HellfireOS v2.0

Generated by Doxygen 1.8.1.2

Tue Aug 23 2016 19:41:09

Contents

1	Data	Struct	ure Index													1
	1.1	Data S	Structures					 	 	 	 	 	 			1
2	File	Index														3
	2.1	File Lis	st					 	 	 	 	 	 			3
3	Data	Struct	ure Docun	nenta	ation											5
	3.1	condva	ar Struct Ro	efere	nce .			 	 	 	 	 	 			5
		3.1.1	Detailed	Desc	ription			 	 	 	 	 	 			5
		3.1.2	Field Doo	cume	ntation	١		 	 	 	 	 	 			5
			3.1.2.1	con	d_que	ue .		 	 	 	 	 	 			5
			3.1.2.2	mut	tex .			 	 	 	 	 	 			6
	3.2	list Str	uct Refere	nce .				 	 	 	 	 	 			6
		3.2.1	Detailed	Desc	ription			 	 	 	 	 	 			6
		3.2.2	Field Doo	cume	ntation	١		 	 	 	 	 	 			6
			3.2.2.1	eler	m			 	 	 	 	 	 			6
			3.2.2.2	nex	t			 	 	 	 	 	 			6
	3.3	mailbo	x Struct Re	efere	nce .			 	 	 	 	 	 			6
		3.3.1	Detailed	Desc	ription			 	 	 	 	 	 			7
		3.3.2	Field Doo	cume	ntation	١		 	 	 	 	 	 			7
			3.3.2.1	cou	nt			 	 	 	 	 	 			7
			3.3.2.2	mre	CV .			 	 	 	 	 	 			7
			3.3.2.3	mse	end .			 	 	 	 	 	 			7
			3.3.2.4	msg	j			 	 	 	 	 	 			8
			3.3.2.5	n_w	vaiting_	_task	s.	 	 	 	 	 	 			8
	3.4	mem_l	block Struc	ct Ref	ierence			 	 	 	 	 	 			8
		3.4.1	Field Doo	cume	ntation	١		 	 	 	 	 	 			8
			3.4.1.1	nex	t			 	 	 	 	 	 			8
			3.4.1.2	size				 	 	 	 	 	 			8
	3.5	mem_	chunk Stru	ıct Re	eferenc	e		 	 	 	 	 	 			8
		3.5.1	Field Doo	cume	ntation	١		 	 	 	 	 	 			9
			0=44													_

ii CONTENTS

3.6	mem_c	chunk_ptr	Struct Reference	 !	9
	3.6.1	Field Doo	cumentation	 !	9
		3.6.1.1	free	 !	9
		3.6.1.2	heap	 !	9
3.7	mem_h	neader_un	nion Union Reference	 	9
	3.7.1	Field Doo	cumentation	 1	0
		3.7.1.1	align_dummy	 1	0
		3.7.1.2	next	 1	0
		3.7.1.3	s	 1	0
		3.7.1.4	size	 1	0
3.8	mtx Str	uct Refere	ence	 1	0
	3.8.1	Detailed	Description	 1	0
	3.8.2	Field Doo	cumentation	 1	1
		3.8.2.1	level	 1	1
		3.8.2.2	lock	 1	1
		3.8.2.3	waiting	 1	1
3.9	pcb_en	ntry Struct	t Reference	 1	1
	3.9.1	Field Doo	cumentation	 1	1
		3.9.1.1	coop_cswitch	 1	1
		3.9.1.2	interrupts	 1	1
		3.9.1.3	preempt_cswitch	 1	1
3.10	queue	Struct Ref	ference	 1	1
	3.10.1	Detailed	Description	 1	2
	3.10.2	Field Doo	cumentation	 1	2
		3.10.2.1	data	 1	2
		3.10.2.2	elem	 1	2
		3.10.2.3	head	 1	2
		3.10.2.4	size	 1	2
		3.10.2.5	tail	 1	2
3.11	sem St	ruct Refer	rence	 1	2
	3.11.1	Detailed	Description	 1	3
	3.11.2	Field Doo	cumentation	 1	3
		3.11.2.1	count	 1	3
		3.11.2.2	sem_queue	 1	3
3.12	tcb_ent	try Struct I	Reference	 1	3
	3.12.1	Detailed	Description	 1	4
	3.12.2	Field Doo	cumentation	 1	4
		3.12.2.1	bgjobs	 1	4
		3.12.2.2	capacity	 1	4
		3.12.2.3	capacity_rem	 1	4

CONTENTS

			3.12.2.4 deadline	14
			3.12.2.5 deadline_misses	14
			3.12.2.6 deadline_rem	14
			3.12.2.7 delay	14
			3.12.2.8 id	15
			3.12.2.9 name	15
			3.12.2.10 other_data	15
			3.12.2.11 period	15
			3.12.2.12 pstack	15
			3.12.2.13 ptask	15
			3.12.2.14 rtjobs	15
			3.12.2.15 stack_size	15
			3.12.2.16 state	15
			3.12.2.17 task_context	15
4	File	Docum	entation	17
•	4.1			17
	4.1	4.1.1		18
		4.1.2		18
		4.1.3		18
		4.1.4		18
				18
				18
				19
			- -	19
			-	19
				19
			-	20
			-	20
			4.1.4.9 ni_init	20
			4.1.4.10 ni_isr	21
		4.1.5	Variable Documentation	21
			4.1.5.1 pktdrv_ports	21
			4.1.5.2 pktdrv_queue	21
			4.1.5.3 pktdrv_tqueue	21
	4.2	drivers	/noc/noc.c File Reference	21
		4.2.1	Detailed Description	22
		4.2.2	LICENSE	22
		4.2.3	DESCRIPTION	22
		4.2.4	Function Documentation	22

iv CONTENTS

		4.2.4.1	hf_comm_create	22
		4.2.4.2	hf_comm_destroy	23
		4.2.4.3	hf_cpuid	23
		4.2.4.4	hf_ncores	23
		4.2.4.5	hf_recv	23
		4.2.4.6	hf_recvack	23
		4.2.4.7	hf_send	24
		4.2.4.8	hf_sendack	24
		4.2.4.9	ni_init	25
		4.2.4.10	ni_isr	25
4.3	sys/inc	lude/cond	var.h File Reference	25
	4.3.1	Typedef I	Documentation	26
		4.3.1.1	$cond_t \ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$	26
	4.3.2	Function	Documentation	26
		4.3.2.1	hf_condbroadcast	26
		4.3.2.2	hf_conddestroy	26
		4.3.2.3	hf_condinit	26
		4.3.2.4	hf_condsignal	26
		4.3.2.5	hf_condwait	27
4.4	sys/inc	lude/ecod	es.h File Reference	27
4.5	sys/inc	lude/hellfir	re.h File Reference	27
	4.5.1	Detailed	Description	27
	4.5.2	LICENSE	:	27
	4.5.3	DESCRI	PTION	27
4.6	sys/inc	lude/kerne	el.h File Reference	27
	4.6.1	Detailed	Description	28
	4.6.2	LICENSE	E	28
	4.6.3	DESCRI	PTION	28
	4.6.4	Variable	Documentation	28
		4.6.4.1	krnl_current_task	28
		4.6.4.2	krnl_delay_queue	28
		4.6.4.3	krnl_event_queue	28
		4.6.4.4	krnl_free	28
		4.6.4.5	krnl_heap	28
		4.6.4.6	krnl_pcb	29
		4.6.4.7	krnl_rt_queue	29
		4.6.4.8	krnl_run_queue	29
		4.6.4.9	krnl_schedule	29
		4.6.4.10	krnl_task	29
		4.6.4.11	krnl_tasks	29

CONTENTS

		4.6.4.12	krnl_tcb	. 29
4.7	sys/inc	lude/kprint	tf.h File Reference	. 29
	4.7.1	Function	Documentation	. 29
		4.7.1.1	dprintf	. 29
		4.7.1.2	kprintf	. 30
4.8	sys/inc	lude/list.h	File Reference	. 30
	4.8.1	Function	Documentation	. 30
		4.8.1.1	hf_list_append	. 30
		4.8.1.2	hf_list_count	. 31
		4.8.1.3	hf_list_get	. 31
		4.8.1.4	hf_list_init	. 31
		4.8.1.5	hf_list_insert	. 31
		4.8.1.6	hf_list_remove	. 31
		4.8.1.7	hf_list_set	. 32
4.9	sys/inc	lude/mailb	oox.h File Reference	. 32
	4.9.1	Typedef I	Documentation	. 32
		4.9.1.1	mail_t	. 32
	4.9.2	Function	Documentation	. 32
		4.9.2.1	hf_mboxaccept	. 32
		4.9.2.2	hf_mboxinit	. 32
		4.9.2.3	hf_mboxrecv	. 32
		4.9.2.4	hf_mboxsend	. 32
4.10	sys/inc	lude/main.	.h File Reference	. 33
	4.10.1	Function	Documentation	. 33
			app_main	
4.11	sys/inc	lude/mallo	oc.h File Reference	. 33
	4.11.1	Detailed	Description	. 33
	4.11.2	LICENSE	E	. 33
			PTION	
	4.11.4		Documentation	
			align	
			mem_header_t	
	4.11.5	Function	Documentation	
		4.11.5.1		
		4.11.5.2		
		4.11.5.3	-	
		4.11.5.4	hf_malloc	
			hf_realloc	
	4.11.6		Documentation	
		4.11.6.1	ff	. 34

vi CONTENTS

		4.11.6.2	krnl_heap_ptr	. 34
4.12	sys/incl	lude/mute	x.h File Reference	. 34
	4.12.1	Typedef I	Documentation	. 35
		4.12.1.1	mutex_t	. 35
	4.12.2	Function	Documentation	. 35
		4.12.2.1	hf_mtxinit	. 35
		4.12.2.2	hf_mtxlock	. 35
		4.12.2.3	hf_mtxunlock	. 35
4.13	sys/incl	lude/panic	c.h File Reference	. 35
	4.13.1	Function	Documentation	. 35
		4.13.1.1	panic	. 35
4.14	sys/incl	lude/proce	essor.h File Reference	. 35
	4.14.1	Function	Documentation	. 36
		4.14.1.1	hf_cpuload	. 36
		4.14.1.2	hf_freecpu	. 36
		4.14.1.3	hf_freemem	. 36
		4.14.1.4	hf_schedlock	. 36
4.15	sys/incl	lude/queu	e.h File Reference	. 37
	4.15.1	Function	Documentation	. 37
		4.15.1.1	hf_queue_addtail	. 37
		4.15.1.2	hf_queue_count	. 37
		4.15.1.3	hf_queue_create	. 38
		4.15.1.4	hf_queue_destroy	. 38
		4.15.1.5	hf_queue_get	. 38
		4.15.1.6	hf_queue_remhead	. 38
		4.15.1.7	hf_queue_remtail	. 39
		4.15.1.8	hf_queue_set	. 39
		4.15.1.9	hf_queue_swap	. 39
4.16	sys/incl	lude/sched	duler.h File Reference	. 39
	4.16.1	Function	Documentation	. 40
		4.16.1.1	dispatch_isr	. 40
		4.16.1.2	sched_be	. 40
		4.16.1.3	sched_rt	. 40
4.17	sys/incl	lude/sema	aphore.h File Reference	. 41
	4.17.1	Typedef [Documentation	. 41
		4.17.1.1	$sem_t \dots $. 41
	4.17.2	Function	Documentation	. 41
		4.17.2.1	hf_semdestroy	. 41
		4.17.2.2	hf_seminit	. 41
		4.17.2.3	hf_sempost	. 42

