ESIC™ - Embedded Systems Infrastructure Component

Generated Documentation

Pre-emptive Operating System v 1.4.0

Embedded Artists

Embedded Artists AB

Västerås Technology Park
Glödgargränd 14
SE-721 30 Västerås
Sweden
Phone +46 (21) 470 22 00
Fax +46 (21) 470 22 00

info@EmbeddedArtists.com
http://www.EmbeddedArtists.com

Table of Contents

1 Configuration

The ESIC **Pre-emptive Operating System v 1.4.0** was generated on 2005-03-15 20:27:10. The configuration settings used in the code generation are summarized in Table 1 below.

Configuration	**Value**
Instances	Multiple
Restrictions	A process can have any priority
Specify number of priorities	5
Specify number of processes	5
Include support for cyclic scheduling	false
Process Creation	Dynamic
Hardware Abstraction Layer	Philips LPC2xxx Series
Compiler	GCC
Instruction Mode for Processes	Thumb mode
Tick Timer	Timer 0
Prescale	0
Ticks	147459
Disable/Enable Interrupt	Save current setting
Counting semaphore	true
Binary semaphore	false
Signal	false
Queue	true
API	Both
Counter Size	16-bit
Enable semaphore counter limit	false
Queue API	Both
Idle process	false
Timer process	true
Suspend/resume	true
Dynamic Memory Manager	None
Enable stack usage statistics	true
Error Reporting	Return error code via supplied pointer
Check that supplied process identification numbers are correct	true
Check that supplied priorities are correct	true
Check interrupt context	true
Check pointers	true
Check process allocation	true
Include debug support	false
Generate a sample application that illustrates the functionality	false

Table 1: Configurations

2 Description

2.1 Process Control Block

The process control block (PCB) contains all relevant information about a process instance that the RTOS needs in order to control the execution. The table below lists the data fields in the process control block.

pStk	Pointer to the top of the stack.
pid	The process identification descriptor, which is a sequential number.
pNextPrioQueueReady	The PCB is placed in a double linked list - one list for each priority level. The list function as a priority list/queue. This forward pointer is used for the ready queue.
pPrevPrioQueueReady	Previous pointer in the double linked list (prioritized queue). Also see pNextPrioQueueReady description above.
pNextPrioQueueEvent	The PCB is placed in a double linked list - one list for each priority level. The list function as a priority list/queue. This forward pointer is used for event queues. A process may only be placed in one event queue at the time.
pPrevPrioQueueEvent	Previous pointer in the double linked list (prioritized queue). Also see pNextPrioQueueEvent description above.
pNextTimeQueue	If the PCB is placed in a (single linked) timeout list, this pointer points to the next PCB in the timeout list. The list is a delta list, i.e., each position only contains the difference (in time) from the previous position.
prio	The process priority. The lower number, the higher priority.
sleep	Number of system ticks to sleep. This number is relative to the previous PCB in the timeout list, i.e., the list is a delta list.
flag	Process state flags.
pStkOrg	Pointer to the start of the original stack area, i.e., the stack area supplied when the process was created. The information is used when checking stack usage.
stackSize	The size of the initial stack area. The information is used when checking stack usage.

Table 2: Process Control Block Data Structure

The stack pointer (pStk) is placed first in the data structure since this makes it more convenient to access it from assembly language. Some functions in the HAL is typically written in assembly language.

The RTOS must be able to access a PCB quickly, since it is a very common operation. All PCB:s in a system is typically organized in a data structure (described further in section 'Priorities'). This data structure is also commonly called the 'ready list'.

2.2 Process Model

The process model defines the lifecycle and certain properties of the processes. This section describes the entities and properties related to the process model. The process entry function is one fundamental entity in a multiprocessing (also commonly called multitasking) environment and is described in a subsection of its own. In the priority subsection the ready list organization and scheduling is described since these are closely related to priority handling. The last subsection of this section describes how process creation works internally.

2.2.1 Process Function

The process function is the entry point for the process. It is supplied to the operating system when a process is created.

A process in the system must run forever, i.e., the process function is not allowed to do an exit or return. However, even with this restriction it is possible for the process to terminate itself by calling the deleteProcess operating system function.

Since a process can have multiple instantiations there must be a way to pass different information to each instance. A process function takes one argument of type: void pointer, in which such information can be passed. It is up to the application designer/programmer to decide what information to pass. The actual information that is passed to an instance is specified when the process instance is created (by a call to the processCreate operating system function).

2.2.2 Priorities

The operating system always runs the process with the highest priority (selected amongst the processes that are ready to run). If a process with higher priority (than the currently running process) gets ready to run, the running process will be pre-empted and the process with the higher priority is run instead.

A process in the system is allowed to have any priority from zero to the maximum number of priorities minus one. The lower the number, the higher the priority (which is the de facto standard). Zero is hence the highest priority and the maximum number minus one is the lowest priority.

