eMOS User Manual

INTRODUCTION

eMOS is a Real-Time Operating System (RTOS) from ImageCraft with the following features:

- No limits on most of its functions and features except by the size of the available SRAM of the target device.
- Multitasking kernel with round robin priority-based preemptive scheduling. Up to 4 priority levels defined in binary release. Source release has no preset limits.
- System calls use a single kernel stack, minimizing tasks' stack overhead.
- Message passing primitives provide synchronizing and interprocess communication capabilities, minimizing the problem with asynchronous events and data copying. Automatic priority inheritance eliminates priority inversion issues.
- Mutex, also with automatic priority inheritance, allows access to critical resources.
- System safety features including: memory resource tracking, virtual watchdog system, and stack checking to help you discover bugs and increase the robustness of your system.
- Additional plug-in modules including System Status task, TCP/IP, USB, File System, etc. will be created using this underlying OS technology.

Design Philosophy

eMOS is a modern RTOS designed for embedded projects. Features such as a preemptive kernel, message passing for synchronization and interprocess communication, and mutex with priority inheritance make for a clean design; use of a single kernel stack, careful design of the internal data structures and use of compiler pragma minimizes resource consumption; and system safety features such as stack checking, virtual watchdog and memory tracking set eMOS apart from other RTOSes.

eMOS can be used on devices ranging from low-end 8-bit AVR microcontrollers with 8K of flash[1] and 2K of RAM to 32-bit ARM devices with lots of memory. The kernel API takes just a few kilobytes and can be used by itself in a minimal system to provide basic multitasking services.

1. With minimal functionality.

In designing the eMOS tasking model, we feel that a preemptive priority-based round robin scheduler is the simplest model for users, because it allows the natural “functions as tasks” style of writing programs. It places no restriction on how users may structure their code, and users do not need to work around the tasking model. The scheduler gets called whenever a high-priority task becomes ready to run, providing very fast response to real-time events.

To minimize resource consumption, a separate kernel stack is used by the kernel calls, so that each task does not need to provide the stack resources needed by the kernel.

For process synchronization and interprocess communication, we have adopted the message passing semantics made popular by OSes such as QNX. A set of 3 primitives provides a robust and fast solution to both synchronization and interprocess communication requirements, plus it handles the problems with priority inversion. For simpler needs, eMOS also provides mutex with priority inheritance.

As many embedded systems are used in low-power situations, eMOS provides hooks to the idle task so that you may place the system in low-power mode to conserve power, to be awakened by interrupts or timer events.

The general philosophy is to provide flexibility rather than to optimize the design for a particular niche (e.g., memory constraint devices). Thus, eMOS uses dynamic memory allocation (via a best-fit always-merge allocator to lessen memory fragmentation). It places no arbitrary limits on the number of tasks or task priorities except as constrained by the memory size or datatype size, and allows dynamic task creation and deletion. This approach allows eMOS to scale up and down except in the most memory-constrained situations.

Memory overwrite and, in particular, stack overflow are common sources of problems in embedded software. As most CPUs do not have any support for stack checking, this problem can manifest in mysterious system crashes that are difficult to track down. eMOS provides stack checking at every task switch (which may be turned off for final production builds) to ensure stack overflows are caught.

A slow or resource-constrained CPU can be replaced with a faster CPU or one with more resources, but there is no substitute for 100% reliable embedded software. Failing that, a system should at least allow postmortem analysis and failsafe recovery. As more embedded products are deployed in the world and in mission-critical environments, it is of the upmost importance for the RTOS to provide as many system safety features as possible without compromising the performance of the system. In the worst case scenario, a hardware anomaly such as alpha particles affecting SRAM cells is a real possibility under some conditions. eMOS provides a virtual watchdog to

check the health of the processes. You can combine the virtual watchdog with a hardware watchdog and increase the chance of preserving the integrity of the system even if the system fails.

Product Versions

You license the use of eMOS binary in your products. eMOS is licensed in multiple versions with no royalties for both commercial and non-commercial uses. Features such as changing the number of priority levels, or removing the stack checking for production release are available only in source licenses.

For non-commercial use and evaluation, you may not develop or deliver products with these versions:

- Free binary release with support for up to 5 tasks, available on our web site `http://www.imagecraft.com` and as part of our compiler product demos.
- Low-cost binary-only release for evaluation only.

For commercial uses:

- STD - Binary release with unlimited-end-user product distribution for a single developer (one person) and a single embedded product.
- ADV - Source release with unlimited-end-user product distribution for a single developer (one person) and a single embedded product.
- PRO - Source release with unlimited-end-user product distribution for a single developer or company for use in multiple products.

Purchase includes six months of upgrades and support. Additional upgrades and support may be purchased on a per annum basis. You may continue to use your licensed copy of the product and distribute the end user products after the support contract expires if you choose not to renew. Please see our web site for current pricing information.

eMOS is written using our ImageCraft line of embedded C compilers. It takes advantage of the compiler pragmas and calling conventions for efficient code handling. However, it should be easily portable to other compilers and other architectures. Please visit our web site for information on porting process and fees.

Change Logs

Changes in V1.01

- Added eMOS_MsgDiscardAsyncMsg().
- Added timeout_ms argument to eMOS_MsgReceiveMsg().

Software License Agreement

This is a legal agreement between you, the end user, and ImageCraft. If you do not agree to the terms of this Agreement, please promptly return the package for a full refund.

GRANT OF LICENSE. This ImageCraft Software License Agreement permits you to use the ImageCraft eMOS for AVR (“SOFTWARE”) in your product according to the license types:

- STD - this license allows unlimited-end-user product distribution for a single developer (one person) and a single embedded product. You may not reverse-engineer the source code of the SOFTWARE.
- ADV - this license allows unlimited-end-user product distribution for a single developer (one person) and a single embedded product. A copy of the source code is provided for your archiving and modification purposes for product development only. You may not transfer the source code to another party (person or company) or otherwise allow another party to have access to the source code without explicit written permission from ImageCraft.
- PRO - this license allows unlimited-end-user product distribution for a single developer or company for use in multiple products. A copy of the source code is provided for your archiving and modification purposes for product development only. You may not transfer the source code to another party (person or company) or otherwise allow another party to have access to the source code without explicit written permission from ImageCraft.
- Non-Commercial Use - this license allows you to evaluate the SOFTWARE, not for product development or delivery. You may not reverse-engineer the source code of the SOFTWARE.

COPYRIGHT. The SOFTWARE is owned by ImageCraft and is protected by United States copyright laws and international treaty provisions. You must treat the SOFTWARE like any other copyrighted material (e.g., a book). You may not copy written materials accompanying the SOFTWARE.

OTHER RESTRICTIONS. The SOFTWARE is not for re-sale. You may not rent, lease, or sell the SOFTWARE license. Except for the PRO license, you are licensing the SOFTWARE for use on a single product. Multiple developers working on the same product require a separate license for each developer unless you purchase a PRO license, or unless the developers do not use the SOFTWARE at the same time. A single developer working on multiple products using the SOFTWARE requires a separate license for each product, unless you purchase a PRO license.

LIMITED WARRANTY

LIMITED WARRANTY. ImageCraft warrants that the SOFTWARE will perform substantially in accordance with the accompanying written materials and will be free from defects in materials and workmanship under normal use and service for a period of thirty (30) days from the date of receipt. Any implied warranties on the SOFTWARE are limited to 30 days. Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

CUSTOMER REMEDIES. ImageCraft's entire liability and your exclusive remedy shall be, at ImageCraft's option, (a) return of the price paid or (b) repair or replacement of the SOFTWARE that does not meet ImageCraft's Limited Warranty and that is returned to ImageCraft. This Limited Warranty is void if failure of the SOFTWARE has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warranty period or 30 days, whichever is longer.

NO OTHER WARRANTIES. ImageCraft disclaims all other warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the SOFTWARE, the accompanying written materials, and any accompanying hardware.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. In no event shall ImageCraft or its supplier be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the SOFTWARE, even if ImageCraft has been advised of the possibility of such damages. The SOFTWARE is not designed, intended, or authorized for use in applications in which the failure of the SOFTWARE could create a situation where personal injury or death may occur. Should you use the SOFTWARE for any such unintended or unauthorized application, you shall indemnify and hold ImageCraft and its suppliers harmless against all claims, even if such claim alleges that ImageCraft was negligent regarding the design or implementation of the SOFTWARE.

GETTING STARTED

Header Files

The primary header file is `emos.h`. All your source files that use eMOS should include this file.

Other header files that you do not need to explicitly include in your source files (because they are included by `emos.h`) are: `_emos.h` for internal use (e.g. private header file for the eMOS source), `target.h` with device-specific defines, and `emos_func.h` with function declarations. These files should be copied to your `c:\iccv7avr\include` directory if they are not there already.

"Strings in Flash" Option

You must enable this option in your `Project->Options->Target` dialog box as eMOS is compiled with this option set to save SRAM space.

Using the Binary (Library) Version

The library file `libemos_avr.a` contains the eMOS functions. Copy the file to your `c:\iccv7avr\lib` directory (if it is not there already) and add `emos_avr` to the `Project->Options->Target->Additional Libs` edit box of your project.

`libemos_avr.a` is for all Mega AVR with flash memory 16K bytes or larger and smaller than the ATMega256. For ATMega256 users, however, you should use the alternate library file `libemos_atm256.a`, as the M256 uses 3 bytes to store the function call return address and the library differs slightly.

To create your project, at the minimum, you will need to add (to your project file list) a file with your `main` function, and the supplied file `avr_usermod.c` (or `atm256_usermod.c` for M256 users) with modifications, if any. If you use printf or puts, per usual ICC usage, you will need to include a copy of putchar also (you can find samples of putchar in `c:\iccv7avr\examples.avr`).

The linker links in library code modules only if they are actually referenced (an entire object module is linked in if any part of it is referenced). So, if you do not use some of the eMOS features (outside of the core multitasking kernel), you will not be incurring the overhead.

The demo version of eMOS allows up to 5 tasks to be created. The licensed versions have no preset limit.

Using the Source Version

Unzip the archive file `emos_avr_src.zip` to a directory of your choice and add all the C and asm source files (ones with `.c` and `.s` extensions) to the project file list of your project.

Source File Structure

main() Function

When using eMOS, your `main()` function typically looks something like the following:

```
#include “emos.h”
...
void task1(void);
...
main()
   {
   extern int _bss_end;
   eMOS_SysInit(&_bss_end, 1024*8);
   eMOS_TaskCreate(EFN(task1), ...
   eMOS_TaskCreate(EFN(task2), ...
   ...
   eMOS_SysStart();
   // never return
   }
```

You must call `eMOS_SysInit()` before calling any other eMOS functions with the arguments being the beginning address and size of the free SRAM space to be used by eMOS. In this code fragment, the address of `_bss_end` is the starting address of the free SRAM and 8K bytes is allocated for eMOS. eMOS uses SRAM to allocate process structures, stacks, mutex etc.

You then use `eMOS_TaskCreate()` to create your initial tasks. Tasks or processes can be created and destroyed dynamically, even within other processes. The macro `EFN()` takes a function name (e.g., `task1`) and expands it to

```
“task1”, task1
```

corresponding the first two arguments to the `eMOS_TaskCreate` function which are the ASCII string name of the task and the address of the task function (i.e., the name of the function).

Finally, you call `eMOS_SysStart()` to start the eMOS multitasking kernel. From that point on, eMOS kernel will select tasks to run. `eMOS_SysStart` will not return.