CONTENTS vii

		4.17.2.4 hf_semwait	12
4.18	sys/inc	lude/task.h File Reference	12
	4.18.1	Function Documentation	43
		4.18.1.1 hf_block	43
		4.18.1.2 hf_delay	43
		4.18.1.3 hf_dlm	43
		4.18.1.4 hf_id	14
		4.18.1.5 hf_jobs	14
		4.18.1.6 hf_kill	14
		4.18.1.7 hf_name	14
		4.18.1.8 hf_resume	14
		4.18.1.9 hf_selfid	45
		4.18.1.10 hf_selfname	45
		4.18.1.11 hf_spawn	45
		4.18.1.12 hf_state	46
		4.18.1.13 hf_yield	46
4.19	sys/ker	nel/main.c File Reference	46
	4.19.1	Detailed Description	46
	4.19.2	LICENSE	46
	4.19.3	DESCRIPTION	46
	4.19.4	Function Documentation	46
		4.19.4.1 main	46
4.20	sys/ker	nel/panic.c File Reference	47
	4.20.1	Detailed Description	47
	4.20.2	LICENSE	47
	4.20.3	DESCRIPTION	47
	4.20.4	Function Documentation	17
		4.20.4.1 panic	47
4.21	sys/ker	nel/processor.c File Reference	47
	4.21.1	Detailed Description	48
	4.21.2	LICENSE	48
	4.21.3	DESCRIPTION	48
	4.21.4	Function Documentation	48
		4.21.4.1 hf_cpuload	1 8
		4.21.4.2 hf_freecpu	1 8
		4.21.4.3 hf_freemem	49
		4.21.4.4 hf_schedlock	49
4.22	sys/ker	nel/scheduler.c File Reference	49
	4.22.1	Detailed Description	49
	4.22.2	LICENSE	49

viii CONTENTS

	4.22.3	DESCRIPTION	49
	4.22.4	Function Documentation	49
		4.22.4.1 dispatch_isr	49
		4.22.4.2 sched_be	50
		4.22.4.3 sched_rt	50
4.23	sys/ker	rnel/task.c File Reference	51
	4.23.1	Detailed Description	51
	4.23.2	LICENSE	51
	4.23.3	DESCRIPTION	51
	4.23.4	Function Documentation	52
		4.23.4.1 hf_block	52
		4.23.4.2 hf_delay	52
		4.23.4.3 hf_dlm	52
		4.23.4.4 hf_id	52
		4.23.4.5 hf_jobs	53
		4.23.4.6 hf_kill	53
		4.23.4.7 hf_name	53
		4.23.4.8 hf_resume	53
		4.23.4.9 hf_selfid	54
		4.23.4.10 hf_selfname	54
		4.23.4.11 hf_spawn	54
		4.23.4.12 hf_state	54
		4.23.4.13 hf_yield	55
4.24	sys/lib/	kprintf.c File Reference	55
	4.24.1	Detailed Description	55
	4.24.2	LICENSE	55
	4.24.3	DESCRIPTION	55
	4.24.4	Function Documentation	55
		4.24.4.1 dprintf	55
		4.24.4.2 kprintf	56
4.25	sys/lib/	list.c File Reference	56
	4.25.1	Detailed Description	56
	4.25.2	LICENSE	56
	4.25.3	DESCRIPTION	57
	4.25.4	Function Documentation	57
		4.25.4.1 hf_list_append	57
		4.25.4.2 hf_list_count	57
		4.25.4.3 hf_list_get	57
		4.25.4.4 hf_list_init	57
		4.25.4.5 hf_list_insert	58

CONTENTS

		4.25.4.6 hf_list_remove	58
		4.25.4.7 hf_list_set	58
4.26	sys/lib/	malloc.c File Reference	58
	4.26.1	Function Documentation	58
		4.26.1.1 heapinit	59
		4.26.1.2 hf_calloc	59
		4.26.1.3 hf_free	59
		4.26.1.4 hf_malloc	59
		4.26.1.5 hf_realloc	59
4.27	sys/lib/	queue.c File Reference	59
	4.27.1	Detailed Description	59
	4.27.2	LICENSE	59
		DESCRIPTION	59
	4.27.4	Function Documentation	60
		4.27.4.1 hf_queue_addtail	60
		4.27.4.2 hf_queue_count	60
		4.27.4.3 hf_queue_create	60
		4.27.4.4 hf_queue_destroy	60
		4.27.4.5 hf_queue_get	61
		4.27.4.6 hf_queue_remhead	61
		4.27.4.7 hf_queue_remtail	61
		4.27.4.8 hf_queue_set	61
		4.27.4.9 hf_queue_swap	62
4.28		nc/condvar.c File Reference	62
		Detailed Description	62
		LICENSE	62
		DESCRIPTION	62
	4.28.4	Function Documentation	63
		4.28.4.1 hf_condbroadcast	63
		4.28.4.2 hf_conddestroy	63
		4.28.4.3 hf_condinit	63
		4.28.4.4 hf_condsignal	63
4.00	,	4.28.4.5 hf_condwait	63
4.29		nc/mutex.c File Reference	64
		Detailed Description	64
		LICENSE	64
		DESCRIPTION	64
	4.29.4	Function Documentation	64
		4.29.4.1 hf_mtxinit	64
		4.29.4.2 hf_mtxlock	64

X CONTENTS

4.29.4.3	3 hf_mtxunlock	65
4.30 sys/sync/semap	phore.c File Reference	65
4.30.1 Detailed	d Description	65
4.30.2 LICENS	SE	65
4.30.3 DESCR	RIPTION	65
4.30.4 Function	n Documentation	65
4.30.4.1	1 hf_semdestroy	65
4.30.4.2	2 hf_seminit	66
4.30.4.3	3 hf_sempost	66
4.30.4.4	4 hf_semwait	66

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

condvar
Condition variable data structure
list
List data structure
mailbox
Mailbox data structure
mem_block
mem_chunk
mem_chunk_ptr
mem_header_union
mtx
Mutex data structure
pcb_entry
queue
Queue data structure
sem
Semaphore data structure
tcb_entry
Task control block (TCB) and processor control block (PCB) entry data structures

Data Structure Index

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

drivers/noc/noc.c	21
drivers/noc/include/noc.h	17
sys/include/condvar.h	25
-,	27
sys/include/hellfire.h	27
,	27
, , , , , , , , , , , , , , , , , , , ,	29
-,	30
•	32
,	33
•	33
,	34
- , -,	35
- y	35
-,	37
•	39
-,	41
,	42
-,	46
, ,	47
, ,	47
,	49
- ,	51
	55
,	56
- ,	58
	59
, ,	62
-,,	64
sys/sync/semaphore.c	65

File Index

Chapter 3

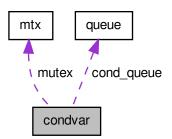
Data Structure Documentation

3.1 condvar Struct Reference

Condition variable data structure.

#include <condvar.h>

Collaboration diagram for condvar:



Data Fields

- struct queue * cond_queue
- mutex_t mutex

3.1.1 Detailed Description

Condition variable data structure.

3.1.2 Field Documentation

3.1.2.1 struct queue* condvar::cond_queue

queue for tasks waiting on the condition variable

3.1.2.2 mutex_t condvar::mutex

mutex used for the critical section associated with the condition variable

The documentation for this struct was generated from the following file:

• sys/include/condvar.h

3.2 list Struct Reference

List data structure.

#include <list.h>

Collaboration diagram for list:



Data Fields

- void * elem
- struct list * next

3.2.1 Detailed Description

List data structure.

3.2.2 Field Documentation

3.2.2.1 void* list::elem

pointer to list node data

3.2.2.2 struct list* list::next

pointer to the next list node

The documentation for this struct was generated from the following file:

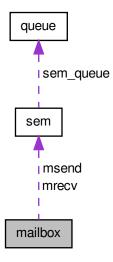
• sys/include/list.h

3.3 mailbox Struct Reference

Mailbox data structure.

#include <mailbox.h>

Collaboration diagram for mailbox:



Data Fields

- void * msg
- uint16_t n_waiting_tasks
- uint16_t count
- sem_t msend
- sem_t mrecv

3.3.1 Detailed Description

Mailbox data structure.

3.3.2 Field Documentation

3.3.2.1 uint16_t mailbox::count

number of elements on the mailbox

3.3.2.2 sem_t mailbox::mrecv

synchronization semaphore for mail receive

3.3.2.3 sem_t mailbox::msend

synchronization semaphore for mail send

3.3.2.4 void* mailbox::msg

pointer to a message buffer

3.3.2.5 uint16_t mailbox::n_waiting_tasks

number of waiting tasks in the mailbox

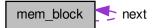
The documentation for this struct was generated from the following file:

• sys/include/mailbox.h

3.4 mem block Struct Reference

#include <malloc.h>

Collaboration diagram for mem_block:



Data Fields

- struct mem_block * next
- size_t size

3.4.1 Field Documentation

3.4.1.1 struct mem_block * mem_block::next

3.4.1.2 size_t mem_block::size

The documentation for this struct was generated from the following file:

• sys/include/malloc.h

3.5 mem_chunk Struct Reference

#include <malloc.h>

Data Fields

• uint32_t size

3.5.1 Field Documentation

3.5.1.1 uint32_t mem_chunk::size

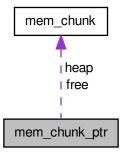
The documentation for this struct was generated from the following file:

• sys/include/malloc.h

3.6 mem_chunk_ptr Struct Reference

```
#include <malloc.h>
```

Collaboration diagram for mem_chunk_ptr:



Data Fields

- mem_chunk * free
- mem_chunk * heap

3.6.1 Field Documentation

3.6.1.1 mem_chunk* mem_chunk_ptr::free

3.6.1.2 mem_chunk* mem_chunk_ptr::heap

The documentation for this struct was generated from the following file:

• sys/include/malloc.h

3.7 mem_header_union Union Reference

#include <malloc.h>

Collaboration diagram for mem_header_union:

Data Fields

```
    struct {
        union mem_header_union * next
        uint32_t size
    } s
```

· align align_dummy

3.7.1 Field Documentation

- 3.7.1.1 align mem_header_union::align_dummy
- 3.7.1.2 union mem_header_union* mem_header_union::next
- 3.7.1.3 struct { ... } mem_header_union::s
- 3.7.1.4 uint32_t mem_header_union::size

The documentation for this union was generated from the following file:

• sys/include/malloc.h

3.8 mtx Struct Reference

```
Mutex data structure.
```

```
#include <mutex.h>
```

Data Fields

```
• int32_t lock
```

- uint8_t level [MAX_TASKS]
- uint8_t waiting [MAX_TASKS-1]

3.8.1 Detailed Description

Mutex data structure.

3.8.2 Field Documentation

3.8.2.1 uint8_t mtx::level[MAX_TASKS]

3.8.2.2 int32_t mtx::lock

mutex lock, atomically modified

3.8.2.3 uint8_t mtx::waiting[MAX_TASKS-1]

The documentation for this struct was generated from the following file:

• sys/include/mutex.h

3.9 pcb_entry Struct Reference

```
#include <kernel.h>
```

Data Fields

- uint32_t coop_cswitch
- uint32_t preempt_cswitch
- uint32_t interrupts

3.9.1 Field Documentation

3.9.1.1 uint32_t pcb_entry::coop_cswitch

cooperative context switches

3.9.1.2 uint32_t pcb_entry::interrupts

number of non-masked interrupts

3.9.1.3 uint32_t pcb_entry::preempt_cswitch

preeptive context switches

The documentation for this struct was generated from the following file:

• sys/include/kernel.h

3.10 queue Struct Reference

Queue data structure.