If two processes have identical priority they will be run in a round-robin fashion. A re-scheduling occurs every operating system tick, as well as in some operating system functions. On average, processes with the same priority will get the same amount of processor time (provided that they do not enter a blocking state).

The operating system maintains a so called ready list, which containing all processes ready to run. The ready list is organized as an array indexed with priority and every element of the array points to a double linked list of process control blocks (PCB) for processes with the same priority. Conceptually this is the same as having one ready-list for each priority level, as illustrated below.

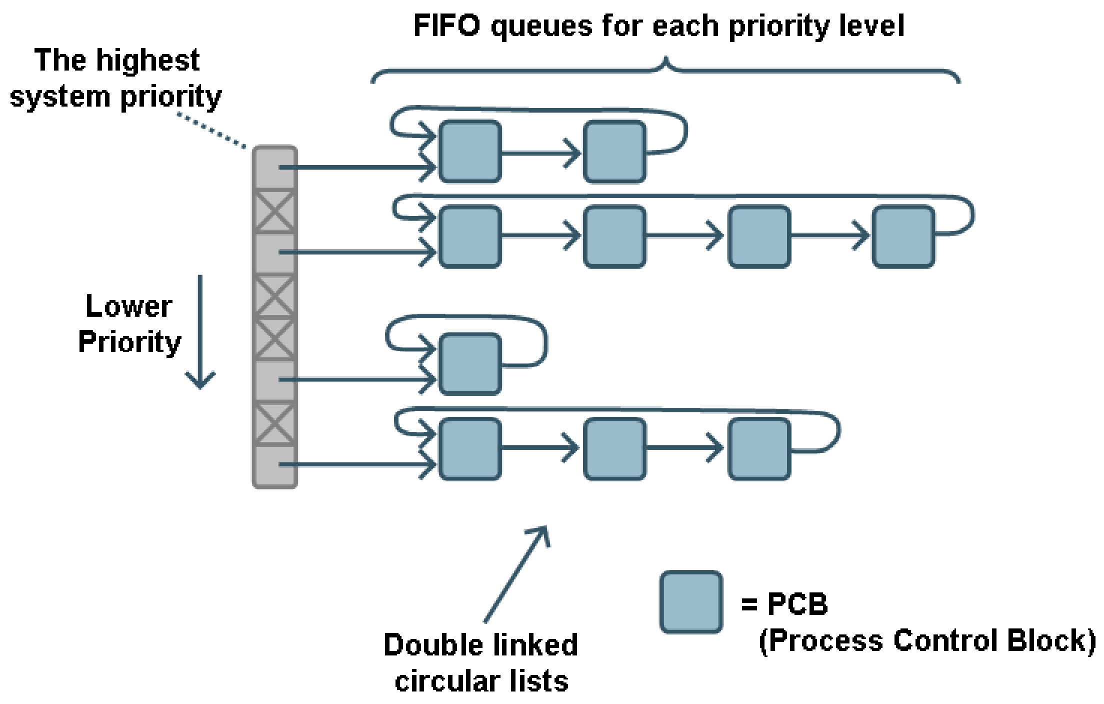
FIFO queues for each priority level
The highest system priority
Lower Priority
Double linked circular lists
= PCB (Process Control Block)

Figure 1: PCB Ready List Organization

During a context switch the operating system searches the array, implementing the ready list, starting with the highest priority, i.e. priority zero, and continues until a non-empty list is found. The first process is picked from the list and the list is circulated to obtain round-robin scheduling.

Observe that only PCB:s for processes that are ready to run are placed in the ready list. Processes in a blocking state have their respective PCB:s placed in other data structures (typically in the timeout list, which is a delta sorted list).

2.2.3 Process Creation and System Startup

It is a good practice when initializing a multiprocessing application to have a separate initialization process. Consequently, the 'main'-function only creates the initialization process, starts it, and finally starts the operating system. In addition, some specific hardware initialization can also be performed first in the 'main'-function. This is typically done when initialization is time critical.

The initialization process, in turn, creates and starts the application processes and creates shared operating system objects like semaphores, queues, and timers. When the initialization process has performed all initialization and started all application processes, it can safely delete itself. In order to get a controlled startup sequence, the initialization process has the highest priority (i.e., priority zero). This guarantees that no other process will start executing until the initialization process is done and has deleted itself.

```
int main(...)
{
  //if needed, perform time critical hardware initialization
  ...

  //create and start initialization process (prcInit)
  ...

  //start operating system
  //(before this call, only create and start process calls are allowed)
  osStart();

  return 0;
}

//highest priority process
void prcInit(...)
{
  //create and start all application processes
  ...

  //delete itself
  ...
}
```

This startup structure is also a good practice since no other operating system calls than create process and start process is allowed until the operating system has been started.

