument to *`func.`*

`flags :` can be *`FREE_SCHED_DATA from sched_flags_t.`*

Returns : a pointer which is now useless from outsite but in the future may allow for an event to be modified or canceled.

sched_delay ()

```
gpointer    sched_delay                     (gint delay,
                                             sched_callback_t func,
                                             gpointer data,
                                             gint flags);
```

function *func* will be called within *delay* time units with the current time as first argument and *data* as the second.

delay : relative time from now.

func : function to be called when the time comes.

data : will be used as the second argument to *func.*

flags : can be *FREE_SCHED_DATA and SCHED_RELOAD from sched_flags_t.*

Returns : a pointer which is now useless from outsite but in the future may allow for an event to be modified or canceled.

sched_callback_t ()

```
void        (*sched_callback_t)             (gint time,
                                             gpointer data);
```

Funtion pointer type for the callbacks used in the sched facility.

time : current time.

data : data pointer specified when requesting the event.

enum sched_flags_t

```
typedef enum {
FREE_SCHED_DATA = 1<<0, /* The data pointer will be used
```

```
 * as argument to g_free() when done*/

SCHED_RELOAD
= 1<<1  /* The callback will be called every "delay"
 * time units */
} sched_flags_t;
```

Flags for the scheduling facility

Property Widget Facility

Name

`Property Widget Facility` — Easy property widget creation.

Synopsis

```
struct      property_t;
properties_t* properties_create            (const prop-
erty_t property[],
                                            const gint num_properties,
                                            void (*no-
tify) (void));
gint        properties_destroy             (proper-
ties_t *properties);
GtkWidget*  properties_get_widget          (const proper-
ties_t *properties);
```

```
gfloat*     properties_get_values           (const proper-
ties_t *properties);
gint        properties_set_values           (proper-
ties_t *properties,
                                             const gfloat value[]);
struct      properties_t;
```

Description

This functions allow the creation and management of a widget to show and get numerical properties from the user without having to learn how to create a widget with Gtk+. It is specialy usefull within algorithms allowing the user with no Gtk+ knowledge to write algorithms which properties.

Details

struct property_t

```
typedef struct {
gchar *label; /* Some text to identify de value */
gfloat min; /* The minimun value */
gfloat max; /* The maximun value */
gfloat step; /* Value increments allowed */
} property_t;
```

Descrives a single property value.

properties_create ()

```
properties_t* properties_create             (const prop-
erty_t property[],
```

```
                                                const gint num_properties,
                                                void (*no-
tify) (void));
```

Creates a GtkWidget for the properties described in *property*.

property : properties descriptions.
num_properties : number of elemente in *property*.
notify : to be called when the properties change.
Returns : a pointer which identifies the newly created properties_t.

properties_destroy ()

```
gint        properties_destroy              (properties_t *prop-
erties);
```

Destroys the properties widget, freeing all it's memory.

properties : a properties pointer obtained with properties_create.
Returns : nothing interesting.

properties_get_widget ()

```
GtkWidget*  properties_get_widget           (const proper-
ties_t *properties);
```

Retrives the widget created to hold all the properties.

properties : a properties pointer obtained with properties_create.

Returns : A pointer to the widget.

properties_get_values ()

```
gfloat*     properties_get_values           (const proper-
ties_t *properties);
```

Retrives the current values of all the properties represented in *properties*.

properties : a properties pointer obtained with properties_create.
Returns : An array with the values.

properties_set_values ()

```
gint        properties_set_values           (properties_t *prop-
erties,
                                             const gfloat value[]);
```

Sets the values of all the properties stored in *properties*.

properties : a properties pointer obtained with properties_create.
value : and array with the values.
Returns : nothing interesting.

struct properties_t

```
typedef struct {
```

```
void (*notify)(void); /* will be called when ever a properties
 * change.*/
GtkWidget *widget; /* is the main widget.*/
gint num_properties; /* number of entry and value elements */
GtkWidget **entry; /* is an array of all entry widgets */
gfloat *value; /* is an array of all property values */
} properties_t;
```

Descrives everything there is to know about a properties widget.

NOTE: it's fields should not be accessed directly.

Drawings

Name

`Drawings` — Multiple representation facility.

Synopsis

```
struct      drawing_style_t;
enum        drawing_flags_t;
GtkWidget*  create_drawing                  (void);
void        register_drawing_style          (GtkWidget *drawing,
                                             draw-
ing_style_t *style);
void        update_drawing                  (GtkWidget *draw-
ing);
```

Description

This allow easy impletentation of multiple representations for the same subsystem.