#include <queue.h>

Data Fields

- int32 t size
- int32_t elem
- int32_t head
- int32_t tail
- void ** data

3.10.1 Detailed Description

Queue data structure.

3.10.2 Field Documentation

3.10.2.1 void** queue::data

pointer to an array of pointers to node data

3.10.2.2 int32_t queue::elem

number of elements queued

3.10.2.3 int32_t queue::head

first element of the queue

3.10.2.4 int32_t queue::size

queue size (maximum number of elements)

3.10.2.5 int32_t queue::tail

last element of the queue

The documentation for this struct was generated from the following file:

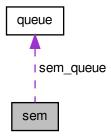
• sys/include/queue.h

3.11 sem Struct Reference

Semaphore data structure.

#include <semaphore.h>

Collaboration diagram for sem:



Data Fields

- struct queue * sem_queue
- int32_t count

3.11.1 Detailed Description

Semaphore data structure.

3.11.2 Field Documentation

3.11.2.1 int32_t sem::count

semaphore counter

3.11.2.2 struct queue* sem::sem_queue

queue for tasks waiting on the semaphore

The documentation for this struct was generated from the following file:

• sys/include/semaphore.h

3.12 tcb_entry Struct Reference

Task control block (TCB) and processor control block (PCB) entry data structures.

#include <kernel.h>

Data Fields

- uint16_t id
- int8_t name [20]
- uint8_t state

- · uint32_t delay
- uint32_t rtjobs
- uint32_t bgjobs
- uint32_t deadline_misses
- uint16_t period
- uint16_t capacity
- uint16_t deadline
- uint16_t capacity_rem
- uint16_t deadline_rem
- context task_context
- void(* ptask)(void)
- int32_t * pstack
- · uint32_t stack_size
- void * other data

3.12.1 Detailed Description

Task control block (TCB) and processor control block (PCB) entry data structures.

3.12.2 Field Documentation

3.12.2.1 uint32_t tcb_entry::bgjobs

total BE jobs executed. BE tasks share processor idle time and execute in the background.

3.12.2.2 uint16_t tcb_entry::capacity

task capacity

3.12.2.3 uint16_t tcb_entry::capacity_rem

remaining capacity on period

3.12.2.4 uint16_t tcb_entry::deadline

task deadline

3.12.2.5 uint32_t tcb_entry::deadline_misses

task realtime deadline misses

3.12.2.6 uint16_t tcb_entry::deadline_rem

remaining time slices on period

3.12.2.7 uint32_t tcb_entry::delay

delay to enter in the run/RT queue

```
3.12.2.8 uint16_t tcb_entry::id
task id
3.12.2.9 int8_t tcb_entry::name[20]
task description (or name)
3.12.2.10 void* tcb_entry::other_data
pointer to other data related to this task
3.12.2.11 uint16_t tcb_entry::period
task period
3.12.2.12 int32_t* tcb_entry::pstack
task stack area (bottom)
3.12.2.13 void(* tcb_entry::ptask)(void)
task entry point, pointer to function
3.12.2.14 uint32_t tcb_entry::rtjobs
total RT task jobs executed
3.12.2.15 uint32_t tcb_entry::stack_size
task stack size
3.12.2.16 uint8_t tcb_entry::state
0 - idle, 1 - ready, 2 - running, 3 - blocked, 4 - delayed, 5 - waiting
3.12.2.17 context tcb_entry::task_context
```

The documentation for this struct was generated from the following file:

• sys/include/kernel.h

task context



Chapter 4

File Documentation

4.1 drivers/noc/include/noc.h File Reference

Functions

· void ni init (void)

NoC driver: initializes the network interface.

void ni_isr (void *arg)

NoC driver: network interface interrupt service routine.

uint16 t hf cpuid (void)

Returns the current cpu id number.

uint16_t hf_ncores (void)

Returns the number of processors in the system.

• int32_t hf_comm_create (uint16_t id, uint16_t port, uint16_t packets)

Creates a communication queue for a task, using a port number as an alias.

int32_t hf_comm_destroy (uint16_t id)

Destroys a communication queue, returning packets buffered on a task message queue to the shared pool of packets.

- int32_t hf_recv (uint16_t *source_cpu, uint16_t *source_port, int8_t *buf, uint16_t *size, uint16_t channel)

 *Receives a message from a task (blocking receive).
- int32_t hf_send (uint16_t target_cpu, uint16_t target_port, int8_t *buf, uint16_t size, uint16_t channel)

 Sends a message to a task (blocking send).
- int32_t hf_recvack (uint16_t *source_cpu, uint16_t *source_port, int8_t *buf, uint16_t *size, uint16_t channel)

Receives a message from a task (blocking receive) with acknowledgement.

int32_t hf_sendack (uint16_t target_cpu, uint16_t target_port, int8_t *buf, uint16_t size, uint16_t channel, uint32_t timeout)

Sends a message to a task (blocking send) with acknowledgement.

Variables

uint16_t pktdrv_ports [MAX_TASKS]

Array of associations between tasks and reception ports.

struct queue * pktdrv_tqueue [MAX_TASKS]

Array of queues. Each task can have its own custom sized queue.

• struct queue * pktdrv queue

Queue of free (shared) packets. The number of packets is NOC_PACKET_SLOTS.

18 File Documentation

4.1.1 Detailed Description

Author

Sergio Johann Filho

Date

April 2016

4.1.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.1.3 DESCRIPTION

Network-on-Chip driver error codes and packet header offsets.

4.1.4 Function Documentation

4.1.4.1 int32_t hf_comm_create (uint16_t id, uint16_t port, uint16_t packets)

Creates a communication queue for a task, using a port number as an alias.

Parameters

id	is the task id which will own the communication queue
port	is the receiving port for the task
packets	is the communication queue size, in packets

Returns

ERR_OK when successful, ERR_INVALID_ID if no task matches the specified id, ERR_COMM_UNFEASIBLE if there is already a communication queue for the task, ERR_COMM_ERROR if there is already another task using the specified port and ERR_OUT_OF_MEMORY if the systems runs out of memory.

The queue created for the task will be used for the reception of data. Both ni_isr() and hf_recv() routines will manage the queue, putting and pulling packets from the queue on demand. The communication subsystem is configured by the association of a task id to a receiving port (alias) and the definition of how many packet slots a task has on its queue.

4.1.4.2 int32_t hf_comm_destroy (uint16_t id)

Destroys a communication queue, returning packets buffered on a task message queue to the shared pool of packets.

Parameters

id	is the task id which owns the communication queue

Returns

ERR_OK when successful, ERR_INVALID_ID if no task matches the specified id, ERR_COMM_ERROR if the queue could not be destroyed.

4.1.4.3 uint16_t hf_cpuid (void)

Returns the current cpu id number.

Returns

the current cpu id, defined by the CPU_ID macro.

4.1.4.4 uint16_t hf_ncores (void)

Returns the number of processors in the system.

Returns

the number of cores, defined by the dimensions of the NoC mesh.

4.1.4.5 int32_t hf_recv (uint16_t * source_cpu, uint16_t * source_port, int8_t * buf, uint16_t * size, uint16_t channel)

Receives a message from a task (blocking receive).

Parameters

source_cpu	is a pointer to a variable which will hold the source cpu
source_port	is a pointer to a variable which will hold the source port
buf	is a pointer to a buffer to hold the received message
size	a pointer to a variable which will hold the size (in bytes) of the received message
channel	is the selected message channel of this message (must be the same as in the sender)

Returns

ERR_OK when successful, ERR_COMM_UNFEASIBLE when no message queue (comm) was created and ERR_SEQ_ERROR when received packets arrive out of order, so the message is corrupted.

A message is build from packets received on the ni_isr() routine. Packets are decoded and combined in a complete message, returning the message, its size and source identification to the calling task. The buffer where the message will be stored must be large enough or we will have a problem that may not be noticed before its too late.

4.1.4.6 int32_t hf_recvack (uint16_t * source_cpu, uint16_t * source_port, int8_t * buf, uint16_t * size, uint16_t channel)

Receives a message from a task (blocking receive) with acknowledgement.

Parameters

source_cpu	is a pointer to a variable which will hold the source cpu
source_port	is a pointer to a variable which will hold the source port
buf	is a pointer to a buffer to hold the received message
size	a pointer to a variable which will hold the size (in bytes) of the received message
channel	is the selected message channel of this message (must be the same as in the sender)

Returns

ERR_OK when successful, ERR_COMM_UNFEASIBLE when no message queue (comm) was created and ERR_SEQ_ERROR when received packets arrive out of order, so the message is corrupted.

A message is build from packets received on the ni_isr() routine. Packets are decoded and combined in a complete message, returning the message, its size and source identification to the calling task. The buffer where the message

20 File Documentation

will be stored must be large enough or we will have a problem that may not be noticed before its too late. After the reception of the whole message is completed, an acknowledgement is sent to the sender task. This works as a flow control mechanism, avoiding buffer/queue overflows common to the raw protocol. Message channel 65535 will be used for the flow control mechanism.

4.1.4.7 int32_t hf_send (uint16_t target_cpu, uint16_t target_port, int8_t * buf, uint16_t size, uint16_t channel)

Sends a message to a task (blocking send).

Parameters

target_cpu	is the target processor
target_port	is the target task port
buf	is a pointer to a buffer that holds the message
size	is the size (in bytes) of the message
channel	is the selected message channel of this message (must be the same as in the receiver)

Returns

ERR OK

A message is broken into packets containing a header and part of the message as the payload. The packets are injected, one by one, in the network through the network interface.

4.1.4.8 int32_t hf_sendack (uint16_t target_cpu, uint16_t target_port, int8_t * buf, uint16_t size, uint16_t channel, uint32_t timeout)

Sends a message to a task (blocking send) with acknowledgement.

Parameters

target_cpu	is the target processor
target_port	is the target task port
buf	is a pointer to a buffer that holds the message
size	is the size (in bytes) of the message
channel	is the selected message channel of this message (must be the same as in the receiver)
timeout	is the time (in ms) that the sender will wait for a reception acknowledgement

Returns

ERR_OK

A message is broken into packets containing a header and part of the message as the payload. The packets are injected, one by one, in the network through the network interface. After that, the sender will wait for an acknowledgement from the receiver. This works as a flow control mechanism, avoiding buffer/queue overflows common to the raw protocol. Message channel 65535 will be used for the flow control mechanism.

4.1.4.9 void ni_init (void)

NoC driver: initializes the network interface.

A queue for the packet driver is initialized with NOC_PACKET_SLOTS capacity (in packets). The queue is populated with empty packets (pointers to dinamically allocated memory areas) which will be used (shared) among all tasks for the reception of data. The hardware is reset and the NoC interrupt handler is registered. This routine is called during the system boot-up and is dependent on the architecture implementation.

4.1.4.10 void ni_i (void * arg)

NoC driver: network interface interrupt service routine.

This routine is called by the second level of interrupt handling. An interrupt from the network interface means a full packet has arrived. The packet header is decoded and the target port is identified. A reference to an empty packet is removed from the pool of buffers (packets), the contents of the empty packet are filled with flits from the hardware queue and the reference is put on the target task (associated to a port) queue of packets. There is one queue per task of configurable size.

4.1.5 Variable Documentation

4.1.5.1 uint16_t pktdrv_ports[MAX_TASKS]

Array of associations between tasks and reception ports.

4.1.5.2 struct queue* pktdrv_queue

Queue of free (shared) packets. The number of packets is NOC PACKET SLOTS.

