TTPos Operating System Model

J. Jun J. Hlavicka,
Czech Technical University in Prague

Our team received the break-by-wire application in a form of C-functions. The functions should be run as tasks on the host computer. We decided to implement a simplified version of the TTPos (see Subsection 1.2.1) in our model in order to be able to run tasks in a specific time relatively to the beginning of the cluster cycle. The secondary benefit of the TTPos implementation is making further development of applications easier even without a deeper knowledge of the C-Sim environment. The C-Sim environment is in the most part screened out by the functions offered by the operating system.

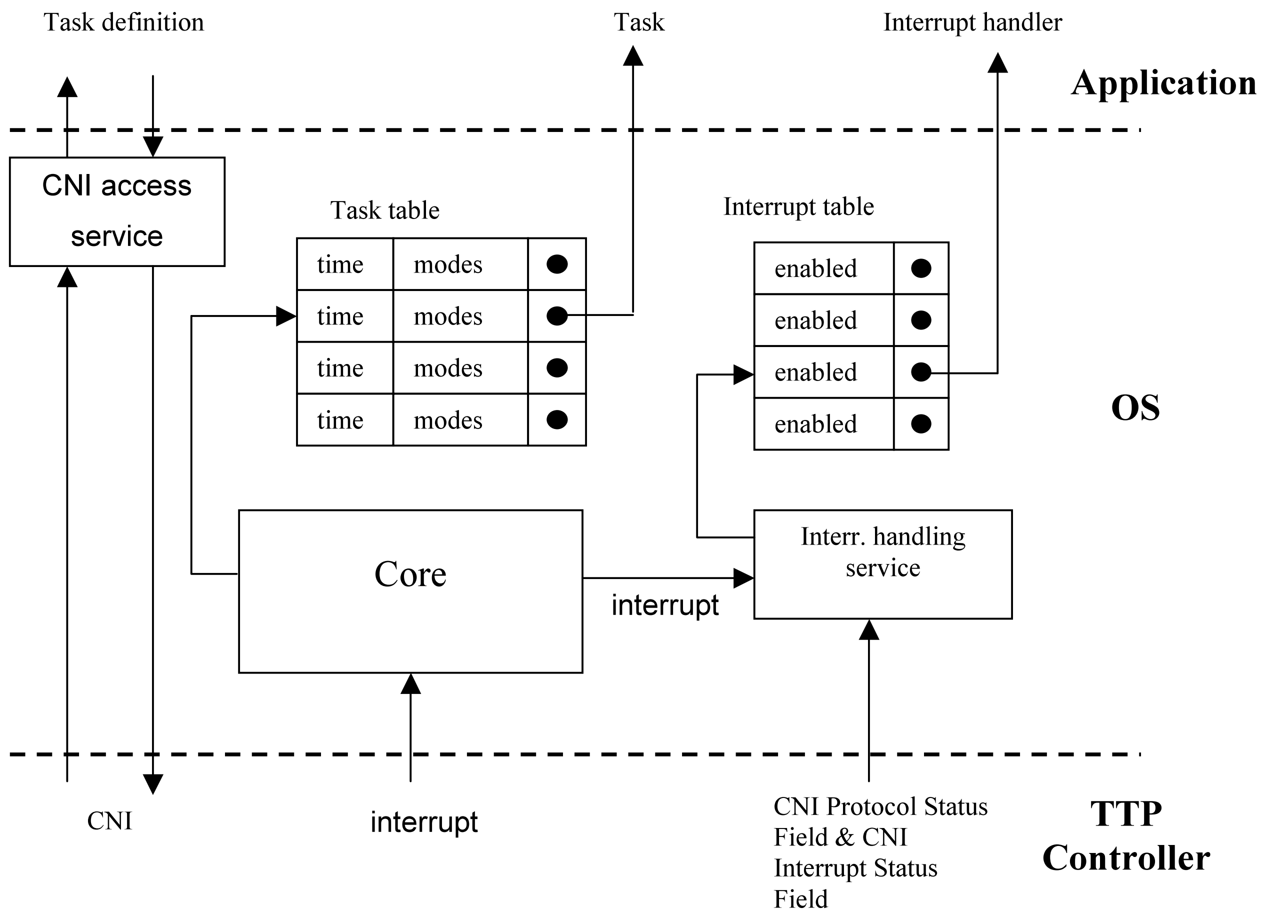

Figure 1: TTPos simulation

1 Structure

The model of the operating system consists of three conceptual parts, the core of the operating system, the interrupt handling service, and the CNI access service (**Figure 1**). The real TTPos runs a fixed number of tasks at time points defined in the TTPBuild tool. In the TTP/C simulation a task has a form of a C-function. The pointer to this function is stored in the *Task table* data structure. The *core* of the operating system has the task to run cyclically all tasks registered in the Task table. An entry of the Task table contains also the time field and the modes field. The *time field* defines the time point of the task execution. Its value is in macroticks