Task Function

A task function is a normal C function and usually executes in an infinite loop. If it ever returns, the task is terminated:

```
#pragma ctask:task1
void task1(void)
```

```
{
while (1)   // forever loop
      {
      eMOS_MsgReceive(...
      ...
      eMOS_MsgReply(...
      ...
      }
}
```

You should use the `ctask` pragma (as shown above) to direct the compiler not to generate any unneeded register saving and restoring code. Typically a task can be categorized as a server task, a client task, or a processing task (other categorizations are of course possible). The client-server model would most likely use the message passing API to send and receive requests. For example, you may use a server task to handle accesses to a LCD controller so that multiple processes can use the LCD without worrying about arbitration. In such an example, the server task would use `eMOS_MsgReceive()` to wait for a request and the client tasks use `eMOS_MsgSend()` to send requests. The interpretation of the message content is entirely determined by the message senders and receivers.

Mutex is also provided to provide exclusive access to shared resources.

Where To Go Next

The remaining portion of this document is the technical manual for eMOS. Most of the document is divided by sections of different API subsystems. First the subsystem is described in some details, followed by the list of the API functions.

There are a few sample projects in `examples.avr\emos\`; they should give you some ideas on how to write your own programs. There are a small number of target specific changes that you may need to make; see USER-SUPPLIED CODE and OPTIMIZING YOUR SYSTEM. The eMOS resource requirements are summarized in eMOS RESOURCE USAGE.

STACK CHECKING

A common cause of error in embedded programming is stack overflow. Stacks are used for local variable allocation and storing function call return addresses. Since the amount of SRAM is limited, the stack may overflow into space already allocated for other uses. In a normal C program for the AVR, the hardware stack (which is used for function calls) may run into the software stack and the software stack may run into the C global data.

With an RTOS, the situation generally worsens because each task has its own stacks. However, we can also turn a disadvantage into a big advantage for users: we can perform stack checking at task switch time and ensure that the stacks overflow have not occurred. Since eMOS is preemptive, it can catch most if not all of the stack pointer problems right when they occur. This can be optionally switched off for production builds (if you have a source license).

When a task is created, eMOS places two sentinals at the hardware and software stack bottoms. Both sentinals have the same value as the constant TASK_IS_HEALTHY used for the virtual watchdog, or the value 0x6B. During task switching, before running the selected task, eMOS checks if the task's stack pointers are within bounds, and it also checks if the sentinal values are still there. If either condition is not met, then an error has occurred.

When eMOS detects an improper stack pointer value, it calls the function `eMOS_UserSOS` to process the info and the `eMOS_UserSysReset` to reset the system. See USER-SUPPLIED CODE.

ERROR MODULE

Catastrophic Failures

In the case of a catastrophic failure as detected by the stack checking or the virtual watchdog, a user-supplied function `eMOS_UserSOS` is called (see USER-SUPPLIED CODE) and the following error code (defined in `emos.h`) is supplied.

- BAD_RUNNABLE *process state not marked as Runnable*
- BAD_BLOCKED *process state not marked as Wait Blocked*
- BAD_SLEEP *process state not marked as Sleep*
- BAD_HIBERNATE *process state not marked as Hibernate*
- BAD_HEALTH *process did not set Healthy status*
- BAD_STACK *process hardware stack out-of-bounds*
- BAD_SWSTACK *process software stack out-of-bounds*
- BAD_COOKIE *bad call to eMOS_VWatchdogFeedCookie*
- BAD_DELAY_TIMER_EXPIRES *process virtual watchdog delay timer expired*

System Call Errors

In the early stages of program development, a common source of errors might come from incorrect arguments to eMOS functions. eMOS calls the user supplied function `eMOS_UserSyscallError(char __flash *func, int retval)` to process the error. `func` is the ASCII name of the eMOS function, and `retval` is a negative error number. Typically, in initial development stage, `eMOS_UserSyscallError` should print out the function name and return `retval`. The error codes are described in the function descriptions below.

If you have a source license, you can define the macro NO_USERSYSCALLERROR in your `project->options->compiler->Macro Defines`, and a system call simply returns the error code when there is an error, thus saving the space used by the ASCII names for the `eMOS_UserSyscallError` calls.

Error Function

You can call `eMOS_ErrorString` to convert an error code to an ASCII string:

- `char __flash *eMOS_ErrorString(int code)`

  returns an ASCII string based on the error code. It returns “unknown failure code” if `code` is invalid.

API SUMMARY

eMOS API are divided into different modules. The naming convention is that the user-callable API has the form `eMOS_<Module><Action>`, e.g. `eMOS_TaskCreate()` to create a task. eMOS has the following modules:

- The System Module (API names begin with eMOS_Sys...) contains the system initialization and startup functions.
- The Multitasking kernel (eMOS_Task...) implements the tasking features of eMOS.
- The Scheduling API (eMOS_Sched...) allows you to change the scheduler's behavior.
- The Message Passing primitives (eMOS_Msg...) provide the interprocess communication and process synchronization functions.
- The Mutex (eMOS_Mutex...) is a simple process synchronization mechanism with priority inheritance support.
- The Com unit (eMOS_Com...) provides a uniform buffered input/output IO interface to serial devices.
- The Memory Management unit (eMOS_Mem...) provides dynamic memory allocation using a best-fit always-merge algorithm for combination of fast performance and minimized fragmentation.

  In addition, dynamic memory can be tracked per process basis, eliminating user bookkeeping errors.
- The Virtual Watchdog unit (eMOS_VWatchdog...) provides a virtual watchdog to check the health status of the system. It can also optionally work with the hardware watchdog.
- Most user-supplied functions have the form eMOS_User... or user functions are assigned to eMOS global function pointers (e.g. void (*eMOS_SysTickHook)()).

List of all the functions in eMOS:

System

```
int eMOS_SysInit(void *begin, unsigned size);
int eMOS_SysStart(void);
void eMOS_SysIdleHook(void (*func)(void));
void eMOS_SysTickISRHook(void (*func)(void));
long eMOS_SysGetTicks(void);
```

Kernel

```
int eMOS_TaskCreate(char FLASH *name, void (*func)(void),
unsigned char prio, unsigned stacksize, unsigned
hw_stacksize);
int eMOS_TaskGetID(void);
char FLASH * eMOS_TaskGetName(void);
int eMOS_TaskGetPIDByName(char FLASH *);
char FLASH * eMOS_TaskGetNameByPID(int);
void eMOS_TaskYield(void);
void eMOS_TaskHibernate(void);
int eMOS_TaskWakeup(int pid);
int eMOS_TaskKill(int pid);
void eMOS_TaskSleep(int secs);
void eMOS_TaskSleepMs(int msecs);
PROC_DUMP *eMOS_TaskProcDump(void *p);
PROC_DUMP *eMOS_TaskProcDumpByID(int pid);
void eMOS_SchedOff(void);
void eMOS_SchedOn(void);
```

Message Passing

```
int eMOS_MsgSend(int pid,void *sendbuf, int sendlen,void
*reqbuf, int reqlen);
int eMOS_MsgAsyncSend(int pid, unsigned msg);
int eMOS_MsgReceive(int *ppid, void *recbuf, int reclen, int
timeout_ms);
int eMOS_MsgReply(int pid, void *reqbuf, int reqlen);
int eMOS_MsgDiscardAsyncMsg(void);
```

Mutex

```
int eMOS_MutexCreate(void);
void eMOS_MutexDestroy(int mid);
int eMOS_MutexLock(int mid);
int eMOS_MutexUnlock(int mid);
```

Memory Management

```
void eMOS_MemInit(void *begin, unsigned size);
void *eMOS_MemAlloc(unsigned size);
void eMOS_MemFree(void *ptr);
void eMOS_MemEnableTracking(void);
void eMOS_MemDisableTracking(void);
void eMOS_MemFreeAll(void);
```

```
int eMOS_MemSpaceAvail(void);
int eMOS_MemSpaceUsed(void);
```

Com Unit

```
void eMOS_ComInit(void);
void eMOS_ComTerm(void);
int eMOS_ComOpen(int dev, COM_DESC_TYPE *cd);
int eMOS_ComClose(int dev );
int eMOS_ComRead(int dev, unsigned char *buf, unsigned
size );
int eMOS_ComWrite(int dev, const unsigned char *buf,
unsigned size );
void eMOS_ComISRPut(int dev);
void eMOS_ComISRGet(int dev);
```

Virtual Watchdog

```
void eMOS_VWatchdogStart(void);
void eMOS_VWatchdogFeedCookie(int status);
void eMOS_VWatchdogDelayTimer(unsigned msec);
```

Error Function

```
char __flash *eMOS_ErrorString(int code);
```

User Supplied Functions

```
void eMOS_UserSysInit(void);
void eMOS_UserSysStart(void);
void eMOS_UserSOS(int code);
void eMOS_UserSyscallError(char __flash *func, int code);
```

CORE SYSTEM

The system API provides functions to initialize and start the system. Almost all eMOS system functions (API) execute with interrupts disabled, so that eMOS functions can use a single kernel stack to minimize SRAM resource consumption.

Using External Memory

We have optimized eMOS' memory footprint in designing the process data structure and in using the single kernel stack for system calls etc. A rough estimate is that each task needs about 100 bytes in SRAM for its data structure and stacks. This allows you to run ten to twenty tasks on a small system with 2K of SRAM. The gain is that by using eMOS, you may simplify your design and coding, resulting in a more robust program faster.

You can also use external memory with eMOS. In the simplest case, you will need to set up the memory interface registers (e.g. the XMCRA and XMCRB registers) [1]in your main function or in a modified C startup file, and then call `eMOS_SysInit` (see below) with the starting address and the size of your memory. If you have discontiguous memory chunks, you can use the function `eMOS_MemInit` to tell eMOS about additional memory chunks.

Kernel Stack

Since a task may be preempted by another task at any moment through the system tick interrupt, a task must have enough stack space to hold the CPU context (e.g., the registers) as part of the interrupt processing and task switching.

When you make a system call, it normally would use stack space on the task stack too. To minimize a task's stack usage, we have designed eMOS such that all system calls use a single kernel stack.

If you are writing your own interrupt handlers, you can also use the kernel stack API to switch your handlers to use the kernel stack, lessening the stack requirements of the host task.

