# Small and simple real time operating systems

**Henrik Hoffström:** hhm00001@student.mdh.se

## Foreword

The four articles contained in this document describe a masters thesis in computer science done at Mälardalen University. The first document is a survey and comparison of six real time operating systems, the second is a description of how parts of the eCos interface were adapted for use with the Sierra Rtos. The third document is the user manual to the new interface and the fourth document is a reference description to how the interface is implemented.

## Contents

# A survey of Real-time operating systems

**Henrik Hoffström:** hhm00001@student.mdh.se

## Abstract

This report looks at the features available in the Real time operating systems: OSEK-VDX (illustrated by Ercos), Qnx, eCos, "VCB" and Symo. The focus is on the use of these rtos:es in small (<30Kb) single node embedded systems.  The report then goes on to discuss the changes that should be made to Symo an rtos for small-embedded systems currently in use at Mälardalen University.

# Contents

# 1 Introduction

There are two purposes for this report. The first is to make a survey of a number of the more widely used rtos for embedded systems, on the market. To be included in the report an rtos had to fit within 30Kb of memory. Note that this report only deals with these systems in a single processor single node environment. Even though several of the examined rtos:es has features enabling them to function in a networked/multiprocessor environment those features will be left out of the report. Also, note that the report will be made using only documentation freely obtainable. A more through report could undoubtedly be made by purchasing a copy of each of the examined rtos:es and using the included documentation but unfortunately that would require the expenditure of quite a lot of money. As it stands the complete published, documentation was available for OSEK-VDX, Ercos, eCos and Symo. The "VCB" and Qnx documentation was partially incomplete. This, of course, affects the accuracy of the information contained in this report. However, care has been taken to ensure that the information is as accurate as possible.

The second purpose of this report is to use the information from the survey to decide what, if any changes should be done to Symo a small simple rtos for embedded systems currently in use at Mälardalen university.

# 2 Terms

| | |
|---|---|
| **API** | Application Programmers Interface, The sum of all function calls available to an application programmer |
| **Embedded system** | A computer system that forms a component of a larger system and is expected to function without human intervention. |
| **Exception** | A software interrupt. |
| **Interrupt service routine (ISR)** | The routine that's called when an interrupt occurs. |
| **Application mode** | A description of a system complete with schedule, tasks etc. some rtos allows the programmer to specify more than one mode. I.e. an aircraft control system may have different modes for takeoff, landing and level flight. |
| **Task/Thread** | A task is a sequential programming performing certain functions, a real time application is usually made up of one or more sets of communicating tasks |

| | |
|---|---|
| **TCB** | Task control block, a structure containing information about a task, it's state, stack owned resources, the value of the processor registers etc. |
| **Real-time system** | A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at witch the results are generated [26] |
| **Round robin** | A simple scheduling algorithm were each task get to run for a pre allotted amount of time. |
| **Rtos** | Real time operating system, an operating system designed to be used in real time systems |

# 3 Outline

The rest of this report consists of an overview of four real-time operating systems currently available on the market or as freeware, namely eCos, Qnx, VxWorks and The OSEK-VDX standard, illustrated by a look at Ercos. The report will also includes a look at SYMO and VCB academic rtos:es currently in use at Mälardalen University. The reasons these particular rtos:es were chosen are:

**OSEK-VDX:** The Open System in Automotive Networks specification is a joint project of the automotive industry at the time of writing 62 different companies have contributed to the OSEK-VDX technical committee [1] The size of this project is reason alone to include it in this rapport. For better or worse the OSEK-VDX standard is bound to have a huge impact on the embedded systems market. The OSEK-VDX standard is illustrated by Ercos.

**Qnx:** The first version of this rtos was released in 1981 and has a, for a Posix compliant commercial rtos, small kernel.

**eCos:** Open source, supports EL/IX Level I, a Linux compatibility interface, for embedded applications in devices that are too small for even stripped down versions of Linux or that require real-time capabilities. eCos also has an ITRON compatibility layer, a popular standard in Japan. (eCos follows an older (3.02) definition of the ITRON standard. probably because the latest version (4.0) is only available in Japanese at the moment. [6]

**VxWorks:** According to [2] VxWorks is the most popular and complete Rtos in the embedded industry. This makes it worth looking at

**"VCB":** An academic rtos capable of functioning in a multiprocessor environment, or more correctly real time unit as it's implemented in hardware, and currently used in the Sara project [21]

**SYMO:** This system is used at Mälardalen University and is a simplified version of the "VCB" system that only handles systems with one node. One of the purposes of this report is to decide how the system should be developed in the near future.

# 4 Template

For each Rtos, the following aspects will be examined:

**4.01 Overview:** The examination of each rtos begins with a brief overview of the rtos

**4.02 Basic facts:** The examination continues by presenting a few basic facts namely: memory requirement, max number of task in the system, and the number of priority levels.

- **Memory requirements:** The amount of ram/rom needed by the rtos is a major issue for small-embedded systems since the amount of memory available is limited. Here the important figures are the minimum memory footprint and the amount of ram needed per task.

- **Priority levels:** Hard real-time systems need to be predictable and, since deadline driven schedulers are not yet commercially available [4], the best solution is to use a pre-emptive priority driven scheduler. This requires that each task be given a unique priority meaning that the number of available priority levels puts a limit on the maximum number of concurrent tasks in the system

- **Maximum number of tasks:** The maximum number of task may differ from the number of available priority level (e.g. the rtos RTAI has 2^30 priority levels [2], and no system can handle that many tasks)

**4.03 API richness:** An interesting issue is the richness of system calls available. Some rtos:es have very few as the minimum is just a scheduler and a synchronization primitive along with primitives to create, delete suspend and resume tasks. All other synchronization and communication primitives can be built on top of that. The more system call you have the less complex it is to write applications the trade of is that more system calls increases the complexity and, more important size of the kernel. On the other hand, the kernel code is likely to be better designed and debugged than the code of individual applications. Meaning that the more of the code that sits in the kernel the less errors [4]

For each Rtos the report only looks at the native API other supported API will be left out. E.g. eCos supports both the uITRON 3.02 and Posix 1003.1b API as well as it's own native API. However only the native API will be examined in the eCos section, other supported API will be examined in their own sections or left as future work.

The API features of each rtos will be overviewed in this section and summed up in a table in appendix A, the table was inspired by appendix 6 [5]. Note that Appendix A (and this section) only indicates the presence of certain feature i.e. a checked square say that a feature is present but a blank square doesn't mean that a feature is absent only that it's presence could not be verified.

**4.04 Task management:** How tasks are created, deleted, activated, resumed etc.

**4.05 Task states:** Which states a task can be in and what causes a state change.

**4.06 Synchronisation:** Real-time tasks often need to share some mutually exclusive resources (e.g. devices, memory.) Task may also be dependent of each other (i.e. don't execute statement A in task X until statement B in task Y has been executed). Synchronisation can also be used to ensure that some threads are executed one after the other. This creates the need for some sort of synchronization mechanism to ensure coordination. Below is a list of common synchronisation mechanism (from [4]).

- **Semaphore:** synchronization and exclusion
- **Mutex:** exclusion
- **Condition variables (in combination with mutexes):** conditional exclusion
- **Event flags:** synchronization of multiple events (can contain high level logic.)
- **Signals:** asynchronous event processing and exception handling.

An important feature is that the temporal properties of a system does not change depending on how many task are in the waiting list of the synchronisation objects. Good rtos design means that for each thread that starts pending on an object, the waiting list of that object is reorganized at that moment, so that the time it takes to release the object is independent of the queue list length

**4.07 Priority inversion:** A high priority task tries to take control of a mutually exclusive resource only to find that it's currently held by a task of lower priority. The high priority task now has no choice but to wait for the low priority task to release the semaphore. Unfortunately a mid priority task (that doesn't need the mutually exclusive resource) pre-empts the low priority task causing the high priority task to be further delayed since it can't continue executing until the low priority task releases the semaphore. This problem, that a high priority task can be delayed by a later arriving mid-priority task is called priority inversion.

**4.08 Memory management:** Most older rtos:es didn't support virtual memory due to the lack of an MMU on earlier processors, As most processors now come with a programmable MMU Virtual memory support is feasible in an rtos. Dynamic memory means that there is need for some sort of garbage collection. Garbage collection has the unfortunate effect that the timing of memory allocation calls can be highly varying as calls have to be blocked while garbage collection is in progress. Several rtos allows restricted use of Dynamic memory allocation. Another important fact is weather or not memory allocation is time limited. If the application is anything but none real-time, the asked for memory should be available immediately or at least with in a few milliseconds of the allocation call being made. Worst-case execution time of memory allocation must be known. Some systems therefore support a time out on memory allocation calls. If that feature isn't present, the Rtos will return an error code to the application stating that there wasn't any memory available. The application will then have to try again until it times out. This greatly adds to the complexity of the application code.

**4.09 Timers and clocks:** Almost all real-time systems works with relative time, the BEFORE and AFTER relations between events are known by the system. In a fully event driven system there is no need for a timer but as soon as you want to introduces system calls that deal with absolute time (such as wait a defined number of seconds) there is need of a clock and a timer.

**4.10 Supported standards:** There are several standards for rtos:es. If a rtos conforms to one or more of them, it will be noted in this section.

**4.11 Interrupt handling:** Real-time systems are expected to react to external event with in the given deadlines. These events are handled by interrupt service routines (ISR). In this section it is examined how an rtos deals with more than one simultaneous event.

An application designer often has to write his/her own interrupt routines. The routines tend to be difficult to debug, some rtos tires to limit the potential for errors by not allowing the programmer to change the interrupt vector table. This add some protection against programmer error at the added prices of extra indirect jumps and thus a reduction in interrupt handling performance

**4.12 Inter task communication:** Data exchange between tasks is necessary, and under, common mechanism for inter process communication are mailboxes, shared memory, message queues. There are several desirable properties and an rtos should offer most of them for example. Non-blocking communication, bounded operation latency and asynchronous communication. If shared memory is used the rtos must provide an API that help the programmer avoid the safety problems of shared memory.

**4.13 Scheduling:** If, as usually is the case, more than one task wants to use a resource (e.g. the processor, the communications buss) at once some sort of scheduling algorithm had to be use to decide which task gets the processor. A deadline driven scheduling algorithm (e.g. Earliest deadline first [9] ) would be ideal but none of the current available rtos support this. Most rtos therefore use some sort of pre-emptive based scheduler.

# 5 OSEK-VDX

**5. 01 Overview:** The OSEK-VDX standard is a joint project of the automotive industry. The standard consists of four parts The OSEK-VDX operating system (OS), OSEK-VDX communication (COM), OSEK-VDX network management (NM) and OSEK-VDX implementation language (OIL). The term implementation language as used in [1] is misleading, as OIL is actually a configuration language. There are many version of each of the four components and to avoid confusion the versions belonging together is defined by [1]. As this is a report about rtos:es the main interest is the (OS) part of the standard even though parts of COM also will be included as inter task communication is describe in that part of the standard, COM also deals describes a standard for communication between different nodes and that falls outside the scope of this report.. At the time of writing the current version of OSEK-VDX is 1.2 (Binding document) If not otherwise mentioned the information in this section comes from [13],[14],[15]

## 5.02 Basic facts:

- **Memory requirements:** This category is irrelevant as the subject is a standard and not an implementation.
- **Priority levels:** Minimum 8
- **Maximum number of tasks:** Minimum 8 / 16 depending on conformance class (see below)

**5.03 API richness:** OSEK-VDX has a small As task are defined offline there are not many features for manipulating them at runtime only two of the seventeen (2/17) features used as richness criteria is present. The API is somewhat richer when it's comes to intra-task communication but it's still (With the exception of SYMO that has no built in support for intra task communication.) the most limited in of the rtos:es examined (5/16) this also true for synchronisation (2/14) the second lowest of the examined rtos:es only "VCB" has less. As with most of the examined rtos:es the standard don't include any calls for memory management (0/10). When it comes interrupt handling the amount of available features is average (3/8). The features for handling clocks is poor (1/4) but when it comes to timers the API has a rich amount of features for handling timers. All the features used as richness criteria were present (6/6). Furthermore OSEK-VDX is one of the few rtos:es that provides support for event flags (5/9). Last, OSEK-VDX allows different application modes can be defined at generation time but mode changes are not possible at runtime (1/2).

**5.04 Task management:** Tasks are defined off-line using OIL. Tasks are activated by using the ActivateTask or the ChainTask system service. Tasks can't be terminated only temporary suspended by use of the perhaps badly named TerminateTask system call or the ChainTask call. All occupied resources must be released before a call to TerminateTask. Ending a task with out calling ChainTask or TerminateTask will cause the system to enter an undefined state. Chain task differs from terminate task in that it requires the programmer to specify a task that will be immediately activated upon the suspension of the calling task.

OSEK-VDX provides system specific hook routines to allow user-defined actions with in the Rtos internal processing hook routines are not interrupted by category 2 interrupts and may be used at: Startup (StartupHook), shutdown (ShutdownHook), debugging (PreTaskHook and PostTaskHook, called when entering/leaving a task) error handling (ErrorHook, called when a system called when an error is detected.) A hook can't use the full set of kernel call, see [13] p 41 for a complete list of which calls can be made by a hook routine.

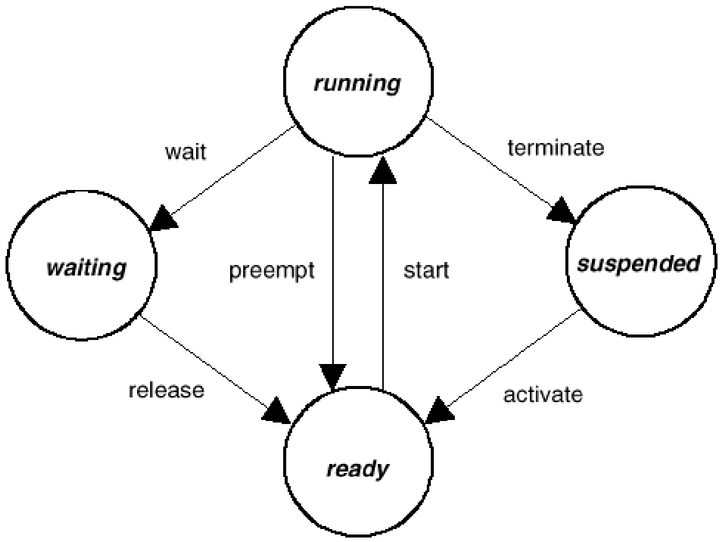
**5.05 Task states:** OSEK-VDX provides two types of tasks. The first type of task is Basic tasks basic task only release the processor if when:

1. They terminate.
2. They get pre-empted by a higher priority task
3. An interrupt occurs

A basic task can be in the following states:

- **Ready:** All tasks that potentially can be run are in this state
- **Suspended:** all tasks that currently can't be run are in this state. A task leaves the *suspended* state upon activation.
- **Running:** only one task can be in this state, if the currently running task is pre-empted it's moved to the ready state if it's terminated it moves to the suspended state.

The other type of task, called Extended task have one more state, the waiting state. A task will be moved to the waiting state if has to wait for one or more events. The task will be moved to the ready state when the event/events occur.



*The OSEK-VDX task state transitions*

OSEK-VDX defines four different conformance classes An Rtos implementing the standard has to state which conformance classes it adheres to.

1. **BCC1:** One task per priority level, no multiple activations, no extended task allowed

2. **BCC2:** More than one task per priority level, multiple activations allowed for basic tasks, no extended tasks allowed

3. **ECC1:** One task per priority level, no multiple activation, extended tasks

4. **ECC2:** More than one task per priority level, multiple activations allowed for basic tasks, extended tasks.