and is relative to the beginning of the cluster cycle. The *modes field* defines the modes in which the task will be executed. Another important data structure of the operating system is the *Interrupt table*. The pointers to the particular interrupt handlers are stored in this table. Each entry has an enable switch which determines whether the interrupt handler will be called if the controller raises the corresponding interrupt. The access module facilitates storing and reading data from the message area of the CNI and some other important fields of the CNI.

2 Core

2.1 Functionality

The core of the operating system fulfills the following tasks:

1. turns the controller on
2. waits for the controller ready field to be set
3. sets the interrupt enable field
4. synchronizes with the controller
5. schedules tasks and host life-sign update by means of time interrupt TI1
6. guards the timeout for the interrupts
7. activates the interrupt handling module

Items 1 to 4 are specified in [Spec99] section 10.1.1. We received a more detailed information about the mechanism of the task scheduling from TTTech GmbH.

2.2 Implementation

The core of the OS was implemented as a state machine whose behavior is depicted in **Figure 3-2**.

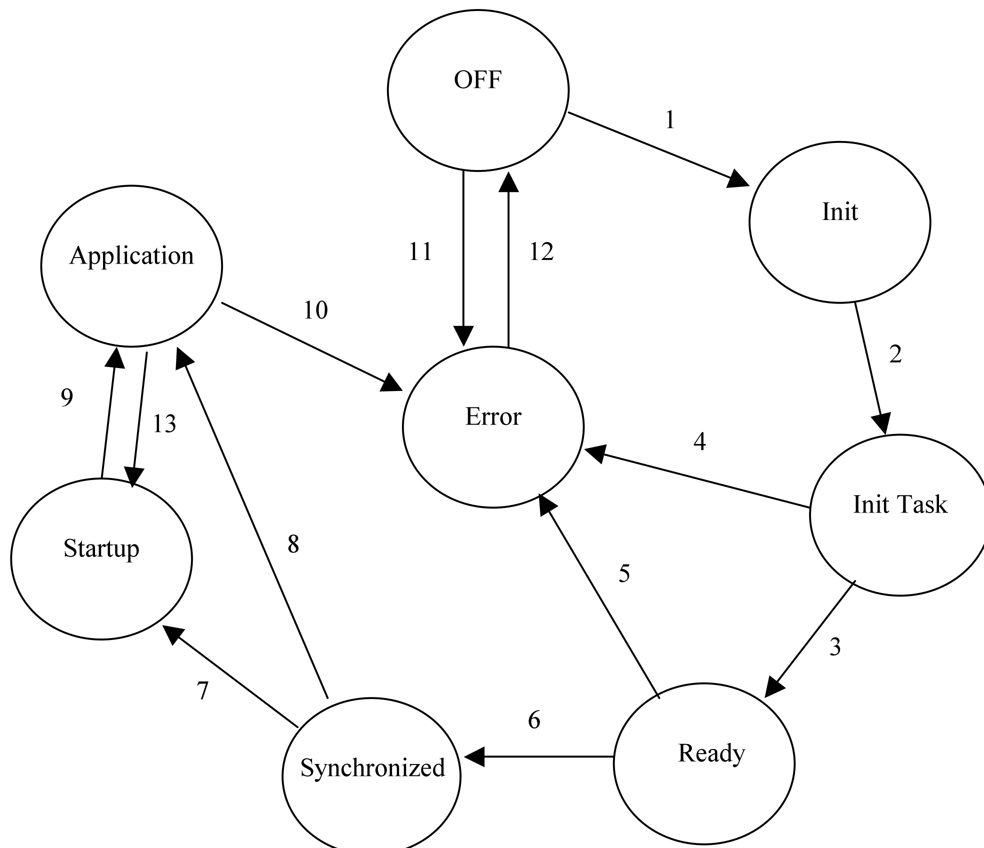

Figure 2: Core states

The actions associated with the states are listed in **Table1**. The conditions of the transitions between these states are listed in **Table 2**.