4.1.5.3 struct queue* pktdrv_tqueue[MAX_TASKS]

Array of queues. Each task can have its own custom sized queue.

4.2 drivers/noc/noc.c File Reference

Functions

· void ni init (void)

NoC driver: initializes the network interface.

void ni_isr (void *arg)

NoC driver: network interface interrupt service routine.

uint16_t hf_cpuid (void)

Returns the current cpu id number.

uint16_t hf_ncores (void)

Returns the number of processors in the system.

int32_t hf_comm_create (uint16_t id, uint16_t port, uint16_t packets)

Creates a communication queue for a task, using a port number as an alias.

int32_t hf_comm_destroy (uint16_t id)

Destroys a communication queue, returning packets buffered on a task message queue to the shared pool of packets.

- int32_t hf_recv (uint16_t *source_cpu, uint16_t *source_port, int8_t *buf, uint16_t *size, uint16_t channel)

 *Receives a message from a task (blocking receive).
- int32_t hf_send (uint16_t target_cpu, uint16_t target_port, int8_t *buf, uint16_t size, uint16_t channel)

 Sends a message to a task (blocking send).
- int32_t hf_recvack (uint16_t *source_cpu, uint16_t *source_port, int8_t *buf, uint16_t *size, uint16_t channel)

Receives a message from a task (blocking receive) with acknowledgement.

• int32_t hf_sendack (uint16_t target_cpu, uint16_t target_port, int8_t *buf, uint16_t size, uint16_t channel, uint32_t timeout)

Sends a message to a task (blocking send) with acknowledgement.

22 File Documentation

4.2.1 Detailed Description

Author

Sergio Johann Filho

Date

April 2016

4.2.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.2.3 DESCRIPTION

NoC (Network-on-Chip) interconnect driver. This driver assumes a 16-bit channel width and a basic communication protocol between the cores and the network interface, provided by _ni_read(), _ni_write() and _ni_status() helper functions (defined on the architecture HAL). A 2D mesh NoC and a buffered (1 packet) network interface are assumed.

Packet format is as follows:

```
2 bytes ....

|tgt_cpu |payload |src_cpu |src_port |tgt_port |msg_size |seq | |channel | ... data ... |
```

The platform should include the following macros:

NOC_INTERCONNECT intra-chip interconnection type CPU_ID a unique sequential number for each core NOC_WIDTH number of columns of the 2D mesh NOC_HEIGHT number of rows of the 2D mesh NOC_PACKET_SIZE packet size (in 16 bit flits) NOC_PACKET_SLOTS number of slots in the shared packet queue per core

4.2.4 Function Documentation

4.2.4.1 int32_t hf_comm_create (uint16_t id, uint16_t port, uint16_t packets)

Creates a communication queue for a task, using a port number as an alias.

Parameters

id	is the task id which will own the communication queue
port	is the receiving port for the task
packets	is the communication queue size, in packets

Returns

ERR_OK when successful, ERR_INVALID_ID if no task matches the specified id, ERR_COMM_UNFEASIBLE if there is already a communication queue for the task, ERR_COMM_ERROR if there is already another task using the specified port and ERR_OUT_OF_MEMORY if the systems runs out of memory.

The queue created for the task will be used for the reception of data. Both ni_isr() and hf_recv() routines will manage the queue, putting and pulling packets from the queue on demand. The communication subsystem is configured by the association of a task id to a receiving port (alias) and the definition of how many packet slots a task has on its queue.

4.2.4.2 int32_t hf_comm_destroy (uint16_t id)

Destroys a communication queue, returning packets buffered on a task message queue to the shared pool of packets.

Parameters

ic	id is the task id which owns the communication queue
----	--

Returns

ERR_OK when successful, ERR_INVALID_ID if no task matches the specified id, ERR_COMM_ERROR if the queue could not be destroyed.

4.2.4.3 uint16_t hf_cpuid (void)

Returns the current cpu id number.

Returns

the current cpu id, defined by the CPU_ID macro.

4.2.4.4 uint16_t hf_ncores (void)

Returns the number of processors in the system.

Returns

the number of cores, defined by the dimensions of the NoC mesh.

4.2.4.5 int32_t hf_recv (uint16_t * source_cpu, uint16_t * source_port, int8_t * buf, uint16_t * size, uint16_t channel)

Receives a message from a task (blocking receive).

Parameters

source_cpu	is a pointer to a variable which will hold the source cpu
source_port	is a pointer to a variable which will hold the source port
buf	is a pointer to a buffer to hold the received message
size	a pointer to a variable which will hold the size (in bytes) of the received message
channel	is the selected message channel of this message (must be the same as in the sender)

Returns

ERR_OK when successful, ERR_COMM_UNFEASIBLE when no message queue (comm) was created and ERR_SEQ_ERROR when received packets arrive out of order, so the message is corrupted.

A message is build from packets received on the ni_isr() routine. Packets are decoded and combined in a complete message, returning the message, its size and source identification to the calling task. The buffer where the message will be stored must be large enough or we will have a problem that may not be noticed before its too late.

4.2.4.6 int32_t hf_recvack (uint16_t * source_cpu, uint16_t * source_port, int8_t * buf, uint16_t * size, uint16_t channel)

Receives a message from a task (blocking receive) with acknowledgement.

24 File Documentation

Parameters

	source_cpu	is a pointer to a variable which will hold the source cpu
H		2 2 2 4 2 2 1 1 1 2 1 2 1 1 1 1 1 1 1 1
	source_port	is a pointer to a variable which will hold the source port
r	buf	is a pointer to a huffer to hold the received massage
L	Dui	is a pointer to a buffer to hold the received message
Г	cizo	a pointer to a variable which will hold the size (in bytes) of the received message
L	3120	a pointer to a variable which will hold the size (in bytes) of the received message
Г	channel	is the selected message channel of this message (must be the same as in the sender)
- [Charine	is the selected message charmer or this message (must be the same as in the sender)

Returns

ERR_OK when successful, ERR_COMM_UNFEASIBLE when no message queue (comm) was created and ERR_SEQ_ERROR when received packets arrive out of order, so the message is corrupted.

A message is build from packets received on the ni_isr() routine. Packets are decoded and combined in a complete message, returning the message, its size and source identification to the calling task. The buffer where the message will be stored must be large enough or we will have a problem that may not be noticed before its too late. After the reception of the whole message is completed, an acknowledgement is sent to the sender task. This works as a flow control mechanism, avoiding buffer/queue overflows common to the raw protocol. Message channel 65535 will be used for the flow control mechanism.

4.2.4.7 int32_t hf_send (uint16_t target_cpu, uint16_t target_port, int8_t * buf, uint16_t size, uint16_t channel)

Sends a message to a task (blocking send).

Parameters

target_cpu	is the target processor
target_port	is the target task port
buf	is a pointer to a buffer that holds the message
size	is the size (in bytes) of the message
channel	is the selected message channel of this message (must be the same as in the receiver)

Returns

ERR_OK

A message is broken into packets containing a header and part of the message as the payload. The packets are injected, one by one, in the network through the network interface.

4.2.4.8 int32_t hf_sendack (uint16_t target_cpu, uint16_t target_port, int8_t * buf, uint16_t size, uint16_t channel, uint32_t timeout)

Sends a message to a task (blocking send) with acknowledgement.

Parameters

target_cpu	is the target processor
target_port	is the target task port
buf	is a pointer to a buffer that holds the message
size	is the size (in bytes) of the message
channel	is the selected message channel of this message (must be the same as in the receiver)
timeout	is the time (in ms) that the sender will wait for a reception acknowledgement

Returns

ERR_OK

A message is broken into packets containing a header and part of the message as the payload. The packets are injected, one by one, in the network through the network interface. After that, the sender will wait for an acknowledgement from the receiver. This works as a flow control mechanism, avoiding buffer/queue overflows common to the raw protocol. Message channel 65535 will be used for the flow control mechanism.

```
4.2.4.9 void ni_init ( void )
```

NoC driver: initializes the network interface.

A queue for the packet driver is initialized with NOC_PACKET_SLOTS capacity (in packets). The queue is populated with empty packets (pointers to dinamically allocated memory areas) which will be used (shared) among all tasks for the reception of data. The hardware is reset and the NoC interrupt handler is registered. This routine is called during the system boot-up and is dependent on the architecture implementation.

```
4.2.4.10 void ni_isr ( void * arg )
```

NoC driver: network interface interrupt service routine.

This routine is called by the second level of interrupt handling. An interrupt from the network interface means a full packet has arrived. The packet header is decoded and the target port is identified. A reference to an empty packet is removed from the pool of buffers (packets), the contents of the empty packet are filled with flits from the hardware queue and the reference is put on the target task (associated to a port) queue of packets. There is one queue per task of configurable size.

4.3 sys/include/condvar.h File Reference

Data Structures

· struct condvar

Condition variable data structure.

Typedefs

• typedef struct condvar cond_t

Functions

```
• int32 t hf condinit (cond t *c)
```

Initializes a condition variable.

int32_t hf_conddestroy (cond_t *c)

Destroys a condition variable.

void hf_condwait (cond_t *c, mutex_t *m)

Wait on a condition variable.

void hf condsignal (cond t *c)

Signal a condition variable.

void hf_condbroadcast (cond_t *c)

Signal (broadcast) a condition variable.

4.3.1 Typedef Documentation

4.3.1.1 typedef struct condvar cond_t

4.3.2 Function Documentation

4.3.2.1 void hf_condbroadcast (cond_t * c)

Signal (broadcast) a condition variable.

Parameters

c is a pointer to a condition variable.

Implements the condition signal broadcast operation for all waiting tasks. The call unblocks and removes all tasks from the waiting queue. If no tasks are waiting for the condition, the signal is lost.

4.3.2.2 int32_t hf_conddestroy ($cond_t * c$)

Destroys a condition variable.

Parameters

c is a pointer to a condition variable.

Returns

ERR_OK on success and ERR_ERROR if the condition variable could not be removed from memory.

4.3.2.3 int32_t hf_condinit (cond t * c)

Initializes a condition variable.

Parameters

c is a pointer to a condition variable.

Returns

ERR_OK on success and ERR_ERROR if the condition variable could not be allocated in memory.

4.3.2.4 void $hf_condsignal(cond_t * c)$

Signal a condition variable.

Parameters

c is a pointer to a condition variable.

Implements the condition signal operation for one waiting task. The call removes a task from the waiting queue and unblocks it. If no tasks are waiting for the condition, the signal is lost.

4.3.2.5 void hf_condwait (cond_t * c, mutex_t * m)

Wait on a condition variable.

Parameters

С	is a pointer to a condition variable.
т	is a pointer to a mutex.

Implements the atomic condition wait operation. The call should always be invoked with the mutex locked. The current task is put in a queue on the condition variable, its state is set to blocked and unlocks the mutex atomically, then yields the processor. When woke up (by a signalling task), the task locks the mutex and returns.

4.4 sys/include/ecodes.h File Reference

4.5 sys/include/hellfire.h File Reference

4.5.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.5.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.5.3 DESCRIPTION

Default system wide include file and error code definitions.

4.6 sys/include/kernel.h File Reference

Data Structures

struct tcb_entry

Task control block (TCB) and processor control block (PCB) entry data structures.

struct pcb_entry

Variables

• struct tcb_entry krnl_tcb [MAX_TASKS]

The task control block and processor control block.

- struct pcb_entry krnl_pcb
- struct tcb_entry * krnl_task
- uint16_t krnl_tasks

- uint16_t krnl_current_task
- uint16_t krnl_schedule
- struct queue * krnl_run_queue
- struct queue * krnl_delay_queue
- struct queue * krnl_rt_queue
- struct queue * krnl_event_queue
- uint8_t krnl_heap [HEAP_SIZE]
- · uint32 t krnl free

4.6.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.6.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.6.3 DESCRIPTION

Kernel data structures.