Observe that the 'main'-function's stack size will be set by the startup code (the crt0-file, or similar). In order not to waste valuable RAM space, the stack should be set as small as possible. Since only time critical hardware initialization and startup of the initialization process is performed in the main-function. The stack size can be very small (unless the hardware initialization is very complex and nested). It is for example not recommended to make a printf()-call in the main-function since this will typically need a lot of stack space. Also observe that the operating system call to create and start the initialization process as well as the 'start operating system' call requires some stack space.

Besides a structured system initialization, it is also advised to have a structured process initialization. A typical process prototype is listed below. Before entering the forever loop all initialization related to the specific process is performed.

```
void prcA(...)
{
  //initialization related to the process
  ...

  //enter forever loop
  while(1)
  {
    //body of process
    ...
  }
}
```

2.3 Synchronization Primitives

This section described the different synchronization primitives.

2.3.1 Event Structure

An event structure is an abstraction that the operating system uses for implementing several different synchronization primitives. There are three functions that operates on an event structure: initializing the structure, waiting on an event, and signaling an event. If there is a process waiting on a specific event, and that event is signaled, the process will exit the blocked state and be flagged as ready to run. I.e., the PCB will be placed in the ready list. If there are more than one process waiting on the specific event the process with the highest priority will be moved to the ready list. The other processes will remain blocked.

2.3.2 Counting Semaphore

The counting semaphore keeps an internal counter, as the name suggests, that is incremented each time the 'give'-function is called and is decremented for each 'take'-call. The operating system will not check for overruns resulting from counter wrap-around, it will just silently wrap around. The counting semaphore makes use of the event mechanism described above.

2.3.3 Queue

A queue is a mechanism to send messages between processes much like the signal synchronization primitive, but with one distinction. When a message is posted to a queue any process can read the message (i.e., receive it), as opposed to a signal that is addressed to a specific process.

The queue implementation makes use of the event mechanism described above.

2.4 Additional Functionality

This section describes some additional functionality that has been included in the operating system. The functionality is not considered as core functionality in an operating system, and can easily be implemented with the basic primitives. However, the functionality presented below has been pre-coded for increased convenience.

2.4.1 Timer Process

A timer process adds support for timers in the operating system. A timer allows an application to execute a specific piece of code (which is the timer's callback function) after a certain specified time delay.

A timer is said to be armed when it is activated and it is said to fire when the delay expires. A timer can either be single-shot or periodic. A periodic timer is re-armed after the callback function returns. The user supplies the callback function and the time delay value when the timer is created.

The timer process runs at the highest priority (i.e., at priority value zero). The process is responsible for calling timer callback functions. This means that all callbacks execute at the highest system priority. The execution time of a callback function should hence be kept as short as possible in order not to block lower priority processes.

Internally, the timer counters are decremented by the system tick function. When a timer's counter reach zero the system tick function signal a counting semaphore. This semaphore represents the number of timers that have reached zero (and should be fired). The timer process, in turn, waits on this counting semaphore and is therefore only awaken when there is a timer to fire.

2.5 Statistics

2.5.1 Stack Usage

When a process is created the operating system will fill the stack area with a predefined bit-pattern. When a process requests the stack usage (calculated as a percentage value) the operating system will investigate how much of the stack that is unused (i.e., has the original bit-pattern intact).

2.6 Critical Sections

In addition to semaphores (that can be selected in the configuration process) the operating system also offers critical sections by disabling interrupts. Internally, the operating system is protecting critical sections by disabling interrupts. Two strategies can be used when disabling/enabling interrupts. The first is just to turn off interrupts when entering a critical section and then turn them on when leaving the critical section. Another way is to save the current state somewhere, for example push it on the stack or storing it in a local variable. The current state of the interrupt enable/disable flag is typically found in the processor status register. When leaving the critical section, the state is restored to its previous/original state. This way, interrupts will remain disabled if they were disabled before the critical section was entered. The second solution is hence more general and lower the risk of unintentionally enable interrupts.

As mentioned before the operating system disables interrupts when entering a critical section and therefore it is important to be aware of which method that is in use since it will affect the interrupt status after a system call. If the first method is used the user cannot assume that interrupts will remain disabled after a system call regardless of context; application processes or interrupt service routines. If the second method is used the user cannot assume that interrupts will remain disabled after a system call in application process context (if a context switch occurs and anoher process starts executing with interrupts enabled), but on the other hand if the system call is made from an interrupt service routine the user may assume that the interrupt status is preserved.

The preferred way of creating critical sections is to use binary semaphores, since they have only slightly larger overhead than simply modifying the interrupt enable/disable flag in the processor status register. However, if data is shared between a process and an interrupt service routine that is not set-up to allow for system calls the preferred way is just to turn off that particular interrupt instead of turning off all interrupts.

2.6.1 Enable / Disable Interrupts

There exists three routines for controlling interrupt enable/disable. These are:

- tSR halDisableInterrupts_oshal(void); That returns the current status register, before disabling interrupts (both IRQ and FIQ).
- void halEnableInterrupts_oshal(void); That unconditionally enable interrupts (both IRQ and FIQ).
- void halRestoreInterrupts_oshal(tSR restoreValue); That restore the state to the parameter value.