**5.06 Synchronisation:** Event flags, an individual event is identified by its owner and it's name (or event mask), resources. Resources are similar to Mutexes, there are two calls for managing them GetResource and ReleaseResource. Tasks failing to claim a resource are probably moved to the ready state. (The OSEK-VDX specification is unclear on this point)

ChainTask, Terminate task or WaitEvent may not be called while a resource is occupied. ISR are not allowed to return without first releasing held resources.

OSEK-VDX forbids nested access to the same resource. In case a task occupies more than one resource, they have to be released in LIFO order.

**5.07 Priority inversion:** Priority ceiling protocol, this can optionally be extended to interrupts as well.

**5.08 Memory management:** The OSEK-VDX standard doesn't say anything about how rtos:es conforming to the standard should deal with memory management.

**5.09 Timers and clocks:** OSEK-VDX provides a two-stage concept to handle reoccurring events. Implementation specific counters register events based on which the kernel offers an alarm mechanism. A counter is represented by a counter value, measured in ticks. There is no standardised way to directly manipulate a counter. There are however several system services to manipulated alarms, an alarm can be set to activate either after an absolute number of ticks or after a number of ticks relative to the current counter value. Alarms can be defined as either single or cyclic. In addition, the rtos provides services for getting the status of an alarm and to cancel an alarm.

**5.10 Supported standards:** OSEK-VDX

**5.11 Interrupt handling:** Interrupts have precedence over tasks. Higher priority interrupts can pre-empt lower priority interrupts. All ISR have a static priority assigned at system generation.

There are three categories of ISR in OSEK-VDX

1. The ISR doesn't use any system calls. After the ISR is finished, execution is resumed where the interrupt occurred. I.e. there is no impact to task management.

2. The ISR is allowed access to a subset of the system calls. The OSEK-VDX kernel provides a runtime environment for a dedicated user routine. This routine is assigned an interrupt at compiled time.

3. This category is a mix between 1 and 2, however as: "The OSEK-VDX committee has conceded that the specification might lead to improper implementations, so the category 3 interrupts will be eliminated in version 3.0 of the OSEK-VDX specification" [12]. The category 3 interrupts will be left out of this report

Interrupt sources can be enabled and disabled.

**5.12 Inter task communication:** Communication is based on message passing. There are two types of messages, queued messages that are stored in a messages queue and read in FIFO order (If a message is sent to a full message queue that message will be lost) and unqueued messages that behave a lot like global variables in that they are not consumed once read but stay the same until overwritten. This means that more than one receiver can read unqueued messages. Once a message transmission is preformed, a notification mechanism is set.

Message objects (queues and variables) are treated in the same way as resources there are two calls, GetMessageResource and ReleaseMessageResource to mange them. There is also a call, GetMessageStatus that checks the status of a message object. This can among other things check if a queue is full or empty or if the message resources is free.

OSEK-VDX defines five conformance classes for communication. However, as this report only deals with communication within a single node only two of them are relevant.

1. **CCCA:** Only unqueued messages, only one notified consumer for each message, GetMessage Status call not available, no message resources.

2. **CCCB:** All features of CCCA plus queued messages, unlimited notification, the GetMessagestatus call and messages resources.

**5.13 Scheduling:** OSEK-VDX supports both pre-emptive and none pre-emptive scheduling.

During none pre-emptive scheduling, a task switch can only be done under in the following cases.

- The currently running task is moved to the suspended state.

- Explicit request to do so made by using the system call Schedule.

- The currently running task is moved to the waiting state.

Task assigned the same priority is scheduled in order of activation; a pre-empted task will be scheduled before all other tasks at that priority level. A task moving from the waiting state to the ready state will be considered that last activated task at its priority level.

During pre-emptive scheduling rescheduling will be done when.

- The currently running task is moved to the suspended state.
- Explicit request to do so made by using the system call Schedule.
- The currently running task is moved to the waiting state.
- A task is activated by another task.
- An extended task performs a wait call.
- An event is set to a waiting task.
- A task releases a resource.
- An ISR returns.

Pre-emptive and non pre-emptive scheduling can be mixed. In that case scheduling policy depends on if the currently running task is pre-emptive or not.

Task can prevent them self from being pre-empted by locking the scheduler. The scheduler is in this case treated like a resource. This doesn't prevent interrupts for occurring but hinders the rescheduling of tasks.

# 6 Ercos

**6.1 Overview:** Ercos V4.1 developed by Etas [10] will be used as an example of an Rtos following the OSEK-VDX standard. If not otherwise stated the information in this section is from [10], [12]

Ercos currently supports the following target processors.

- Infineon C16x
- Motorola MPC555
- NEC V85x ATOMIC
- NEC V85x PHOENIX

- NEC V85x SF1
- Hitatchi SH7055F
- Motorola 683x6 and 6837x
- Motorola 68HC12
- Texas Instruments TMS 470
- ARM 7 based controller targets

Etas have developed Escape an essential configuration tool. Ercos cannot be run with out escape as Escape generates the code for inter task communication.

## 6.02 Basic facts:

- **Memory requirements:** Rom 3kb-7kb, ram 18-unknown, ram per task information not available, these figures are from [11]
- **Priority levels:** No information was found.
- **Maximum number of tasks:** No information was found.

**6.03 API richness:** Ercos claims to follow the OSEK-VDX standard. However, it only full fills CCCA for communication and BCC1 for tasks. Furthermore, Ercos implements a few features not supported by the OSEK-VDX standard. This means that compared to OSEK-VDX:

- The features for managing task are a little richer (3/17).
- There is full support for application modes (2/2) (Ercos is the only examined rtos than fully supports them.).
- The features for inter task communication is severely limited (2/16).
- There is a little support for memory management. (1/10)
- Lesser features for interrupt handling (2/8).
- There is no support for event flags (0/9).
- There are calls to allow the user to implement deadline monitoring. These are:

| | |
|---|---|
| SetDeadlineAbs | Set a deadline to an absolute time |
| SetDeadlineRel | Set a deadline to a relative time |
| CancelDeadline | Cancels a deadline |

| | |
|---|---|
| CheckDeadline | Checks if a specified deadline has been overrun. |
| CheckAndSetDeadline | Checks if a deadline has been overrun and resets it. |

**6.04 Task management**: In addition to what's specified by the standard Ercos implements a RestartTask call that places the currently running task in the ready state. This is a valid way to leave a task in addition to TerminateTask and ChainTask

**6.05 Task states:** After activation a task is in the ready state, if the tasks priority his higher than the currently executing task and a task switch is possible. The task is transferred to the *running* state and the previously running task is transferred to the ready state. Upon termination, a task is transferred to the suspended state. As seen, Ercos lacks a blocked state[1]. Thus, a task is activated statically and then run for a predictable amount of time. NB this doesn't say anything about how much time it takes from that that a task is activated until it finished it execution. What it says are how much time a task will spend in the running state until it finishes its execution.

**6.06 Synchronisation:** Mutual exclusion in Ercos is achieved with Resources. A resource is simply a special kind of variable defined in the offline tool Escape. There are two simple calls for managing resources, one to claim and one to release. A task trying to claim a resource held by another task will probably be moved to the ready state. (What happens to a task that fails to claim a resource is described in [12].) The application programmer is responsible for ensuring that locked resources are released in the correct order.

**6.07 Priority inversion:** Ercos makes use of a stack-based priority ceiling protocol. It differs from the normal priority ceiling protocol by letting the new task always inherit the maximum priority when it accesses a resource instead of only doing so when an access conflict occurs. This makes it impossible for a process that holds a resource to be pre-empted.

The ceiling priority for each resource is calculated offline by escape..

**6.08 Memory management:** There are system calls for checking both the Kernel and user stack for the conditions of overflow and almost overflow the later meaning that the stack is at least 75% full. Constant stack supervision can also be enabled.

**6.09 Timers and clocks:** Ercos provides two types of timer services:

[1] Ercos doesn't support OSEK-VDX Extended Conformance Classes (ECC1/ECC2) and thus doesn't support any event mechanism except interrupts.

1. A timetable service for time-triggered task with fixed repetition rates. The timetable is defined in the offline tool escape. There are online calls to start and stop the processing of a timetable. Multiple timetables can be defined and it's possible to switch between them at runtime.

2. An alarm service for tasks with varying repetition rates, this timer service allows for single delayed task activation by setting the start delay and no repetition period. It's also possible to change the repetition period of an alarm during execution[2]

**6.10 Supported Standards:** OSEK-VDX (partially)

**6.11 Interrupt handling:** Interrupts are assigned priorities in the same way as tasks. Tasks which are requested by the occurrence of interrupts are called Hardware tasks, these task represents an extension of the category 2 ISR as defined by the OSEK-VDX standard as they are allowed to occupy resources. Category 3 ISR as defined by the standard is not supported by Ercos.

**6.12 Inter task communication:** Communication in Ercos is not handled by the kernel but is handled by an offline tool used to generate optimised application code. Communication between objects is based on messages. The message semantics are similar to global variables. When a message is received it's not consumed, it's therefore possible to send the same message to multiple receivers, as the message isn't changed until overwritten. As can be seen Ercos conforms to CCCA. (See 5.12)

**6.13 Scheduling:** Ercos offers a mix of dynamic and static scheduling by making use of something called, a scheduling-sequence. A scheduling is sequence of processes to be executed in a certain order at a given priority level at the occurrence of some activation event. These scheduling-sequences are the result of offline analysis. Dynamic scheduling is then done on these scheduling sequences as a whole. The point of this mechanism is to reduce the numbers of different entities that have to be scheduled at runtime. **Note that in Ercos terminology a task is a scheduling-sequence.**

There are two ways to switch from a running task to a ready task.

1. Use cooperating scheduling; switches are then preformed at the borders between the processes in the scheduling-sequence. It's also possible to insert scheduling points in the application code by using the schedule call.

2. Use pre-emptive scheduling.

Ercos contains a mechanism for detecting if two tasks arrive closer than their minimum inter arrival time, if they are the system can be set to simply ignore tasks that arrive to early. The

[2] Changing the repetition period of an alarm is an extension to the OSEK-VDX standard.

rational behind this is that tasks that arrive to early are the product of wrongly estimated peek load scenario or malfunctioning hardware. The Ercos kernel also has a mechanism that limits the maximum number of instances of the same task than can be active at the same time. The reasoning behind this is to protect the kernel from overload.

# 7 QNX

**7.01 Overview:** QNX [16] has been developed by QNX Software Systems Ltd. The first version of QNX was released 1981 QNX is available for the x86, PowerPC and MIPS platforms QNX consists of a small kernel that in charge of a group of cooperating processes, a file system manager, a device manager, a network manager, and a process manager. As the subject of this report is rtos:es for small, embedded systems, the network manager and file system manager will be left out. The minimum QNX system is the kernel and a process manager. All QNX services except those handled by the micro kernel are provided by standard QNX process. These are for all purposes no different from user processes. This means it's possible to extend the rtos just writing a process that provides the required services. If not otherwise noted the information in this section comes from [16]

## 7.02 Basic facts:

- Memory requirements: 12k RAM/ROM [17]
- Priority levels: 32 [2] [16]
- Maximum number of tasks: Information not available.

**7.03 API richness**: The QNX documentation [16] gives the impression of a very rich api. Unfortunately, the documentation available when this report was written is incomplete in this respect and it therefore not possible to include even am estimate about the richness of the QNX API. Other than to say that QNX seems to contain at least 24/84 of the features selected as richness criteria.

**7.04 Task management:** The process manager is responsible for creating new task and managing the resources associated with each task. If a task wants to create a new task, it sends a message to the process manager.

There are three task creation primitives:

| | |
|---|---|
| fork() | Creates an exact copy of the parent task. |
| exec() | Replaces the calling tasks TCB with a new. It's a common practise in Posix systems to first do a fork and then have the newly created task call |

| | |
|---|---|
| | exec(). |
| spawn() | This call creates a new task as a child of the calling task. spawn() is not a part of the Posix standard. |

This table, from [16], describes the three kinds of task creation.

| Item inherited | fork() | exec() | spawn() |
|---|---|---|---|
| task ID | no | yes | no |
| open files | yes | optional* | optional |
| file locks | no | yes | no |
| pending signals | no | yes | no |
| signal mask | yes | optional | optional |
| ignored signals | yes | optional | optional |
| signal handlers | yes | no | no |
| environment variables | yes | optional | optional |
| session ID | yes | yes | optional |
| task group | yes | yes | optional |
| real UID, GID | yes | yes | yes |
| effective UID, GID | yes | optional | optional |
| current working directory | yes | optional | optional |
| file creation mask | yes | yes | yes |
| priority | yes | optional | optional |
| scheduler policy | yes | optional | optional |
| virtual circuits | no | no | no |
| symbolic names | no | no | no |
| real time timers | no | no | no |

‘
A task can be terminated in one of three ways.

- A signal is received whose defined action is to terminate the task.
- The task makes the call exit().
- The task returns from its main() function

Upon termination, all memory, proxies, interrupt handlers and timers held by the task are released.

## 7.05 Task states:

**READY** tasks that potentially can be run are in this state.

**RUNNING:** The Currently run task is in this state.

**SEND-blocked:** The task has sent a message but the recipient hasn't received the message

**REPLY-blocked:** The task has sent a message that has been received and the task is now waiting for a reply.

**RECEIVE-blocked:** The task has issued a receive request but there was no message available yet.

**SEMAPHORE-blocked:** The task is blocked waiting for a semaphore

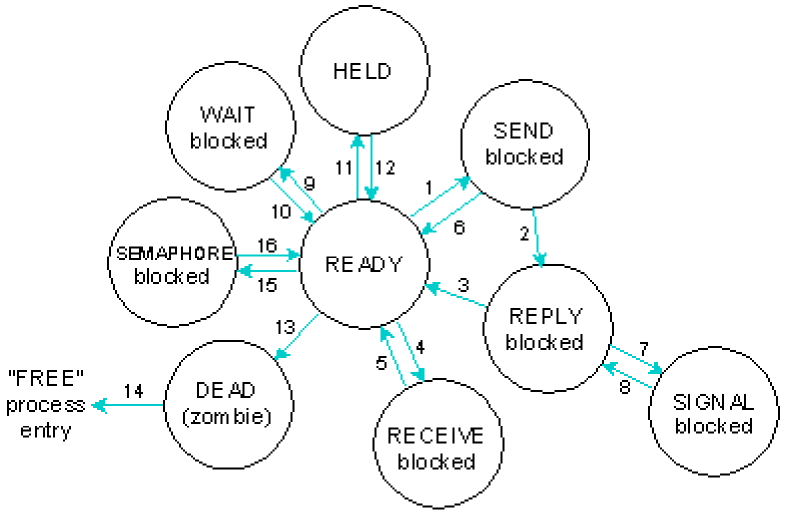
**SIGNAL- blocked:** While a task is processing a signal all signals of that type will be blocked.

**HELD:** The process has received a SIGSTOP signal. The task will remain in this state until it's terminated or receives a SIGCONT signal.

**WAIT:** blocked the task has issued a wait() or waitpid call to wait for status from one of it's child tasks.

**DEAD:** The task has terminated but it's parent hasn't issued a wait() or waitpid(). A DEAD process has a state, but the memory it once occupied has been released.