4.6.4 Variable Documentation

4.6.4.1 uint16_t krnl_current_task

the current running task id

4.6.4.2 struct queue* krnl_delay_queue

pointer to a queue of delayed tasks

4.6.4.3 struct queue* krnl_event_queue

pointer to a queue of tasks waiting for an event

4.6.4.4 uint32_t krnl_free

amount of free heap memory, in bytes

4.6.4.5 uint8_t krnl_heap[HEAP_SIZE]

contiguous heap memory area to be used as a memory pool. the memory allocator (malloc() and free()) controls this data structure

```
4.6.4.6 struct pcb_entry krnl_pcb
4.6.4.7 struct queue* krnl_rt_queue
pointer to a queue of real time tasks
4.6.4.8 struct queue* krnl_run_queue
pointer to a queue of best effort tasks
4.6.4.9 uint16_t krnl_schedule
scheduler enable / disable flag
4.6.4.10 struct tcb_entry* krnl_task
pointer to a task control block entry
4.6.4.11 uint16_t krnl_tasks
number of tasks in the system
4.6.4.12 struct tcb_entry krnl_tcb[MAX_TASKS]
The task control block and processor control block.
       sys/include/kprintf.h File Reference
Functions
    • int32_t kprintf (const int8_t *fmt,...)
          Kernel short version of printf().
    int32_t dprintf (const int8_t *fmt,...)
          Kernel debug version of printf().
```

4.7.1 Function Documentation

4.7.1.1 int32_t dprintf (const int8_t * fmt, ...)

Kernel debug version of printf().

Parameters

fmt is a pointer to formatted data to be printed on the debug output.

Returns

0.

```
4.7.1.2 int32_t kprintf ( const int8_t * fmt, ... )
```

Kernel short version of printf().

Parameters

fmt	is a pointer to formatted data to be printed.
- 1	F

Returns

0.

4.8 sys/include/list.h File Reference

Data Structures

struct list

List data structure.

Functions

struct list * hf_list_init (void)

Initializes a list.

int32_t hf_list_append (struct list *lst, void *item)

Appends a new node to the end of the list.

int32_t hf_list_insert (struct list *lst, void *item, int32_t pos)

Inserts a new node to an arbitrary position in a list.

int32_t hf_list_remove (struct list *lst, int32_t pos)

Removes an arbitrary node from a list.

void * hf_list_get (struct list *lst, int32_t pos)

Returns the address of the data belonging to a list node.

int32_t hf_list_set (struct list *lst, void *item, int32_t pos)

Changes the address of the data belonging to a list node.

• int32_t hf_list_count (struct list *lst)

Returns the number of nodes in a list.

4.8.1 Function Documentation

4.8.1.1 int32_t hf_list_append (struct list * lst, void * item)

Appends a new node to the end of the list.

Parameters

lst	is a pointer to a list structure.
item	is a pointer to data belonging to the list node.

Returns

0 when successful and -1 otherwise.

4.8.1.2 int32_t hf_list_count (struct list * lst)

Returns the number of nodes in a list.

Parameters

lst	is a pointer to a list structure.

Returns

The number of elements in the list.

4.8.1.3 void* hf_list_get (struct list * lst, int32_t pos)

Returns the address of the data belonging to a list node.

Parameters

lst	is a pointer to a list structure.
pos	is the n-th element position in the list.

Returns

0 when the element is not found and the address to data otherwise.

4.8.1.4 struct list* hf_list_init (void) [read]

Initializes a list.

Returns

a pointer to a list structure.

4.8.1.5 int32_t hf_list_insert (struct list * lst, void * item, int32_t pos)

Inserts a new node to an arbitrary position in a list.

Parameters

lst	is a pointer to a list structure.
item	is a pointer to data belonging to the list node.
pos	is the n-th element position in the list.

Returns

0 when successful and -1 otherwise.

4.8.1.6 int32_t hf_list_remove (struct list * lst, int32_t pos)

Removes an arbitrary node from a list.

Parameters

lst	is a pointer to a list structure.
pos	is the n-th element position in the list.

Returns

0 when successful and -1 otherwise.

```
4.8.1.7 int32_t hf_list_set ( struct list * lst, void * item, int32_t pos )
```

Changes the address of the data belonging to a list node.

Parameters

lst	is a pointer to a list structure.
item	is an address to data belonging to the list node.
pos	is the n-th element position in the list.

Returns

-1 when the element is not found and 0 if the element was updated.

4.9 sys/include/mailbox.h File Reference

Data Structures

struct mailbox

Mailbox data structure.

Typedefs

• typedef struct mailbox mail_t

Functions

- void hf_mboxinit (mail_t *mbox, uint16_t n_waiting_tasks)
- void hf_mboxsend (mail_t *mbox, void *msg)
- void * hf_mboxrecv (mail_t *mbox)
- void * hf_mboxaccept (mail_t *mbox)

4.9.1 Typedef Documentation

4.9.1.1 typedef struct mailbox mail_t

mailbox type definition

4.9.2 Function Documentation

- 4.9.2.1 void* hf_mboxaccept (mail t* mbox)
- 4.9.2.2 void hf_mboxinit (mail_t * mbox, uint16_t n_waiting_tasks)
- 4.9.2.3 void* hf_mboxrecv (mail_t * mbox)
- 4.9.2.4 void hf_mboxsend (mail_t * mbox, void * msg)

4.10 sys/include/main.h File Reference

Functions

• void app_main (void)

4.10.1 Function Documentation

4.10.1.1 void app_main (void)

4.11 sys/include/malloc.h File Reference

Data Structures

- struct mem_chunk
- struct mem_chunk_ptr
- union mem_header_union
- struct mem_block
- struct mem_block

Typedefs

- typedef uint32_t align
- typedef union mem_header_union mem_header_t

Functions

- void hf free (void *ptr)
- void * hf_malloc (uint32_t size)
- void heapinit (void *heap, uint32_t len)
- void * hf_calloc (uint32_t qty, uint32_t type_size)
- void * hf_realloc (void *ptr, uint32_t size)

Variables

- mem_chunk_ptr krnl_heap_ptr
- struct mem_block * ff

4.11.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.11.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.11.3 DESCRIPTION

Data structures of several memory allocators.

```
4.11.4 Typedef Documentation
4.11.4.1 typedef uint32_t align
4.11.4.2 typedef union mem_header_union mem_header_t
4.11.5 Function Documentation
4.11.5.1 void heapinit ( void * heap, uint32_t len )
4.11.5.2 void* hf_calloc ( uint32_t qty, uint32_t type_size )
4.11.5.3 void hf_free ( void * ptr )
4.11.5.4 void * hf_malloc ( uint32_t size )
4.11.5.5 void* hf_realloc ( void * ptr, uint32_t size )
```

4.11.6 Variable Documentation

4.11.6.1 struct mem_block* ff

4.11.6.2 mem_chunk_ptr krnl_heap_ptr

4.12 sys/include/mutex.h File Reference

Data Structures

struct mtx

Mutex data structure.

• struct mtx

Mutex data structure.

Typedefs

· typedef struct mtx mutex_t

Functions

void hf_mtxinit (mutex_t *m)

Initializes a mutex, defining its initial value.

void hf_mtxlock (mutex_t *m)

Locks a mutex.

void hf_mtxunlock (mutex_t *m)

Unlocks a mutex.

4.12.1 Typedef Documentation

4.12.1.1 typedef struct mtx mutex_t

4.12.2 Function Documentation

4.12.2.1 void hf_mtxinit (mutex_t * m)

Initializes a mutex, defining its initial value.

Parameters

s is a pointer to a mutex.

4.12.2.2 void hf_mtxlock (mutex_t * m)

Locks a mutex.

Parameters

s is a pointer to a mutex.

If the mutex is not locked, the calling task continues execution. Otherwise, the task spins.

4.12.2.3 void $hf_mtxunlock (mutex_t * m)$

Unlocks a mutex.

Parameters

s is a pointer to a mutex.

4.13 sys/include/panic.h File Reference

Functions

• void panic (int32_t cause)

Causes the kernel to panic.

4.13.1 Function Documentation

4.13.1.1 void panic (int32_t cause)

Causes the kernel to panic.

Interrupts are disabled, the panic cause is presented to the user and the system is locked forever.

4.14 sys/include/processor.h File Reference

Functions

void hf_schedlock (int32_t lock)

Enables or disables the task scheduler.

• int32_t hf_freecpu (void)

Returns the percentage of free processor time. Only realtime tasks are accounted as processor load.

int32_t hf_cpuload (uint16_t id)

Returns the percentage of processor time used by a given task. Both realtime and best effort tasks are accounted. Best effort tasks lose processor time when realtime tasks are part of the task set, as they only run in the background (idle time).

• uint32_t hf_freemem (void)

Returns the amount of free memory.

4.14.1 Function Documentation

```
4.14.1.1 int32_t hf_cpuload ( uint16_t id )
```

Returns the percentage of processor time used by a given task. Both realtime and best effort tasks are accounted. Best effort tasks lose processor time when realtime tasks are part of the task set, as they only run in the background (idle time).

Parameters

id is the task id number

Returns

a number representing the percentage of processor usage or ERR_INVALID_ID if the referenced task does not exist.

```
4.14.1.2 int32_t hf_freecpu ( void )
```

Returns the percentage of free processor time. Only realtime tasks are accounted as processor load.

Returns

a number representing the percentage of free processor.

```
4.14.1.3 uint32_t hf_freemem ( void )
```

Returns the amount of free memory.

Returns

free heap memory, in bytes.

```
4.14.1.4 void hf_schedlock ( int32_t lock )
```

Enables or disables the task scheduler.

Parameters

lock defines the scheduler activation (a value of 1 disables task scheduling).

4.15 sys/include/queue.h File Reference

Data Structures

struct queue

Queue data structure.

Functions

struct queue * hf_queue_create (int32_t size)

Creates a queue of specified size.

int32_t hf_queue_destroy (struct queue *q)

Destroys a queue.

int32_t hf_queue_count (struct queue *q)

Counts the number of nodes in a queue.

• int32_t hf_queue_addtail (struct queue *q, void *ptr)

Adds a node to the tail of the queue.

void * hf_queue_remhead (struct queue *q)

Removes a node from the head of the queue.

void * hf_queue_remtail (struct queue *q)

Removes a node from the tail of the queue.

void * hf_queue_get (struct queue *q, int32_t elem)

Returns a node from the queue.

• int32_t hf_queue_set (struct queue *q, int32_t elem, void *ptr)

Updates a node on the queue.

• int32_t hf_queue_swap (struct queue *q, int32_t elem1, int32_t elem2)

Swap the position of two nodes in the queue.

4.15.1 Function Documentation

4.15.1.1 int32_t hf_queue_addtail (struct queue * q, void * ptr)

Adds a node to the tail of the queue.

Parameters

q	is a pointer to a queue structure.
ptr	a pointer to data belonging to the queue node.

Returns

0 when successful and -1 otherwise.

4.15.1.2 int32_t hf_queue_count (struct queue * q)

Counts the number of nodes in a queue.

Parameters

q	is a pointer to a queue structure.

Returns

the number of nodes.

4.15.1.3 struct queue* hf_queue_create (int32_t size) [read]

Creates a queue of specified size.

Parameters

size	is the maximum number of elements.

Returns

pointer to the queue on success and NULL otherwise.