There are two different ways of using these routines. Either enabling or disabling of interrupts is done explicitly, regardless of previous state. Then the functions: halDisableInterrupts_oshal and halEnableInterrupts_oshal are used. Alternatively, the current state is stored when disabling interrupts. This state is then used when enabling again. Then the function: halDisableInterrupts_oshal and halRestoreInterrupts_oshal are used. The macros m_os_dis_int() and m_os_ena_int() are connected to these functions:

```
#define m_os_dis_int() {localSR = halDisableInterrupts_oshal();}
#define m_os_ena_int() {halRestoreInterrupts_oshal(localSR);}
```

There exist a define that can be used:

```
#define tSR tU32
```

This define declares a variable type that equals the status register of the processor. All functions/processes that must enable/disable interrupts must declare a variable of this type, and the name must be localSR, see example below.

```
void anyFunction(void)
{
  tSR localSR;
  ...                  //other local variables

  ...                  //some code that execute with enabled IRQ
  m_os_dis_int();
  ...                  //some code that must execute with disabled IRQ
  m_os_ena_int();
  ...                  //some code that execute with enabled IRQ

}
```

It is recommended not to enable interrupts in an ISR (Interrupt Service Routine). Nested interrupts require more stack space and can be difficult to analyze.

2.7 Interrupt Service Routines

Some system calls are allowed from interrupt service routines and others are not. The rule is that a system call that can result in a wait state is not allowed, since an interrupt service routine cannot be put idle (i.e., block waiting for a specific event).

Before a system call is allowed to be used in an interrupt service routine the operating system must be aware of that the system is running in foreground context (i.e., in interrupt context). The operating system has an internal variable that specifies the interrupt service routine nesting depth (isrNesting). If isrNesting is zero the system is executing in background context (i.e., an application process) and if it is greater than zero the system is executing in foreground context. If system calls need to be executed from an interrupt service routine the isrNesting variable must be increased by one when entering the ISR and decreased by one when leaving the ISR. In addition the stack frame must have a correct layout. The actual stack layout is completely hardware abstraction layer dependent. By mimicking the tick interrupt service routine (implemented by the hardware abstraction layer) the stack layout will be correct and no time will be wasted correcting the stack layout (when system calls are used).

To have total control of the stack layout the interrupt handler(s) must generally be written in assembly language. The interrupt handler creates the correct stack layout, increment isrNesting (HAL ISR Enter function), calls the actual interrupt service routine (ISR), calls the HAL ISR Exit function, and finally restores the registers and the stack. Observe that the actual ISR may very well be written in ANSI-C.

The HAL ISR Exit function will check if a process with higher priority than the currently running process has been made ready to run. If that is the case a context switch is initiated. This is the reason why the stack must have a certain layout before the ISR exit function is called.

The HAL ISR Enter function is called: osISREnter and the HAL ISR Exit function is called: osISRExit.

Interrupt handlers (ISRs) that make use of functions in a pre-emptive operating system are sometimes said to be 'operating system aware'.

2.7.1 LPC2xxx Processor Family Interrupts

Philips LPC2xxx series of microcontrollers includes a Vectored Interrupt Controller (VIC), which is a very flexible interrupt controller. It takes 32 interrupt request inputs that can be assigned into 3 categories, FIQ, vectored IRQ, and non-vectored IRQ. The programmable assignment scheme means that priorities can be dynamically assigned and adjusted. The VIC can also supply the address of the respective ISR handler directly to the processor. This will minimize the time spent figuring out which interrupt source that actually requested the interrupt.

The pre-designed HAL for ARM7 processors (like Philips LPC2xxx) is built around one common IRQ handler. All interrupts execute this common code, which is listed below. First the correct stack frame layout is created (observe that this is done in System execution mode), then isrNesting is incremented in order to inform the operating system that an interrupt is executing. The stack pointer of the interrupted process is saved if it is the first nesting of interrupts, i.e., if isrNesting equals one. By doing this, the code becomes independent of compiler code optimization (that sometimes changes the stack layout in order to optimize the code). At this point, the stack frame layout is correct and the operating system is informed that an interrupt is executing. This means that ISR may call functions in the pre-emptive operating system.

