

Implementation of the ART Real Time Micro Kernel

This document describes the functions constituting the application interface of the ART kernel.

Task administration

exception `init_kernel()`

This function initializes the kernel and its data structures and leaves the kernel in start-up mode. The `init_kernel` call must be made before any other call is made to the kernel.

Argument

none

Return parameter

Int: Description of the function's status, i.e. FAIL/OK.

Function

Set tick counter to zero
 Create necessary data structures
 Create an idle task
 Set the kernel in start up mode
 Return status

exception `create_task(void(*task_body)(), uint deadline)`

This function creates a task. If the call is made in start-up mode, i.e. the kernel is not running, only the necessary data structures will be created. However, if the call is made in running mode, it will lead to a rescheduling and possibly a context switch.

Argument

`*task_body`: A pointer to the C function holding the code of the task.

`deadline`: The kernel will try to schedule the task so it will meet this deadline.

Return parameter

Description of the function's status, i.e. FAIL/OK.

Function

Allocate memory for TCB
 Set deadline in TCB
 Set the TCB's PC to point to the task body
 Set TCB's SP to point to the stack segment
IF start-up mode **THEN**
 Insert new task in Readylist
 Return status
ELSE
 Disable interrupts
 Save context
 IF "first execution" **THEN**
 Set: "not first execution any more"
 Insert new task in Readylist
 Load context
ENDIF
ENDIF
 Return status

void run()

This function starts the kernel and thus the system of created tasks. Since the call will start the kernel it will leave control to the task with tightest deadline. Therefore, it must be placed last in the application initialization code. After this call the system will be in running mode.

Argument

none

Return parameter

none

Function

Initialize interrupt timer
Set the kernel in running mode
Enable interrupts
Load context

void terminate()

This call will terminate the running task. All data structures for the task will be removed. Thereafter, another task will be scheduled for execution.

Argument

none

Return parameter

none

Function

Remove running task from Readylist
Set next task to be the running task
Load context

Inter-Process Communication

Mailbox* create_mailbox(int nof_msg, int size_of_msg)

This call will create a Mailbox. The Mailbox is a FIFO communication structure used for asynchronous and synchronous communication between tasks.

Argument

nof_msg: Maximum number of Messages the Mailbox can hold.

Size_of msg: The size of one Message in the Mailbox.

Return parameter

Mailbox*: a pointer to the created mailbox or NULL.

Function

Allocate memory for the Mailbox
Initialize Mailbox structure
Return Mailbox*

exception remove_mailbox(Mailbox* mBox)

This call will remove the Mailbox if it is empty and return OK. Otherwise no action is taken and the call will return NOT_EMPTY.

Argument

Mailbox*: A pointer to the Mailbox to be removed.

Return parameter

OK: The mailbox was removed

NOT_EMPTY: The mailbox was not removed because it was not empty.

Function

IF Mailbox is empty **THEN**
Free the memory for the Mailbox
Return OK

ELSE
Return NOT_EMPTY

ENDIF

exception send_wait(Mailbox* mBox, void* Data)

This call will send a Message to the specified Mailbox. If there is a receiving task waiting for a Message on the specified Mailbox, send_wait will deliver it and the receiving task will be moved to the Readylist. Otherwise, if there is not a receiving task waiting for a Message on the specified Mailbox, the sending task will be blocked. In both cases (blocked or not blocked) a new task schedule is done and possibly a context switch. During the blocking period of the task its deadline might be reached. At that point in time the blocked task will be resumed with the exception: DEADLINE_REACHED. *Note: send_wait and send_no_wait Messages shall not be mixed in the same Mailbox.*

Argument

*mBox a pointer to the specified Mailbox.

*Data: a pointer to a memory area where the data of the communicated Message is residing.

Return parameter

exception: The exception return parameter can have two possible values:

- OK: Normal behavior, no exception occurred.
- DEADLINE_REACHED: This return parameter is given if the sending tasks' deadline is reached while it is blocked by the send_wait call.

Function

Disable interrupt

Save context

IF first execution **THEN**

Set: "not first execution any more"

IF receiving task is waiting **THEN**

Copy sender's data to the data area of the receivers Message

Remove receiving task's Message struct from the mailbox

Move receiving task to Readylist

ELSE

Allocate a Message structure

Set data pointer

Add Message to the Mailbox

Move sending task from Readylist to

Waitinglist

ENDIF

Load context

ELSE

IF deadline is reached **THEN**

Disable interrupt

Remove send Message

Enable interrupt

Return DEADLINE_REACHED

ELSE

Return OK

ENDIF

ENDIF

exception receive_wait(Mailbox* mBox, void* Data)

This call will attempt to receive a Message from the specified Mailbox. If there is a send_wait or a send_no_wait Message waiting for a receive_wait or a receive_no_wait Message on the specified Mailbox, receive_wait will collect it. If the Message was of send_wait type the sending task will be moved to the Readylist. Otherwise, if there is not a send Message (of either type) waiting on the specified Mailbox, the receiving task will be blocked. In both cases (blocked or not blocked) a new task schedule is done and possibly a context switch. During the blocking period of the task its deadline might be reached. At that point in time the blocked task will be resumed with the exception: DEADLINE_REACHED.

Argument

*mBox: a pointer to the specified Mailbox.

*Data: a pointer to a memory area where the data of the communicated Message is to be stored.

Return parameter

exception: The exception return parameter can have two possible values:

- OK: Normal function, no exception occurred.
- DEADLINE_REACHED: This return parameter is given if the receiving tasks' deadline is reached while it is blocked by the receive_wait call.