This figure from [16] shows the possible task states in QNX



**7.06 Synchronisation:** Message passing (See Inter task communication below.) provides a means of synchronisation. As a task blocks when it uses the Send() primitive and doesn't unblock until it receives a reply, message passing can be used to ensure that the sender task remain blocked until the receiver task has preformed some part of its execution. As the Receive () primitive also is blocking message passing can also be used to ensure that the receiver task doesn't continue its execution until the sender task has reach a certain point.

QNX also supports semaphores.

**7.07 Priority inversion:** No protection mechanism is set by default.

**7.08 Memory management:** All processes run in their own MMU protected space, dynamic allocation of memory is allowed.

**7.09 Timers and clocks:** Tasks can pause for a specified number of seconds by using the sleep or the delay function. A task can create and remove timers. These facilities are based on Posix 1003.1b. A timer can be set to expire either on an absolute time or at a time relative to the current clock value. A timer can also be set to expire cyclical; it's possible to change the value of a timer

**7.10 Supported standards:** QNX conforms fully to Posix 1003.1, 1003.2 (shells and utilities) and partially to Posix 1b/1.c (real-time)

**7.11 Interrupt handling:** All hardware interrupts are first routed through the kernel and the passed on to the appropriate driver or system manager. Interrupts are given priority levels, higher priority interrupts will pre-empt lower priority interrupts. Several tasks may be attached to the same interrupt if this is supported by the hardware

**7.12 Inter task communication:** message passing, proxies, and signals.

**Messages:** the fundamental form of communication between tasks. Provides synchronous communication and can also supply proof of reception and potentially a reply to the message
In addition to the usual Send () and Receive () primitives there is a special primitive Reply() that sends a reply to tasks that have sent messages. When a task uses the send primitive it will be blocked until it has received a Reply().There is a Creceive() call it's differs from a normal Receive() in that the task doesn't get blocked if there are no message available, instead the call returns immediately and the process continues it's execution. Messages are normally received in the order they were sent, however a task can specify that it will receive messages in the order based on the priority of the senders.

Tasks that don't need to communicate directly don't need to use these special message primitives. The QNX C library is built on top of messaging, task can use standard services such as pipes

**Proxies:** A special form of message especially suited for event notification where the sending task doesn't need to interact with the receiving task the only function of a proxy is to send a fixed message to the task that owns the proxy. By using a proxy a task or interrupt handler can send a message with out being blocked. Proxies are created by the qnx_proxy_attach() function. Any other task or interrupt handler that knows the identity of the proxy can then cause the proxy to deliver its predefined message by using the Trigger() function.

**Signals:** These provide support of asynchronous communication. A task can receive a signal in one of three ways.

1. If no special action to handle signals has been taken, the default action for that signal is taken.

2. The task can beset to ignore a signal, not all signals can be ignored

3. The task calls a signal handler for the signal; a signal handler is a function in the task that's invoked when the signal is delivered. When a process contains a signal handler for a signal it's said to be able to catch the signal. The signal handler for any given signal can be changed at any time. While a task is processing a signal, all signals of that type will be blocked. As a process that catches a signal is in fact receiving a form of software interrupt, no data is transferred with the signal.

This list of signals is from [16]

| Signal: | Description: |
|---|---|
| SIGABRT | Abnormal termination signal such as issued by the abort() function. |
| SIGALRM | Timeout signal such as issued by the alarm() function. |
| SIGBUS | Indicates a memory parity error (QNX-specific interpretation). Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated. |
| SIGCHLD | Child task terminated. The default action is to ignore the signal. |
| SIGCONT | Continue if HELD. The default action is to ignore the signal if the process isn't HELD. |
| SIGDEV | Generated when a significant and requested event occurs in the Device Manager |
| SIGFPE | Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow. Note that if a second fault occurs while your process is in a signal handler for this fault, the task will be terminated. |
| SIGHUP | Death of session leader, or hang up detected on controlling terminal. |
| SIGILL | Detection of an invalid hardware instruction. Note that if a second fault occurs while your process is in a signal handler for this fault, the task will be terminated. |
| SIGINT | Interactive attention signal (Break) |
| SIGKILL | Termination signal - should be used only for emergencies. This signal cannot be caught or ignored. Note that a server with super user privileges may protect itself from this signal via the qnx_pflags() function. |
| SIGPIPE | Attempt to write on a pipe with no readers. |
| SIGPWR | Soft boot requested via Ctrl-Alt-Shift-Del or shutdown utility. |
| SIGQUIT | Interactive termination signal. |
| SIGSEGV | Detection of an invalid memory reference. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated. |

| | |
|---|---|
| SIGSTOP | HOLD task signal. The default action is to hold the task. Note that a server with super user privileges may protect itself from this signal via the qnx_pflags() function. |
| SIGTERM | Termination signal |
| SIGTSTP | Not supported by QNX. |
| SIGTTIN | Not supported by QNX. |
| SIGTTOU | Not supported by QNX. |
| SIGUSR1 | Reserved as application-defined signal 1 |
| SIGUSR2 | Reserved as application-defined signal 2 |
| SIGWINCH | Window size changed |

Signals can be blocked; blocked signals will remain pending on the task it was sent to.

There is an important interaction between signals and messages. If a task is SEND or RECEIVE -blocked when a signal is generated the following will happen.

1. The process is unblocked
2. Signal handling takes place
3. The Send()/Receive() returns with an error.

If a process is reply blocked this causes problems as there is no way to know if the messages has been send or if the Send() should be retried.

It's therefore possible for a process receiving messages to be notified if the sending task receives a signal while reply blocked. The receiving task can then either:

- Complete the original request normally

- Return an error to the sender.

## 7.13 Scheduling:

QNX supports pre-emptive scheduling, if more than two or more task one the same priority level is ready one of the below strategies will be used to determine which task gets the CPU.

- **FIFO scheduling:** A task continues to be run until it gives up control of the CPU. i.e. makes a kernel call or is pre-empted.

- **Round robin scheduling:** A task continues to be run until it gives up control of the CPU. i.e. makes a kernel call, is pre-empted or uses up it's time slice.

- **Adaptive scheduling:** If a task consumes its time (i.e. doesn't block) it's priority is decreased by 1. Note this is only done when the task consumes its first time slice. The priority doesn't continue to drop. As soon as the task is blocked, it reverts to its original priority.

Each task on the system may be scheduled using any one of these methods. Scheduling strategy is assigned per task, not globally.

Rescheduling is done when:

- A task becomes unblocked.
- The time slice for a running task expires (If sliced scheduling are used)
- The running task is pre-empted

In QNX a task, receiving messages can be set to run at the priority of the task that send the message.

# 8 eCos 1.3.1

**8.01 Overview:** eCos developed by Redhat inc. is an open source real-time operating system for embedded applications. If not otherwise state the information in this section comes from [27] and [28]. eCos has both an uITRON 3.02 and a EL/IX level 1 Posix, -compatibility layer. eCos development kit (also open source) includes:

- **Red Hat GNUPro:** This includes a GCC compiler and a thread-aware version of GDB. This means that multi-threaded debugging of applications running on eCos is supported. In some cases, special hardware interfaces are also supported.
- **eCos Configuration Tool:** This tool provides a graphical view of the elements of the eCos rtos The eCos source code incorporates Red Hat's Component Description Language (CDL), and the Configuration Tool uses that information to assure that the configuration choices made will successfully build, link to the application and run on the target hardware. A memory layout tool is also available which assists the developer in modifying the memory map from that of the example evaluation board and quickly adapting it to the actual memory map of the final target board.

EL/IX is an application-programming interface (API) based on the Linux API allowing embedded applications to be ported easily and quickly between any EL/IX compliant operating systems.

eCos is available for the following platforms:

- ARM and Intel StrongARM
- Fujitsu SPARClite
- Hitachi SH-3

- Intel i386 PC
- Matsushita AM31/AM33
- Motorola PowerPC
- NEC V850
- MIPS

## 8.02 Basic facts:

- **Memory requirements:** 10 KB for start up plus a further target dependent amount. Minimum ram per thread 64 bytes, minimum ram per queue 8 bytes
- **Priority levels:** 32
- **Maximum number of threads:** No information was found.

**8.03 API richness:** The eCos api is of average richness. There is an average amount of features for managing threads (5/17) and a fair amount of features for inter task communication (7/16). As with most of the examined rtos:es the standard don't include any application modes (0/2) and event flags (0/9). However eCos is the only of the examined rtos:es that has a rich amount features for memory management built in (9/10) eCos also provides the largest amount of features of the examined rtos:es for interrupt handling (4/8) and the second largest amount of features for synchronisation (5/14). The eCos api is also rich with features for managing clocks (3/4) and the support for timers is good (5/6).

**8.04 Thread management:** The following calls are used to manage threads

| | |
|---|---|
| cyg_thread_create | Dynamically creates a thread, a newly created thread will be placed in the suspended state. |
| cyg_thread_suspend | Suspends a thread and moves it to the suspended state. |
| cyg_thread_delay | Suspends a thread for a specified amount of time |
| cyg_thread_kill | Terminates a thread, The programmer is responsible for freeing up all resources held by a thread prior to termination. |
| cyg_thread_delete. | Terminates a thread, The programmer is responsible for freeing up all resources held by a thread prior to termination. |
| cyg_thread_resume | Restarts a suspended thread. If a thread has been suspended multiple times since it last were run it will to be resumed an equal amount of times. |
| cyg_thread_release | Breaks the thread from any wait it's currently in. exactly how this works depends on the synchronization object the thread was waiting on. |
| cyg_thread_yield | Gives control to the next run able thread of equal priority if no such |

| | |
|---|---|
| | thread exists this call has no effect. |
| cyg_thread_set_priority | Dynamically changes the priority of a thread |

**8.05 Thread states:** No clear information can be gained from the eCos documentation [28] however there seems to be at lest 4 states:

1. A "running" state containing the currently executing thread
2. A "ready" state containing all threads that possible can be run
3. A "suspended" state for threads that can't be run
4. A "waiting state" for threads waiting on synchronisation objects

As the available information is sketchy, it’s not possible to provide a state transition diagram.

**8.06 Synchronisation:** semaphores mutexes, event flags, condition variables. Condition variables are used along with mutexes to grant several threads exclusive access to shared data.

A typical example of the use of condition variables is when one thread (the producer) is producing data and several other (consumer) threads are waiting for that data to be ready. The consumers will wait by invoking `cyg_cond_wait()`. The producer will lock access to the data with a mutex, and when it has generated enough data for the other processes to consume, it will invoke `cyg_cond_broadcast()` to wake up the consumers. It's also possible to wake up all the threads waiting on a condition variable by using the cyg_cond_broadcast call

cyg_cond_timed_wait makes it possible to wait for a predetermined amount of time on a condition variable and the return an error if the thread wasn't awakened before the time out..

Event flags are also supported allowing a thread to wait for a condition or combination of conditions. A "consumer side" thread can wait for a "producer side" thread to set any combination of flags.

When a thread set some combination of flags all threads whose requirements now are satisfied are awakened. Thus, flags have broadcast semantics. A variation of the wait call can specify that the flag value be cleared when the wait condition is satisfied in which case the setting of the flag bits would not be a broadcast. Blocking, none blocking and blocked with time out versions of these calls are provided.

As a thread can prevent it self from being rescheduled this can also be used to achieve mutual exclusion

**8.07 Priority inversion:** Simple, mutex priority inheritance this will not handled nested mutexes correctly.

**8.08 Memory management:** dynamic memory allocation based on memory pools. If the standard C library is used the malloc() call is available. Both fixed block size memory pools and variable blocks size memory pools are available.

**8.09 Timers and clocks:** eCos makes a strict distinction between counter clocks alarms and timers.

- A counter maintains a monotonically increasing counter that's driven by some possible irregular sources of ticks.
- A clock is a counter driven by a regular source of ticks i.e. it counts timer
- An alarm is associated with a counter and provides a mechanism for generating single or reoccurring events based on the counters value.
- A Timer is an alarm attached to a clock.

**8.10 Supported standards:** EL/IX Level 1, ISO C, (POSIX.1a, 1b, 1c, 1d) subset. uITRON 3.02 however neither uITRON 3.02 nor POSIX /(EL/IX 1) is fully supported.

**8.11 Interrupt handling:** Interrupt handlers are actually a pair of functions, one of which. (The interrupt service routine, or ISR) is executed immediately and run with interrupt services disabled. After the ISR exits but before the scheduler is invoked again, a delayed service routine (DSR) will be invoked. It executes with scheduling disabled but interrupts enabled, so that further invocations of the same DSR can be queued. A DSR cant use any call the will put it to sleep.

**8.12 Inter thread communication:** Message boxes, a message box can be created by cyg_mbox_create before the scheduler is started. Two threads in the typical producer-consumer relationship can then access the message box. The producer uses cyg_mbox_put to make data available to the consumer who uses cyg_mbox_get to access the data. The size of the internal message queue is configured by the CYGNUM_KERENEL_SYNCH_MBOX_QUEUE_SIZE parameter. Blocking, none blocking, and blocking with timeout versions of the message box access calls are available.

**8.13 Scheduling:** There are two schedulers. One that only accepts one thread per priority level and one that allows multiple threads per priority level with time slicing between threads at the same priority level. This scheduler supports a limit form of priority inheritance (see above)

The scheduler must be protected from concurrent access. The traditional mechanism to ensure this is to disable the interrupts. However, this has the drawback of increasing the interrupt latencies. The mechanism implemented by eCos is to maintain a counter Scheduler::sched_lock as long as this counter is positive rescheduling is prevented. A thread can prevent it self from being rescheduled by calling the Scheduler::lock() which increment the counter. Scheduling is reallowed by calling Scheduler::unlock() that decrements the counter.

# 9 VxWorks 5.4

**9.01 Overview:** VxWorks 5.4 developed by WindRiver systems, is the fundamental component of the Tornado II development platform and is according to several sources, [2], [19] the most adopted rtos for embedded systems in the world. VxWorks 5.4 contains more components than can be listed here. The focus will be on the components that make up the minimum VxWorks 5.4 system. If not otherwise stated the information in this section comes from [20]

## 9.02 Basic facts:

- **Memory requirements:** Information not available
- **Priority levels:** 256.
- **Maximum number of tasks:** Information not available

**9.03 API richness:** The VxWorks 5.4 api has the most features for task management of the examined rtos:es (8/17) the same goes for inter task communication (13/16) and synchronisation (7/14). As with most of the examined rtos:es the standard don't include any calls for memory management (0/10), application modes (0/2) and event flags (0/9). There is an average amount of features for handling interrupts (3/8) The support for timer are however the worst of all the examined rtos:es (2/6) and there are no features for handling clocks (0/4).

**9.04 Task management:** A task context includes

- The program counter
- CPU registers
- A stack for dynamic variables and function calls.
- A delay timer
- A Time slice timer
- Kernel control structures
- Signal handlers
- Debugging and performance monitoring values.

The following calls are used to create tasks:

| | |
|---|---|
| taskSpawn() | Creates and activates a new task |
| taskInit() | Initialise a new task |
| taskActivate() | Activate an initialised task. |

When a task is spawned, an option parameter can be specified the options are

| | |
|---|---|
| VX_FP_TASK | Executed with FPU |
| VX_NO_STACK_FILL | Doesn't fill the stack with 0xEE |
| VX_PRIVATE_ENV | Executes the task within a private environment. |
| VX_UNBREAKABLE | Disable breakpoints for the task. |

The task options can be altered after task creation by use of taskOptionsGet() and taskOptionsSet(). This feature is not fully implemented and currently only the VX_UNBREAKABLE option can be changed.