After this initial handling, the actual ISR is executed (function: handleIRQs). Observe that this is done in IRQ execution mode. Finally, the function osISRExit is called in order to check if a context switch should be performed. If not the common IRQ handler returns to the originally interrupted process.

```
HandlerIRQ:
  STMFD   SP!,{R1-R3}
  MOV     R1,SP
  ADD     SP,SP,#12
  SUB     R2,LR,#4
  MRS     R3,SPSR
  MSR     CPSR_c,#(NO_INT | MODE_SYS)

  @
  @ Save interrupted process contex
  @
  STMFD   SP!,{R2}            @ Push adjusted return PC
  STMFD   SP!,{R4-R12,LR}
  LDMFD   R1!,{R4-R6}         @ Move R1-R3 from IRQ to SYS stack
  STMFD   SP!,{R4-R6}
  STMFD   SP!,{R0}            @ Push R0
  STMFD   SP!,{R3}            @ Push CPSR (actually IRQ's SPSR)

  @
  @ isrNesting++  =>  block scheduling during interrupts
  @
  LDR     R0,addr_isrNesting  @ R0 = &isrNesting
  LDRB    R1,[R0]             @ R1 = isrNesting
  ADD     R1,R1,#1            @ R1 = R1 + 1
  STRB    R1,[R0]             @ Store new value of 'isrNesting'

  @
  @ Check if (isrNesting == 1)
  @
  CMP     R1,#1
  BNE     HandlerIRQ_cont

  @
  @ Store SP (only done if isrNesting == 1)
  @
  LDR     R4,addr_pRunProc    @ R4 = &pRunProc
  LDR     R5,[R4]             @ R5 = pRunProc
  STR     SP,[R5]             @ Store SP of the pre-empted process

HandlerIRQ_cont:
  @
  @ Switch back to IRQ mode
  @ Execute specific ISR for the interrupt
  @ Switch back to SYSTEM mode
  @
  MSR     CPSR_c,#(NO_INT | MODE_IRQ)

  BL      handleIRQs          @ Now jump to the specific ISR

  MSR     CPSR_c,#(NO_INT | MODE_SYS)

  @
  @ Inform the OS that the interrupt is (soon) over.
  @ Check if time to perform a context switch.
  @
  BL      osISRExit

  @
  @ Restore interrupted process context and return
  @
  LDMFD   SP!,{R4}
  MSR     CPSR_cxsf,R4
  LDMFD   SP!,{R0-R12,LR,PC}  @ Restore regs of interrupted context
```

The ISR handler is actually a C function, as listed below. It just reads the actually ISR handler addresses from VIC directly. It is a very simple and effective solution. The individual ISR handlers (their starting addresses) are registered in VIC registers. This must of course be done by the application code.

```
/* Declare a function pointer to a: void XXX(void); function */
typedef void (*PFNCT)(void);

void
handleIRQs(void)
{
  PFNCT pfnct;
  /* Read the interrupt vector from the VIC */
  pfnct = (PFNCT)VICVectAddr;

  /* Handle ALL interrupting devices */
  while (pfnct != (PFNCT)0)
  {
    /* Call ISR for interrupting device */
    (*pfnct)();
    /* Read the interrupt vector from the VIC */
    pfnct = (PFNCT)VICVectAddr;
  }
}
```

The code below illustrates how a typical ISR handler is written. First the interrupt condition is handled, the interrupt flag is reset (may also be done first), and then finally the VIC is informed that the ISR has reached its end. VIC must be informed of this since there can be other pending interrupts. Observe that all interrupt sources must be acknowledged before returning from an interrupt handler. Else, the IRQ/FIQ will be re-entered immediately on return.

```
/* Exemple of the general structure of an ISR */
void
MyISR_Handler(void)
{
  /* Service the interrupting device                            */
  /* Buffer the data (if any) and signal to process the data    */
  /* Clear the interrupting device (i.e. acknowledge the device) */
  /* Inform VIC                                                 */
}

/* Exemple of the specific ISR handler for OS timer ticks */
void
timerIsr(void)
{
  TIMER0_IR   = 0xff;  /* reset all IRQ flags in timer #0         */
  osTick();
  VICVectAddr = 0;     /* Inform VIC that ISR has reached its end */
}
```

The structure above also applies for FIQ interrupts. It is advised to only have one interrupt source for the FIQ interrupt.

Observe that ISR handlers can (with advantage) be written as C-functions and thay should not be declared as interrupts functions (just plain C-function). The ISR handlers must neither call the osISREnter nor osISRExit function.

3 API

3.1 API Overview

The interfaces of an ESIC are defined from a service perspective, i.e., the application code accesses an ESIC function through the **high level interface** (requests a service) and the ESIC returns the results (if any). If the ESIC in turn requires some other (low-level) service to carry out the request, this service is accessed through the **low level interface** of the ESIC. When carrying out the requested service, an ESIC supports calling optional user-defined functions called **hooks**. The figure below shows an overview of the interfaces of the Pre-emptive Operating System.

Note: The interfaces of an ESIC are governed by the user configuration.