1. Be aware that some external memory may require a waiting time before the memory can be accessed.

Estimating Stack Size

As eMOS provides stack checking, you can always test the program with different stack sizes for the tasks to see how they perform. To estimate the stack size, estimate the largest amount of local variable space in the task's execution paths. If you look at the program's .lst file (View->"Listing File" under the IDE) and search for the task function's name, you should see the instruction to reserve space on the stack for that function, e.g.

```
_mut1:
  lockid               --> R14
  t                    --> R12
  id                   --> R10
   393 940E 0BD7 CALL push_xgset00FC
   395 9724 SBIW R28,4
```

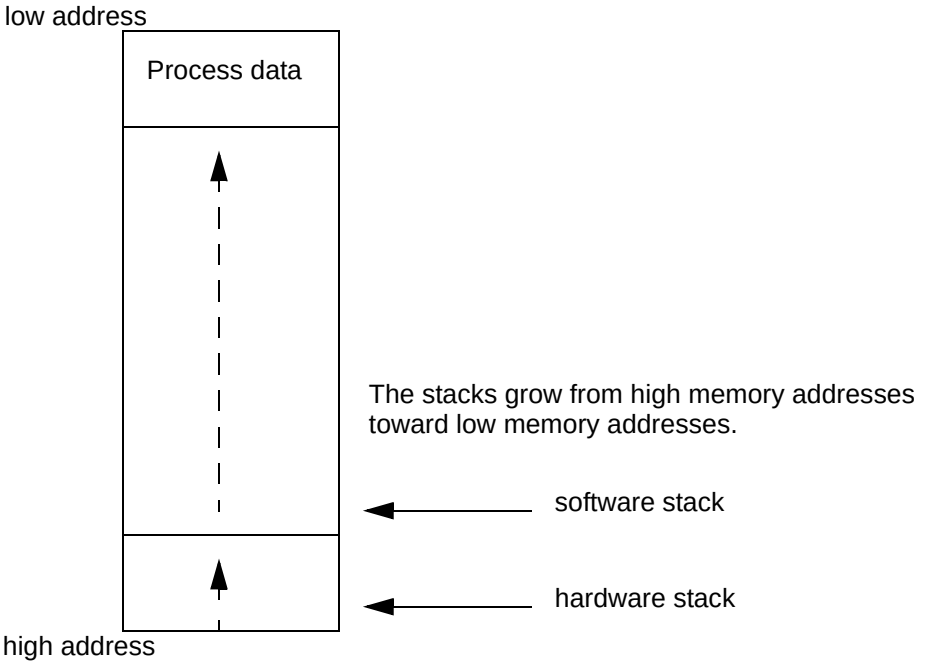
The `call push_xgset...` saves 6 registers on the stack (the name contains the bit patterns, e.g. `0x00FC` has 6 bits on) and then the `sbiw` instruction allocates 4 bytes on the stack. Thus, the local variable space for this function is 10. If the function calls other functions, you will need to trace through the paths.

If you are calling `printf`, then you should add 40 bytes to the software stack and 10 bytes to the hardware stack, as `printf` is fairly expensive. Other library functions are generally less expensive, and 6-10 bytes would be sufficient for most.

You will also need to add space to hold the CPU context, i.e., all the registers during a task switch. The #defines `MIN_STACKSIZE` and `MIN_HW_STACKSIZE` are good starting values for a task function that may have a small number of local variables and may contain one or two function calls. You can always start with these values, or add some increments to these values, and see how the system behaves. Invalid stack pointers are detected by the system's stack checking function; however, even if that capability has been turned off, unchecked stack overflow will probably cause a system to crash.

Internally, the stacks are allocated as part of the same block of memory that holds the process data structure.

The PROCESS DATA STRUCTURE



`printf` and Kernel Stack

Since it uses internal buffers, the library function `printf` is not reentrant (i.e., it cannot be interrupted and run by another task). Moreover, it uses ~40 bytes of software stack and ~6 bytes of hardware stack (on top of whatever your task needs). Therefore, if your task function uses `printf`, it should increase the stack sizes accordingly and also disable scheduling around the call:

```
...
eMOS_SchedOff();
printf(...
eMOS_SchedOn();
...
```

We also provide a version of `printf` that uses the kernel stack and disables interrupts. Its use should be limited, since it turns off interrupts. Therefore, if you have high-priority tasks or use the watchdog, or have time-critical tasks that are sleeping or hibernating, etc., then you should not use this version of `printf`! However, it does eliminate the need to allocate extra stack space, since it uses the kernel stack.

- `int eMOS_Printf(char *fmt, ...)`

  is the interface to use the library `printf` using the kernel stack and with interrupt disabled.

The System API

- `int eMOS_SysInit(void *address, unsigned size)`

  initializes the eMOS system. This must be called before other eMOS functions. The address and size are the location and size of the free memory space for the memory allocator (see MEMORY MANAGEMENT). A typical call may look like:

  ```
  extern int _bss_end; // defined by ICC
  ...
  eMOS_SysInit(&_bss_end, 1024*6); // 6K of free space
  ...
  ```

  The function will check the accessibility of the memory by writing and reading the content of the first and last byte of the memory region specified.

  Return Values:

  0 : success

  ERR_PID_NOT_ZERO : the next process ID is not zero. Either indicates a configuration error (e.g. the C startup did not initialize global variables correctly) or that the user calls `eMOS_TaskCreate` prior to calling `eMOS_SysInit` (`eMOS_SysInit` must be called first).

  ERR_CANNOT_WRITE_FREEMEM : cannot correctly write to the first byte of the free memory.

  ERR_CANNOT_WRITE_FREEMEM_END : cannot correctly write to the last byte of the free memory.

- `int eMOS_SysStart(void)`

  This starts the eMOS kernel. Typically this is done as the last line in the `main()` function. Once started, the CPU execution is controlled by eMOS and will not return to the original `main()` function.

Return Values:

(normally does not return once multitasking starts)

ERR_NULLTASK_NOT_FOUND : cannot find the NullTask, an unexpected internal error.

- `void eMOS_SysTickISRHook(void (*func)(void))`

  This arranges `func` to be called whenever the system tick interrupt handler is called, typically once every 10 ms. The function should be made as short as possible to minimize the impact to the system. See OPTIMIZING YOUR SYSTEM.

  IMPORTANT: since the function is called using the kernel stack, it must not call `eMOS_TaskWakeup` as it will cause the system to crash. Your own ISR, not using the kernel stack, may call `eMOS_TaskWakeup` without problems.

- `void eMOS_SysIdleHook(void (*func)(void))`

  This arranges `func` to be called whenever the system is quiescent and has no runnable user task. This is useful for putting the system in the low-power mode to conserve power and for only having interrupt handlers responding to events. See OPTIMIZING YOUR SYSTEM.

- `long eMOS_SysGetTicks(void)`

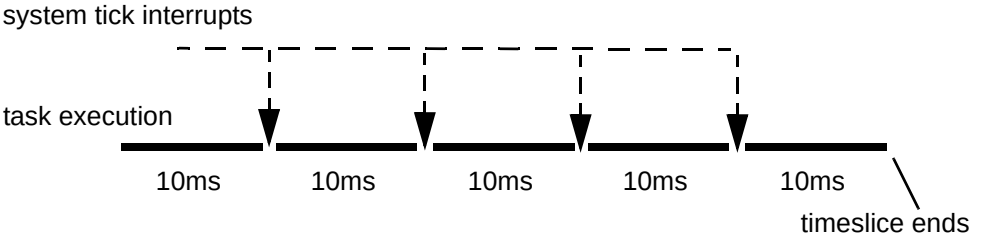
  This returns the number of ticks executed since the system start.

MULTITASKING KERNEL

Using an RTOS allows you to divide your program into multiple tasks or processes.[1] Since a (single core) CPU has only one path of execution at a time, the kernel of an RTOS manages the tasks and uses a scheduler to decide which task is run by the CPU. With eMOS, any C function can become a task. In fact, multiple tasks can share the same C function if desired.

eMOS is a preemptive priority-based multitasking kernel. It is called preemptive because periodically the kernel takes control of the CPU and the scheduler examines the list of all the tasks that are eligible to run (e.g., not waiting for a message, or blocked) and makes another task run when the control returns to the user tasks. The opposite of a preemptive kernel is a cooperative kernel, where a task explicitly gives up control before another task can be run. A preemptive kernel is far more flexible, since you do not need to partition your code such that it gives up processing at the right moment. The cost of a preemptive kernel is the overhead of needing to store the CPU context on each task's stack when a task switch occurs.

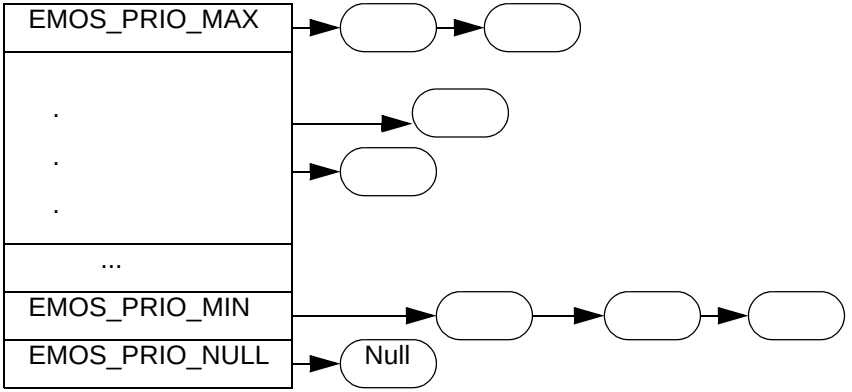
The frequency of the periodic system interrupt is called a tick or quantum, and is usually once every 10 milliseconds. Typically a task is allowed to run for a period of multiple ticks called a timeslice, and the system tick interrupt only does some housekeeping and does not call the scheduler unless either the process's timeslice is finished or a higher-priority process becomes eligible to run. When a task runs for completion of its timeslice period, a new task is selected to run for its timeslice period. This is called round-robin scheduling.



1. This document uses the terms task and process interchangeably.

To provide further control and flexibility, a task is given a (not necessarily unique) priority at task creation time. If a higher-priority task becomes runnable, the scheduler is called almost immediately, even before the next tick interrupt and without waiting for the current (lower-priority) task to finish its timeslice. This gives the system a very fast response time to real-time events.

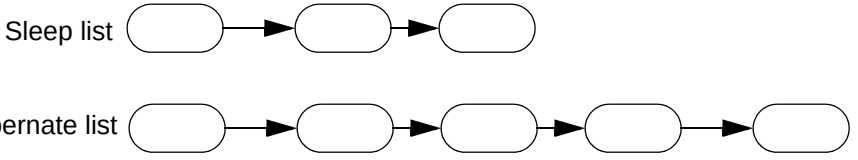
PROCESS TABLE of all runnable processes



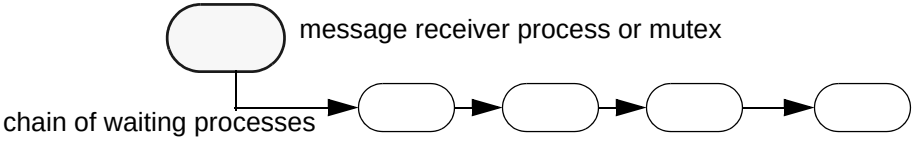
Each rounded rectangle denotes a runnable process. The process table is indexed by the priority values and each entry is a link to all runnable processes of that priority. The Scheduler always selects the highest priority runnable process to run. If the table is sparse and has empty entries, it takes time to skip over those entries.

The “null” task is the system idle task that runs when no other task is runnable.

The Sleep and Hibernate lists keep track of the sleeping and hibernating processes.



Processes blocking on message send are linked off the receiver process and processes waitinf on a mutex are linked off the mutex:



Interrupt Handlers

The preferred method of handling real-time events is to use high-priority tasks. As they become ready to run, they are scheduled almost immediately, so the response time is very fast. For system events that need even more immediate attention, you may use the system interrupt capability directly.

eMOS provides API functions for a user-written ISR to save the user context and switch to use the kernel stack, and an exit function to either cause an immediate task switch or to restore the user context.

Another option is to leverage the 10-ms system tick interrupt and have it call a function of your choosing. This allows you to run a lightweight function at the system tick frequency.

Interrupts and eMOS Functions

Most eMOS functions are run with interrupts disabled, and will blindly enable interrupts before returning. Therefore, generally, you should not call an eMOS function inside an interrupt handler. The exceptions are:

- `eMOS_SchedOn()`, and (less useful in an ISR) `eMOS_SchedOff()`
- `eMOS_TaskWakeup()`

`eMOS_TaskWakeup` is especially useful in a low-power system. See The Tasking API below.