The following calls are used for task identification:

| | |
|---|---|
| taskName() | Get the task name associated with a task ID |
| taskNameToId() | Get the ID associated with a task name |
| taskIdSelf() | Get the ID of the calling task |
| taskIdVerify(). | Verifies the existence of a specified task. |

These calls can be used to get information about a task. Note this is a snapshot and the actual value can change at any moment after the call has been executed.

| | |
|---|---|
| taskIdListGet | Returns an array of the ID of all active tasks |
| taskInfoGet() | Get information about a task |
| taskPriorityGet() | Get the priority of a task |
| taskRegsGet() | Get the values of a tasks "registers" |
| taskRegsSet() | Set the values of a tasks "registers" |
| taskISupsended() | Checks if a task is suspended |
| taskIsReady() | Checks if a task is ready |
| taskTcb() | Gets a pointer to the tasks TCB |

Task can be dynamically deleted from the system using the following calls: (It's the application programmers responsibility to ensure that a tasks release all shared resources before it's deleted.)

| | |
|---|---|
| exit() | Terminates the calling task |
| taskDelete | Terminates the specified task |
| taskSafe() | Makes it impossible to delete the calling task. Any task that tries to delete the calling task is blocked until the calling task preformes a taskUnsafe() call. Nested calls are supported. For each call to taskSafe() a counter is incremented. Each call to taskUnsafe() decrement the counter as long as the counter is positive it's impossible to delete the task. |
| taskUnsafe() | Undoes a taskSafe() call. |

The following calls directly control a tasks execution:

| | |
|---|---|
| taskSuspend() | Suspends a task |
| taskResume() | Resumes a task |
| taskRestart() | Restarts a task |
| taskDelay() | Delays a task for a specified amount of ticks |
| nanosleep() | Delays a task for a specified amount of nanoseconds. |

Tasking Extensions

To allow additional task-related facilities to be added to the system, the wind kernel provides task create, switch, and delete hooks which allows additional routines to be called whenever a task is created, deleted or a context switch occurs

| | |
|---|---|
| taskCreateHookAdd() | Adds a routine called upon task creation |
| taskCreateHookDelete() | Removes the above routine |
| taskSwitchHookAdd() | Adds a routine called on task switch |
| taskSwitchHookRemove() | Removes the above routine |
| taskDeleteHookAdd() | Adds a routine called on task deletion |
| taskDeleteHookDelete() | Removes the above routine. |

All routines not involving the kernel can be called from a task Hook se [20] page 31 for a complete list.

## 9.05 Task states:

| | |
|---|---|
| **READY:** | The state of tasks not waiting for any resource except the cpu. The executing task is strange, as it may seem also in the ready state. |
| **PEND:** | The state of tasks waiting for some resources |
| **DELAY:** | The state of a task that is asleep |
| **SUSPEND:** | The state of a task that is unavailable for execution. This state is used primarily for debugging. |

| | |
|---|---|
| **DELAY + S:** | This state contains task that are both delayed and suspended |
| **PEND + S:** | This state contains tasks that are both pending and suspended |
| **PEND + T:** | This state contains tasks pending with timeout. |
| **PEND + S + T:** | This state contains tasks suspended and pending with timeout. |
| **Any of the above + I** | As above plus an inherited priority. |

These are some of the possible state transitions and the calls that cause them

| | |
|---|---|
| **READY ⇒ PENDING** | semTake(), msgQReceive() |
| **READY ⇒ DELAYED** | taskDelay() |
| **READY ⇒ SUSPENDED** | taskSuspend() |
| **PENDING ⇒ READY** | semGive(), mdgQSend() |
| **PENDING ⇒ SUSPENDED** | taskSuspend() |
| **DELAYED ⇒ READY** | expired delay |
| **DELAYED ⇒ SUSPENDED** | taskSuspend() |
| **SUSPENDED ⇒ READY** | taskResume(), taskActivate |
| **SUSPENDED ⇒ PENDING** | taskResume() |
| **SUSPENDED ⇒ DELAYED.** | taskResume() |

As there are 16 possible states and because there are many possible transitions, it's not possible to include a state transition diagram in this report. (Note that list above only contains the transitions between four of the 16 possible states)

## **9.06 Synchronisation:** signals, semaphores.

There are there types of semaphore.

1. **Binary:** A general purpose semaphore

2. **Mutual exclusion:** A special binary semaphore optimised to avoid problems with priority inversion, task deletion and recursion.

3. **Counting:** Like a binary semaphore but keeps track of the number of times a semaphore is given, optimised for guarding multiple instances of a resource.

Task queuing on a semaphore can be handled in priority order or FIFO order.

| | |
|---|---|
| semBCreate() | Creates a binary semaphore |
| semMCreate() | Creates a mutual exclusion semaphore |
| semCCreate() | Creates a counting semaphore |
| semDelete() | Deletes a semaphore |
| semTake() | Takes a semaphore, a task can be set to only wait for a specified amount of time before returning with an error. |
| semGive() | Releases a semaphore. |
| semFlush() | Unblocks all tasks pending on a semaphore |

A mutual exclusion semaphore differs from a binary semaphore in that

- It can't be used for synchronization
- It only the task that took it can issue a semGive() call.
- semGive() can't be called from an ISR
- The semFlush() operation is illegal.

To prevent that a task is deleted while it's in a critical region mutual exclusion semaphores have an option SEM_DELETED_SAFE, that when set automatically performs a taskSafe() with each semTAKE() and a taskUnsafe() with each semGice().Mutual exclusion can be taken recursively i.e. more than once by the same task. A recursively taken semaphore must be freed as many times as it was taken. Mutual exclusion semaphores can also be used to avoid the problem of priority inversion see next section.

A counting semaphore differs from a binary semaphore in the following way: It keeps track of the number of times the semaphore is given. Every time the semaphore is freed a counter is incremented every time a semaphore is taken a counter is decremented When the count reaches zero a task trying to take the semaphore is blocked.

As a task can prevent it self from being rescheduled this can be used to achieve mutual exclusion.

**9.07 Priority inversion:** Mutual exclusion semaphores has an option that enables the Priority inheritance protocol when this protocol is used task must be set to queue on semaphores in priority order.

**9.08 Memory management**: virtual memory supports is provided if it's supported by the target hardware.  All code executes in a single common address space. There is a checkStack() call that returns the amount of free space left on the stack.

**9.09 Timers and clocks:** VxWorks 5.4 includes a watchdog-timer mechanism that allows any function to be connected to a time delay. Watchdog timers are maintained as part of the system clock ISR.

| | |
|---|---|
| wdCreate() | Creates a watchdog |
| wdStart() | Starts a watchdog. |
| wdDelete() | Deletes a watchdog |
| wdCancel() | Cancels a watchdog |

**9.10 Supported standards:** Posix 1003.1b Real-Time Extensions.

**9.11 Interrupt handling:** The following routines are used to handle interrupts

| | |
|---|---|
| intConnect() | Attaches an ISR to an interrupt vector. |
| intContext() | Returns TRUE if called from an ISR otherwise FALSE. |
| intCount() | Get the current interrupt nesting dept. |
| intLevelSet() | Sets the processor interrupt mask level. |
| intLock() | Disable interrupts. |
| intUnlock() | Re-enable interrupts. |
| intVecBaseSet() | Get the vector base address |
| intVecSet() | Set an exception vector. |
| intVecGet() | Get an exception vector. |
| intLockLevelSet() | There are seven levels of interrupts this call locks all interrupt below the specified level, note it's impossible to lock level seven in this way |

Whenever the target architecture allows it all ISR uses the interrupt stack. This stack is allocated by the system at start-up time. However as some architectures doesn't permit the usage of a separate interrupt stack. In these cases, an ISR uses the interrupted tasks stack.

There are some calls that cannot be used inside an ISR for a complete listing of these see [20] page 87. The basic restriction is that an ISR cannot use calls that may cause the caller to block. Further more an ISR can't call routines that use an FPU as floating point registers are not restored when the ISR exits.

**9.12 Inter task communication:** Shared memory (As all code executes in the same address space this is trivial), message queues. Message queues multiple tasks can send and receive from the same queue. Messages can be queued either in order based upon the priorities of the senders or in simple FIFO order.

The following routines are used for handling message queues.

| | |
|---|---|
| msgQCreate() | Creates a message queue, the possible parameter to this function includes the maximum number of messages in the queue and the max size of each message. |
| msgQDelete() | Deletes a message queue |
| msgSend | Puts a message in the queue, this call can take an optional timeout parameter. a message can be specified as urgent, this ,means that message will automatically be placed first in the queue. |
| msgReceive | Gets the first message in the queue, this call can take an optional timeout parameter |

**Signals:** Any task or ISR can raise a signal. The task being signalled immediately suspends it's Execution and starts it's signal handler the next time the task is scheduled for execution. The Signal handler executes in the task context and makes use of the tasks stack. A blocked task re Receiving a signal will be moved to the ready state so signal handler can run.

The wind kernel supports UNIX-BSD style signals.

The following calls are used to manage signals

| | |
|---|---|
| signal() | Specify the handler to be associated with the signal. |
| kill() | Signals a task. (This is not a misprint, the kill call is used to raise a signal) |
| sigvec() | Examine or set the signal handler for a signal. |
| pause() | Suspend a task until a signal is delivered. |
| sigsetmask() | Manipulate a signal mask. |
| sigblock() | Blocks a signal. |

Task can attach their own handlers for certain hardware exceptions through the signal facility. Signals are used for hardware exceptions as well as software exceptions.

### 9.13 Scheduling: Pre-emptive as default but can be set to round robin.

The following calls control the scheduling:

| | |
|---|---|
| kernelTimeSlice() | Control round robin scheduling. |
| taskPrioritySet() | Changes the priority of a task. |
| taskLock() | Disable task rescheduling. |

# 10 “VCB”

**10.01 Overview:** "VCB" for lack of a better name ("VCB" is really only the name of the communication system and stands for Virtual Communication Buss.)  is a hardware implemented rtos used in the SARA project [21]. The rtos is designed for use in a multiprocessor environment and has support for up to three CPU at once. If not otherwise noted the information is this part of this report is from [22] and [23]

## 10.02 Basic facts:

- **Memory requirements:** N/A "VCB" is implemented in hardware
- **Priority levels:** 64
- **Maximum number of tasks:** 128

**10.03 API richness:** “VCB” has an average amount of features for task management (5/17) and as with almost all the examined rtos:es “VCB” don’t contain features for memory management (0/9), application modes (0/2) or event flags (0/9). The amount of features for handling interrupts is average (3/8) and the same goes for clocks (2/4). Furthermore, “VCB” has no support for synchronisation (0/14) and poor support for timers (3/6). Support for intra task communication is average (5/16)

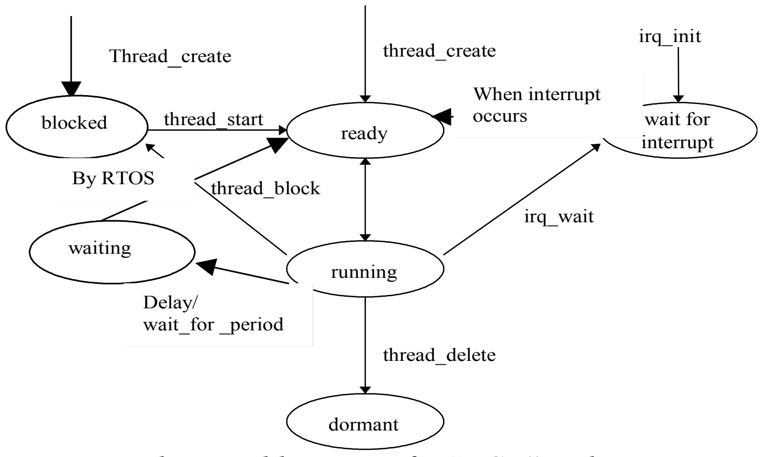
**10.04 Task management:** These calls are used to manage tasks

| | |
|---|---|
| rtu_thread_create | Creates a thread, the new be placed in the blocked, suspended or ready state. |
| rtu_thread_delete | Threads are removed with this call. The rtu doesn't deallocate any resources held by the thread. |
| rtu_thread_start | Starts s thread |
| rtu_thread_block | Blocks a thread |
| rtu_thread_suspend | Suspends a thread |
| rtu_thread_resume | Resumes the execution of a thread |

| | |
|---|---|
| rtu_thread_set prio | Changes the priority of a thread |
| rtu_thread_yield | The thread will be moved to the blocked state and control of the CPU will be given to another thread at the same priority level. if no such thread exists nothing happens. |
| rtu_thread_getinfo | Gets the status of a thread |

**10.05 Task states:** A task can be in the following states.

- Suspended: A task is moved here as a result of a thread_suspend call.
- Blocked/waiting: A task is moved here as a result of a thread_block call
- Ready: a task is moved here as a result of a thread_start or thread_resume call depending on if it was in the blocked/waiting state or in the suspendend state
- Wait for interrupt: A task is moved here as a result of a irq init or if it was in the running state a irq wait. Only interrupt handler thread can be in this state.
- Dormant: a task is moved here as a result of a thread_delete call.



*The possible states of a "VCB" task.*

**10.06 Synchronisation:** The available "VCB" documentation doesn't mention any synchronization mechanisms

**10.07 Priority inversion:** The available "VCB" documentation doesn't mention of any mechanism to avoid priority inversion.

**10.08 Memory management:** No information available

## 10.09 Timers and clocks: 
The following calls are used to handler timers and clocks

| | |
|---|---|
| rtu_time_settimneoutthread | This call specifies the thread that handles all time out functionality. When the timeout expires, this thread is sent to the ready queue a scheduled as a normal thread. |
| rtu_time_gettimeoutthread | Returns the Id of the timeout thread. |
| rtu_time_removetimeoutthread. | Removes the timeout thread. |
| rtu_timesetfrequency. | Specifies the clock frequency of the rtu. |
| rtu_timegetfrequency | Gets the clock frequency of the rtu. |
| rtu_timesetbasetime | Sets the resolution of the internal clock |
| rtu_timegetbasetime | Get the frequency of the internal clock. |
| rtu_time_setperiodicthread | Specifies the thread that handles all periodic timers in the system. The periodic thread can be set under supervision. If so the thread will only be sent to, the ready queue if it's made an rtu_time_waitnextperiod call before the timer has expired. If the timer expires before the call is made a counter will keep track of the number of missed time periods |
| rtu_time_getperiodicinfo | Returns the id of the current periodic timer thread |
| rtu_time_removeperiodicthread | Removes the periodic timer handler thread |
| rtu_time_waitnextperiod | When a periodic timer expires, this thread is send to the ready queue and scheduled as a normal task. |

All timers can be set can be set with a resolution of 1 us.

## 10.10 Supported standards: 
none

## 10.11 Interrupt handling: 
Interrupt handlers are threads in all respect identical to normal threads. They get their status as interrupt handler by having the rtu_irq_init call performed on them.

The following calls are available to handle interrupts..

| | |
|---|---|
| rtu_irq_init | Specifies the thread that will be called when an external interrupt at a given level occurs i.e. installs an interrupt handler. This thread is placed in the wait for interrupt state |
| rtu_irq_wait | This call is used to indicate than an interrupt service thread has finished it's execution and |

| | |
|---|---|
| | is ready to process a new interrupt as a result of this the interrupt service thread will be blocked. |
| rtu_irq_remove. | This calls removes a thread from the wait for interrupt state. i.e. uninstalls an interrupt handler. |
| rtu_irq_getinfo | Obtains the status of an interrupt handler thread |

## 10.12 Inter task communication:

The following calls are available

| | |
|---|---|
| send() | Sends a message, its possible to specify the priority of the message. |
| rsend() | Sends a message, this is a nonblocking send. |
| receive() | Waits a given number of ticks for a message |
| broadcast() | Sends the message to all receivers. |
| sendwait() | Sends a message and waits (with timeout) for a reply. |

The maximum size of a message can be specified at system configuration.