Details

struct drawing_style_t

```
typedef struct { /* descrives a drawing style */
GtkWidget *widget; /* widget to be shown */
gchar *name; /* name for this style */
void (*update)(GtkWidget *widget); /* function to update the
   style */
drawing_flags_t flags; /* one of drawing_flags_t */
} drawing_style_t;
```

enum drawing_flags_t

```
typedef enum {
DRAWING_FIXED_SIZE=1, /* the widget will not change it’s
   size scrollbars should be used */
DRAWING_FIXED_RATIO=1<<1 /* the widget will change it’s size
   if needed but wants to keep it’s
   width/height ratio */
} drawing_flags_t;
```

create_drawing ()

```
GtkWidget*  create_drawing                  (void);
```

Creates a widget capable of containing multiple representations for a certain subsystem.

Returns : a widget ready to be handled with the apropiate GTK+ functions.

register_drawing_style ()

```
void        register_drawing_style          (GtkWidget *drawing,
                                             draw-
ing_style_t *style);
```

Adds *style* to *drawing*.

drawing : a widget returned by *create_drawing.*

style : the structure descriving a drawing style.

update_drawing ()

```
void        update_drawing                  (GtkWidget *draw-
ing);
```

Updates the styles (representations) in a "drawing".

drawing : a widget returned by create_drawing.

Gdk

Name

Gdk — Gdk addons

Synopsis

```
GdkGC*      new_gdk_GC_with_color          (guint8 red,
                                            guint8 green,
                                            guint8 blue);
void        resize_gdk_pixmap              (GdkPixmap **pixmap,
                                            gint new_width,
                                            gint new_height,
                                            GdkGC *fill);
void        enlarge_gdk_pixmap             (GdkPixmap **pixmap,
                                            gint new_width,
                                            gint new_height,
                                            GdkGC *fill);
void        draw_gdk_text_centered         (GdkDraw-
able *drawable,
                                            GdkFont *font,
                                            GdkGC *gc,
                                            gint x,
                                            gint y,
                                            gint width,
                                            gint height,
                                            const gchar *text,
                                            gint text_length);
void        fill_gdk_window                (GdkPixmap *pixmap,
                                            GdkGC *fill);
```

Description

This is the functionality which I wanted from GDK but wasn't there.

Details

new_gdk_GC_with_color ()

```
GdkGC*      new_gdk_GC_with_color           (guint8 red,
                                             guint8 green,
                                             guint8 blue);
```

Creates a new GdkGC (Graphic Context) with the foreground color set to the color descrived by *red*, *green* and *blue* components.

red :	red component of the color.
green :	green component of the color.
blue :	blue component of the color.
Returns :	the new Graphic Context.

resize_gdk_pixmap ()

```
void        resize_gdk_pixmap               (GdkPixmap **pixmap,
                                             gint new_width,
                                             gint new_height,
                                             GdkGC *fill);
```

Will resize **pixmap* and, if enlarging, will fill the extra area with the foreground color of *fill*.

If the size realy changes, it will copy the old pixmap to a new one, destroing it and setting *`pixmap` to the new pixmap.

`pixmap :` pixmap to resize.
`new_width :` the new width of the pixmap.
`new_height :` the new height of the pixmap.
`fill :` a GdkGC with the foreground color set.

enlarge_gdk_pixmap ()

```
void        enlarge_gdk_pixmap              (GdkPixmap **pixmap,
                                             gint new_width,
                                             gint new_height,
                                             GdkGC *fill);
```

Same as `resize_gdk_pixmap` but will only do the resizing if the requested size is bigger then the current size.

This is usefull for eficiency. See src/MEM/drawings/virtual.c:draw_page_tables for an example.

`pixmap :` pixmap to resize.
`new_width :` the new width of the pixmap.
`new_height :` the new height of the pixmap.
`fill :` a GdkGC with the foreground color set.

draw_gdk_text_centered ()

```
void        draw_gdk_text_centered          (GdkDrawable *draw-
able,
```

```
                                                GdkFont *font,
                                                GdkGC *gc,
                                                gint x,
                                                gint y,
                                                gint width,
                                                gint height,
                                                const gchar *text,
                                                gint text_length);
```

Draw *text* of length *text_length* centered in the area defined by *x*, *y*, *width* and *height* of *drawable* using font *font* and the foreground color of *gc*.

drawable : a GdkDrawable.
font : any font.
gc : a GdkGC with the foreground color set.
x : x coordinate of the area.
y : y coordinate of the area.
width : width of the area.
height : height of the area.
text : text to draw.
text_length length of *text.*

fill_gdk_window ()

```
void        fill_gdk_window                 (GdkPixmap *pixmap,
                                             GdkGC *fill);
```

Paint all the surface of *pixmap* with the foreground color of *fill*.

pixmap : pixmap to fill.
fill : a GdkGC with foreground color set.

Glib

Name

`Glib` — Glib addons

Synopsis

```
GSList*     g_slist_dup                     (GSList *list);
```

Description

This is the functionality which I wanted from Glib but wasn't there.