state	action
OFF	no action
Init	Initialization of the OS
Init Task	enabling the interrupts, running the initialization task
Ready	updates the life-sign every TDMA round, waits for valid C-State (node is synchronized)
Synchronized	starts the tasks execution
Startup	passes interrupt signals, executes tasks, guards the timeouts, triggers the mode change
Application	passes interrupt signals, executes tasks, guards the timeouts
Error	executes the error handler

Table 1 States of the core

No.	From	To	Event
1	OFF	Init	start allowed by application
	Init	Init Task	always
3	Init	Ready	controller ready
4	Init	Error	controller not ready
5	Ready	Error	node did not synchronize within one cluster cycle
6	Ready	Synchronized	node is synchronized with the cluster
7	Synchronized	Startup	current mode is not the application mode
8	Synchronized	Application	current mode is the application mode
9	Startup	Application	current mode is the application mode
10	Application	Error	protocol error or bus guardian error occurred or timeout expired.
11	OFF	Error	controller ON
12	Error	OFF	always
13	Application	Startup	current mode is the startup mode

Table 2 State transitions of the core

3 Task Scheduling

3.1 Mechanism

The mechanism of scheduling the tasks uses the time interrupt ti1. The controller rises a time interrupt, the OS runs a task and sets the timer1 according to the time difference between the current task and the following one. Then the OS process falls to sleep and waits in a sleep state to be woken up by the scheduled time interrupt. If no interrupt is received for an interval longer then one TDMA round, the OS wakes up and transits into the ERROR state.

3.2 Life-Sign Task and Ordinary Task

There are two kinds of tasks. The “host life-sign update” task and the ordinary tasks. Both are scheduled by the same mechanism. The “host life-sign update” task updates the life-sign of the host which must be done every TDMA round. Hence every “host life-sign update” task should be set active in all used modes (i.e. the matching bit in the *modes* field should be set). No tasks should be run in the startup mode except the life-sign update.

4 Interrupt Handling Service

4.1 Mechanism

The core of the OS activates the interrupt handling service if the controller raised an interrupt[1] signal or if an internal operating system interrupt occurred. If the controller raised the interrupt the interrupt handling service can gather information on the cause of the interrupt from the interrupt status field of the CNI. This information is stored in the internal data structure called *EDM-counter* accessible to the application. If an internal OS interrupt occurs the cause is stored in the *EDM-counter* as well. Finally a corresponding interrupt handler is triggered.

4.2 EDM-Counter fields

The fields of the *EDM-counter* data structure are described in **Table 3-3**:

name	**description**
mi	Mode change interrupt
ml	Membership lost interrupt
mc	Membership changed interrupt
ui1	User interrupt 1
ui2	User interrupt 2
ti1[2]	Time interrupt 1
ti2	Time interrupt 2
be	BIST error
pe_be	Protocol error: bus guardian error
pe_ae	Protocol error: acknowledgement error
pe_se	Protocol error: clock synchronization error
pe_me	Protocol error: max. successive membership failure reached.
pe_mc	Protocol error: MEDL CRC error
pe_cb	Protocol error: communication blackout
pe_dc	Protocol error: download completed
he_so	Host error: slot occupied
he_mv	Host error: mode violation
he_ne	Host error: NBW protocol error or incorrect EIF
os_error	OS detected errors[3]

Table 3-3 EDM-counter fields

5 CNI Access Service

The CNI Access Service is a set of functions facilitating the following operations:

a) Storing and reading data into/from the message area of the CNI

b) Triggering a mode change

c) Reading information about the state of the controller

[1] The TTP/C C1 Controller uses one signal line to inform the host about an interrupt occurrence.

[2] The ti1 should not be used by the application since it is used for task scheduling

[3] In the current OS implementation only the time-out OS error can be raised and no further distinction is necessary.