**10.13 Scheduling:** Pre-emptive priority based with Round robin scheduling as an option. Unfortunately, the documentation doesn't go into detail.

# 11 Symo

**11.01 Overview:** Symo is a trimmed version of the same rtu that used in the "VCB" system. The main purpose of this paper is to examine a number of rtos in order to find if there is any features that should be implemented in Symo. This section describes Symo as it is to day. If not otherwise, state the information in this section is from [24]. Symo exists both as a hardware and as a software -implementation.

Symo is written for the Motorola M68332 processor

## 11.02 Basic facts:

- **Memory requirements:** Information not available

- **Priority levels:** 8.

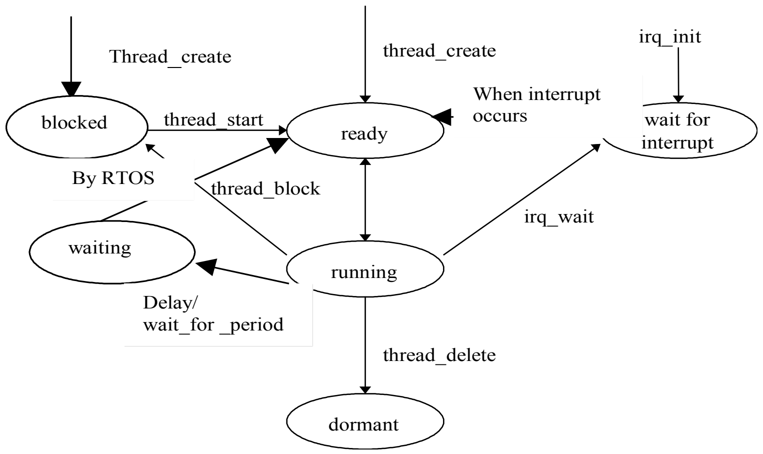
- **Maximum number of tasks:** 16

**11.03 API richness:** Symo has the smallest api of the examined rtos:es, the amount of features for task management is average (5/17) and there are no features for application modes (0/2), inter task communication (0/16), memory management (0/10) or event flags (0/9). The support for timers, is average (3/6) the same goes for clocks (2/4), interrupt handling (3/8) and synchronisation (3/14).

**11.04 Task management**: These calls are available

| | |
|---|---|
| thread_create | Creates a task, when a task is created it can either be put in the ready, blocked or suspended state |
| thread_delete | Delets a thread, it the programers responsiblity to free any resources held by the thread |
| thread_block() | Blocks the calling thread |
| thread_yield | The thread will be moved to the ready state and the new thread will begin it's execution. if there are no ready threads at the same priority as the calling thread nothing will happen. |
| thread_getinfo | Returns information (Part of the TCB) of the calling thread |

**11.05 Task states:** Task can be in six different states, the six states and the transitions between them is illustrated by the following figure from [24]



The six states are:

- **Running:** the currently executed task is in this state.

- **Blocked:** A task is placed in this state when it blocks it self or fails to take control of a mutually exclusive resource. The calls thread_block, pend_semaphore and thread_create can all cause a task to be placed in the blocked state.

- **Wait for interrupt:** A task will be placed in this state when it waiting for an interrupt. i.e. the task functions as an interrupt handler for that task. The only way for a task to be moved out of this state is when the interrupt occurs o0r if an irq_remove call is made. a task is placed in the wait for interrupt state as a result of a wait_for_interrupt call.

- **Ready:** All tasks currently waiting for the Cpu are in this state. A thread can be moved to the ready state. by the thread_create, the thread_start and the irq_remove() calls

- **Waiting:** The calls delay and wait_for_next_period, will place a task in this state.

- **Dormant:** This state is where deleted task will be placed. This is done by the thread_delete call

## 11.06 Synchronisation: Symo supports only one synchronisation mechanism, semaphores.

The following calls are available:

| | |
|---|---|
| pend_semaphore() | Attempt to take a semaphore if the semaphore isn't free the calling task will be blocked |
| release_semaphore() | Releases a held semaphore. |
| read_semaphore | Checks the counter value for a semaphore, the value of the counter = the number of task currently pending on the semaphore. |

## 11.07 Priority inversion: The Symo documentation makes no mention of any mechanism for avoiding priority inversion.

## 11.08 Memory management: Each task has its own stack, apart from that nothing is specified in the documentation.

## 11.09 Timers and clocks: 
The following calls are used to manage clocks and timers

| | |
|---|---|
| set_frequency() | Sets the frequency of the system clock |
| get_frequeny() | Gets the frequency of the system clock |
| set_time_base() | Set the resolution of the internal clock |
| get_time_base | Gets the resolution of the internal clock |
| init_period_timer() | Initialises the periodic timer for a periodic thread |
| wait_for_next_period() | Suspends the task until the current period has expired |
| stop_period | This call disables the periodic restart for a thread |
| enable_periodic_start | This call enables the periodic restarting of a thread. |
| delay() | This call suspends the executing thread for a number of ticks; the thread is moved to the waiting state until the delay expires then its move back to the ready state. Periodic threads are not allowed to use delay () |
| remove_from_timeq() | Removes a task from the delay / period queue thus activating/terminating the task. |

## 11.10 Supported standards: 
None

## 11.11 Interrupt handling: 
There are seven levels of interrupts. If two or more interrupts occur at once, the one at the highest level will be processed first. As ISR are normal task they can be interrupt by task with higher priority.

In Symo ISR are as previously seen no different from other task a task can be set as an ISR by using the irq_init call. When the interrupt occurs, the ISR task will be transferred to the ready state and run just like an ordinary task.

irq_init() specifies witch thread will be started when a specified interrupt occurs. i.e. installs an interrupt handler.

There are three calls for managing interrupts:

| | |
|---|---|
| irq_wait() | A call to this routine indicates that an ISR has finished the calling takes will be placed back in the waiting for interrupt state. |
| irq_remove | Moves a thread out of the wait for interrupt state and into the ready state. |
| irq_getinfo() | Gets information about the thread assigned to a specific interrupt level |

| | |
|---|---|

**11.12 Inter task communication:** According to the Symo documentations there are no mechanisms for inter task communication.

**11.13 Scheduling:** Symo has a fixed priority pre-emptive scheduler. A task can disable the scheduler thus making sure it wont be interrupted by using the off_tsw() call. The call on_tsw() re-enables the scheduler.

# 12 Conclusions

There are may rtos:es for embedded systems on the market, however information about them (other than sales speak) is scarce and hard to find, at the time of writing, very few papers comparing different rtos:es exists and those that do such as [2] and [25] are either work in progress or several years old.

As can be seen when the model presented in the overview section is compared to the actual sections about each rtos it has not been possible to do such a throughout report as was originally planned. This is because the information available about the rtos:es even in the cases where the complete published documentation was available did not contain all the required information. For example, the available Qnx documentation gives a good description of the architecture but says little of the interface. The eCos documentation on the other hand says much about the interface but little about the architecture.

It's not possible within the time available to write a detailed comparison between the examined rtos:es, there are just to many details. What can be done is to compare how many of the api features chosen as richness criteria that are present in each rtos. Such a comparison is presented in the overview table at the end of appendix A. A glance at that table reveals that the richness criterion seems to be well chosen. SYMO and VCB were known to be small rtos and accordingly they contain the fewest of the features selected as richness criteria. On the other hand VxWorks 5.4 is know to have a large api and accordingly has the second richest api surpassed only by eCos

# 13 Future work

It would be interesting to look at SSX5 [7]. A request for information on SSX5 was sent to Realogy but unfortunately, they choose to ignore it. The same can be said about Rubus [8]. Another system worth looking at would be QNXNeutrino [16] as both the rtos and developing environment is free for none commercial use. Unfortunately it required slightly too much memory to be include in this report, the minimum footprint is 64KB this is the only reason it was left out.

Furthermore, it would be worthwhile to examine the uITRON standard as well as the Posix 1003.1b Real-time Extension. These were left out of the report due to lack of documentation. The Posix documentation isn't available online and the latest uITRON documentation available in anything else than Japanese is for version 3.0 from 1995 and the current version is 4.0. [6]

It would also be interesting to do a performance comparison between the rtos:es. This information was originally meant to be a part of the report but since the vendors don't supply comparable figures, it had to be left out. It seems like the only way to obtain these figures is to do a time consuming benchmark of all the systems.

# 14 References

[1] osek-vdx binding dokument 1.2.2, www.osek-vdx.org

[2] Realtime-Operating systems: an ongoing review. R. Yerraballi

[3] Evaluation (dedicated-systems.com)

[4] Realtime magazine: What makes a good rtos. (dedicated-systems.com)

[5] Evaluation report definition. (dedicated-systems.com)

[6] ITRON homepage, tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html

[7] www.realogy.com

[8] www.arcticus.se

[9] Scheduling algorithms for multiprogramming in a hard real-time environment; C. Liu and J Layland. Journal of the ACM, 20(1), pages 46-61, Jan. 1973.

[10] www.etas.de

[11] Benchmarking Real-Time Operating Systems For Automotive Use; Niklas Adolfsson, Jakob Mattsson. http://www.ce.chalmers.se/undergraduate/Theses/RTBenchmark.pdf

[12] Ercos V4.1 User's Guide, www.etas.de

[13] OSEK-VDX (OS), www.osek-vdx.org

[14] OSEK-VDX (OIL), www.osek-vdx.org

[15] OSEK-VDX (COM), www.osek-vdx.org

[16] QNX hompage: www.qnx.com

[17] Rtos buyers guide. www.dedicated-systems.com

[18] An architectural Overview of qnx Dan Hildebrand. www.qnx.com

[19] WindRiver systems homepage www.wrs.com

[20] VxWorks Programmers Guide 5.4 edition 1, www.wrs.com.

[21] http://www.mrtc.mdh.se/

[22] BOOSTER RTU Hardware Functional Specification 2000-09-21

[23] The Virtual Communication Bus and SARA.

[24] Symo HW/SW Real-Time Kernel for single processor system, Larisa Rizvanovic

[25] A survey of real-time operating systems -draft Karsten Schwan et. al. 1994

[26] Misconceptions about real-time computing, J. Stankovic, IEEE Computer, 21(10), pages 10-19, Oct. 1988

[27] eCos homepage, http://sources.redhat.com/ecos

[28] eCos 1.3.1 usersguide, http://sources.redhat.com/ecos /docs.html

# Appendix A

**Thread**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Get Thread ID | | √ | √ | √ | √ | √ | √ | √ |
| Get TCB | | √ | | | | | √ | √ |
| Set TCB | | | | | | | √ | |
| Get State | | | √ | √ | √ | | | √ |
| Set State | | | | | | | | |
| Get priority | | | | √ | | √ | √ | √ |
| Set priority | | | √ | | | √ | √ | √ |
| Get stack pointer | | | | | | | | |
| Get stack size | | | | | | | | |
| Set stack size | | | | | | | | |
| Get stack address | | | | | | | | |
| Set stack address | | | | | | | | |
| Set scheduling policy | If the rtos supports more than one. | | | | | √ | | |
| Disable scheduling | Can be used as mutex | √ | | | | | | √ |
| Set max CPU usage | Puts a limit on how much of the CPU resources a thread can use. | | | | | | | |
| Suspend | Halts the thread for a predefined amount of time | √ | √ | | | √ | | √ |
| Resume | Resumes execution of a halted thread. | √ | √ | | | | | √ |

**Application modes**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Get current mode | | | | √ | √ | | | |
| Change mode | | | | √ | | | | |

**Synchronisation**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Get max counter value | | | | | | | | |
| Set max counter value | | | | | | | | |
| Set initial counter value | | | | | | | √ | √ |
| Share | Makes it possible for more than one process to hold the semaphore, Useful for synchronisation purposes | | | | | | | |
| Wait blocking | | √ | | | | √ | √ | √ |
| Wait non blocking | | | | | | | √ | √ |
| Wait with timeout | | | | | | | √ | √ |
| Release | | √ | | √ | √ | √ | √ | √ |
| Release, broadcast | Increments counter by more than '1' releasing more than one of the thread that were pending on the semaphore | | | | | | | √ |
| Get counter value | | √ | | | | | | |
| Avoid Priority inversion | | | | √ | √ | | | √ |
| Get owner ID | | | | | | | | |
| Get blocked ID | Get the ID of all threads blocked on the object | | | | | | | |
| Deletion safety | Auto release to object when the thread holding it is deleted | | | | | | | |

**Event Flags**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Set flag | | | | | √ | | √ | |
| Set multiple flags | | | | | √ | | √ | |
| Pend | Block until flag is set | | | | √ | | √ | |
| Pend multiple | Block until multiple event flags are set | | | | √ | | √ | |
| Pend OR | Pend until '1' flag is set | | | | | | √ | |
| Pend AND | Pend until all flags are set | | | | | | √ | |
| Pend AND/OR | Pend until specified AND/OR combination of flags are set | | | | | | | |
| Pend with timeout | Block until flags are set out or timeout expires. | | | | | | √ | |
| Get flags | Checks the current status of a tasks event flags | | | | √ | | | |

**Inter task communication**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Set max message size | | | | | √ | √ | | √ |
| Get max message size | | | | | | √ | | √ |
| Set queue size* | | | | | | | | √ |
| Get queue size* | | | | | | | | √ |
| Get number of messages in queue* | | | | | | | √ | √ |
| Share | Makes it possible for several processes to share the queue/mailbox | | | √[1] | √ | | | √ |
| Receive blocking | Blocks until message is available | | | | | √ | √ | √ |
| Receive non-blocking | Returns immediately if no message is available | | | | √ | √ | √ | √ |
| Receive with timeout | Blocks until message is available or timeout expires | | √ | | | | √ | √ |
| Send blocking | Blocks until message can be send | | √ | | | √ | √ | √ |
| Send non-blocking | Returns immediately with error message if queue is full | | √ | | √ | √ | √ | √ |
| Send with ACK | The sender gets an ACKnowledgement from the receiver when the message is received | | | | | | | |
| Send with priority* | Makes it possible to make messages be read in order of importance instead of just FIFO | | √ | | | | | √ |
| Send with timeout | If the queue is full the sender blocks until message can be sent or the timeout expires | | | | | | √ | √ |
| Send broadcast | Send to all threads pending on the queue/mailbox | | √ | √[1] | √ | | | |

[1] Ercos only supports mailboxes.
[1]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Timestamp | Attach a timestamp to the message | | | | | | | |

*Not applicable on Mailboxes.