Details

g_slist_dup ()

```
GSList*     g_slist_dup                     (GSList *list);
```

Duplicates *`list`*.

`list :` GSList to duplicate.

Returns : the new list.

Messaging

Name

`Messaging` — Messaging facility.

Synopsis

```
struct      message_t;
#define     MESSAGE_MAXSIZE
enum        subsystem_t;
enum        mesg_message_type_t;
enum        misc_message_type_t;
void        (*receive_callback)             (const mes-
sage_t *m);
gint        mesg_pre_setup                  (subsys-
tem_t subsystem);
gint        mesg_messanger_setup            (subsys-
tem_t subsystem);
gint        mesg_subsystem_setup            (subsys-
tem_t subsystem,
                                             gint flags);
```

```
gint        mesg_quit                       (subsys-
tem_t subsystem);
gint        mesg_send                       (subsystem_t dest,
                                             mesg_type_t type,
                                             const gpointer data,
                                             gint size);
gint        mesg_broadcast                  (mesg_type_t type,
                                             const gpointer data,
                                             gint size);
gboolean    mesg_loop                       (void);
gint        mesg_callback_register          (mesg_type_t type,
                                             re-
ceive_callback func);
subsystem_t mesg_who_am_i                   (void);
gchar*      mesg_subsystem_name             (subsys-
tem_t subsystem);
gint        mesg_block                      (void);
gint        mesg_unblock                    (void);
```

Description

This could be seen as another subsystem, and in deed is a different process. It makes comunication between the other subsystem posible in a transparent maner, currently it uses sockets but it could use any means of comunication without modifing any other code.

Details

struct message_t

```
typedef struct { /* struct describing a message */
subsystem_t sender; /* sender of the message */
```

```
subsystem_t dest; /* target of the message */
mesg_type_t type; /* type of message */
gint data_size; /* size of the data */
gint8 data[MESSAGE_MAXSIZE]; /* data of the message */
} message_t;
```

MESSAGE_MAXSIZE

```
#define MESSAGE_MAXSIZE 8
/* maximun size of the data to send in a message */
```

enum subsystem_t

```
typedef enum {
/* identification for the subsystems will also be used
   as the higher byte for message types */
ALL=-2, /* broadcast messages */
NOSUBSYSTEM=-1,
/* used within the messaging code to indicate that
   there is no subsystem */
MESG=0, /* messages to be handled by the messager */
CPU, /* CPU subsystem */
MEM, /* Memory subsystem */
IO, /* I/O subsystem */
CLOCK, /* Clock subsystem */
REQUESTOR, /* Requestor subsystem */
N_SUBSYSTEMS,
/* used within the messaging code as the total number
   of subsystems */
MISC_MESSAGES /* messages that don't belong to any subsystem in
   particular */
} subsystem_t;
```

enum mesg_message_type_t

```
typedef enum { /* message types to be handled by the
   messager */
MESG_QUIT=MESG<<8, /* each subsystem should send this message
   to the messenger before quiting */
MESG_RESET_SYSTEM /* send by the CPU tells the messager to
   terminate all subsystems and restart
   the system */
} mesg_message_type_t;
```

enum misc_message_type_t

```
typedef enum { /* message types not belonging to a
   particular subsystem */
MISC_SHOW=MISC_MESSAGES<<8, /* tells a subsystem to show it’s
   main window */
MISC_QUIT /* tells a subsystem to quit */
} misc_message_type_t;
```

receive_callback ()

```
void        (*receive_callback)            (const mes-
sage_t *m);
```

Function pointer type to be used in mesg_callback_register.

m : message.

mesg_pre_setup ()

```
gint        mesg_pre_setup                  (subsystem_t subsys-
tem);
```

Should be called before the fork. It registers the subsystem and gets ready for comunication.

subsystem : the subsystem ID.

Returns : nothing important.

mesg_messanger_setup ()

```
gint        mesg_messanger_setup            (subsystem_t subsys-
tem);
```

Should be called after the fork on the parent side. Finishes setting up comunications.

subsystem : the subsystem ID.

Returns : nothing important.

mesg_subsystem_setup ()

```
gint        mesg_subsystem_setup            (subsystem_t subsys-
tem,
                                             gint flags);
```

Should be called after the fork on the child side. Finishes setting up comunications.

If *flags* is MESG_WITH_GTK GTK I/O monitoring facilities will be used.