Process States

A process state may be in one of the following states:

- running - the current process
- runnable - the process is ready to run
- blocked - the process is waiting for a message, or a mutex, etc.
- sleep - the process is sleeping for X system ticks
- hibernate - the process is hibernating
- zombie - the process has been killed but the resources (i.e. process data structure and any tracked memory) have not been reclaimed by the system yet.

Task Priorities

The lowest task priority is 0 and is reserved for the system's idle task (known as the "null task"). `EMOS_PRIO_MIN` is 1 and is the lowest priority you can specify when creating a task. `EMOS_PRIO_MAX` is the highest priority and is set to 3 by default. If you have a source license, you may change the `EMOS_PRIO_MAX` value to be any number that can be represented in the datatype of the `prio` field in the internal process structure, which is by default an `unsigned char`.

You can always start with a large priority range (e.g. set the `EMOS_PRIO_MAX` to a high value such as 256) and use values sparsely to accomodate future needs. Or, you can start with just a few priority levels and modify them later if needed. There is some minor performance impact with a large range, since runnable tasks are linked in a table indexed by the priority levels, and thus the more levels there are, the longer it takes to traverse the table.

The Tasking API

- `int eMOS_TaskCreate(char __flash *name, void (*func)(), unsigned char priority, unsigned stacksize, unsigned hw_stacksize)`

  This creates a task and returns its process ID.

  `name` is for your reference only, and it may be null (0). If non-null, you may search for a process by its name. The system does not check for duplicate names. To minimize data SRAM usage, `name` should reside in flash. Since usually you would use a literal string as the name, the option `Project->Options->Target->Strings in Flash` should be enabled.

  `func` is the task function. It should be a normal C or asm function. If the latter, it must follow the calling conventions of the C compiler. For a C function, to minimize stack usage, you should use the `#pragma ctask` to declare the function as a ctask. This directs the compiler not to save and restore Preserved Registers as dictated by the C calling convention, e.g.:

```
#pragma ctask task1
...
void task1() { ... }
// somewhere in your code
eMOS_TaskCreate("task1", task1, EMOS_PRIO_NORM,
MIN_STACKSIZE, MIN_HW_STACKSIZE);
```

The last two arguments are the stack sizes. For the Atmel AVR, since it uses two stacks (a software stack for data and a hardware stack for function calling), you must specify sizes for both stacks.

The stack must be large enough to hold the CPU context plus the deepest memory used by the functions called by the task function. If the stack is too small, the eMOS stack checking will reset the system, or the system will crash.

The macros `MIN_STACKSIZE` and `MIN_HW_STACKSIZE` can be used as defaults if the task function does not call other functions.

Return Values:

> 0 : process ID

ERR_OUT_OF_MEMORY : cannot allocate memory for process structure.

ERR_MAX_TASKS_CREATED: only applicable in the demo version, which limits the maxiumum number of tasks to 5.

- `int eMOS_TaskGetID(void)`

  This returns the current process ID.

- `char *eMOS_TaskGetName(int process_id)`

  This returns the name of the task.

- `int eMOS_TaskGetPIDByName(char *name)`

  Given a task name, this returns the process ID. Note: the behavior is undefined if you have multiple tasks with the same name.

  Return Values:

  >= 0 : process ID.

  ERR_PID_NOT_FOUND : no process with the name is found.

- `char *eMOS_TaskGetNameByPID(int id)`

  Given a task ID, this returns the process name. Returns 0 if there is no process with that ID.

- `void eMOS_TaskYield(void)`

  This gives up execution and allows other tasks to run. The task is still in a runnable state.

- `void eMOS_TaskSleepMs(int msec)`

  This put the task to sleep for `msec` milliseconds. The resolution is that of the system tick ISR.

- `void eMOS_TaskSleep(int sec)`

  This put the task to sleep for `sec` seconds. The resolution is that of the system tick ISR and may not be accurate for real-time clock (RTC) purposes.

- `void eMOS_TaskHibernate(void)`

  This gives up execution and puts the process into hibernation. The process will not be run again until awakened by another process.

- `int eMOS_TaskWakeup(int process_id)`

  This wakes up a hibernating process. This causes a task scheduling, so the awakened task may be run “immediately,” depending on its priority.

  You should not call this function while using the kernel stack; i.e. do not call this in a function that is hooked to the system tick interrupt using `eMOS_SysTickISRHook.`

  In a low-power system, you may put the system in a power-down mode and use an interrupt handler to wake up a key process as needed.

  Return Values:

  0 : success.

  -1 : cannot find process with ID. Note: this function does not call the `eMOS_UserSyscallError` function, since it may be common to blindly wakeup a process regardless whether it is actually hibernating or not.

- `int eMOS_TaskKill(int pid)`

  This kills a process. `pid` must not be zero.

  Return Values:

  0 : success.

  ERR_PID_NOT_FOUND : cannot find the process with ID or if `pid` is zero.

Interrupt Handler (ISR) API

These API functions allow your interrupt handler to use the kernel stack, and thus do not take up any resources on the process stacks, and on exit, cause an immediate task switch if needed. This can be useful if the ISR wakes up a task using `eMOS_TaskWakeup()` and wants the new task to run as soon as possible.

- `void eMOS_ISREntry(void)`

  This saves the user context (on the current process' stacks) and switches to the kernel stack.

- `void eMOS_ISRExit(int schedule)`

  This exits from the ISR. If `schedule` is non-zero, it causes an immediate task scheduling. Otherwise, it restores the user context.

The Scheduler API

The Scheduler API can be called by in interrupt handler.

- `void eMOS_SchedOff(void)`

  This temporarily stops the scheduling. This is less drastic than disabling interrupts and not allowing the system tick interrupt handler to run. This may be useful to prevent data corruption of library functions due to multitasking, e.g., `printf`.

  Note that the system tick ISR still operates and the sleeping tasks' sleep timer and any health monitor still gets decremented.

- `void eMOS_SchedOn(void)`

  This restarts the scheduler.

- `INTR_OFF()`

  This disables interrupts. It stops the multitasking kernel and should only be used in cases where eMOS kernel data is accessed. Since there is only one single kernel stack, interrupts are disabled during system calls.

  Interrupt should only be disabled for as short a time period as possible.

- `INTR_ON()`

  This enables interrupts.

MESSAGE PASSING

eMOS uses the QNX style of message passing: a message is an arbitrary number of bytes of data that is copied between a sender and the receiver. Unlike mailboxes in other RTOSes, messages are directly passed between the clients without kernel overhead and without using a separate construct (e.g., a mailbox) in the kernel. Messages are not interpreted by the system in any way. Message passing is more robust than other forms of Interprocess Communication (IPC), as tasks have well-defined synchronization points.

Message Passing is synchronous: the sender sends a message using `eMOS_MsgSend()` and waits until the receiver receives the message and replies to it before execution resumes. A receiver uses `eMOS_MsgReceive()` to wait for a message. If a receiver calls `eMOS_MsgReceive()` and there is no pending message, the receiver waits until a message is sent. Once received, the receiver continues execution and eventually would use `eMOS_MsgReply()` to reply to the original sender. Once a reply reaches the original sender, it becomes unblocked and its execution continues.

sender:

```
....
// sends & waits until receiver reply
eMOS_MsgSend(...
// continues processing
....
```

receiver:

```
...
// gets a message
eMOS_MsgReceive(...
// does processing
...
// replies to and unblocks sender
eMOS_MsgReply(...
```

While a message received is usually followed closely by the reply, the receiver may choose to interleave receive and reply calls with other receive and reply calls as long as a receive call is followed by its corresponding reply call at some points.

```
Typical call pattern:

...
eMOS_MsgReceive(&sender_id, ...
... // does something
eMOS_MsgReply(sender_id, ...
...
```

```
Unusual but also valid call pattern:

...
eMOS_MsgReceive(&sender1_id, ...
eMOS_MsgReceive(&sender2_id, ...
... // does something
// reply out of order, valid
eMOS_MsgReply(sender2_id, ...
eMOS_MsgReply(sender1_id, ...
...
```

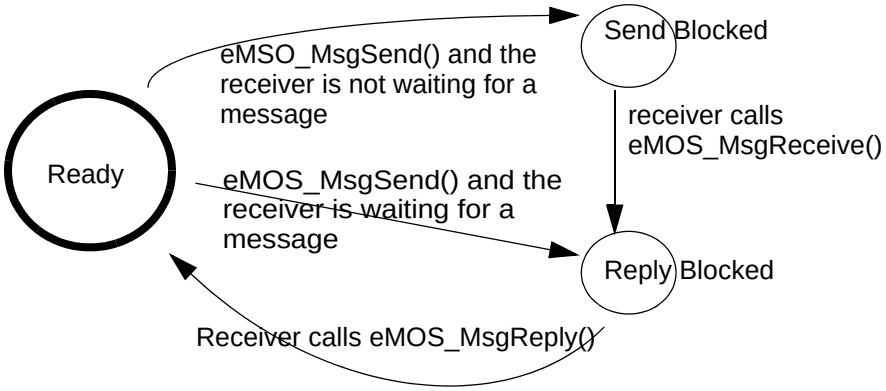
Typically with the message passing system, you divide your tasks into server tasks and client tasks. A server task uses the “receive” function to listen for requests and client tasks use the “send” function to ask for services.

An example:

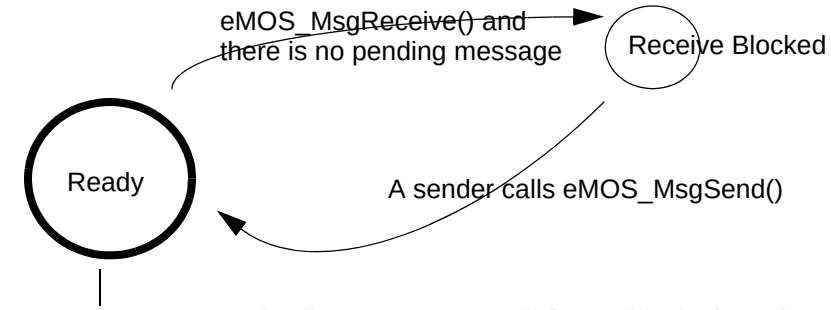
```
server:

while (eMOS_ReceiveMsg(&sender_id, buf, sizeof (buf)) >= 0)
   {
   ... // perform task
   eMOS_ReplyMsg(sender_id, replybuf, sizeof (replybuf));
   }
...

client:

if (eMOS_SendMsg(server_id, sendbuf, sizeof (sendbuf),
requestbuf, sizeof (requestbuf)) >= 0)
   {
   ... // perform task
   }
```

To summarize:

- Sending a message always blocks the sender until the receiver replies.
- Receiving a message blocks the receiver if there is no message pending. Otherwise, the data is copied immediately from a pending “message send.”
- Once a receiver receives a message, it needs to reply to the original sender so to unblock the sender. Message reply is not blocking, although it causes scheduling to happen so that the original sender may run.



State transitions of a task sending a message via eMOS_MsgSend()



State transitions of a task receiving a message via eMOS_MsgReceive()

The message passing primitives are the fundamental IPC mechanisms in eMOS, and they are tightly coupled to scheduling. When there is no need for task switching or blocking, data is copied immediately with the effect that the operation is very fast. If there is no message available, tasks are immediately blocked from execution so other tasks may run. Neither the user nor the kernel has to make independent decisions regarding sychronization and task scheduling, increasing the clarity of the user code and the robustness of the system design. Overall, this gives a very responsive system performance while avoiding many of the pitfalls with other IPC mechanisms such as mailboxes and semaphores.