**Memory**

| Feature | Comments | SYMO | VCB | Ercos/OSEK -VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Set pool size | | | | √ | | | √ | |
| Get pool size | | | | | | | √ | |
| Make new pool | | | | | | | √ | |
| Get memory block -blocking | Gets memory from pool, blocks until memory is available | | | | | | √ | |
| Get memory block -non blocking | Gets memory from pool, returns immediately if no memory is available | | | | | | √ | |
| Get memory block with timeout | Get memory from pool, blocks until memory is available or timeout expires | | | | | | √ | |
| Release memory block | | | | | | | √ | |
| Extend pool | | | | | | | √ | |
| Extend block | | | | | | | √ | |
| Get number of free bytes | | | | | | | √ | |

**Interrupt handling**

| VxWorks | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Install interrupt handler | | √ | √ | √ | √ | √ | √ | √ |
| Uninstall | | √ | √ | | | √ | √ | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| interrupt handler | | | | | | | | |
| Wait blocking | Block until specified interrupt occurs | √ | √ | | | | | |
| Wait with timeout | Block until specified interrupt occurs or timeout expires | | | | | | | |
| Raise interrupt | Raises a hardware interrupt | | | | | | | |
| Disable/enable interrupt | | | | √ | √ | | √ | √ |
| Mask/Unmask interrupt | | | | | √ | | √ | √ |
| Interrupt sharing | Allow more than on ISR to use each interrupt vector | | | | | √ | | |

**Clock**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Get time of day | | | | | | √ | √ | |
| Set time of day | | | | | | | | |
| Set resolution | | √ | √ | √ | √ | √ | √ | |
| Get resolution | | √ | √ | | | | √ | |

**Timer**

| Feature | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Expire, absolute | Set timer to expire on an absolute date | | √ | √ | √ | √ | √ | |
| Expire, relative | Set timer to expire on a relative date | √ | √ | √ | √ | √ | √ | √ |
| Expire, cyclical | Set timer to expire periodically e.g. once every sec. | √ | √ | √ | √ | √ | √ | |
| Get time remaining | Get the time remaining until timer expire | | | √ | √ | √ | | |
| Change timer value | Adds an amount of tick to the timer value | | | √ | √ | √ | √ | √ |
| Cancel timer | | √ | | √ | √ | √ | √ | |

**Overall**

| Number of features | Comments | SYMO | VCB | Ercos/OSEK-VDX | | QNX | eCos | VxWorks |
|---|---|---|---|---|---|---|---|---|
| Thread | | 5/17 | 5/17 | 3/17 | 2/17 | ?[3] | 5/17 | 8/17 |
| Application modes | | 0/2 | 0/2 | 2/2 | 1/2 | ? | 0/2 | 0/2 |
| Synchronisation | | 3/14 | 0/14 | 2/14 | 2/14 | ? | 5/14 | 7/14 |
| Event flag | | 0/9 | 0/9 | 0/9 | 5/9 | ? | 0/9 | 0/9 |
| Inter task communication | | 0/16 | 5/16 | 2/16 | 5/16 | ? | 7/16 | 13/16 |
| Memory management | | 0/10 | 0/10 | 1/10 | 0/10 | ? | 9/10 | 0/10 |
| Interrupt handling | | 3/8 | 3/8 | 2/8 | 3/8 | ? | 4/8 | 3/8 |
| Clock | | 2/4 | 2/4 | 1/4 | 1/4 | ? | 3/4 | 0/4 |
| Timer | | 3/6 | 3/6 | 6/6 | 6/6 | ? | 5/6 | 2/6 |
| Overall | | 16/84 | 18/84 | 19/84 | 25/84 | ? | 38/84 | 33/84 |

[3] Reliable information on the api richness of QNX were not available.

# Partial implementation of the eCos interface for the Sierra RTOS

Henrik Hoffström hhm00001@student.mdh.se

## Abstract

This document describes the work of adding a partial eCos [1] interface, called SECOS, to the Sierra Rtos [2]. There is also a section about how the implementation could be improved. Detailed descriptions of the finished interface are not part of this document. The user manual is [3] and the implementation is described in [4].

# Contents

## 1 Introduction

The initial purpose of the project was to add a partial eCos interface to the Symo [5] Rtos. The decision to use the eCos interface was based upon the results of [6]. However, when half of the work was complete (The design was finished but the implementation was not yet underway) a new generation of Symo called Sierra was released. The existence of Sierra, made it obvious that an eCos interface implementation for Symo would not see much use therefore it was decided to change the target platform for SECOS to Sierra. What was done was that the existing design aimed at Symo was adapted to Sierra. The rest of this document contains an overview of the design decisions for the original Symo design, the design decisions made when changing to eCos and finally a description of the problems faced when implementing the eCos design

## 2 Designing for Symo

After deciding, to use eCos as the new interface to be added to Symo the first problem was to decide what parts of the eCos interface to implement. As eCos contains about 100 system calls and Symo contains about 25, it obviously would not be possible to implement the complete eCos interface. Besides, Symo is intended for use in systems with very small amounts of ram so a full eCos interface implementation would be too large anyway. Moreover, it would not be possible as the new interface was to be constructed on top of the Symo system and the data required by some eCos calls cannot be output by Symo. Lastly, there was a limited time available for the project. After considering these factors, it was decided that it would be possible to implement roughly one third of the eCos interface.

The next problem was to decide which eCos functions to implement. All Rtos need some basic functionality; they need to: Control the scheduler, manage tasks, manage interrupts, and provide mutual exclusion. The first step was therefore to go through the eCos documentation and select all calls for scheduler control, thread management, interrupt management, semaphores and mutexes that:

1) **Would be possible to implement on top of Symo.**
2) **Would be useful in the limited implementation.**

It was found that the required functionality could be provided by including less than 30 calls. Because of this it was decided to also include, a mechanism for inter task communication. The selection fell on the Message box mechanism, as it seemed to be the simplest eCos communication mechanism to implement. With the Message boxes added, SECOS contained around 35 calls. Now, the next stage was begun.

A design was made for the implementation of each eCos call. During this process, it turned out that some of the calls planned for inclusion were not useful or not possible to implement so the final design for SECOS contained about 30 calls.

## 3 Adapting to Sierra

The change to Sierra actually simplified the design. Symo and Sierra both use a numerical ID to keep track of the tasks and semaphores. In Symo, the ID was a parameter to the create task/semaphore system call. This made it necessary to keep track of which ID belonged to which "object" and each time an eCos Create task/semaphore call were made assign a free ID and use that when calling the Symo "Creation calls". Sierra on the other hand automatically keeps track of the free ID in the system; the ID of a newly created object is delivered as output from the Sierra create task/semaphore function. In other words, the change to Sierra meant that the mechanisms and data structures used to keep track of and assign free ID to objects could be removed from the design. The result was a smaller system (Both in kb and in complexity) than originally planned. In other words, if Sierra had been available when the design was originally begun more functionality could have been added as more both more time and system resources would have been available.

## 4 Implementing the interface

Implementation was a straightforward and mostly uncomplicated process. Unfortunately, Sierra proved to be far from reliable and bug free. In fact, in the first version of Sierra obtained for this project it was impossible to make any systems calls as all system calls returned an error code regardless of input. The problem was eventually overcome by obtaining newer versions of Sierra but even in the last one obtained there was at least one bug. One of the system calls still always returned Error regardless of input. This bug meant that some of the functionality of SECOS could not be tested as it depended on the nonworking system call. There may also be more bugs as some odd behaviour were noticed during the testing of the new interface. At one time, a variable mysteriously changed value when a Sierra system call was made even though that variable was not a parameter to the call and should therefore not have been affected. Other strange behaviour from Sierra was that the itoa function, that is supposed to convert an integer to an asci string and place the resulting string in a buffer provided as a parameter to the function and finally return a pointer to the converted string, didn't place the string in the buffer. The string was instead placed at some other location in the memory and a pointer to that location was returned. However as the purpose of this project were not to debug and verify Sierra, something that would not have been possible to do any way as the Sierra source code wasn't available, these problems had to be left unsolved.

## 5 Future work

The current implementation of SECOS can probably be improved in the areas of speed and size. The focus has been on getting a working version and now when that is accomplished it should be possible to optimise it. Furthermore, support for dynamic memory is to be added to Sierra, as soon as that is done SECOS should be updated to make use of it. In addition, as soon as the above-mentioned bug in Sierra is fixed the functions that were impossible to test due to the bug must be tested. When all these changes have been done, it should prove useful to measure the worst-case execution time of the SECOS system calls.

# 6 References

[1] eCos homepage, http://sources.redhat.com/ecos

[2] http://www.idt.mdh.se/kurser/ct1550/kursinfo/sumo/doc/sierra-api.pdf

[3] User manual for the SECOS system, Henrik Hoffström

[4] Implementation of, and test procedures for, the SECOS system, Henrik Hoffström

[5] Symo HW/SW Real-Time Kernel for single processor system, Larisa Rizvanovic

[6] A survey of Real-time operating systems

# User manual for the SECOS system

Henrik Hoffström: hhm00001@student.mdh.se

## Abstract

This document describes how to use SECOS, a partial eCos [1] interface implemented on top of the Sierra Rtos [2].

# Contents

# 1 Configuring a SECOS system

SECOS is configured by including the file SECOSinterface.h in all files and then editing the skeleton function SECOSmain in the file SECOSmain.c. Before compiling, add your files to the LIB_USER_OBJS line of the Make file. As SECOS executes in the address space of the user threads it's important that all threads are given large enough stacks.

# 2 Types used when programming SECOS

Any deviation from the standard native eCos interface will be written in *italics*

### 2.1 cyg_addrword_t

A type that is large enough to store the larger of an address and a machine word. This is used for convenience when a function is passed data, which could be either a pointer to a block of data or a single word.

### 2.2 cyg_handle_t

A *handle* is a variable used to refer to **eCos** system objects (such as a thread or an alarm). Most **eCos** system calls that create system objects will return a handle that is used to access that object from then on.

### 2.3 cyg_priority_t

A numeric type used to represent the priority of a thread, or the priority of an interrupt level. A lower number means a higher (i.e. more important) priority thread.

### 2.4 cyg_vector_t

A numeric type used to identify an interrupt vector. Its value is called the interrupt vector *id.*

### 2.5 cyg_thread_entry_t

A function type for functions that are entry points for threads. It is used in the thread creation call **`cyg_thread_create()`** .

### 2.6 cyg_sem_t, cyg_mutex_t

These types are of the appropriate size to contain the memory used by their respective kernel objects. These objects are always referred to by a pointer to an object of this type.

### 2.7 cyg_interrupt, cyg_thread, cyg_mbox

These types are of the appropriate size to contain the memory used by the respective kernel objects. These types are only used in the corresponding create call

### 2.8 cyg_bool_t

A boolean type.

### 2.9 cyg_tick_count_t

*A numeric type used to count counter ticks. This is a 32 bit type.*

### 2.10 cyg_ISR_t, and cyg_DSR_t

These are function types used interrupt and delayed service routines are installed.

### 2.11 cyg_ucount32

This type is used both in eCos and in SECOS although it is not descried in the eCos documentation. In SECOS, it's implemented as signed long int.

### 2.12 A note about the types: cyg_thread, cyg_interrupt, cyg_mbox

These types are only used in the corresponding "create call", after creation the provided handle is used to reference the object. *Note, cyg_interupt, cyg_thread, and cyg_mbox are of type void in this implementation, the memory required for these objects are managed by the underlying Sierra kernel. These types are included in SECOS even though they serve no purpose, as they are part of the eCos interface. Removing them would break the compatibility with eCos making it impossible to move code written for SECOS to a real eCos system*

# 3 Systemcalls

Any deviation from the standard native eCos interface will be written in *italics*

## 3.1 Thread operations

### 3.1.1 void cyg_scheduler_start( void )

Starts the scheduler with the threads that have been created. Interrupts are not enabled until the scheduler has been started with `cyg_scheduler_start()`.

### 3.1.2 void `cyg_scheduler_lock`( void )

Locks the scheduler so that a context switch cannot occur, this can be used to protect data shared between a threads by `surrounding` the critical region with `cyg_scheduler_lock()` and `cyg_scheduler_unlock()` .

### 3.1.3 void `cyg_scheduler_unlock`( void )

Unlocks the scheduler so that context switch can occur again.

### 3.1.4 void `cyg_thread_create`(cyg_addrword_t *sched_info,* cyg_thread_entry_t **entry,* cyg_addrword_t *entry_data,* char **name,* void **stack_base,* cyg_ucount32 *stack_size,* cyg_handle_t **handle,* cyg_thread **thread )*

Creates a thread in the suspended state, the thread will not run until it has been resumed with `cyg_thread_resume()` and the scheduler has been started with `cyg_scheduler_start()` .

*Due to the limitations of Sierra, the max number of threads is at the moment 16. The number of priorities is 15 (0-14) 0 being the highest.*

The parameters of **`cyg_thread_create()`** :

*sched_info*

Information to be passed to the scheduler, this is a simple priority value

*entry*

This is the entry function of the thread. The entry function takes a single argument of type cyg_addrword_t, which is usually a pointer to a block of data.

Example entry function:

cyg_thread_entry_t thread1(cyg_addrword_t *entrydata)

{/*User code*/}

*entry_data*

A data value passed to the *entry* function. This may be either a machine word datum or the address of a block of data.

*name*

A C string with the name of this thread, as this string is not used by SECOS in any way it should be set to NULL

*stack_base*

The address of the stack base.

*stack_size*

The size of the stack for this thread.

*handle*

**`cyg_thread_create()`** returns the thread handle in this variable.

*thread.*

***The required memory of the thread, SECOS `supports a NULL value, eCos doesn't (yet).`***

### 3.1.5 void `cyg_thread_exit`( void )
Exits the current thread, at present this simply puts the thread into the suspended state.

### 3.1.6 void `cyg_thread_resume`(cyg_handle_t *thread )*
Resumes a suspended *thread,* threads are created in a suspended state and must be resumed before they will run.

### 3.1.7 void `cyg_thread_yield`( void )
Yields control to the next runable thread of equal priority. If no such thread exists, this function has no effect.

### 3.1.8 void `cyg_thread_kill`( cyg_handle_t *thread )*

Kills the thread. *Due to limitations imposed by Sierra a thread is only allowed to kill it self.*

#### 3.1.9 cyg_handle_t `cyg_thread_self`( void )
Returns the handle of the currently executing thread.

#### 3.1.10 cyg_priority_t `cyg_thread_get_priority`( cyg_handle_t *thread* )
Returns the priority of the given thread.

#### 3.1.11 void `cyg thread_delay` ( cyg_tick_count_t *delay* )
Puts the current thread to sleep for *delay* ticks. The amount of time between two ticks is 100uS. *For some reason the underlying, Sierra kernel limits the max delay to 255.*

## 3.2 Interrupt handling
*Due to Sierra, there can only be 7 interrupts. Interrupts can have priority 0-14 (NB. in the underlying Sierra kernel the interrupt service routines are executed as normal threads. It's therefore recommended that the interrupt are given higher priority than any of the "normal" threads)*

#### 3.2.1 void `cyg_interrupt_create`(cyg_vector_t *vector,* cyg_priority_t *priority,* cyg_addrword_t *data* cyg_ISR_t **isr,* cyg_DSR_t **dsr,* cyg_handle_t **handle,* cyg_interrupt **intr* )

```
(The vector is the same as the interrupt level.)
```

Creates an interrupt and returns a handle to it. The interrupt is not immediately attached; it must be attached with the `cyg_interrupt_attach()` call. As SECOS dosen't support, the full eCos interrupt model it's not possible to use a DSR, if the DSR parameter is given any other value than NULL an error will occur.

#### 3.2.2 void `cyg_interrupt_delete`(cyg_handle_t *interrupt* )
Detaches the interrupt from its vector and deletes the ISR.

#### 3.2.3 void `cyg_interrupt_attach`(cyg_handle_t *interrupt* )
Attaches the interrupt.

#### 3.2.4 void `cyg_interrupt_detach`(cyg_handle_t *interrupt* )
Detaches the interrupt from the vector. *Due to a limitation of Sierra, (Not mentioned in the Sierra documentation!) an interrupt is only allowed to detach it self.*

#### 3.2.5 void `cyg_interrupt_acknowledge`(cyg_vector_t *vector* )
*Should be used as the first instruction of an ISR to acknowledge the reception of the interrupt. The interrupt must be acknowledged! If it isn't the system will become unstable after exiting the ISR.*

## 3.3 Semaphores

The semaphores in SECOS are counting semaphores. They are not referred to by handles, but by a pointer to the variable in which the semaphore is created.