Function

Disable interrupt

Save context

IF first execution **THEN**

Set: "not first execution any more"

IF send Message is waiting **THEN**

Copy sender's data to receiving task's data area

Remove sending task's Message struct from the Mailbox

IF Message was of wait type **THEN**

Move sending task to Ready list

ELSE

Free senders data area

ENDIF

ELSE

Allocate a Message structure

Add Message to the Mailbox

Move receiving task from Readylist to

Waitinglist

ENDIF

Load context

ELSE

IF deadline is reached **THEN**

Disable interrupt

Remove receive Message

Enable interrupt

Return DEADLINE_REACHED

ELSE

Return OK

ENDIF

ENDIF

exception send_no_wait(Mailbox* mBox, void* Data)

This call will send a Message to the specified Mailbox. The sending task will continue execution after the call. When the Mailbox is full, the oldest Message will be overwritten. The send_no_wait call will imply a new scheduling and possibly a context switch. *Note: send_wait and send_no_wait Messages shall not be mixed in the same Mailbox.*

Argument

*mBox: a pointer to the specified Mailbox.

*Data: a pointer to a memory area where the data of the communicated Message is residing.

Return parameter

Description of the function's status, i.e. FAIL/OK.

Function

Disable interrupt

Save context

IF first execution **THEN**

Set: "not first execution anymore"

IF receiving task is waiting **THEN**

Copy data to receiving tasks' data area.

Remove receiving task's Message struct from the Mailbox

Move receiving task to Readylist

Load context

ELSE

Allocate a Message structure

Copy Data to the Message

IF mailbox is full **THEN**

Remove the oldest Message struct

ENDIF

Add Message to the Mailbox

ENDIF

ENDIF

Return status

exception receive_no_wait(Mailbox* mBox, void* Data)

This call will attempt to receive a Message from the specified Mailbox. The calling task will continue execution after the call. When there is no send Message available, or if the Mailbox is empty, the function will return FAIL. Otherwise, OK is returned. The call might imply a new scheduling and possibly a context switch.

Argument

*mBox: a pointer to the specified Mailbox.

*Data: a pointer to the Message.

Return parameter

Integer indicating whether or not a Message was received (OK/FAIL).

Function

Disable interrupt

Save context

IF first execution **THEN**

Set: "not first execution any more"

IF send Message is waiting **THEN**

Copy sender's data to receiving task's data area

Remove sending task's Message struct from the Mailbox

IF Message was of wait type **THEN**

Move sending task to Readylist

ELSE

Free sender's data area

ENDIF

ENDIF

Load context

ENDIF

Return status on received Message

Timing functions

exception wait(uint nTicks)

This call will block the calling task until the given number of ticks has expired.

Argument

nTicks: the duration given in number of ticks

Return parameter

exception: The exception return parameter can have two possible values:

- OK: Normal function, no exception occurred.
- DEADLINE_REACHED: This return parameter is given if the receiving tasks' deadline is reached while it is blocked by the receive_wait call.

Function

Disable interrupt
Save context

IF first execution THEN

Set: "not first execution any more"
Place running task in the Timerlist
Load context

ELSE

IF deadline is reached THEN
Status is DEADLINE_REACHED

ELSE

Status is OK

ENDIF

ENDIF

Return status

void set_ticks(uint nTicks)

This call will set the tick counter to the given value.

Argument

nTicks: the new value of the tick counter

Return parameter

none

Function

Set the tick counter.

uint ticks(void)

This call will return the current value of the tick counter

Argument

none

Return parameter

A 32 bit value of the tick counter

Function

Return the tick counter

uint deadline(void)

This call will return the deadline of the specified task

Argument

none

Return parameter

the deadline of the given task

Function

Return the deadline of the current task

void set_deadline(uint deadline)

This call will set the deadline for the calling task. The task will be rescheduled and a context switch might occur.

Argument

deadline: the new deadline given in number of ticks.

Return parameter

none

Function

Disable interrupt

Save context

IF first execution **THEN**

Set: "not first execution any more"

Set the deadline field in the calling TCB.

Reschedule Readylist

Load context

ENDIF

void TimerInt(void)

This function is not available for the user to call. It is called by an ISR (Interrupt Service Routine) invoked every tick. *Note, context is automatically saved prior to call and automatically loaded on function exit.*

Function

Increment tick counter

Check the Timerlist for tasks that are ready for execution, move these to Readylist

Check the Waitinglist for tasks that have expired deadlines, move these to Readylist and clean up their Mailbox entry.

void SaveContext (void) [Ⓢ]

This function is hardware dependent. All relevant registers are saved to the TCB of the currently running task.

Function

Save PC in Running->PC

Save A0-A15 in Running->Context[0-15]

Save B0-B15 in Running->Context[16-31]

void LoadContext (void) [Ⓢ]

This function is hardware dependent. All relevant registers are restored from the TCB of the currently running task to the CPU registers.

Function

Load B0-B15 from Running->Context[16-31]

Load A0-A15 from Running->Context[0-15]

Set SP to Running->SP

Enable Interrupts

Restore PC from Running->PC

Jump to Running->PC

void timer0_isr (void) [Ⓢ]

This function is not available for the user to call. It is an ISR (Interrupt Service Routine) invoked every tick. *Note. It calls the C-Function TimerInt().*

Function

Disable Interrupts

Save A0-A15 in Running->Context[0-15]

Save B0-B15 in Running->Context[16-31]

Save SP in Running->SP

Call C-function TimerInt()

Load B0-B15 from Running->Context[16-31]

Load A0-A15 from Running->Context[0-15]

Set SP to Running->SP

Enable Interrupts

Restore PC from Running->PC

Jump to Running->PC

[Ⓢ] Provided by course admin.