4.15.1.4 int32_t hf_queue_destroy (struct queue * q)

Destroys a queue.

Parameters

q	is a pointer to a queue structure.
---	------------------------------------

Returns

0 when successful and -1 otherwise.

4.15.1.5 void* hf_queue_get (struct queue * q, int32_t elem)

Returns a node from the queue.

Parameters

q	is a pointer to a queue structure.
elem	is the n-th element from the queue.

Returns

pointer to node data success and 0 otherwise.

4.15.1.6 void* hf_queue_remhead (struct queue * q)

Removes a node from the head of the queue.

Parameters

q	is a pointer to a queue structure.

Returns

pointer to node data on success and 0 otherwise.

4.15.1.7 void* hf_queue_remtail (struct queue * q)

Removes a node from the tail of the queue.

Parameters

q	is a pointer to a queue structure.

Returns

pointer to node data on success and 0 otherwise.

4.15.1.8 int32_t hf_queue_set (struct queue * q, int32_t elem, void * ptr)

Updates a node on the queue.

Parameters

q	is a pointer to a queue structure.
elem	is the n-th element from the queue.
ptr	a pointer to data belonging to the queue node.

Returns

0 success and -1 otherwise.

4.15.1.9 int32_t hf_queue_swap (struct queue * q, int32_t elem1, int32_t elem2)

Swap the position of two nodes in the queue.

Parameters

q	is a pointer to a queue structure.
elem1	is the first n-th element from the queue.
elem2	is the second n-th element from the queue.

Returns

0 when successful and -1 otherwise.

4.16 sys/include/scheduler.h File Reference

Functions

void dispatch_isr (void *arg)

The task dispatcher.

int32_t sched_be (void)

Best effort (BE) scheduler.

int32_t sched_rt (void)

Real time (RT) scheduler.

4.16.1 Function Documentation

```
4.16.1.1 void dispatch_isr (void * arg)
```

The task dispatcher.

The job of the dispatcher is simple: save the current task context on the TCB, update its state to ready and check its stack for overflow. If there are tasks to be scheduled, process the delay queue and invoke the real-time scheduler. If no RT tasks are ready to be scheduled, invoke the best effort scheduler. Update the scheduled task state to running and restore the context of the task.

Delayed tasks are in the delay queue, and are processed in the following way:

- · The number of elements (tasks) in queue is counted;
- The a task from the head of the queue is removed and its delay is decremented;
 - If the decremented delay of a task reaches 0, it is put on RT or BE run queue;
 - It is put it back on the tail of the delay queue otherwise;
- · Repeat until the whole queue is processed;

```
4.16.1.2 int32_t sched_be ( void )
```

Best effort (BE) scheduler.

Returns

Best effort task id.

The algorithm is Lottery Scheduling.

- Take a task from the run queue, copy its entry and put it back at the tail of the run queue.
- If the task is in the blocked state (it may be simply blocked or waiting in a semaphore) or its not the ticket, it is put back at the tail of the run queue and the next task is picked up.
- So, if all tasks are blocked, at least the idle task can execute (it is never blocked, at least it is what we hope!).
- Tasks in the blocked state are never removed from the run queue (they are ignored), although they may be in another queue waiting for a resource.

```
4.16.1.3 int32_t sched_rt ( void )
```

Real time (RT) scheduler.

Returns

Real time task id.

The scheduling algorithm is Rate Monotonic.

- · Sort the queue of RT tasks by period;
- Update real time information (remaining deadline and capacity) of the whole task set.
- If the task at the head of the queue fits the requirements to be scheduled (not blocked, has jobs to execute and no task with higher priority according to RM was selected) then register the task to be scheduled.

4.17 sys/include/semaphore.h File Reference

Data Structures

· struct sem

Semaphore data structure.

Typedefs

• typedef struct sem sem_t

Functions

```
• int32_t hf_seminit (sem_t *s, int32_t value)
```

Initializes a semaphore and defines its initial value.

int32_t hf_semdestroy (sem_t *s)

Destroys a semaphore.

void hf_semwait (sem_t *s)

Wait on a semaphore.

void hf_sempost (sem_t *s)

Signal a semaphore.

4.17.1 Typedef Documentation

4.17.1.1 typedef struct sem sem_t

4.17.2 Function Documentation

4.17.2.1 int32_t hf_semdestroy ($sem_t * s$)

Destroys a semaphore.

Parameters

s	is a pointer to a semaphore.

Returns

ERR_OK on success and ERR_ERROR if the semaphore could not be removed from memory.

```
4.17.2.2 int32_t hf_seminit ( sem_t * s, int32_t value )
```

Initializes a semaphore and defines its initial value.

Parameters

S	is a pointer to a semaphore.
value	is the semaphore initial value.

Returns

ERR_OK on success and ERR_ERROR if the semaphore could not be allocated in memory or its initial value is less than zero

```
4.17.2.3 void hf_sempost(sem_t * s)
```

Signal a semaphore.

Parameters

```
s is a pointer to a semaphore.
```

Implements the atomic V() operation. The semaphore count is incremented and the task from the head of the semaphore queue is unblocked if the count is less than or equal to zero.

```
4.17.2.4 void hf_semwait ( sem_t * s )
```

Wait on a semaphore.

Parameters

s is a pointer to a semaphore.

Implements the atomic P() operation. The semaphore count is decremented and calling task is blocked and queued on the semaphore if the count reaches a negative value. If not, the task continues its execution.

4.18 sys/include/task.h File Reference

Functions

```
• int32 t hf id (int8 t *name)
```

Get a task id by its name.

int8_t * hf_name (uint16_t id)

Get a task name by its id.

uint16_t hf_selfid (void)

Get the current task id.

int8_t * hf_selfname (void)

Get the current task name.

int32_t hf_state (uint16_t id)

Get the current state of a task.

• int32_t hf_jobs (uint16_t id)

Get the number of executed jobs of a task.

int32_t hf_dlm (uint16_t id)

Get the number of deadline misses of a task.

• int32_t hf_spawn (void(*task)(), uint16_t period, uint16_t capacity, uint16_t deadline, int8_t *name, uint32_t stack_size)

Spawn a new task.

void hf_yield (void)

Yields the current task.

• int32_t hf_block (uint16_t id)

Blocks a task.

• int32_t hf_resume (uint16_t id)

Resumes a blocked task.

int32_t hf_kill (uint16_t id)

Kills a task.

• int32_t hf_delay (uint16_t id, uint32_t delay)

Delays a task for an amount of time.

4.18.1 Function Documentation

4.18.1.1 int32_t hf_block (uint16_t id)

Blocks a task.

Parameters

id	is a	task	id	number.
----	------	------	----	---------

Returns

ERR_OK on success, ERR_INVALID_ID if the referenced task does not exist or ERR_ERROR if the task is already in the blocked state.

The task is marked as TASK_BLOCKED so the scheduler doesn't select it as a candidate for scheduling. The blocking state is acomplished without removing the task from the run queue, reducing the cost of the operation in cases where the task state is switched frequently (such as in semaphore primitives).

4.18.1.2 int32_t hf_delay (uint16_t id, uint32_t delay)

Delays a task for an amount of time.

Parameters

id	is a task id number.
delay	is the amount of time (in quantum / tick units).

Returns

ERR_OK on success or ERR_INVALID_ID if the referenced task does not exist.

A task is removed from its run queue and its state is marked as TASK_DELAYED. The task is put on the delay queue and remains there until the dispatcher places it back to its run queue. Time is managed by the task dispatcher, which counts down delays, controls the delay queue by cycling the tasks and removing them when the task delay has passed.

4.18.1.3 int32_t hf_dlm (uint16_t id)

Get the number of deadline misses of a task.

Parameters

id	is a task id number.

Returns

deadlines missed by the task if found and ERR_INVALID_ID otherwise.

```
4.18.1.4 int32_t hf_id ( int8_t * name )
```

Get a task id by its name.

Parameters

name	is a pointer to an array holding the task name.

Returns

task id if the task is found and ERR_INVALID_NAME otherwise.

```
4.18.1.5 int32_t hf_jobs ( uint16_t id )
```

Get the number of executed jobs of a task.

Parameters

id is a task id number.	
-------------------------	--

Returns

jobs executed by the task if found and ERR_INVALID_ID otherwise.

4.18.1.6 int32_t hf_kill (uint16_t id)

Kills a task.

Parameters

```
id is a task id number.
```

Returns

ERR_OK on success or ERR_INVALID_ID if the referenced task does not exist.

All memory allocated during the task initialization is freed, the TCB entry is cleared and the task is removed from its run queue.

4.18.1.7 int8_t* hf_name (uint16_t id)

Get a task name by its id.

Parameters

id	is a task id number.

Returns

task name if the task is found and NULL otherwise.

4.18.1.8 int32_t hf_resume (uint16_t id)

Resumes a blocked task.

Parameters

io	d is a task id number.

Returns

ERR_OK on success, ERR_INVALID_ID if the referenced task does not exist or ERR_ERROR if the task is not in the blocked state.

The task must be in the TASK_BLOCKED state in order to be resumed. The task is marked as TASK_BLOCKED so the scheduler doesn't select it as a candidate for scheduling. The blocking state is acomplished without removing the task from the run queue, reducing the cost of the operation in cases where the task state is switched frequently (such as in semaphore primitives).

4.18.1.9 uint16_t hf_selfid (void)

Get the current task id.

Returns

current task id.

4.18.1.10 int8_t* hf_selfname (void)

Get the current task name.

Returns

current task name.

4.18.1.11 int32_t hf_spawn (void(*)() task, uint16_t period, uint16_t capacity, uint16_t deadline, int8_t * name, uint32_t stack_size)

Spawn a new task.

Parameters

task	is a pointer to a task function / body.
period	is the task RT period (in quantum / tick units).
capacity	is the amount of work to be executed in a period (in quantum / tick units).
deadline	is the task deadline to complete the work in the period (in quantum / tick units).
name	is a string used to identify a task.
stack_size	is the stack memory to be allocated for the task.

Returns

task id if the task is created, ERR_EXCEED_MAX_NUM if the maximum number of tasks in the system is exceeded, ERR_INVALID_PARAMETER if impossible RT parameters are specified or ERR_OUT_OF_MEMORY if the system fails to allocate memory for the task resources.

If a task has defined realtime parameters, it is put on the RT queue, if not (period 0, capacity 0 and deadline 0), it is put on the BE queue. WARNING: Task stack size should be always configured correctly, considering data declared on the auto region (local variables) and around 1024 of spare memory for the OS. For example, if you declare a buffer of 5000 bytes, stack size should be at least 6000.

```
4.18.1.12 int32_t hf_state ( uint16_t id )
```

Get the current state of a task.

Parameters

```
id is a task id number.
```

Returns

task state the task if found (TASK_IDLE, TASK_READY, TASK_RUNNING, TASK_BLOCKED, TASK_DELAY-ED or TASK_WAITING) and ERR_INVALID_ID otherwise.

```
4.18.1.13 void hf_yield (void)
```

Yields the current task.

The current task gives up execution and the best effort scheduler is invoked.

4.19 sys/kernel/main.c File Reference

Functions

• int main (void)

HellfireOS kernel entry point and system initialization.

4.19.1 Detailed Description

Author

Sergio Johann Filho

Date

January 2016

4.19.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.19.3 DESCRIPTION

The HellfireOS realtime operating system kernel.

4.19.4 Function Documentation

4.19.4.1 int main (void)

HellfireOS kernel entry point and system initialization.

Returns

should not return.

We assume that the following machine state has been already set before this routine.