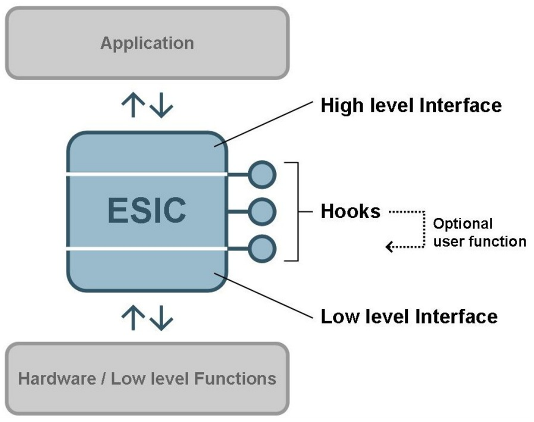

Figure 2: API Overview

The following interface functions are exposed by the Pre-emptive Operating System ESIC.

High-level Interface

3.2 Structures and Defines

3.2.1 Error Codes

- OS_OK (0x00) - Operation completed successfully
- OS_ERROR_NULL (0x01) - A NULL pointer was supplied as an argument that is not allowed to be NULL.
- OS_ERROR_ISR (0x02) - The operation is not allowed inside an interrupt service routine.
- OS_ERROR_PID (0x04) - An illegal pid was supplied to the function.
- OS_ERROR_ALLOCATE (0x05) - Out of process control blocks
- OS_ERROR_STATE (0x06) - Trying to resume a process that is not suspended.
- OS_ERROR_QUEUE_FULL (0x07) - The queue is full.
- OS_ERROR_TIMEOUT (0x08) - The operation returned due to a timeout.
- OS_ERROR_PRIO (0x09) - The priority level is out of range.

3.3 High-level

The configurable interface towards the application enables the developer to choose how to interface towards the infrastructure function (call-back or blocking interface, copying or non-copying interface, etc.). This greatly minimizes the integration work for the application programmer, since the application interface can be customized to match the system architecture of the target application.

3.3.1 osInitTimers

`void osInitTimers( tU8* pError )`

This function initializes the timer process. The timer process runs on the highest priority, i.e. 0, and executes the timer callback function when a timer expires. No other process should be run on this priority.

Parameters:

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_ALLOCATE - The timer process could not be created since there was no free process control blocks available. The number of process control blocks is specified during operating system configuration (maximum number of processes).`

3.3.2 osCreateTimer

```
void osCreateTimer( tTimer* pTimer, void (*callback)(void),
tBool repeat, tU32 time )
```

This function initializes a timer. A timer is initialized with a timer value, specified in timer ticks. When the specified time has elapsed the timer is said to fire. When a timer fire the callback function of the timer is executed. A timer can be set to be repeatable. A repeatable timer is reactivated once the callback function has returned. The timer structure must be allocated, statically or dynamically, by the user before this function is used. osCreateTimer does not allocate the structure, it initializes and queues the timer.

Parameters:

[in] pTimer - A pointer to an allocated timer structure.

[in] callback - The callback to be used when the timer fires.

[in] repeat - If TRUE the timer will be reactivated as soon as the callback function has returned.

[in] time - The initial timer value, specified in system ticks.

3.3.3 osDeleteTimer

`void osDeleteTimer( tTimer* pTimer, tU8* pError )`

This function deletes a timer. The timer structure is not de-allocated, only removed from the timer queue. If the timer has already fired and is not repeatable there is no need to call this function. It is only meaningful to call this function on a timer that is armed but not fired.

Parameters:

[in] pTimer - A pointer to the timer to delete.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_NULL - A NULL pointer was supplied to the function where it was not allowed.`

3.3.4 osSemInit

```
void osSemInit( tCntSem* pSem, tU16 initial )
```

This function initializes a counting semaphore and must be called before any other function is used on the semaphore.

Parameters:

[in] pSem - A pointer to an allocated counting semaphore structure.

[in] initial - The initial counter value.

3.3.5 osSemTake

```
tBool osSemTake( tCntSem* pSem, tU32 timeout, tU8* pError )
```

This function takes a counting semaphore, i.e. decreasing the semaphore counting. If the semaphore counter is zero the function will block until another process or an ISR gives the semaphore or a timeout occurs.

Parameters:

[in] pSem **-** A pointer to an initialized semaphore structure.

[in] timeout **-** After timeout ticks the operation will timeout. A timeout of zero means no timeout at all.

[out] pError **-** The return status of the function.

Returns:

TRUE if semaphore was taken and FALSE if timeout or error.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_ISR - The function was called from an interrupt service routine.`

`OS_ERROR_NULL - A NULL pointer was supplied to the function where it was not allowed.`

3.3.6 osSemGive

```
void osSemGive( tCntSem* pSem, tU8* pError )
```

This function gives a counting semaphore, i.e. increases the semaphore counter. If there are one or more processes waiting for the semaphore the process with highest priority is made ready to run.

Parameters:

[in] pSem - A pointer to an initialized semaphore structure.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_NULL - A NULL pointer was supplied to the function where it was not allowed.`

3.3.7 osSemTryTake

`tU8 osSemTryTake( tCntSem* pSem, tU8* pError )`

This function tries to take a counting semaphore. If the semaphore cannot be taken the function immediately returns instead of blocking. This function can be used from an ISR (interrupt service routine).