The lengths of the buffers used in a send or reply operation may differ between the message sender and receiver. An eMOS message passing function always uses the shorter of the lengths specified (to avoid any possibility of buffer overflow) and returns the actual length used as its return value. In the case of `eMOS_MsgSend`, there are two buffers (the send buffer and the request buffer), and the system returns the length of the reply message.

Knowing the actual lengths of the messages passed between the sender and receiver allows your code to determine if there are mismatches due to data structure changes, and makes it easier to implement protocol such as transmitting a large buffer by breaking it into smaller chunks.

Asynchronous Send

While synchronous message passing is the norm, sometimes it is useful to have an asynchronous alternative. eMOS provides the function `eMOS_MsgAsyncSend` for sending an unsigned word (16 bits for AVR) to a message receiver. This function does not block the sender.

The receiver uses the function `eMOS_MsgReceive` to receive a message, regardless of whether is synchronous or asynchronous. A receiver determines the message type by checking the process id of the sender, which is set to negation of the process id if the message is asychronous. A receiver should not reply to an asynchronous message, as the sender is not waiting for a reply.

To read the 16-bit message, the receiver may use the following code fragment, (assuming the buffer is larger than 16 bits):

```
...
eMOS_MsgReceive(&pid, buffer, sizeof (buffer));
if (pid < 0) // asynchronous message
   {
   int msg = *(unsigned *)buffer;
   pid = -pid; // recover the actual pid of the sender
```

Avoid Deadlocks

If two tasks send to each other, then they are in a deadlock, since both tasks would be send-blocked. This applies to more than 2 tasks too: if you arrange all the tasks that send to others in a chain, the chain should not form a loop.

Priority Inversion

Message passing also avoids priority inversion, one of the sources of latent bugs in a priority-based OS. The most famous example of a priority inversion problem is when the Mars Pathfinder robot would reset itself after some hours of operation.

Priority inversion happens when a high-priority task cannot execute because a critical resource it needs is held by a low-priority task. The problem comes in when that low-priority task in turn is interrupted and cannot run because a medium-priority task becomes runnable. The medium-priority task then continues to run, preventing the low-priority task from running, which in turn never releases the critical resource that the high-priority task needs to resume execution.

This is exactly what happened on Mars: a shared information bus was used for communication between different components of the Pathfinder and priority inversion prevented a high-priority task from running, making the watchdog timer reset the system. Since the information bus was shared, access to it was regulated by using a mutex. In the Pathfinder, a high-priority but infrequently run Information Bus Management task moved certain data in and out of the information bus and a low-priority meterological task used the information bus to publish its data. Under some conditions, the low-priority task held the mutex and the high-priority task waited until the mutex was released. However, in the brief period that the low-priority task had the mutex, a medium-priority communication task, which did not use the information bus, preempted the low-priority task's execution. Thus, even though the high-priority task had a higher priority than the communication or the meterological task, it did not get run.

Under a message passing system like eMOS, once a task receives a message, the receiver task assumes the task priority of the sender task, since it is performing the task on the sender's behalf. The task priority reverts once a reply message is sent back to the sender. The tasks are in well-defined runnable or blocked states.

The Message Passing API

- `int eMOS_MsgSend(int receiver_id, void *sendbuf, int sendlen, void *requestbuf, int reqlen)`

  sends a message to the process with the process ID `pid`. `sendbuf` is the data to be sent and `requestbuf` is the buffer to store the reply result.

  If the receiver process is waiting for a message, i.e. in the `RECEIVE_BLOCKED` state, then the data is copied to the receiver's buffer immediately and the receiver process is unblocked. The sender is blocked and put into the `REPLY_BLOCKED` state. The scheduler is then invoked and the receiver process will run at some point depending on its priority and the priorities of the other runnable processes.

  If the receiver process is not waiting for a message, then the sender is blocked and put into the `SEND_BLOCKED` state. Pending messages are linked in priority order of the sender processes to the receiver process, so a higher priority sender will have its message processed first.

  Return Values:

  >= 0 : the length of the message actually copied into `requestbuf.`

  ERR_PID_NOT_FOUND : cannot find process with ID.

- `int eMOS_MsgAsyncSend(int receiver_id, unsigned msg)`

  asynchronously sends a 16-bit message `msg` to the process with the process ID `pid`. The call returns without blocking the sender.

  If the receiver process is waiting for a message, i.e. in the `RECEIVE_BLOCKED` state, then `msg` is copied to the receiver's buffer immediately, and the receiver process is unblocked.

  If the receiver process is not waiting for a message, then eMOS creates an internal data structure to store the message and chains the information to the receiver process. eMOS treats pending messages from `eMOS_MsgSend` and `eMOS_AsyncSend` the same way, and links them in the priority order of the sender processes to the receiver process, so a higher priority sender will have its message processed first.

  Return Values:

  0 : success.

  ERR_PID_NOT_FOUND : cannot find process with ID.

  ERR_OUT_OF_MEMORY : cannot allocate memory for internal data structures.

- `int eMOS_MsgReceive(int *ppid, void *receivebuf, int reclen, int timeout_ms)`

  receives a message. If `timeout_ms` is non-zero, then waits at most for `timeout_ms` milliseconds.

  The data is copied from the sender call to the receiver's buffer `receivebuf` directly. If the sender and receiver's message lengths differ, the shorter of the two will be used.

  If there is no message pending, the process is blocked and put into the `RECEIVE_BLOCKED` state until a message arrives or until the `timeout_ms` value expires.

  If there are pending messages from sender processes, the highest-priority sender is removed from the pending sender list and the data is copied to the argument buffer. The function returns without blocking the calling task.

  Return Values:

  >= 0 : the length of the message actually copied to `receivebuf`.

  ERR_SENDER_NOT_SEND_BLOCKED : the sender process internal state is not in SEND_BLOCKED. This is an internal eMOS system error.

  ERR_TIMEOUT_EXPIRES: the `timeout_ms` value expires.

  *`ppid` : the process ID of the message sender. If the received message is from `eMOS_MsgAsyncSend`, then the returned sender ID is the negation of the actual sender's process ID.

- `int eMOS_MsgReply(int sender_id, void *requestbuf, int reqlen)`

  replies to a previously sent message. Usually a reply call is made as soon as a message is received and processed. However, it is possible to reply to a sender in an arbitrary order differing from the message receive order.

  Keep in mind, though, that a receiver process executes with the priority of the sender process until a reply message is sent to the sender. At that time, the priority of the receiver process reverts to its original priority. If you interleave multiple `eMOS_MsgReceive` with `eMOS_MsgReply` calls in an unusual order, the running priority of the receiver process may not be as expected.

  The data is copied from the receiver to the original sender's buffer directly. If the sender and receiver's message lengths differ, the shorter of the two will be used.

  The reply function does not block the calling process, and unblocks the original sender.

It is your responsibility to reply to received messages. Otherwise, the sender process will wait forever. It is, however, an error to reply to an asynchronous sender.

Return Values:

>= 0 : the length of the message actually copied into the original sender's `requestbuf.`

ERR_PID_NOT_FOUND : cannot find sender process with ID.

ERR_SENDER_NOT_REPLY_BLOCKED : the sender process internal state is not in REPLY_BLOCKED. This means that either you use an incorrect sender ID, or there is an internal eMOS system error.

MUTEX

Mutex (Mutually Exclusive access to shared resources) is a traditional mechanism for controlling shared resource access. A mutex is either available or not available. If a task tries to obtain a lock on a mutex, the mutex is returned immediately if it is available. Otherwise, the task waits until it becomes available. When a task is done using the resources, it releases the mutex so other tasks may continue.

An example of mutex usage is a double buffer display: a normal task receives command input to “draw” on a memory buffer. At some interval (e.g. 10 times a second or every 100 milliseconds), another task copies the memory buffer out to the actual display buffer. In this scenario, a mutex can be used to control access to the memory buffer:

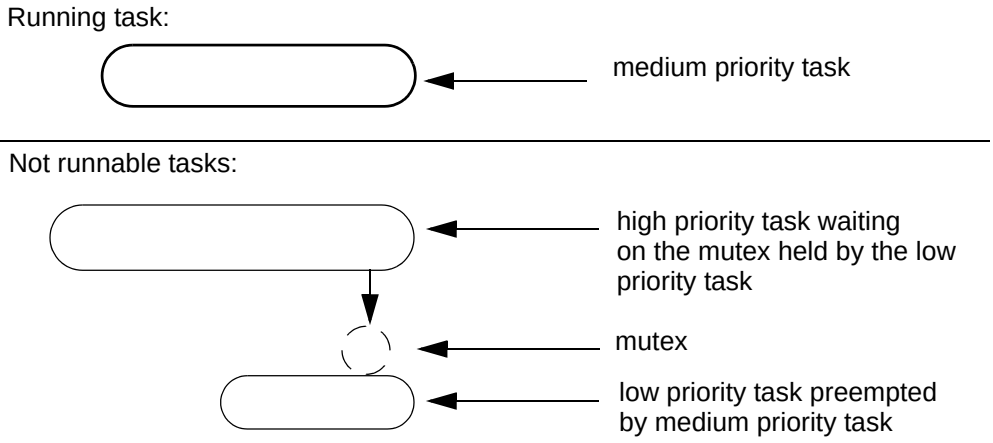
```
unsigned char memory_buffer[...];

Drawing Task:
...
while (1)
   {
   // commands come in as messages
   eMOS_MsgReceive(....

   eMOS_MutexLock(...
   memory_buffer[...] = ...
   eMOS_MutexUnlock(...
   eMOS_MsgReply(...
   ...
   }

Display Buffer Update Task:
...
while (1)
   {
   eMOS_MutexLock(...
   hardware_display... = memory_buffer...
   eMOS_MutexUnlock(...
   eMOS_TaskSleepMS(100);
   }
```

Priority Inversion and Mutex

Priority Inversion is a major problem in systems using mutex. In fact, the aforementioned Mars Pathfinder used a mutex to arbitrate exclusive access to the shared information bus. Therefore, this issue must be addressed for a mutex to be truly useful.

Two possible solutions exist: a) priority inheritance, where the priority of a task holding a mutex is temporarily raised to the priority of a task wishing to obtain a lock on the mutex, and b) priority ceiling, where the user at mutex creation time specifies the maximum priority level a task holding a mutex should be raised to, and promises that no task requesting the mutex would have a priority higher than this preset limit.

Priority inheritance is more flexible and uses the same semantics as used by the message passing API and therefore is the adopted solution for the eMOS mutex. The downside is that the mutex priority may get raised multiple times depending on the tasks requesting a lock.



Potential Priority Inversion Caused by Mutex Without Priority Inheritance

The Mutex API

- `int eMOS_MutexCreate(void)`

  This creates a mutex and returns a mutex ID.

  Return Values:

  >= 0 : mutex ID.

ERR_OUT_OF_MEMORY : cannot allocate memory for mutex structure.

- `int eMOS_MutexDestroy(int mutex_id)`

  Given its ID, this destroys a mutex. Any pending tasks will be unblocked.

  Return Values:

  0 : success.

  ERR_MUTEX_NOT_FOUND : cannot find the mutex with ID.

- `int eMOS_MutexLock(int mutex_id, int timeout_ms)`

  This locks a mutex and, if it is not available, waits until it becomes available. If `timeout_ms` is non-zero, then it will wait at most for `timeout_ms` milliseconds.