NOTE: GTK+ I/O monitoring facilities can only be used if the subsystem uses GTK+ and calls *gtk_main*.

subsystem : the subsystem ID.
flags : could be MESG_WITH_GTK or 0.
Returns : nothing important.

mesg_quit ()

```
gint        mesg_quit                       (subsystem_t subsys-
tem);
```

Should be called by all subsystems before quitting.

subsystem : ID of the subsystem.
Returns : nothing important.

mesg_send ()

```
gint        mesg_send                       (subsystem_t dest,
                                             mesg_type_t type,
                                             const gpointer data,
                                             gint size);
```

Send a message from one subsystem to another.

dest : target subsystem ID.
type : type for the message.
data : data for the message.

size : size of *data.*
Returns : nothing important.

mesg_broadcast ()

```
gint        mesg_broadcast                  (mesg_type_t type,
                                             const gpointer data,
                                             gint size);
```

Send a message from one subsystem to all the other subsystems.

type : type for the message.
data : data for the message.
size : size of *data.*
Returns : nothing important.

mesg_loop ()

```
gboolean    mesg_loop                       (void);
```

Function to be called last on the messanger process, it will handle messages between the different subsystems and won't return until the system is reset or terminated.

Returns : TRUE if the system should be reset insted of terminated.

mesg_callback_register ()

```
gint        mesg_callback_register          (mesg_type_t type,
                                             re-
ceive_callback func);
```

Registers the function *`func`* to be called when messages of type *`type`* are received.

`type :` message type.
`func :` function to be called.
Returns : nothing important.

mesg_who_am_i ()

```
subsystem_t mesg_who_am_i                   (void);
```

This is usefull so the code which is usable by all subsystems can find out where it is executing.

This is specialy usefull in combination with *`mesg_subsystem_name`*.

Returns : the identification number of the current subsystem.

mesg_subsystem_name ()

```
gchar*      mesg_subsystem_name             (subsystem_t subsys-
tem);
```

The string returned should not be modified of freed.

`subsystem :` the identification number of a subsystem.
Returns : a string with the name of the subsystem.

mesg_block ()

```
gint        mesg_block                          (void);
```

In case GTK I/O monitoring facilities are not been used, this function can be used to protect critical sections of code from been interrupted by I/O signals.

Returns : nothing important.

mesg_unblock ()

```
gint        mesg_unblock                        (void);
```

Restarts normal messaging behaviour after calling *`mesg_block`*.

Returns : nothing important.

System Events

Name

`System Events` — Event dispatching facility.

Synopsis

```
enum        sys_event_t;
void        system_event                    (sys_event_t type,
                                             gpointer data);
void        system_event_receive            (sys_event_t type,
                                             sys_event_callback func);
```

Description

This facility allows for non related code to be informed of some events.

For an example look at: src/CPU/drawings/bars.c, src/CPU/drawings/overlapped_bars.c or /src/CPU/drawings/state.c

Details

enum sys_event_t

```
typedef enum {
SYS_EVENT_PROC_CREATE=0, /* there is a new process */
SYS_EVENT_PROC_DESTROY, /* a process terminated */
SYS_EVENT_PROC_READY, /* a process is now ready to run */
SYS_EVENT_PROC_QUEUED, /* a process put in a ready queue */
SYS_EVENT_PROC_RUNNING, /* a process is now running */
SYS_EVENT_PROC_WAITING, /* a process is now blocked */
SYS_EVENT_PROC_SELECT, /* a process has been selected by
   the user */
SYS_EVENT_FRAME_SELECT /* the user selected a memory frame */
}sys_event_t;
```

system_event ()

```
void        system_event                    (sys_event_t type,
                                             gpointer data);
```

Generate a system event of type *`type`* with argument *`data`*.

`type :` type of the event.
`data :` agument of the event.

system_event_receive ()

```
void        system_event_receive            (sys_event_t type,
                                             sys_event_callback func);
```

Instruct the system events code to call *`func`* when ever an event of type *`type`* ocurs.

`type :` type of event to receive.
`func :` function to call.

BitOps

Name

`BitOps` — Bitmap operations.

Synopsis

```
void        set_bit                         (gint nr,
                                             guint32 *addr);
gboolean    test_bit                        (gint nr,
                                             guint32 *addr);
void        clear_bit                       (gint nr,
                                             guint32 *addr);
```

Description

Confortable functions to set, test and clear bits in a bitmap.

Details

set_bit ()

```
void        set_bit                         (gint nr,
                                             guint32 *addr);
```

Set bit number *nr* in the bitmap pointed by *addr*.

nr : index of the bit to be set to 1.

addr : address of the bitmap.

test_bit ()

```
gboolean    test_bit                        (gint nr,
```

```
                                                guint32 *addr);
```

Test if bit number *nr* of the bitmap pointed by *addr* is 1 or 0.

nr :	index of the bit to be tested.
addr :	address of the bitmap.
Returns :	TRUE if the bit is 1 and FALSE otherwise.

clear_bit ()

```
void        clear_bit                       (gint nr,
                                             guint32 *addr);
```

Clear bit number *nr* in the bitmap pointed by *addr*.

nr :	index of the bit to be cleard to 0.
addr :	address of the bitmap.

black