Parameters:

[in] pSem - A pointer to an initialized semaphore structure.

[out] pError - The return status of the function.

Returns:

0 if the semaphore was taken, else 1.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_NULL - A NULL pointer was supplied to the function where it was not allowed.`

3.3.8 osSleep

```
void osSleep( tU32 ticks )
```

This function puts a process to sleep for the specified number of ticks.

Parameters:

[in] ticks - The number of ticks to put the process to sleep.

3.3.9 osPid

```
tU8 osPid( tU8* pError )
```

This function returns the process identification descriptor for the running process.

Parameters:

[out] pError - The return status of the function.

Returns:

The process identification descriptor of the currently running process.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_ISR - The function was called from an interrupt service routine.`

3.3.10 osInit

```
void osInit( void )
```

This function must be called before any other call to the operating system.

3.3.11 osStart

```
void osStart( void )
```

This function starts the operating system. There must be at least one process created and started before this function is called. A process is created by calling osCreateProcess and started by calling osStartProcess. osStart, osCreateProcess and osStartProcess are the only operating system functions that may be called before the operating system is started. If other operating system functions are called before the operating system is started the behavior is undefined. The preferred way of starting up a multitasking system is to only create and start an init process in the main function before osStart is called. The init process initializes the system and starts other processes needed. The init process can safely initialize other operating system objects like semaphores, queues, etc.

3.3.12 osISREnter

`void osISREnter( void )`

This function is used to notify the operating system that the application has entered an interrupt service routine (ISR). This is important if the ISR is using services from the operating system, since some services need to know if they are executed from an ISR or not. The function osISRExit should be used before the ISR returns to notify the operating system about the ISR exit.

3.3.13 osISRExit

```
void osISRExit( void )
```

This function is used to notify the operating system that the currently serviced interrupt is about to exit. The function is always used in conjunction with the function osISREnter, which should always be called before osISRExit. It is important to notify the OS about ISRs (Interrupt Service Routines) if they are using services from the operating system (since some services need to know if they are executed from an ISR or not).

3.3.14 osDeleteProcess

```
void osDeleteProcess( void )
```

This function deletes the currently running process. The process control block used by the process will be freed and is therefore available for new processes.

3.3.15 osCreateProcess

```
void osCreateProcess( void (*pProc)(void* arg), tU8* pStk,
tU16 stkSize, tU8* pPid, tU8 prio, void* pParam, tU8* pError )
```

This function creates a new process. The process is not automatically started. To start the process the osStartProcess function must be called. A new process can only be created if there is a free process control block available. The number of process control blocks is specified during operating system configuration (maximum number of processes).

Parameters:

[in] pProc - The process entry function.

[in] pStk - A pointer to the stack area to use. The stack area must be allocated before the process is created.

[in] stkSize - The size of the stack area in bytes.

[out] pPid - The returned process identification descriptor (pid).

[in] prio - The priority of the process. The priority is a number between 0 and NUM_PRIO-1, where NUM_PRIO is specified during operating system configuration (maximum number of priorities). 0 is the highest priority level and NUM_PRIO-1 is the lowest priority level. The operating system will always run the process that has the highest priority and is ready to run, i.e. is not sleeping, suspended or waiting for a synchronization primitive. If several processes are run on the same priority level they are scheduled in a round-robin fashion.

[in] pParam - This parameter is passed to the process entry function when the process is started.

[out] pError - The return status of the function.

Parameters to (*pProc):

[in] arg - This argument can be used to pass arbitrary information to the process entry function when the process is started.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_PRIO - The supplied priority is not correct.`

```
OS_ERROR_ALLOCATE - The process could not be created
since there are no free process control blocks
available. The number of process control blocks is
specified during operating system configuration (maximum
number of processes).
```

3.3.16 osStartProcess

```
void osStartProcess( tU8 pid, tU8* pError )
```

This function is used to start a process. The process must previously have been created by a call to osCreateProcess.

Parameters:

[in] pid - The process identification descriptor (pid) of the process to start. The pid is returned by osCreateProcess.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

```
OS_OK - The function completed successfully.
OS_ERROR_PID - The supplied pid is not correct.
```

3.3.18 osSuspend

```
void osSuspend( void )
```

This function suspends the currently running process. Another process can resume it by a call to osResume.

3.3.19 osResume

`void osResume( tU8 pid, tU8* pError )`

This function resumes a suspended process. It is valid to do resume on a process that has not been suspended.

Parameters:

[in] pid - The process to resume.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_PID - The supplied pid is not correct.`

3.3.20 osCreateQueue

```
void osCreateQueue( tQueue* pQueue, void** pQueueArea, tU16
size )
```

This function initializes a queue structure.

Parameters:

[in] pQueue **-** A pointer to an allocated queue structure.