  Returns 1 if successful or a negative number otherwise.

  Pending tasks are ordered by the tasks' priorities. If a high-priority task tries to lock a mutex when it is not available, the task currently holding the lock will get its priority raised to the high-priority task's level until it releases the lock.

  If the mutex is already owned by the running task, it returns immediately with a success status.

  Return Values:

  1 : success.

  ERR_MUTEX_NOT_FOUND: cannot find the mutex with ID.

  ERR_OUT_OF_MEMORY : cannot allocate memory for mutex structure.

  ERR_TIMEOUT_EXPIRES: the `timeout_ms` value expires.

- `int eMOS_MutexUnlock(int mutex_id)`

  This unlocks a mutex. If there are tasks waiting to lock the mutex, the one with the highest priority gets the lock and is removed from the wait state.

  Return Values:

  1 : success.

  ERR_MUTEX_NOT_FOUND : cannot find the mutex with ID.

  ERR_MUTEX_NOT_LOCKED : mutex is not in locked state.

  ERR_MUTEX_NOT_OWNER : calling process is not the owner of the mutex.

COM PORT MODULE

The Com unit provides a uniform buffered input/output IO interface to serial devices. At the low level, it requires an interface function (could be an ISR routine) to read or write to the device, which can be an UART, a simple IO port, a SPI port, etc.

The Com unit operates on two levels: it presents a high-level interface (to task functions) for reading and writing bytes of data, and it interfaces to the low-level device port (using asynchronous interrupt handler if available). The purpose of the Com unit is to “even out” the datastream between these two layers. Internally it uses a FIFO buffer to manage the dataflow between these two levels of operations.

The Com Unit Descriptor

Since the Com unit can interface with a diverse number of interfaces with a sizable number of control variables, you use a Com unit descriptor (either flash or RAM based) to describe your device to the Com unit. A device can have an input port, an output port, or both:

```
typedef struct
   {
   unsigned timeout;     /* Recv time out */
   unsigned char eol;      /* End of line charactor */
   unsigned char wr_size;  /* Write FIFO size */
   unsigned char rd_size;  /* Read  FIFO size */
   unsigned char *tx_port; /* Transmit port adderss */
   void (*tx_ena)(void); /* Transmit enable  IRQ
fucntion pointer */
   void (*tx_dis)(void); /* Transmit disable IRQ
fucntion pointer */
   unsigned char *rx_port; /* Receive  port address */
   void (*rx_ena)(void); /* Receive  enable  IRQ
fucntion pointer */
   void (*rx_dis)(void); /* Receive  disable IRQ
fucntion pointer */
   } COM_DESC_TYPE; /* COMM device descriptor */
```

For an input-only device, you would leave the output/receive fields initialized to zero. Likewise, if it is an output-only device, you would leave the input/transmit fields initialized to zero.

To use the Com unit for a device, you open the the device using `eMOS_ComOpen`, and then use `eMOS_ComRead` and `eMOS_ComWrite` to do buffered IO. The low-level single byte IO is done using memory indirection (i.e., through the `tx_port` and `rx_port` addresses above). This works for the AVR even though it has a separate IO space, as the IO space is mapped to the data memory space.

`timeout` is the maximum number of milliseconds a process waits for an `eMOS_ComRead` call.

`eol` is the end of line character. `eMOS_ComRead` would also return if the end of line character is read.

`rd_size` and `wr_size` are the sizes of the internal input and output FIFO. They must be either 0 (if that IO operation is not supported) or a value between 16 and 256.

The IRQ enable and disable functions are called to prevent the internal buffers being corrupted when the Com unit is operating.

Device Number

To simplify operations, the Com unit defines 4 device numbers (0 to 3), which can be associated with a Com unit descriptor by using `eMOS_ComOpen()`. If you have a source license, you may increase the number of devices.

Device Interrupt and FIFO Size

The best throughput is to use your device IO interrupts, if available. The Com unit is written such that the IO interrupts are enabled and disabled as needed.

The read FIFO size `rd_size` should be based on how fast the data is coming in and how fast they are being read. If you expect a lot of data coming in from the device before a task would read the data from the FIFO, then you should choose a large FIFO size.

Likewise, the write FIFO size `wr_size` should be based on how fast the data is coming from the high-level task vs. how fast the low-level output operates. If you expect the task to write a lot of data and the low-level output is relatively slow, then you should choose a large FIFO size.

Com Unit API

- `void eMOS_ComInit(void)`

  This initializes the COM module.

- `void eMOS_ComTerm(void)`

  This terminates the COM module and deallocate all memory.

- `int eMOS_ComOpen(int dev, COM_DESC_TYPE *cd)`

  Given the Com unit descriptor and a device number, this opens a Com device.

  Return Values:

  0 : success.

  ERR_BAD_ARG : incorrect argument (e.g. out-of-bounds `dev` number).

  ERR_ALREADY_OPEN: com port is already open.

  ERR_OUT_OF_MEMORY: cannot allocate memory for the buffers etc.

  ERR_BAD_COM_DESC_TYPE: `cd` does not contain valid or sufficient information.

- `int eMOS_ComClose(int dev)`

  Given the device number, this closes a Com device and deallocates memory.

  Return Values:

  0 : success.

  ERR_BAD_ARG : incorrect argument.

  ERR_COM_NOT_OPEN : the com device has not been opened.

  ERR_INTERNAL_ERROR: some internal error has occurred.

- `int eMOS_ComRead(int dev, unsigned char *buf, int len)`

  This reads a series of bytes into `buf` from the device with device number `dev`. The call waits until either: `len` bytes has been read, or the end of line character (as defined in the Com unit descriptor) has been read, or the timeout period has been reached.

  Return Values:

  > 0 : number of bytes read.

  ERR_BAD_ARG : incorrect argument.

  ERR_COM_NOT_OPEN : com port not open.

- `void eMOS_ComWrite(int dev, unsigned char *buf, int len)`

  This writes a buffer of length `len` to the device with the device number `dev`. The data is copied to the internal FIFO. If there is not enough space in the FIFO, the process waits until more space is freed up by `eMOS_ComISRPut` calls.

Return Values:

> 0 : number of bytes written.

ERR_BAD_ARG : incorrect argument.

ERR_COM_NOT_OPEN : com port not open.

- `void eMOS_ComISRGet(int dev)`

  This reads a byte from the device. May be called by the target device's "byte available" ISR or called as a normal function call. The byte is put into the Com unit buffer, to be read by using `eMOS_ComRead()`.

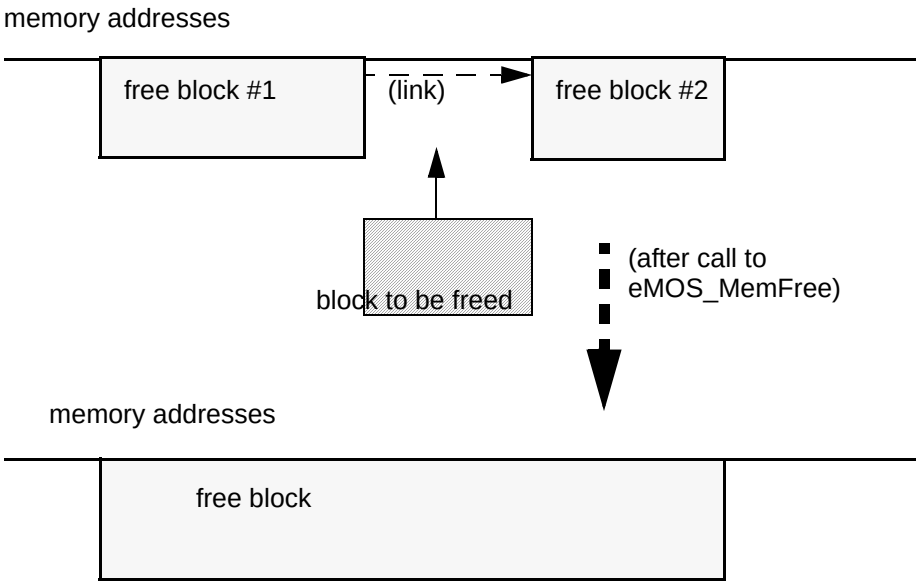
- `void eMOS_ComISRPut(int dev)`

  this takes a byte (if available) from the internal FIFO and writes it to the output port. May be called by the target device's "device available" ISR or called as a normal function call.

MEMORY MANAGEMENT

Many RTOSes avoid using dynamic memory management due to the potential fragmentation issues and the nondeterministic time needed for allocation and deallocation. However, we believe that a well-designed memory allocator will eliminate most fragmentation issues (except the ones caused by the user's usage pattern) and the time used by the typical memory management calls will fall within reasonable bounds. In return, the users do not need to pre-analyze how many tasks are needed, to hand-allocate arrays as stacks, etc., resulting in a cleaner system design in the end.

The memory allocator uses a best-fit always-merge algorithm. The free memory list is searched to find the smallest sized block sufficient for the allocation request and when a memory block is deallocated, it is merged with the neighboring free blocks to obtain the largest size free block possible. The former minimizes holes created by the allocation and the latter minimizes fragmentation.



Free blocks are merged aggressively to reduce fragmentation

eMOS uses memory allocation in three cases: to allocate the Task Control Block for a task and its stack, when a mutex is created (or destroyed), and when an asynchrous message is sent to a process not currently waiting for a message. (Synchronous message passing carries no memory allocation overhead in the kernel.)

When you call `eMOS_SysInit()` to initialize eMOS, you supply the starting address and the size of the free memory space for eMOS's memory allocator to use. `&_bss_end` is a good value to use for the starting address, as `_bss_end` is the label created by the ImageCraft linker to designate the end of the global data area used by your program:

```
extern int _bss_end; // defined by ICC
...
eMOS_SysInit(&_bss_end, 1024*6); // 6K of free space
...
```

You can use `eMOS_MemInit()` to add additional free space blocks to the system.

Resource Tracking

You can optionally enable eMOS to provide resource tracking. If enabled, allocated memory is chained off the process structure. You can free all the memory with a single API call, or all the memory will be freed if the process is killed or exits.

The Memory Management API

- `int eMOS_MemInit(void *address, unsigned size)`

  This adds an additional block to the free space pool. Returns 0 if successful, or a negative number otherwise (e.g.: memory cannot be correctly written).

  Return Values:

  0 : success.

  ERR_CANNOT_WRITE_FREEMEM : cannot correctly write to the first byte of the free memory.

  ERR_CANNOT_WRITE_FREEMEM_END : cannot correctly write to the last byte of the free memory.

- `void eMOS_MemEnableTracking(void)`

  This enables memory tracking for the current process. The behavior is undefined if not called within a task context.

- `void eMOS_MemDisableTracking(void)`

  This disables memory tracking for the current process. The behavior is undefined if not called within a task context.

- `void *eMOS_MemAlloc(unsigned size)`

  This returns a memory block or zero if none is available. The content of the memory blocked will be zeroed out.

  Return Values:

  > 0 : pointer to allocated memory.

  0 : out of memory.

- `void eMOS_MemFree(void *p)`

  This frees a memory block. It is OK to free memory with memory tracking turned on, although it will decrease the performance slightly.

- `void eMOS_MemFreeAll(void)`

  This frees all allocated memory for the current process. Only meaningful if memory tracking is enabled. This is done implicitly when a process is killed or exits.

- `unsigned eMOS_MemSpaceAvail(void)`

  This returns the amount of free space in bytes. Note that due to allocation overhead and fragmentation, not all space can be used.