*The maximum number of semaphores + mutexes is 16 due to limitations imposed by Sierra*

### 3.3.1 void `cyg_semaphore_init`(cyg_sem_t *sem, cyg_ucount32 *val* )

Initialises a semaphore, the initial semaphore count is set to *val.*

### 3.3.2 void `cyg_semaphore_destroy`(cyg_sem_t **sem* )

Destroys a semaphore, this must not be done while there are any threads waiting on it.

### 3.3.3 void `cyg_semaphore_wait`(cyg_sem_t **sem* )

If the semaphore count is zero, the current thread will wait on the semaphore. If the count is non-zero, it will be decremented and the thread will continue running.

### 3.3.4 cyg_bool_t `cyg_semaphore_trywait`(cyg_sem_t **sem* )

A non-blocking version of `cyg_semaphore_wait()` . This attempts to decrement the semaphore count. If the count is positive, then the semaphore is decremented and *true* is returned. If the count is zero then the semaphore remains unchanged and *false* is returned, but the current thread continues to run. *Due to a bug in the current version of Sierra, this function cannot be used.*

### 3.3.5 void `cyg_semaphore_post`(cyg_sem_t **sem* )

If there are threads waiting on this semaphore, this will wake exactly one of them. Otherwise it simply increments the semaphore count.

### 3.3.6 void `cyg_semaphore_peek`(cyg_sem_t **sem,* cyg_count32 **val* )

Returns the current semaphore count in the variable pointed to by *val. Due to a bug in the current version of Sierra this function cannot be used.*

## 3.4 Mutexes

Mutexes (mutual exclusion locks) are used in a similar way to semaphores. A mutex has only two states, locked and unlocked. Mutexes are used to protect accesses to shared data or resources. When a thread locks a mutex, it becomes its owner. Only the mutex's owner may unlock it. While a mutex remains locked, the owner should not lock it again, as the behaviour is undefined. If non-owners try to lock the mutex, they will be suspended until the mutex is available again, at which point they will own the mutex.

*The maximum number of semaphores + mutexes is 16 due to Sierra*

### 3.4.1 void `cyg_mutex_init`(cyg_mutex_t **mutex* )

Initialises a mutex, it is initialised in the unlocked state.

### 3.4.2 void `cyg_mutex_destroy` (cyg_mutex_t **mutex* )

Destroys a mutex, a mutex should not be destroyed in the locked state, as the behaviour is undefined.

#### 3.4.3 cyg_bool_t `cyg_mutex_lock`( cyg_mutex_t **mutex )*
Changes the mutex from the unlocked state to the locked state, when this happens the mutex becomes owned by the current thread. If the mutex is locked, the current thread will wait until the mutex becomes unlocked before performing this operation.

#### 3.4.4 void `cyg_mutex_unlock`(cyg_mutex_t **mutex )*
Changes the mutex from the locked state to the unlocked state, this function may only be called by the thread that locked the mutex and should not be called on an unlocked mutex. *Due to a bug in the current version of Sierra, this function cannot be used.*

## 3.5 Message boxes
Message boxes are a primitive mechanism for exchanging messages between threads, inspired by the $\mu$ITRON specification [3]. A message box can be created with **`cyg_mbox_create()`** before the scheduler is started, and two threads in a typical producer/consumer relationship can access it. One thread, the producer, will use **`cyg_mbox_tryput()`** to make data available to the consumer thread which uses **`cyg_mbox_tryget()`** to access the data.

The size of the internal message queue is configured by the *CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE* parameter *in SECOSinterface.h*. The default value is 5.

*The maximum number of message boxes in the system is specified with the NOF_MBOXES parameter in SECOSinterface.h (Default is 1), This is due to the implementation of SECOS, in the real eCos there is no limit to the number of message boxes.*

*The maximum size of a message is specified in the MAX_MESSAGE_SIZE parameter in SECOSinterface.h. (Default is four bytes) This parameter is a part of this interface implementation and not a part of eCos.*

#### 3.5.1 void `cyg_mbox_create`(cyg_handle_t **handle,* cyg_mbox **mbox )*
Creates a message box using the space provided in the *mbox* parameter, and returns a handle for future access to that message box.

#### 3.5.2 void `cyg_mbox_delete`(cyg_handle_t *mbox )*
Deletes the message box.

#### 3.5.3 void *`cyg_mbox_tryget`(cyg_handle_t *mbox )*
Checks to see if a message is ready, if no message is available it returns immediately with a return value of *NULL.* If a message is available, it retrieves it and returns the address of the data. The data stored in the address returned by this call can be altered at any time after the call completes. It's therefore recommended that the message box is protected by some sort of mutual exclusion mechanism such as a semaphore, mutex or complete locking of the scheduler.

#### 3.5.4 void *`cyg_mbox_peek_item`(cyg_handle_t *mbox )*
Checks to see if a message is ready, and if one is available returns the address of the data *without* removing the message from the queue. If no message is available, it returns *NULL. This call*

*should only be used when there is no chance of the calling function being pre-empted or interrupted. Else, the returned result may be wrong.*

### 3.5.5 cyg_bool_t `cyg_mbox_tryput`(cyg_handle_t *mbox*,void **item* )

Tries to put a message in the given message box. It returns *true* if the message was successfully sent, and *false* if the message could not be sent immediately, usually because the queue was full.

### 3.5.6 cyg_count32 `cyg_mbox_peek`(cyg_handle_t *mbox* )

Peeks at the queue and returns the number of messages waiting in it.

# 4 Error handling

If a call is made using erroneous parameters, a system error will occur causing the SECOS error handler will be called. The default behaviour of the error handler is to lock the scheduler, print an error code indicating which call caused the error and then go into an infinite loop. The default behaviour can be changed by editing the file sierra_error.c. However, it's not recommended that the error handler be allowed to exit as the system is in an undefined state after the occurrence of an error.

## 4.1 Default Error codes

| | |
|---|---|
| 1 | cyg_thread_create |
| 2 | cyg_thread_exit |
| 3 | cyg_semaphore_init |
| 4 | cyg_semaphore_destroy |
| 5 | cyg_semaphore_wait |
| 6 | cyg_semaphore_post |
| 7 | cyg_semaphore_peek |
| 8 | cyg_mbox_create |
| 9 | cyg_interrupt_create |
| 10 | cyg_interrupt_delete |
| 11 | cyg_interrupt_attach |
| 12 | cyg_mbox_peek |
| 13 | cyg_interrupt_detach |
| 14 | cyg_interrupt_acknowledege |
| 15 | cyg_mbox_tryget |
| 16 | cyg_thread_delay |
| 17 | cyg_mbox_peek_item |
| 18 | cyg_mbox_try_put |
| 19 | cyg_thread_kill. |

# 5 References

[1] http://www.redhat.com/eCos

[2] http://www.idt.mdh.se/kurser/ct1550/kursinfo/sumo/doc/sierra-api.pdf

[3] ITRON homepage, tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html

# Implementation of, and test procedures for, the SECOS system

Henrik Hoffström hhm00001@student.mdh.se

# Contents

# 1 Introduction
SECOS is implemented as a number of C functions on top of the Sierra OS [1]. This document contains descriptions of how those functions are implemented and how they were tested. SECOS function calls runs as an ordinary application function calls on the Sierra system. In other words they are executed on the calling threads stack.

# 2 Structure
The SECOS system consists of the following files placed in the src directory. The names are self-explanatory except for as noted.

**SECOSscheduler.c, SECOSthreads.c, SECOSsemaphores.c, SECOSmutexes.c, SECOSmailboxes.c, SECOSinterrupts.c, SECOSdefinitions.c, SECOStypes.h, SECOSmain.h, SECOSdefinitions.h, Makefile.**

**sierraerror.c:** This file contains a function that handle the errors returned by the Sierra functions that SECOS are built upon it also handles some error conditions of SECOS itself.

**SECOSdefinitions.h:** Contains definitions of the macros and global variables used by SECOS.

**sierrainterface.h:** Contains code use to make all the Sierra functions visible. Caveat currently this doesn't work as supposed. The Sierra functions are visible regardless if this file is included or not. This is probably due to how the make file is constructed.

**SECOSinterface.h:** Contains prototypes for all SECOS functions.

**SECOSmain.c:** This file contains the skeleton function were the application code is to be placed

**main.c:** This is the file containing the Sierra main function, that calls SECOSmain, and the idle task used by SECOS, initialisation of the SECOS data structures are also done in this file.

# 3 Types
Here is the implementation of the required eCos types. Not that several types that in a real eCos system are used to store "objects" are here simple integer types used to store the Sierra id of the object. As an "object" id is, what is used in Sierra to refer to "objects" This deception is invisible to the application programmer as long as the limits of the SECOS system are adhered to.

```
typedef int cyg_addrword_t;
typedef void cyg_thread_entry_t;
typedef unsigned long int cyg_ucount32;
typedef int cyg_handle_t;
typedef int cyg_vector_t;
typedef int cyg_priority_t;
typedef int cyg_sem_t;
typedef int cyg_mutex_t;
typedef int cyg_mbox;
typedef void cyg_thread;
typedef signed long int cyg_tick_count_t;
typedef void cyg_ISR_t;
typedef void cyg_DSR_t;
typedef void cyg_interrupt;
typedef short cyg_bool_t;
```

# 4 Definitions and globals

Here are the default definitions and global variables that are used. (This is an excerpt from the file SECOSdefinitions.h)

```
#define INTERRUPT_STACK_SIZE 256
#define NOF_MESSAGEBOXES 2
#define MAX_MESSAGE_SIZE 4
#define CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE 10
#define thread_create 1
#define thread_exit 2
#define semaphore_init 3
#define semaphore_destroy 4
#define semaphore_wait 5
#define semaphore_post 6
#define semaphore_peek 7
#define mbox_create 8
#define interrupt_create 9
#define interrupt_delete 10
#define interrupt_attach 11
#define mbox_peek 12
#define interrupt_detach 13
#define interrupt_acknowledege 14
#define mbox_tryget 15
#define thread_delay 16
#define mbox_peek_item 17
#define mbox_tryput 18
#define thread_kill 19

extern struct MB {
          short free,empty,queue_front,queue_rear;
          char queue[CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE][MAX_MESSAGE_SIZE];
          }mbox_list[];

extern short schedlock;
extern short schedstarted;
extern struct ID {
          short priority;
          short vector;
          } id_table[];
```

# 5 Functions

## 5.1 Template

The SECOS functions are described according to the following template

**Description**
A description of what the function does and how it does it.

**Source code**
The source code of the function

**Test**
How the function was tested, many functions have only the word trivial under this heading since the in data of the function is correct and if all the functions called by the function produces the correct result, a single test run is all that is necessary to verify correct operation.

## 5.2 main

**Description**
This is the main were Sierra user code is placed. It is included in this document as SECOS is nothing but a Sierra application. The function initialises the SECOS data structures and installs an idle task. Finally, it calls eCosmain.

**Source code**

```
int main (void)
{
int x,y,z;
char idlestack[256];
task_create(150,TASK_READY,myIdleTask,NULL,idlestack,256);
set_idle_task(myIdleTask,NULL);
/*initiate messageboxes*/
for(x=0;x<NOF_MESSAGEBOXES; x++) {
           mbox_list[x].queue_front=0;
           mbox_list[x].queue_rear=0;
           mbox_list[x].free=TRUE;
           mbox_list[x].empty=TRUE;
           for(y=0;y<CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE; y++) {
                      for(z=0;z<MAX_MESSAGE_SIZE;z++)
                                 mbox_list[x].queue[y][z]=0;
           }
}
eCosmain();
return OK;
}
```

**Testing**
The initialised data structures were checked so that they contain the correct values, this was done using the debugger. The rest of the testing of this function is trivial.

## 5.3 myIdleTask

**Description**
Sierra requests that an Idle task is present.

**Source code**
```
void myIdleTask(void *data)
{
while(1);
}
```

**Testing**
Trivial.

## 5.4 SECOSmain

**Description**
This is a skeleton function for the application programmer to complete.

**Source code**
```
void eCosmain(void)
{
/*User application code goes here*/
}
```

**Testing**
Nothing to test.

## 5.5 SIERRA_ERROR

**Description**
This function handles unexpected errors, for example if a Sierra function called by a SECOS function returns error. There is one case for each possible error. At default, the function only prints an error code on the terminal and then goes into an infinite loop. It's allowed for the user to change this function if a different kind of error handling is needed. Note that this function is not a part of eCos or a part of Sierra. It is a native SECOS function.

**Source_code**

```
void SIERRA_ERROR(int error)
{
int temp;
csw_off();
switch (error) {

case thread_create:
```

```
                print("ERROR 1\r\n");
                break;

        case thread_exit:
                print("ERROR 2\r\n");
                break;

        case semaphore_init:
                print("ERROR 3\r\n");
                break;
        case semaphore_destroy:
                print("ERROR 4\r\n");
                break;
        case semaphore_wait:
                print("ERROR 5\r\n");
                break;
        case semaphore_post:
                print("ERROR 6\r\n");
                break;
        case semaphore_peek:
                print("ERROR 7\r\n");
                break;
        case mbox_create:
                print("ERROR 8\r\n");
                break;
        case interrupt_create:
                print("ERROR 9\r\n");
                break;
        case interrupt_delete:
                print("ERROR 10\r\n");
                break;
        case interrupt_attach:
                print("ERROR 11\r\n");
                break;
        case mbox_peek:
                print("ERROR 12\r\n")
                break;
        case interrupt_detach:
                print("ERROR 13\r\n");
                break;
        case interrupt_acknowledge:
                print("ERROR 14\r\n");
                break;
        case mbox_tryget:
                print("ERROR 15\r\n");
                break;
        case thread_delay:
                print("ERROR 16\r\n");
                break;
        case mbox_peek_item:
                print("ERROR 17\r\n");
                break;
        case mbox_tryput:
                print("ERROR 18\r\n");
                break;
        case thread_kill:
                print("ERROR 19\r\n");
```

```
            break;
default:
            print("EJECT EJECT EJECT\r\n");
            break;
            }
while(1)
            temp=0;
}
```

**Testing**
Trivial.

## 5.6 cyg_scheduler_start

**Description**
This function changes the value of a global variable to indicate that the scheduler has been started.

**Source code**

```
void cyg_scheduler_start(void)
{
schedstarted=1;
}
```

**Testing**
Trivial.

## 5.7 cyg_scheduler_lock

**Description**
If the scheduler has been started this function disables context switch and increases the variable used to keep track of how many times the scheduler has been locked. If the scheduler has not been started, this function does nothing.

**Source code**
```
void cyg_scheduler_lock(void)
{
if(schedstarted==1) {
            schedlock++;
            csw_off();
            }
}
```

**Testing.**
Trivial.

## 5.8 cyg_scheduler_unlock

**Description**
If the scheduler is started this function decreases the lock variable, if the lock variable goes below 1 it set to 0 i.e. unlocked, and context switch is re-enabled.

**Source code**

```
void cyg_scheduler_unlock(void)
{
if (schedstarted==1) {
           schedlock--;
           if(schedlock<1) {
                      schedlock=0;
                      csw_on();
                      }
           }
}
```

**Testing**
Trivial.

## 5.9 cyg_thread_create

**Description**
Creates a new thread by mapping the in-parameters to a call to task_create. Then it places some information about the newly created thread in id_table using the task id of the newly created task as index.  If task creation fails, the error handling function is called.