5.1 Storing and Reading Data into/from Message Area of CNI

The **message area** of CNI serves for exchanging application data between the host and the communication controller. The layout of this area is determined by the current MEDL, which contains the addresses of all messages sent/received by the SRU. An entry of the CNI message area consists of the message status field and its associated message data field. The message status field provides information on the transmission of the frames. The CNI Access service checks the status field in case of reading and sets correctly the status field in case of storing data into the message area. Various storing/reading functions were implemented for various types of data.

5.2 Mode Change Triggering

The mode change from the startup mode to the application mode is triggered automatically by the core of the OS. This service must be initialized before starting the cluster (i.e. the minimal number of nodes and the number of application mode must be defined). The mode changes between the application modes can be triggered by the call of the *os_request_mode_change* function. So far only deferred mode change was implemented.

5 3 Controller State Information

The CNI Access service offers some functions for obtaining the basic information about the state of the controller. An information about

- current global cluster time
- current slot
- time relative to the beginning of the cluster cycle

can be obtained.

6 Application Programmer Interface

Here we give a description of the most important functions of the operating system model. Description of all implemented functions can be found in the TTPos manual which can be obtained at the CTU Prague or in the header files of the operating system model.

6 1Creating and Initializing Controller and Channel

The controller and channel model creation and initialization is described in detail in [Jez01] and [Gri01]. Following actions have to be performed:

- creating two channel instances
- creating the controller instances
- initializing the channels
  - setting the channel ID to 0 or 1
  - setting the bus guardian callback function
- initializing the controllers
  - attaching the controller to the channels
  - setting the drift rate of the local clock
  - reading the MEDL of the controller

6 2 Creating and Initializing a TTPos

A TTPos instance has to be crated by calling:

OS_PROCESS **os_create**(char * name)*
input: *name*........name of the instance chosen by the programmer
return: pointer to the TTPos instance

The TTPos instance has to be attached to a controller instance by calling:

*void **os_attach_controller**(OS_PROCESS* os, CONTROLLER* cntrl)*
input: *os*…....pointer to the TTPos instance
cntrl…..pointer to the controller instance

During the start up the cluster transits from the STARTUP mode to the APPLICATION mode. We define the conditions of the transition and we choose the exact application mode by calling:

void ***os_set_mode_trigger****(OS_PROCESS *os, unsigned min_cnt, unsigned mode)*

input: *os*..........pointer to the TTPos instance
min_cnt…minimal number of nodes synchronized with the cluster
mode….....the number of the destination mode

6 3 Adding Life-sign Update

unsigned ***os_add_lifesign_update****(OS_PROCESS* os, unsigned time)*

input: *os*..........pointer to the TTPos instance
time…....time of execution in macro-ticks relative to the beginning of the cluster cycle

return: index of the life-sign update task in task table or –1 if an error occurred

6 4 Adding Application Task

unsigned ***os_add_task****(OS_PROCESS *os, char *name, unsigned time, unsigned modes, OS_TASK_FN prog, void *prog_arg)*

input: *os*..........pointer to the TTPos instance
name.....name of the task chosen by the programmer
time.......time of execution in macro-ticks relative to the beginning of the cluster
modes...a 32 bit mask defining in which modes the task should be executed
prog......the pointer to the function defining the body of the task
arg........the arguments of the *prog* function

return: index of the life-sign update task in task table or –1 if an error occurred

The TTPos executes an initialization task at the startup of the node. An initialization task can be added by calling:

void ***os_add_init_task****(OS_PROCESS *os, OS_TASK_FN prog, void *prog_arg)*

input: *os*..........pointer to the TTPos instance
prog......the pointer to the function defining the body of the task
arg........the arguments of the *prog* function

return: index of the life-sign update task in task table or –1 if an error occurred

6 5 Interrupt Handling Service

We can insert a pointer to a interrupt handler into the interrupt handler by calling:

void ***os_add_xx_interrupt****(OS_PROCESS *os, OS_TASK_FN prog, void *arg)*

where xx is the interrupt identifier according to the left column of **Table 3-3**.

input: *os*..........the pointer to the TTPos instance
prog…....the pointer to the function defining the body of the handler
arg….....the arguments of the *prog* function

An interrupt can be enabled or disabled by calling:

void ***os_enable_****xx****_int****(OS_PROCESS *os, unsigned state)*

where xx is the interrupt identifier according to the left column of **Table 3-3**.

input: *state*…...1 for enabling, 0 for disabling

implicitly all interrupts are disabled.