- `unsigned eMOS_MemSpaceUsed(void)`

  This returns the amount of space used. Note that due to allocation overhead and fragmentation, the amount used will be larger than the total number of bytes requested.

Note: Users must be careful not to overwrite memory, or to free a memory pointer that they did not allocate, or to free a memory block twice, etc. Otherwise, data corruption will happen.

THE VIRTUAL WATCHDOG

eMOS provides a virtual watchdog system that can optionally work with the hardware watchdog system.

Virtual Watchdog

In the basic form, a watchdog wakes up periodically and sees if it has been “tickled.” If not, then it assumes the system has failed and reset the system. While software engineers like to view a watchdog only as a failsafe of last resort, in the embedded world, there are too many war stories about insufficient use of watchdogs.

For example, the NASA 1994 Clementine’s mission to visit the asteroid Geographos after 2 successful months surveying the Moon was not completed because floating-point exceptions eventually caused the thruster controllers to dump all the fuel. The software team insisted on not using the built-in hardware watchdog timer, resulting in the error conditions not being detected until it was too late. (That particular situation probably would have called for a more sophisticated watchdog system than the basic one, but the basic watchdog might have helped the situation.)

The eMOS virtual watchdog task executes in a forever loop. Whenever it wakes up, it checks the health of the system and then goes back to sleep. Between the times it runs, all runnable tasks except the ones executing in `EMOS_PRIO_MAX` priority level must “feed a cookie” to the watchdog at least once (this can be delayed per process, see below). The virtual watchdog also checks the process state to ensure that it is correct with respect to the list that the process belongs to, e.g., a process in the sleep list must be in a sleep state.

When you enable the virtual watchdog, you specify a minimum and a maximum sleeping time in milliseconds. The watchdog task is created with a priority of `EMOS_PRIO_MAX-1` and thus may not run if your system has maximum priority tasks that use up all the processing time. This also mean that by definition and design, ***maximum priority tasks do not need to tickle the watchdog***. As the highest priority tasks should be used for dealing with the most urgent events, this allows them to run without the overhead of tickling (AKA “feeding a cookie to”) the virtual watchdog.

Processes that are not runnable, e.g., a server process waiting to receive a message, will not be checked by the virtual watchdog (except for the consistency of its internal state value). However, the process must still feed the watchdog frequently “enough” when it is running.

Delaying the Virtual Watchdog

If enabled, every runnable task with a priority lower than `EMOS_PRIO_MAX` must feed the virtual watchdog at least once between the times when the virtual watchdog task runs by calling `eMOS_VWatchdogFeedCookie()`. This may be cumbersome for a

task that does a sequence of processing steps, then sleeps (or waits for a message), and then does some other processing and sleeps, etc. While it is possible to feed cookies to the watchdog in multiple places, to streamline programming, eMOS allows a process to delay having to feed the virtual watchdog.

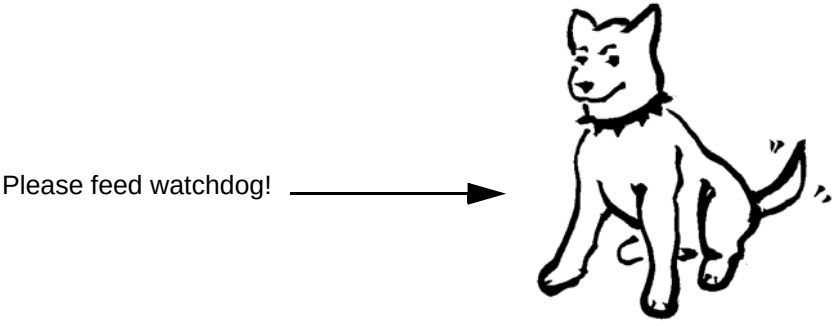
A process may call `eMOS_VWatchdogDelayCheck(unsigned msec)` to disable the watchdog from checking the health status of the process. `msec` has an upper bound of 6554 milliseconds or approximately ~6.5 seconds. The process must call `eMOS_VWatchdogFeedCookie()` before the delay period expires, and the delay period is canceled once the watchdog is fed. The delay is per process basis and does not affect checking of other processes.

Task With Multiple Cookie Calls

```
while (1)
{
eMOS_VWatchdogFeedCookie(..
)     ;
// do something
// sleep
eMOS_VWatchdogFeedCookie(..
)     ;
// do something
// sleep
```

Task with Delay Check Call

```
while (1)
{
eMOS_VWatchdogDelayCheck(..
)     ;
// do something
// sleep
// do something
// sleep
...
eMOS_VWatchdogFeedCookie(..
```



Which one to use is dependent on your program logic. The delay check is convenient as an alternative for multiple feeding, but process sleeping or hibernation time is not deducted from the delay time.

Hardware Watchdog

The virtual watchdog should be used in conjunction with the hardware watchdog if possible. When used in this manner, the virtual watchdog checks for the health of the entire tasking system, and the hardware watchdog provides a hardware failsafe in case the virtual watchdog is not sufficient.

Pitfalls

Although the virtual watchdog is very useful in detecting problems, it is not foolproof. The biggest source of problem results from memory corruption, and since microcontroller MCUs typically do not have memory protection mechanisms, there is no way to eliminate this potential problem completely. For example, the memory might be corrupted such that process data structures are damaged and then the watchdog functions might not operate reliably due to false memory content. It is very likely, however, that memory corruption would cause the system to fail in some obvious ways, so while the watchdog may not catch all the problems, it is still an important feature that you should exploit. In particular, when combined with a hardware watchdog, the chance of the “watchdog team” detecting a misbehaving system is high.

Another pitfall is that watchdog cannot catch logic errors such as indefinite sleep (unless you use a delay check), or processes that are in a deadlocked state (e.g., two processes sending messages to each other). Those are beyond of the scope of the watchdog system.

Since the virtual watchdog is a normal eMOS task (with the highest priority), if you disable the kernel, or if you have a highest priority task hogging the system, it may prevent the virtual watchdog task from running. Therefore, you should use the virtual watchdog in conjunction with the hardware watchdog.

Finally, hardware anomalies (including alpha particles affecting the content of SRAM cells) can be a real problem under certain conditions. Even though the virtual and hardware watchdog may not help, your chances of detecting a problem are higher with them working properly.

The Virtual Watchdog API

- VWATCHDOG_DESC flash descriptor

  The virtual watchdog is configured by initializing a vwatchdog descriptor in flash, which is defined in emos.h:

```
typedef struct
   {
   void (*wd_init)(void);
         /* function to initialize the HW watchdog */
   void (*wd_reset)(void);
         /* function to reset the HW watchdog */
   int  sleep_min;
         /* min number of msec for the watchdog task to
sleep */
   int  sleep_max;
         /* max number of msec for the watchdog task to
sleep */
   int  enabled;
         /* enable the virtual watchdog? */
   } VWATCHDOG_DESC;
extern __flash VWATCHDOG_DESC vwatchdog;
         /* must be initialized by user */
```

In the user module (typically called avr_usermod.c), you would initialize vwtachdog with the desired values, e.g.

```
void wd_init(void);
void wd_reset(void);
...
__flash VWATCHDOG_DESC vwatchdog = {
   wd_init,
   wd_reset,
   1000,
   2000,
   1
};
```

In this example, the virtual watchdog is enabled, with a sleep timeout period of between 1 to 2 seconds. The user also provides the two hardware watchdog functions: `wd_init` for initializing the hardware watchdog, and `wd_reset` to tickle the hardware watchdog. The hardware watchdog timeout period must of course set to be longer than the virtual watchdog's maximum sleep time.

Storing and initializing the descriptor in flash memory lessens the possibility of the virtual watchdog being rendered ineffective by errant program execution.

- `void eMOS_VWatchdogStart(void)`

  Checks to see if the `enabled` field of the `vwatchdog` descriptor is nonzero, and if so, creates the virtual watchdog task and calls the hardware watchdog init function if specified.

  This function should be called the user supplied function `eMOS_UserSysStart`. The reason this is not called by an eMOS function (e.g. `eMOS_SysStart`) is because this would allow the linker not to link in the virtual watchdog code if you do not use the feature (thus reducing your code size requirements).

- `void eMOS_VWatchdogFeedCookie(int status)`

  This "feeds a cookie" to the virtual watchdog and asserts the task's health_status. To provide a safety check, `status` must have the value `TASK_IS_HEALTHY`. Otherwise, this will be viewed as an errant call and the virtual watchdog task will be invoked immediately.

  This also cancels any "delay check."

- `void eMOS_VWatchdogDelayCheck(unsigned msec)`

  This starts a timer and delays virtual watchdog on this process. The maximum `msec` value is 6554 (milliseconds) or approximately 6.55 seconds. The timer is based on the system tick, and thus is only approximate.

  Before the timer expires, the process must feed the virtual watchdog. Note that the delay time is not affected by the process sleeping or hibernating, e.g., if a task sleeps forever (accidentally), the delay timer will still expire.

USER-SUPPLIED CODE

You need to provide the following functions and initialized data to get eMOS running on your target. A basic template is provided in the source file `avr_usermod.c` (`atm256_usermod.c` for ATM256). All user supplied functions have the form `eMOS_User...`. You may also make some changes to `emos.h` if you have a source license to recompile the source code. A copy of `avr_usermod.c` and `atm256_usermod.c` are provided, and you can use them as starting points for your needs.

To make changes to the hardware-specific interface, or to adapt eMOS to a different AVR chip, it is easiest to use the Application Builder in the ICC IDE to generate code for the timer, the watchdog, UART, etc.

Application Builder

If you use the Application Builder to generate the system tick timer interrupt or the UART initializations etc., be sure to:

- remove the `sei()` at the end of the `init_devices` function. `sei()` enables the interrupt and this needs to be delayed under eMOS.
- remove the call to `timer0_init()` in the `init_devices` function. Instead, call the function in `eMOS_UserSysStart` function (see below). The `timer0_init()` function enables the system tick timer interrupt and this needs to be done in the right place.
- change the

  ```
  #pragma interrupt_handler timer0_comp_isr:iv_TIM0_COMP
  ```

  to

  ```
  #pragma interrupt_handler eMOS_SysTickTimer:iv_TIM0_COMP
  ```

- if you use the watchdog, remove the watchdog enable function in the `init_devices` function. Instead, use the Virtual Watchdog to work with the hardware watchdog. See THE VIRTUAL WATCHDOG.

These changes are reflected in the supplied `avr_usermod.c` and `atm256_usermod.c`.

System Initialization and Start

When you call `eMOS_SysInit`, it in turn calls a user-supplied function:

- `void eMOS_UserSysInit(void)`

  This function performs device-specific initializations.

- `void eMOS_UserSysStart(void)`

  This function performs device-specific startup. Typically, you call the timer initialization and enable the timer interrupt here (e.g. call `timer0_init()`). If you use the virtual watchdog, you must also call `eMOS_VWatchdogStart` in this function. Calling `eMOS_VWatchdogStart` in a user-supplied module instead of in the eMOS system code allows the linker not to link in the virtual watchdog code if you do not use the feature (thus reducing your code size requirements).

emos.h Changes: Priority Levels and Tick Time

Priority levels can be changed:

- `EMOS_PRIO_MAX`

  This is the highest-priority level. This defines the number of priority levels in eMOS. The default is 4. `EMOS_PRIO_MIN`, the lowest-priority level, should not be changed from 1, as the priorities are used as indexes to a zero-based array internally, and 0 should be reserved for the system idle task.[1]

- `_TICK_TIME` and `TASK_TIME`

  If you make any changes to the system tick interrupt frequency, you should change these constants to match the changes. `_TICK_TIME` is the number of milliseconds for the tick interrupt. `TASK_TIME` is the number of ticks of the timeslice, or the amount of time a task executes (unless it is blocked) before the scheduler runs a different task. `TASK_TIME` is usually 50 ms or 5 ticks.