- · Kernel BSS section is filled with 0.
- · Kernel stack is configured.
- · All interrupts are disabled.
- Minimum page table is set. (MMU systems only)

4.20 sys/kernel/panic.c File Reference

Functions

void panic (int32_t cause)
 Causes the kernel to panic.

4.20.1 Detailed Description

Author

Sergio Johann Filho

Date

March 2016

4.20.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.20.3 DESCRIPTION

Kernel panic.

4.20.4 Function Documentation

4.20.4.1 void panic (int32_t cause)

Causes the kernel to panic.

Interrupts are disabled, the panic cause is presented to the user and the system is locked forever.

4.21 sys/kernel/processor.c File Reference

Functions

• void hf_schedlock (int32_t lock)

Enables or disables the task scheduler.

int32_t hf_freecpu (void)

Returns the percentage of free processor time. Only realtime tasks are accounted as processor load.

int32_t hf_cpuload (uint16_t id)

Returns the percentage of processor time used by a given task. Both realtime and best effort tasks are accounted. Best effort tasks lose processor time when realtime tasks are part of the task set, as they only run in the background (idle time).

• uint32_t hf_freemem (void)

Returns the amount of free memory.

4.21.1 Detailed Description

Author

Sergio Johann Filho

Date

March 2016

4.21.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.21.3 DESCRIPTION

Processor and scheduler management primitives and auxiliary functions.

4.21.4 Function Documentation

4.21.4.1 int32_t hf_cpuload (uint16_t id)

Returns the percentage of processor time used by a given task. Both realtime and best effort tasks are accounted. Best effort tasks lose processor time when realtime tasks are part of the task set, as they only run in the background (idle time).

Parameters

Returns

a number representing the percentage of processor usage or ERR_INVALID_ID if the referenced task does not exist.

4.21.4.2 int32_t hf_freecpu (void)

Returns the percentage of free processor time. Only realtime tasks are accounted as processor load.

Returns

a number representing the percentage of free processor.

```
4.21.4.3 uint32_t hf_freemem ( void )
```

Returns the amount of free memory.

Returns

free heap memory, in bytes.

4.21.4.4 void hf_schedlock (int32_t lock)

Enables or disables the task scheduler.

Parameters

lock defines the scheduler activation (a value of 1 disables task scheduling).

4.22 sys/kernel/scheduler.c File Reference

Functions

void dispatch_isr (void *arg)

The task dispatcher.

• int32_t sched_be (void)

Best effort (BE) scheduler.

• int32_t sched_rt (void)

Real time (RT) scheduler.

4.22.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.22.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.22.3 DESCRIPTION

Kernel two-level scheduler and task queue management.

4.22.4 Function Documentation

4.22.4.1 void dispatch_isr (void * arg)

The task dispatcher.

The job of the dispatcher is simple: save the current task context on the TCB, update its state to ready and check its stack for overflow. If there are tasks to be scheduled, process the delay queue and invoke the real-time scheduler. If no RT tasks are ready to be scheduled, invoke the best effort scheduler. Update the scheduled task state to running and restore the context of the task.

Delayed tasks are in the delay queue, and are processed in the following way:

- The number of elements (tasks) in queue is counted;
- The a task from the head of the queue is removed and its delay is decremented;
 - If the decremented delay of a task reaches 0, it is put on RT or BE run queue;
 - It is put it back on the tail of the delay queue otherwise;
- · Repeat until the whole gueue is processed;

```
4.22.4.2 int32_t sched_be ( void )
```

Best effort (BE) scheduler.

Returns

Best effort task id.

The algorithm is Lottery Scheduling.

- Take a task from the run queue, copy its entry and put it back at the tail of the run queue.
- If the task is in the blocked state (it may be simply blocked or waiting in a semaphore) or its not the ticket, it is put back at the tail of the run queue and the next task is picked up.
- So, if all tasks are blocked, at least the idle task can execute (it is never blocked, at least it is what we hope!).
- Tasks in the blocked state are never removed from the run queue (they are ignored), although they may be in another queue waiting for a resource.

```
4.22.4.3 int32_t sched_rt ( void )
```

Real time (RT) scheduler.

Returns

Real time task id.

The scheduling algorithm is Rate Monotonic.

- Sort the queue of RT tasks by period;
- · Update real time information (remaining deadline and capacity) of the whole task set.
- If the task at the head of the queue fits the requirements to be scheduled (not blocked, has jobs to execute and no task with higher priority according to RM was selected) then register the task to be scheduled.

4.23 sys/kernel/task.c File Reference

Functions

int32_t hf_id (int8_t *name)

Get a task id by its name.

• int8_t * hf_name (uint16_t id)

Get a task name by its id.

• uint16_t hf_selfid (void)

Get the current task id.

int8_t * hf_selfname (void)

Get the current task name.

• int32_t hf_state (uint16_t id)

Get the current state of a task.

• int32_t hf_jobs (uint16_t id)

Get the number of executed jobs of a task.

int32_t hf_dlm (uint16_t id)

Get the number of deadline misses of a task.

int32_t hf_spawn (void(*task)(), uint16_t period, uint16_t capacity, uint16_t deadline, int8_t *name, uint32_t stack_size)

Spawn a new task.

void hf yield (void)

Yields the current task.

• int32_t hf_block (uint16_t id)

Blocks a task.

int32_t hf_resume (uint16_t id)

Resumes a blocked task.

int32_t hf_kill (uint16_t id)

Kills a task.

• int32_t hf_delay (uint16_t id, uint32_t delay)

Delays a task for an amount of time.

4.23.1 Detailed Description

Author

Sergio Johann Filho

Date

March 2016

4.23.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.23.3 DESCRIPTION

Task management primitives and auxiliary functions.

4.23.4 Function Documentation

4.23.4.1 int32_t hf_block (uint16_t id)

Blocks a task.

Parameters

id	is a task id number.

Returns

ERR_OK on success, ERR_INVALID_ID if the referenced task does not exist or ERR_ERROR if the task is already in the blocked state.

The task is marked as TASK_BLOCKED so the scheduler doesn't select it as a candidate for scheduling. The blocking state is acomplished without removing the task from the run queue, reducing the cost of the operation in cases where the task state is switched frequently (such as in semaphore primitives).

4.23.4.2 int32_t hf_delay (uint16_t id, uint32_t delay)

Delays a task for an amount of time.

Parameters

id	is a task id number.
delay	is the amount of time (in quantum / tick units).

Returns

ERR_OK on success or ERR_INVALID_ID if the referenced task does not exist.

A task is removed from its run queue and its state is marked as TASK_DELAYED. The task is put on the delay queue and remains there until the dispatcher places it back to its run queue. Time is managed by the task dispatcher, which counts down delays, controls the delay queue by cycling the tasks and removing them when the task delay has passed.

4.23.4.3 int32_t hf_dlm (uint16_t id)

Get the number of deadline misses of a task.

Parameters

id	is a task id number.
----	----------------------

Returns

deadlines missed by the task if found and ERR_INVALID_ID otherwise.

4.23.4.4 int32_t hf_id (int8_t * name)

Get a task id by its name.

Parameters

name is a pointer to an array holding the task name.
--

Returns

task id if the task is found and ERR_INVALID_NAME otherwise.

4.23.4.5 int32_t hf_jobs (uint16_t id)

Get the number of executed jobs of a task.

Parameters

id	is a task id number.

Returns

jobs executed by the task if found and ERR_INVALID_ID otherwise.

4.23.4.6 int32_t hf_kill (uint16_t id)

Kills a task.

Parameters

is a		

Returns

ERR_OK on success or ERR_INVALID_ID if the referenced task does not exist.

All memory allocated during the task initialization is freed, the TCB entry is cleared and the task is removed from its run queue.

4.23.4.7 int8_t* hf_name (uint16_t id)

Get a task name by its id.

Parameters

id	is a task id number.
----	----------------------

Returns

task name if the task is found and NULL otherwise.

4.23.4.8 int32_t hf_resume (uint16_t id)

Resumes a blocked task.

Parameters

id	is a task id number.

Returns

ERR_OK on success, ERR_INVALID_ID if the referenced task does not exist or ERR_ERROR if the task is not in the blocked state.

The task must be in the TASK_BLOCKED state in order to be resumed. The task is marked as TASK_BLOCKED so the scheduler doesn't select it as a candidate for scheduling. The blocking state is acomplished without removing the task from the run queue, reducing the cost of the operation in cases where the task state is switched frequently (such as in semaphore primitives).

4.23.4.9 uint16_t hf_selfid (void)

Get the current task id.

Returns

current task id.

4.23.4.10 int8_t* hf_selfname (void)

Get the current task name.

Returns

current task name.

4.23.4.11 int32_t hf_spawn (void(*)() task, uint16_t period, uint16_t capacity, uint16_t deadline, int8_t * name, uint32_t stack_size)

Spawn a new task.

Parameters

task	is a pointer to a task function / body.
period	is the task RT period (in quantum / tick units).
capacity	is the amount of work to be executed in a period (in quantum / tick units).
deadline	is the task deadline to complete the work in the period (in quantum / tick units).
name	is a string used to identify a task.
stack_size	is the stack memory to be allocated for the task.

Returns

task id if the task is created, ERR_EXCEED_MAX_NUM if the maximum number of tasks in the system is exceeded, ERR_INVALID_PARAMETER if impossible RT parameters are specified or ERR_OUT_OF_MEMORY if the system fails to allocate memory for the task resources.

If a task has defined realtime parameters, it is put on the RT queue, if not (period 0, capacity 0 and deadline 0), it is put on the BE queue. WARNING: Task stack size should be always configured correctly, considering data declared on the auto region (local variables) and around 1024 of spare memory for the OS. For example, if you declare a buffer of 5000 bytes, stack size should be at least 6000.

4.23.4.12 int32_t hf_state (uint16_t id)

Get the current state of a task.

Parameters

id is a task id number.

Returns

task state the task if found (TASK_IDLE, TASK_READY, TASK_RUNNING, TASK_BLOCKED, TASK_DELAY-ED or TASK_WAITING) and ERR_INVALID_ID otherwise.

```
4.23.4.13 void hf_yield (void)
```

Yields the current task.

The current task gives up execution and the best effort scheduler is invoked.

4.24 sys/lib/kprintf.c File Reference

Functions

- int32_t kprintf (const int8_t *fmt,...)
 - Kernel short version of printf().
- int32_t dprintf (const int8_t *fmt,...)

Kernel debug version of printf().

4.24.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.24.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.24.3 DESCRIPTION

Kernel printing / conversion auxiliary functions.

4.24.4 Function Documentation

```
4.24.4.1 int32_t dprintf ( const int8_t * fmt, ... )
```

Kernel debug version of printf().

Parameters

fmt is	s a pointer to formatted data to be printed on the debug output.

Returns

0.

```
4.24.4.2 int32_t kprintf ( const int8_t * fmt, ... )
```

Kernel short version of printf().

Parameters

fmt is a pointer to formatted data to be printed.

Returns

0.

4.25 sys/lib/list.c File Reference

Functions

struct list * hf_list_init (void)

Initializes a list.

int32_t hf_list_append (struct list *lst, void *item)

Appends a new node to the end of the list.

int32_t hf_list_insert (struct list *lst, void *item, int32_t pos)

Inserts a new node to an arbitrary position in a list.

int32_t hf_list_remove (struct list *lst, int32_t pos)

Removes an arbitrary node from a list.

void * hf_list_get (struct list *lst, int32_t pos)

Returns the address of the data belonging to a list node.

• int32_t hf_list_set (struct list *lst, void *item, int32_t pos)

Changes the address of the data belonging to a list node.