We can receive the EDM data structure by:

*OS_EDM_COUNTERS ****os_get_edm_counters****(OS_PROCESS *os)*

input: *os*..........the pointer to the TTPos instance

return: the current EDM data structure

We can clear the EDM data structure by calling:

void ***os_clear_edm_counters****(OS_PROCESS *os)*

input: *os*..........the pointer to the TTPos instance

6 6 CNI Access Service

The layout of the message area is determined by the current message descriptor list (MEDL), which contains the addresses of all messages sent/received by the node. An entry of the CNI message area consists of the message **status field** and its associated message **data field**. The message status field provides information on the transmission of the frame(s). It allows the host to detect if the message has been correctly received or not. It consists of the **Error Indication Field (EIF)** and the **Concurrency Control Field (CCF)**. The MSB of the EIF, which is called the reception status (RS) flag, determines whether the frame was received correctly. The value of the EIF is used for the frame from the controller to the host and from the host to the controller as well. Therefore the host application has to set the EIF to a reasonable value every time a frame is copied to the CNI (i.e., the RS flag must be set). When the controller reads a frame from the CNI, it examines the value of the RS flag. If this flag is not set, a host error is raised and the controller transits into the passive state. Only if the RS bit is set a transmission of the frame takes place. When the controller itself writes a frame to the CNI, the EIF is set according to the reception status of the frame. The CCF is used by the non-blocking write protocol when accessing a message [Spec99].

Validating the EIF field for one channel:
void ***os_validate_eif****(OS_PROCESS *os, unsigned addr)*
input: *os*...........the pointer to the TTPos instance
addr....address of the message in the CNI message area
Validating the EIF field for both channels:
void ***os_validate_rep_eif****(OS_PROCESS *os, unsigned addr[2])*
input: *os*...........the pointer to the TTPos instance
addr.......addresses of the messages in the CNI message area

Setting the data field for one channel:
void ***os_set_data****(OS_PROCESS *os, unsigned addr, char *buf, unsigned size);*
input: *os*...........the pointer to the TTPos instance
addr.......address of the message in the CNI message area
buf..........pointer to the data
size........size of the data
Setting the data field for both channels:
void ***os_set_rep_data****(OS_PROCESS *os, unsigned addr[2], char *buf, unsigned size);*
input: *os*...........the pointer to the TTPos instance
addr.......addresses of the messages in the CNI message area
buf..........pointer to the data
size........size of the data

Reading the data field for one channel:
unsigned ***os_get_data****(OS_PROCESS *os, unsigned addr, char *buf, unsigned size);*
input: *os*...........the pointer to the TTPos instance
addr.......address of the message in the CNI message area
size........size of the data
output: buf..........pointer to the data
return: 0 if RS of the EIF not set, >0 if the RS of the EIF set
Reading the data field for both channels:
unsigned ***os_get_rep_data****(OS_PROCESS *os, unsigned addr[2], char *buf, unsigned size);*
input: *os*...........the pointer to the TTPos instance
addr.......addresses of the messages in the CNI message area
size........size of the data
output: buf..........pointer to the data
return: 0 if RS of none EIF not set, >0 if the RS of one of the EIF set
The stores into the 'buf' the value from the first channel if it is valid. If it is not valid the value from the second channel is stored.

We can trigger a mode change by calling:
unsigned ***os_request_mode_change****(OS_PROCESS *os, unsigned mode, unsigned immediate);*
input: *os*...........the pointer to the TTPos instance

mode.....the relative mode address (see [Spec99])

immediate..1 for immediate change, 0 for deferred change

output: 0 if successful, nonzero if not permitted

Following function deliver information about:

- time in macro-ticks relative to the beginning of the cluster cycle:
  unsigned ***os_get_time****(OS_PROCESS *os)*
- my sending slot and the current slot

  unsigned ***os_get_my_slot****(OS_PROCESS *os)*
  unsigned ***os_get_current_slot****(OS_PROCESS *os)*
- current cluster cycle

  unsigned ***os_get_transmission_cycle****(OS_PROCESS *os)*

The input parameter is the pointer to the TTPos instance.