A shorter `TASK_TIME` period may give more tasks a chance to run, but may decrease overall system performance, as task switching takes time, around 450 cycles (see eMOS RESOURCE USAGE). With an 8-MHz clock, 10 millisecond corresponds to approximately 80,000 AVR instructions (minus some eMOS overhead) for each system tick and 400,000 instructions for each time slice. So, a time slice of 50 milliseconds corresponds to just over 0.1% overhead for task scheduling. A shorter system tick interrupt period may increase the responsiveness to certain hard real-time events but will decrease overall system performance as the interrupt handler overhead starts to have more of an effect.

1.The system will prevent you from assigning priority 0 to any other task.

Virtual Watchdog

To increase the robustness of the virtual watchdog system, its operations are controlled by a flash-based descriptor. See THE VIRTUAL WATCHDOG. You need to initialize the flash-based description `vwatchdog` with appropriate values. To enable the virtual watchdog, set the "`enabled`" field with a value of 1. Otherwise, set it to 0 to disable the virtual watchdog.

If the virtual watchdog is enabled and you want to also use the hardware watchdog, you will need to provide a function to initialize the hardware watchdog and a function to reset the hardware watchdog.

Error Functions

If the stack checking detects an error or when the virtual watchdog detects an error, the following user-supplied function is called:

- `void eMOS_UserSOS(int code, PROC_DUMP *pd)`

  This function informs the user of a catastrophic event. `code` is the error code (see GETTING STARTED) and you can use the system function `char *eMOS_ErrorString(int code)` to convert the code into an ASCII string. `PROC_DUMP` is a subset of the internal process data structure:

```
typedef struct
      {
      int pid;
      char __flash *name;
      unsigned char state;
      void *sp, *sp_bot;
      void *sw_sp;
      } PROC_DUMP;
```

  `sp` is the hardware stack pointer, `sw_sp` is the software stack pointer, and `sp_bot` is the lower bound for the stacks.

Of course if the system is very corrupted, the data may not be correct, but it still alerts you that something has gone awry.

The `eMOS_UserSOS` function is responsible for informing the user of the error condition somehow. Obviously, this is most useful during development where the device may blink an LED, log the data in NVRAM, sound a buffer, etc. In regular good practice, it is highly recommend that some observable actions should be taken even in production release, e.g.: in another embedded system "war story," a device was

responding slower than expected (this is in the field, where the device was already in production). Eventually they found that an unexpected error was causing the device to reset hundreds of times in a second, thus giving a very slow response time.

After the `eMOS_UserSOS` function is called, if the error comes from the virtual watchdog and if the hardware watchdog is enabled (i.e., the virtual watchdog is enabled and the hardware watchdog initialize function is defined), eMOS locks out all interrupts and loops indefinitely until the hardware watchdog kicks in. If the error comes from stack checking, or if the hardware watchdog is not used, then eMOS calls the following function:

- `void eMOS_UserSysReset(void)`

  This function resets the system. In the simplest case, it can jump to the system reset vector. However, if you want to put peripheral pins in some safe state and perfoem other system-specific reset considerations, you can do these before resetting.

- `void eMOS_UserSyscallError(char __flash *func, int code)`

  This function processes a system error. During development, you most likely want to display the function name and the error code. A sample implementation looks like this:

```
int eMOS_UserSyscallError(char __flash *func, int code)
      {
      cprintf("Syscall error in function '%S': %d %s\n",
            func,
            code,
            eMOS_ErrorCode(code);

      return code;
      }
```

  For production builds, and if you have a source license, you can define the macro NO_USERSYSCALLERROR in your `project->options->compiler->Macro Defines`. A system call simply returns the error code when there is an error. See STACK CHECKING.

OPTIMIZING YOUR SYSTEM

Minimizing Resource Usage

If you have a source license, you may do the following to minimize resource usage:

- disable stack checking by adding `NO_STACKCHECK` to `Project->Options->Compiler->Macro Define(s)`. This saves 4 bytes per each process structure and a small amount of code.
- Eliminate calls to the system call error function by adding `NO_USERSYSCALLERROR` to `Project->Options->Compiler->Macro Define(s)`. This eliminates call instructions and literal strings associated with the error messages.

As these are useful features, these techniques should only be used if you are critically running out of resources.

Using External Memory

You can also use external memory with eMOS. In the simplest case, you will need to set up the memory interface registers (e.g. the XMCRA and XMCRB registers) in your main function or in a modified C startup file, and then call `eMOS_SysInit` (see below) with the starting address and the size of your memory. If you have discontiguous memory chunks, you can use the function `eMOS_MemInit` to tell eMOS about additional memory chunks.

Avoiding Task Time Dynamic Memory Allocation

If you create all your tasks and mutex in your `main` function before you call `eMOS_SysStart`, you would minimize the time spent in memory allocation when the tasks are running and thus make the system more deterministic.

Declaring a Task Function Using `#pragma ctask`

You should declare your task functions as `ctask` so the compiler will not generate register saving and restore code that is not needed for top-level task functions. The pragma must appear in the same source file where the task function is defined and must appear before the function definition. For example:

```
#pragma ctask task1, task2, ...
void task1(void);
void task2(void);
...
void task1(void)
   {
   ...
```

```
   }
...
// in main() somewhere
eMOS_TaskCreate("task1", task1, ...
eMOS_TaskCreate("task2", task2, ...
...
```

The first two arguments to eMOS_TaskCreate are the task name and the function name. You can find the macro `EFN()` defined in `eMOS.h` to create both arguments:

```
// in emos.h
```

```
// in emos.h
#define EFN(f) #f, f
...

// in your code
eMOS_TaskCreate(EFN(task1), ...
// expands to
// eMOS_TaskCreate("task1", task1, ...
```

System Tick ISR Hook

eMOS provides several hook functions for you to optimize your system's performance.

- `void eMOS_SysTickISRHook(void (*func)(void));`

If you have a periodic function that you need to perform, you can piggyback onto the system tick interrupt (default 10 ms) and not use up another interrupt vector. It is very important that your function does not take too long to operate, so it does not adversely affect eMOS performance. The function will use the eMOS kernel stack for local variable and function calls, and thus should not call `eMOS_TaskWakeup`.

You must not declare the function as an interrupt handler, but you should declare it as `ctask` so that the compiler does not generate any unnecessary register-saving code:

```
#pragma ctask mytick_function
....
void mytick_function(void)
   {
   ...
   }
...
... // somewhere in your code
   eMOS_SysTickISRHook(mytick_function);
   ...
```

System Idle Hook Function

- `void eMOS_SysIdleHook(void (*func)(void));`

eMOS runs a “null task” when it is idle. In its entirety:

```
static void NullTask(void)
    {
    while (1)
    {
        if (emos_idle_function)
              (*emos_idle_function)();
        eMOS_TaskYield();
        }
    }
```

It is created with the lowest priority (`EMOS_PRIO_NULL` or `0`) and whenever it runs, it just gives up and yields the CPU. If this task runs, then all the other tasks in the system are not runnable. The `eMOS_SysIdleHook` function lets you hook into the idle task. When the null task runs, it will call your function.

Disabling the System Tick Interrupt

If you want to minimize power consumption as much as possible, you can use the idle hook function above and either stretch the system tick interrupt frequency or disable the system tick interrupt altogether, and put the system in a low-power standby mode or use the SLEEP instruction to put the CPU to sleep. You can arrange for the system to wake up when a certain interrupt happens and restore the timer interrupt and the rest of the system in a normal operation mode.

Temporarily Disabling the Scheduler

- `void eMOS_SchedOff(void);`

  This temporarily turns off the scheduler.

- `void eMOS_SchedOn(void);`

  This enables the scheduler.

This is useful to prevent multiple tasks from accessing non-reentrant C library functions such as `printf` without the overhead of a mutex. Generally, using `eMOS_SchedOff()` is not recommended, since it may prevent a high-priority task from running. However, it does not disable a sleeping task’s sleep timers, nor does it affect the virtual watchdog’s health monitor.

EXAMPLE

Here are some code fragments to demonstrate eMOS. The system reads a line from the UART port and echoes the line out to the UART port when a linefeed is received. First is the line output task:

```
#pragma ctask LineOutputTask, LineInputTask
...
void LineOutputTask(void)
   {
   int id;
   char buf[20];

   while (eMOS_MsgReceive(&id, buf, sizeof (buf)) >= 0)
         // send buf to the UART
   }
```

The line input task gets a character from the UART receive interrupt handler and puts it in a buffer. When a `\n` is received, it sends it to `LineOutputTask`:

```
int cinput;

void LineInputTask(void)
   {
   char outbuf[20];
   int index = 0;

   while (1)
         {
         eMOS_Hibernate();
         // wake up by UART handler
         outbuf[index++] = cinput;
         if (cinput == '\n' || index == sizeof (outbuf))
               {
               eMOS_SendMsg(output_process_id, outbuf,
index, 0, 0);
               index = 0;
               }
         }
   }
```

Note that there is a “race condition” where if another character-receive interrupt comes in before the `LineOutputTask` finishes receiving the buffer,[1] characters would be lost. The solution is left as an exercise for the reader. ;-)

The code fragment for the UART receive function:

```
#include <iccioavr.h>
....
#pragma interrupt_handler uart_receive:iv_USART0_RX
void uart_receive()
   {
   cinput = UDR;
   eMOS_SchedOn();
   }
```

Finally, to put the system into sleep mode in the idle task hook:

```
void IdleFunction()
   {
   eMOS_SchedOff();
   asm(“SLEEP”);
   }
...

...
main()
   {
   ... // set up for low-power mode etc.
   eMOS_SysInit(...);
   eMOS_TaskCreate(...);
   ... // TaskCreate for all tasks
   eMOS_SysIdleHook(IdleFunction);
   eMOS_SysStart();// never return
   }
```

With eMOS, we provide a framework that allow you to write your programs without too much concern about the tasking model. Our goal is for eMOS to work for you, and not the other way around.

1. To be precise: when `eMOS_MsgReceive` finishes copying the data from the buffer in `LineInputTask` to the buffer in `LineOutputTask`.

eMOS RESOURCE USAGE

These figures should only be used as a guideline, as the actual implementation may change. The timing cycles are usually the best-case scenarios with not many tasks or memory fragmentation. We at ImageCraft will do our best to keep the information updated.

Table 1:

API Module	Size in Bytes (flash)
Kernel API	4600
Message Passing API	800
Mutex	600
Memory Management	1100
Virtual Watchdog	1020
Com Module	2460

Table 2:

Data Structure	Size in Bytes (SRAM)
Process Data Structure	36
Mutex	9
Minimum Stack Space	58
Kernel Stack Size	50

Table 3:

Functions	Approximate # of Cycles
Timer interrupt	100
Timer Interrupt, no scheduling	160

Table 3:

Functions	Approximate # of Cycles
Timer interrupt to running a new task (scheduling)	450
Message send (until blocked)	340
Message receive with message pending	300
Mutex lock	120
Allocating a block (first find)	470
Allocating a block (find in 5th link)	600