[in] pQueueArea **-** A pointer to the queue area. The user must allocate the memory area used by the queue. The queue area is an array of void pointers.

[in] size **-** The size of the queue area. The size is given in number of void pointers in the area.

3.3.21 osPendQueue

`void* osPendQueue( tQueue* pQueue, tU16 timeout, tU8* pError )`

This function retrieves the first message from the queue. The message is removed from the queue. If the queue is empty the function will block until there is a message to retrieve or a timeout occurs.

Parameters:

[in] pQueue **-** A pointer to an initialized queue structure.

[in] timeout **-** The number of ticks to wait on a queue before returning. If a timeout of zero is specified the function will never timeout.

[out] pError **-** The return status of the function.

Returns:

The first message in the queue or NULL if timeout or error.

Possible error situations (what can be identified in an error code):

`OS_OK - The function completed successfully.`

`OS_ERROR_ISR - The function was called from an interrupt service routine.`

`OS_ERROR_NULL - A NULL pointer was supplied to the function where it was not allowed.`

3.3.22 osAcceptQueue

`void* osAcceptQueue( tQueue* pQueue, tU8* pError )`

This function tries to receive the first message from the queue. If the queue is empty the function returns immediately. This function can be called from within an interrupt service routine (ISR).

Parameters:

[in] pQueue - A pointer to an initialized queue structure.

[out] pError - The return status of the function.

Returns:

The retrieved message or NULL if the queue is empty.

Possible error situations (what can be identified in an error code):

```
OS_OK - The function completed successfully.
```

```
OS_ERROR_NULL - A NULL pointer was supplied to the
function where it was not allowed.
```

3.3.23 osFlushQueue

```
void osFlushQueue( tQueue* pQueue, tU8* pError )
```

This function clears a queue from all messages.

Parameters:

[in] pQueue - A pointer to an initialized queue.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

```
OS_OK - The function completed successfully.
OS_ERROR_NULL - A NULL pointer was supplied to the
function where it was not allowed.
```

3.3.24 osPostQueue

```
void osPostQueue( tQueue* pQueue, void* msg, tU8* pError )
```

This function posts a new message to the end of the queue.

Parameters:

[in] pQueue - A pointer to an initialized queue structure.

[in] msg - The message to post.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

```
OS_ERROR_QUEUE_FULL - The queue is full.
OS_OK - The function completed successfully.
OS_ERROR_NULL - A NULL pointer was supplied to the
function where it was not allowed.
```

3.3.25 osPostFrontQueue

```
void osPostFrontQueue( tQueue* pQueue, void* msg, tU8* pError
)
```

This function posts a new message to the front of the queue.

Parameters:

[in] pQueue -

[in] msg - The message to post.

[out] pError - The return status of the function.

Possible error situations (what can be identified in an error code):

```
OS_ERROR_QUEUE_FULL - The queue is full.
OS_OK - The function completed successfully.
OS_ERROR_NULL - A NULL pointer was supplied to the
function where it was not allowed.
```

3.3.26 osStackUsage

`tU8 osStackUsage( tU8 pid )`

This function returns the stack usage. The stack usage is based on the maximum size used so far, i.e. from the application start to the point where this function is called.

Parameters:

[in] pid - The pid of the process to check.

Returns:

The used fraction of the stack area specified in percent.

3.3.27 m_os_ena_int

```
void m_os_ena_int( void )
```

This macro enables interrupts.

3.3.28 m_os_dis_int

```
void m_os_dis_int( void )
```

This macro disables interrupts.

3.4 Low-level

As the generated ESIC code is completely independent of the target hardware or operating system, no hardware specific or operating system commands are utilized. Instead, a highly configurable low level interface is provided. The low level interface of an ESIC enables the selection/configuration of interface aspects such as principal access mechanism, where and how interface data is stored, if and how data is passed between the layers, etc. Only the final access (service request) to the hardware or operating system function must be provided by the user, since this is very application and hardware specific.

3.5 Hooks

To enable user-specific processing hooks are provided which allow a user function to "hook" into the normal execution flow. Hooks are intended primarily for the inclusion of optional functionality, i.e., functions that are not critical for the operation of the ESIC such as gathering statistics or logging of internal events. Since hooks interrupt the normal execution flow of an ESIC, they should be kept as short as possible to ensure that the ESIC can perform its intended function within its given parameters.

3.5.1 m_os_user_tick

```
#define m_os_user_tick()
```

This macro is called every system tick by the operating system and can be used by the application to perform additional work at every tick.

Observe that the amount of code executing in this hook should be kept to a minimum, since the timer tick will occur often.

4 To-do

This section describe the necessary integration work that has to be done before the generated code is fully integrated into a specific system.

4.1 Timer Process

4.1.1 TIMERSTACK_SIZE

This macro defines the size of the timer process' stack. The required stack size is dependent on how much the callback routines require, the number of interrupts (than may need stack space), the target processor, and the compiler.