• int32_t hf_list_count (struct list *lst)

Returns the number of nodes in a list.

4.25.1 Detailed Description

Author

Sergio Johann Filho

Date

January 2016

4.25.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.25.3 DESCRIPTION

List manipulation primitives and auxiliary functions. List structures are allocated dynamically at runtime, which makes them very flexible. Memory is allocated / deallocated on demand, so additional memory management penalties are incurred.

4.25.4 Function Documentation

4.25.4.1 int32_t hf_list_append (struct list * lst, void * item)

Appends a new node to the end of the list.

Parameters

lst	is a pointer to a list structure.
item	is a pointer to data belonging to the list node.

Returns

0 when successful and -1 otherwise.

4.25.4.2 int32_t hf_list_count (struct list * lst)

Returns the number of nodes in a list.

Parameters

lst	is a pointer to a list structure.

Returns

The number of elements in the list.

4.25.4.3 void* hf_list_get (struct list * lst, int32_t pos)

Returns the address of the data belonging to a list node.

Parameters

lst	is a pointer to a list structure.
pos	is the n-th element position in the list.

Returns

0 when the element is not found and the address to data otherwise.

4.25.4.4 struct list* hf_list_init (void) [read]

Initializes a list.

Returns

a pointer to a list structure.

4.25.4.5 int32_t hf_list_insert (struct list * lst, void * item, int32_t pos)

Inserts a new node to an arbitrary position in a list.

Parameters

lst	is a pointer to a list structure.
item	is a pointer to data belonging to the list node.
pos	is the n-th element position in the list.

Returns

0 when successful and -1 otherwise.

4.25.4.6 int32_t hf_list_remove (struct list * lst, int32_t pos)

Removes an arbitrary node from a list.

Parameters

lst	is a pointer to a list structure.
pos	is the n-th element position in the list.

Returns

0 when successful and -1 otherwise.

4.25.4.7 int32_t hf_list_set (struct list * lst, void * item, int32_t pos)

Changes the address of the data belonging to a list node.

Parameters

lst	is a pointer to a list structure.
item	is an address to data belonging to the list node.
pos	is the n-th element position in the list.

Returns

-1 when the element is not found and 0 if the element was updated.

4.26 sys/lib/malloc.c File Reference

Functions

- void hf_free (void *ptr)
- void * hf_malloc (uint32_t size)
- void heapinit (void *heap, uint32_t len)
- void * hf_calloc (uint32_t qty, uint32_t type_size)
- void * hf_realloc (void *ptr, uint32_t size)

4.26.1 Function Documentation

```
4.26.1.1 void heapinit ( void * heap, uint32_t len )
4.26.1.2 void* hf_calloc ( uint32_t qty, uint32_t type_size )
4.26.1.3 void hf_free ( void * ptr )
4.26.1.4 void* hf_malloc ( uint32_t size )
4.26.1.5 void* hf_realloc ( void * ptr, uint32_t size )
```

4.27 sys/lib/queue.c File Reference

Functions

```
    struct queue * hf queue create (int32 t size)
```

Creates a queue of specified size.

int32_t hf_queue_destroy (struct queue *q)

Destroys a queue.

int32_t hf_queue_count (struct queue *q)

Counts the number of nodes in a queue.

int32_t hf_queue_addtail (struct queue *q, void *ptr)

Adds a node to the tail of the queue.

void * hf_queue_remhead (struct queue *q)

Removes a node from the head of the queue.

void * hf_queue_remtail (struct queue *q)

Removes a node from the tail of the queue.

void * hf_queue_get (struct queue *q, int32_t elem)

Returns a node from the queue.

• int32_t hf_queue_set (struct queue *q, int32_t elem, void *ptr)

Updates a node on the queue.

int32_t hf_queue_swap (struct queue *q, int32_t elem1, int32_t elem2)

Swap the position of two nodes in the queue.

4.27.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.27.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.27.3 DESCRIPTION

Queue manipulation primitives and auxiliary functions. Queue structures are allocated only on the creation of queues, so little additional overhead regarding memory management is incurred at runtime.

4.27.4 Function Documentation

4.27.4.1 int32_t hf_queue_addtail (struct queue * q, void * ptr)

Adds a node to the tail of the queue.

Parameters

q	is a pointer to a queue structure.
ptr	a pointer to data belonging to the queue node.

Returns

0 when successful and -1 otherwise.

4.27.4.2 int32_t hf_queue_count (struct queue * q)

Counts the number of nodes in a queue.

Parameters

Returns

the number of nodes.

4.27.4.3 struct queue* hf_queue_create (int32_t size) [read]

Creates a queue of specified size.

Parameters

size	is the maximum number of elements.

Returns

pointer to the queue on success and NULL otherwise.

4.27.4.4 int32_t hf_queue_destroy (struct queue * q)

Destroys a queue.

Parameters

q is a pointer to a queue structure.

Returns

0 when successful and -1 otherwise.

4.27.4.5 void* hf_queue_get (struct queue * q, int32_t elem)

Returns a node from the queue.

Parameters

q	is a pointer to a queue structure.
elem	is the n-th element from the queue.

Returns

pointer to node data success and 0 otherwise.

4.27.4.6 void* hf_queue_remhead (struct queue * q)

Removes a node from the head of the queue.

Parameters

q	is a pointer to a queue structure.

Returns

pointer to node data on success and 0 otherwise.

4.27.4.7 void* hf_queue_remtail (struct queue * q)

Removes a node from the tail of the queue.

Parameters

q	is a pointer to a queue structure.

Returns

pointer to node data on success and 0 otherwise.

4.27.4.8 int32_t hf_queue_set (struct queue * q, int32_t elem, void * ptr)

Updates a node on the queue.

Parameters

	q	is a pointer to a queue structure.
Ī	elem	is the n-th element from the queue.
	ptr	a pointer to data belonging to the gueue node.

Returns

0 success and -1 otherwise.

4.27.4.9 int32_t hf_queue_swap (struct queue * q, int32_t elem1, int32_t elem2)

Swap the position of two nodes in the queue.

Parameters

q	is a pointer to a queue structure.
elem1	is the first n-th element from the queue.
elem2	is the second n-th element from the queue.

Returns

0 when successful and -1 otherwise.

4.28 sys/sync/condvar.c File Reference

Functions

int32_t hf_condinit (cond_t *c)

Initializes a condition variable.

• int32_t hf_conddestroy (cond_t *c)

Destroys a condition variable.

void hf_condwait (cond_t *c, mutex_t *m)

Wait on a condition variable.

void hf_condsignal (cond_t *c)

Signal a condition variable.

void hf_condbroadcast (cond_t *c)

Signal (broadcast) a condition variable.

4.28.1 Detailed Description

Author

Sergio Johann Filho

Date

May 2016

4.28.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.28.3 DESCRIPTION

Condition variable synchronization primitives.

4.28.4 Function Documentation

4.28.4.1 void hf_condbroadcast (cond_t * c)

Signal (broadcast) a condition variable.

Parameters

c is a pointer to a condition variable.

Implements the condition signal broadcast operation for all waiting tasks. The call unblocks and removes all tasks from the waiting queue. If no tasks are waiting for the condition, the signal is lost.

4.28.4.2 int32_t hf_conddestroy (cond_t * c)

Destroys a condition variable.

Parameters

c is a pointer to a condition variable.

Returns

ERR_OK on success and ERR_ERROR if the condition variable could not be removed from memory.

4.28.4.3 int32_t hf_condinit (cond_t * c)

Initializes a condition variable.

Parameters

c is a pointer to a condition variable.

Returns

ERR_OK on success and ERR_ERROR if the condition variable could not be allocated in memory.

4.28.4.4 void hf_condsignal ($cond_t * c$)

Signal a condition variable.

Parameters

c is a pointer to a condition variable.

Implements the condition signal operation for one waiting task. The call removes a task from the waiting queue and unblocks it. If no tasks are waiting for the condition, the signal is lost.

4.28.4.5 void hf_condwait ($cond_t * c$, $mutex_t * m$)

Wait on a condition variable.

Parameters

С	is a pointer to a condition variable.
т	is a pointer to a mutex.

Implements the atomic condition wait operation. The call should always be invoked with the mutex locked. The current task is put in a queue on the condition variable, its state is set to blocked and unlocks the mutex atomically, then yields the processor. When woke up (by a signalling task), the task locks the mutex and returns.

4.29 sys/sync/mutex.c File Reference

Functions

void hf mtxinit (mutex t *m)

Initializes a mutex, defining its initial value.

void hf_mtxlock (mutex_t *m)

Locks a mutex.

void hf_mtxunlock (mutex_t *m)

Unlocks a mutex.

4.29.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.29.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.29.3 DESCRIPTION

Mutex synchronization primitives.

4.29.4 Function Documentation

4.29.4.1 void $hf_mtxinit(mutex_t * m)$

Initializes a mutex, defining its initial value.

Parameters

s	is a pointer to a mutex.

4.29.4.2 void hf_mtxlock (mutex_t * m)

Locks a mutex.

Parameters

```
s is a pointer to a mutex.
```

If the mutex is not locked, the calling task continues execution. Otherwise, the task spins.

```
4.29.4.3 void hf_mtxunlock ( mutex_t * m )
```

Unlocks a mutex.

Parameters

s is a pointer to a mutex.

4.30 sys/sync/semaphore.c File Reference

Functions

```
• int32_t hf_seminit (sem_t *s, int32_t value)
```

Initializes a semaphore and defines its initial value.

int32_t hf_semdestroy (sem_t *s)

Destroys a semaphore.

void hf_semwait (sem_t *s)

Wait on a semaphore.

void hf_sempost (sem_t *s)

Signal a semaphore.

4.30.1 Detailed Description

Author

Sergio Johann Filho

Date

February 2016

4.30.2 LICENSE

This source code is licensed under the GNU General Public License, Version 2. See the file 'doc/license/gpl-2.0.txt' for more details.

4.30.3 DESCRIPTION

Semaphore synchronization primitives.

4.30.4 Function Documentation

4.30.4.1 int32_t hf_semdestroy ($sem_t * s$)

Destroys a semaphore.

Parameters

s	is a pointer to a semaphore.

Returns

ERR_OK on success and ERR_ERROR if the semaphore could not be removed from memory.

4.30.4.2 int32_t hf_seminit ($sem_t * s$, int32_t value)

Initializes a semaphore and defines its initial value.

Parameters

S	is a pointer to a semaphore.
value	is the semaphore initial value.

Returns

ERR_OK on success and ERR_ERROR if the semaphore could not be allocated in memory or its initial value is less than zero.

4.30.4.3 void hf_sempost ($sem_t * s$)

Signal a semaphore.

Parameters

s	is a pointer to a semaphore.

Implements the atomic V() operation. The semaphore count is incremented and the task from the head of the semaphore queue is unblocked if the count is less than or equal to zero.

4.30.4.4 void hf_semwait ($sem_t * s$)

Wait on a semaphore.

Parameters

S	is a pointer to a semaphore.

Implements the atomic P() operation. The semaphore count is decremented and calling task is blocked and queued on the semaphore if the count reaches a negative value. If not, the task continues its execution.

Index

```
list.c
     list_append, 5
     list_count, 5
     list_get, 5
     list_init, 5
     list_insert, 5
     list_remove, 6
     list_set, 6
list_append
     list.c, 5
list_count
     list.c, 5
list_get
     list.c, 5
list_init
     list.c, 5
list_insert
     list.c, 5
list_remove
     list.c, 6
list_set
     list.c, 6
main
     main.c, 3
main.c
     main, 3
sys/kernel/main.c, 3
sys/lib/list.c, 4
```