**Source code**

```
void cyg_thread_create(cyg_addrword_t sched_info, cyg_thread_entry_t *entry,
cyg_addrword_t entry_data, char *name, void *stack_base, cyg_ucount32
stack_size, cyg_handle_t *handle, cyg_thread *thread)
{
int index;
void (*f)(void*);
cyg_scheduler_lock();
f=entry;
index=task_create((int) sched_info,TASK_BLOCKED,f, (void *) entry_data,
stack_base,(int)stack_size);
if(index==ERROR)
           SIERRA_ERROR(thread_create);
else {
           id_table[index].priority=sched_info;
           *handle=index;
           cyg_scheduler_unlock();
           }
}
```

**Testing**
Trivial.

## 5.10 cyg_thread_exit

**Description**
Block the currently executing thread by calling task_block

**Source code**
```
void cyg_thread_exit(void)
{
task_block();
}
```

**Testing**
Trivial.

## 5.11 cyg_thread_resume

**Description**
Unblocks a thread.

**Source code**
```
cyg_thread_resume(cyg_handle_t thread);
{
task_unblock (*thread);
}
```

**Testing**
Trivial.

## 5.12 cyg_thread_kill

**Description**
As tasks in Sierra only are allowed to kill them self, this introduces a limitation to the SECOS interface not present in the real eCos. This function checks if the task specified by the parameter to the function is the currently executing task. If it is, the task is deleted, if not the error handler is called.

**Source code**
```
void cyg_thread_kill(cyg_handle_t thread)
{
if(thread!=self)
           SIERRA_ERROR(thread_kill);
task_delete();
}
```

**Testing**

Trivial.

## 5.13 cyg_thread_self

**Decription**
This function returns the Task id of the currently executing task; in Sierra, this is stored in the self variable.

**Source code**
```
cyg_handle_t cyg_thread_self()
{
return self;
}
```

**Testing**
Trivial.

## 5.14 cyg_thread_yield

**Description**
Makes the currently executing task yield the CPU to another task of the same priority.

**Source Code**
```
void cyg_thread_yield()
{
task_yield();
}
```

**Testing**
Trivial.

## 5.15 cyg_thread_getpriority

**Description**
This function returns the priority of the currently executing task, task priority is not possible to get from the Sierra kernel which is why it's stored in id_table by cyg_thread_create.

**Source Code**
```
cyg_priority_t cyg_thread_get_priority(cyg_handle_t thread)
{
return id_table[thread].priority;
}
```

**Testing**
Trivial.

## 5.16 cyg_thread_delay

**Description.**
Delays the currently executing task.

```
void cyg_thread_delay(cyg_tick_count_t delay)
{
int test;
test=task_delay(delay);
if(test!=OK)
           SIERRA_ERROR(thread_delay);
}
```

**Testing**
Trivial.

## 5.17 cyg_semaphore_init

**Description**
This function use semaphore_create to create a semaphore. The semaphore id returned by semaphore_create is stored in the cyg_sem_t variable pointed to by sem. If the semaphore_create call fails, the error handler is called.

**Source code**
```
void cyg_semaphore_init(cyg_sem_t *sem, cyg_ucount32 val)
{
cyg_scheduler_lock();
*sem=semaphore_create(val);
if(*sem==ERROR)
           SIERRA_ERROR(semaphore_init);
cyg_scheduler_unlock();
}
```

**Testing**
Trivial

## 5.18 cyg_semaphore_destroy

**Description**
Destroys a semaphore, the variable pointed to by the sem pointer is set to –1. (i.e. an illegal semaphore ID) If the delete semaphore_call fails, the error handler is called.

**Source code**
```
void cyg_semaphore_destroy (cyg_sem_t *sem)
{
cyg_scheduler_lock();
if(semaphore_delete(*sem)==ERROR)
```

```
           SIERRA_ERROR(semaphore_destroy);
*sem=-1;
cyg_scheduler_unlock();
}
```

**Testing**
Trivial.

## 5.19 cyg_semaphore_wait

**Description**
Calls semaphore_take, if the semaphore_take call fails, the error handler is called.

**Source code**
```
void cyg_semaphore_wait(cyg_sem_t *sem)
{
if(semaphore_take(*sem)==ERROR)
           SIERRA_ERROR(seamphore_wait);
}
```

**Testing**
Trivial.

## 5.20 cyg_semaphore_peek

**Description**
Reads the value of a semaphore using semaphore read. If the call fails, the error handler is called. Due to a bug in Sierra, (semphore_read always returns -1) this function cannot be used at present.

**Source code**

```
void cyg_semaphore_peek(cyg_sem_t *sem, cyg_ucount32 *val)
{
cyg_scheduler_lock();
*val=semaphore_read(*sem);
if(*val==ERROR)
           SIERRA_ERROR(semaphore_peek);
cyg_scheduler_unlock();
}
```

**Testing**
Trivial.

## 5.21 cyg_semaphore_trywait

**Description**

A nonblocking version of cyg_semaphore_wait, the function calls cyg_semaphore_peek to check if the semaphore is free (count >0), if it is the cyg_semaphore_wait is called and the function returns TRUE. If the semaphore isn't free i.e. count=0; the function simply returns FALSE. Due to a bug in Sierra, (semphore_read always returns -1) this function cannot be used at present.

**Source code**

```
cyg_bool_t cyg_semaphore_trywait(cyg_sem_t *sem)
{
cyg_ucount32 count;
cyg_scheduler_lock();
cyg_semaphore_peek(sem,&count);
if(count<1) {
           cyg_scheduler_unlock();
           return FALSE;
           }
else {
           cyg_semaphore_wait(sem);
           cyg_scheduler_unlock();
           return TRUE;
           }
}
```

**Testing**
Trivial.

## 5.22 cyg_semaphore_post

**Description**
Calls semaphore_release, if the call fails the error handler is called.

**Source code**

```
void cyg_semaphore_post(cyg_sem_t *sem)
{
if(semaphore_release(*sem)==ERROR)
           SIERRA_ERROR(seamphore_release);
}
```

**Testing**
Trivial.

## 5.23 cyg_mutex_init

**Description**
SECOS mutexes are implemented on top of Sierra semaphores. Therefore, this function simply calls cyg_semaphore_init who then creates a semaphore with a count value of one.

**Source code**
```
void cyg_mutex_init(cyg_mutex_t *mutex)
{
cyg_semaphore_init(mutex, 1);
}
```

**Testing**
Trivial.

## 5.24 cyg_mutex_destroy

**Description**
SECOS mutexes are implemented on top of Sierra semaphores. Therefore, this function simply calls cyg_semaphore_destroy who then destroys the semaphore.

**Source code**
```
void cyg_mutex_destroy(cyg_mutex_t *mutex)
{
cyg_semaphore_destroy(mutex);
}
```

**Testing**
Trivial

## 5.25 cyg_mutex_lock

**Description**
Locks a mutex, as the eCos functions used to break an eCos thread from the wait is not implemented in SECOS this function will always return TRUE.

**Source code**
```
cyg_bool_t cyg_mutex_lock(cyg_mutex_t *mutex)
{
cyg_semaphore_wait(mutex);
return TRUE;
}
```

**Testing**
Trivial.

## 5.26 cyg_mutex_unlock

**Description**

As only one thread may lock a mutex, the count of the semaphore the mutex is implemented on cannot be allowed to go higher than one. This function checks the value of the semaphore count and, if it is 0, calls cyg_semaphore_post in effect unlocking the mutex. Due to a bug in Sierra, a function cyg_semaphore read depends on give erroneous results this function cannot be used at present.

**Source code**

```
void cyg_mutex_unlock(cyg_mutex_t *mutex)
{
cyg_ucount32 val;
cyg_scheduler_lock();
cyg_semaphore_peek(mutex, &val);
if(val==0) {
           cyg_semaphore_post(mutex);
           cyg_scheduler_unlock();
           return;
           }
else
           cyg_scheduler_unlock();
           return;
}
```

**Testing**
Trivial

## 5.27 cyg_interrupt_create

**Description**
SECOS doesn't fully support the eCos interrupt model, it is not possible to use a deferred service routine. Any attempt to do so i.e. if the dsr parameter is anything but NULL will result in an error As interrupts in Sierra are in all respects normal tasks this function calls cyg_thread_create to create a new thread then the interrupt vector assigned to the interrupt is stored in id_table.

The parameters to this function are used in the following way: vector is used to pass the irq the interrupt thread is to attached to, priority contains the priority the interrupt thread will have, data contains the address to the data that will be available to the interrupt function when it begins its execution, isr contains a pointer to the entry function of the interrupt thread, the dsr parameter isn't used for anything, the handle variable is a pointer to were the handle of the interrupt thread will be placed once it is created, finally the intr parameter is used to pass the address of the stack of the interrupt task. All interrupt threads have a stack of size INTERRUPT_STACK_SIZE this is necessary as none of the parameters to cyg_interrupt_create can be used to pass information about stack size.

**Source code**

```
void cyg_interrupt_create(cyg_vector_t vector, cyg_priority_t
priority,cyg_addrword_t data, cyg_ISR_t *isr, cyg_DSR_t *dsr, cyg_handle_t
*handle, cyg_interrupt *intr)
```

```
{
cyg_scheduler_lock();
if(dsr!=NULL)
          SIERRA_ERROR(interrupt_create);
cyg_thread_create(priority,isr, data, NULL,
intr,INTERRUPT_STACK_SIZE,handle,NULL);
id_table[*handle].vector=vector;
cyg_scheduler_unlock();
}
```

**Testing**
Trivial.

## 5.28 cyg_interrupt_delete

**Decription**
The effect of this function is that the interrupt task is deleted; as Sierra tasks only are allowed to delete them self, this function can only be called by the interrupt task itself.

**Source code**

```
void cyg_interrupt_delete(cyg_handle_t interrupt)
{
if(irq_remove(id_table[interrupt].vector)==ERROR)
          SIERRA_ERROR(interrupt_delete);
cyg_thread_kill(interrupt);
}
```

**Testing**
Trivial.

## 5.29 cyg_interrupt_attach

### ***Description***
This function first initialises the thread used as an interrupt handler to an irq by using irq_init and then activates the task by using task unblock (eCos tasks are placed in the blocked state when created.)

**Source code**

```
void cyg_interrupt_attach(cyg_handle_t interrupt)
{
if(irq_init(interrupt,id_table[interrupt].vector)==ERROR)
          SIERRA_ERROR(interrupt_attach);
if(task_unblock(interrupt)==ERROR)
          SIERRA_ERROR(interrupt_attach);
}
```

**Testing**
Trivial.

## 5.30 cyg_interrupt_detach

**Description**
Removes an interrupt task by calling irq_remove to move the task out of the wait for interrupt state.

**Source code**

```
void cyg_interrupt_detach(cyg_handle_t interrupt)
{
if(irq_remove(id_table[interrupt].vector)==ERROR)
           SIERRA_ERROR(interrupt_detach);
}
```

**Testing**
Trivial.

## 5.31 cyg_interrupt_acknowledge

**Description**
Another deception, in eCos cyg_interrupt_acknowledge must be called from inside an interrupt service routine in order to make sure that the handler is not run more than once for each occurrence of the interrupt. In Sierra, it's necessary to call irq_wait in order to make the thread handling the interrupt wait for the next occurrence of the interrupt. As SECOS requires that this function is called as the first statement of the main loop of the interrupt handling thread. (A limitation not present in a real eCos system.) It's possible to map a call to cyg_interrupt_acknowledge to a call to irq_wait

**Source code**

```
void cyg_interrupt_acknowledge(cyg_vector_t vector)
{
if(irq_wait(vector)==ERROR)
           SIERRA_ERROR(interrupt_acknowledge);
}
```

**Testing**
Trivial.

## 5.32 cyg_mbox_create

**Description**
As the SECOS message boxes are stored in global variables (dynamic memory management would be more effective but the current version of Sierra doesn't support it) this function checks

if there is a free message box available, if there is the status of that message box are change to not free then the id of the message box is placed in the memory pointed to by handle

**Source code**
```
void cyg_mbox_create(cyg_handle_t *handle, cyg_mbox *mbox)
{
int index=0;
cyg_scheduler_lock();
while(mbox_list[index].free==FALSE) {
           index++;
           if(index==NOF_MESSAGEBOXES)
                      SIERRA_ERROR(mbox_create);
           }
mbox_list[index].free=FALSE;
*handle=index;
cyg_scheduler_unlock();
}
```

**Testing**
Trivial.

## 5.33 cyg_mbox_delete

**Description**
This function marks a message box as free.

**Source code**

```
void cyg_mbox_delete(cyg_handle_t mbox)
{
cyg_scheduler_lock();
mbox_list[mbox].free=TRUE;
mbox_list[mbox].empty=TRUE;
mbox_list[mbox].queue_front=0;
mbox_list[mbox].queue_rear=0;
cyg_scheduler_unlock();
}
```

**Testing**
Trivial.

## 5.34 cyg_mbox_tryget

### ***Description***
The function checks to see if the message box is properly initialised, if it isn't the error handler is called.

1. The function checks if the message box is empty, if it is NULL are returned.

2. If the message box isn't empty, a temporary pointer is set to point to the first message in the message queue.

3. The queue is a circular queue implemented in an array so if the queue_front variable reaches the value of the maximum number of messages in the queue the variable is set to 0.

4. The variable keeping track of the index of the front of the queue is advanced

5. A check is made to see if the queue is empty. (In this case the queue is empty when queue front and queue rear have the same value) if so the empty variable is changed to True.

6. The temporary pointer is returned

**Source code**

```
void *cyg_mbox_tryget(cyg_handle_t mbox)
{
void *temp;
cyg_scheduler_lock();
if(mbox_list[mbox].free==TRUE)
           SIERRA_ERROR(mbox_tryget);
if(mbox_list[mbox].empty==TRUE) {
           cyg_scheduler_unlock();
           return NULL;
           }
else {
           temp=mbox_list[mbox].queue[mbox_list[mbox].queue_front];
                                                   mbox_list[mbox].queue_fr
ont++;
           if(mbox_list[mbox].queue_front==CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE
           )
                      mbox_list[mbox].queue_front=0;
           if(mbox_list[mbox].queue_front==mbox_list[mbox].queue_rear)
                      mbox_list[mbox].empty=TRUE;
           cyg_scheduler_unlock();
           return temp;
           }
}
```

**Testing**

The message box was filled with messages, then the first was removed and a check was made to make sure that this was the first message put in the message box. After that, a further message was added to the message box (This message will be placed at index 0) Then CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE +1 messages were removed. The result of that operation was that all the messages were returned in the order they were inserted in the queue including the last one verifying that queue front is set to 0 when it reaches the value CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE. Then another message was removed from the queue. The result of this operation was NULL, which was the desired result as the queue was empty.

## 5.35 cyg_mbox_peek_item

**Description**
This function returns a pointer to the first message in the message queue without removing the message from the queue. If the message box is empty, NULL is returned.

**Source code**

```
void *cyg_mbox_peek_item(cyg_handle_t mbox)
{
void *temp;
cyg_scheduler_lock();
if(mbox_list[mbox].free==TRUE)
           SIERRA_ERROR(mbox_peek_item);
if(mbox_list[mbox].empty==TRUE) {
           cyg_scheduler_unlock();
           return NULL;
           }
else {
           temp=mbox_list[mbox].queue[mbox_list[mbox].queue_front];         cy
g_scheduler_unlock();
           return temp;
           }
}
```

**Testing**
The function was first used on an empty message box, which correctly resulted in a return result of NULL. Then a message was put in the message box and this function was called twice. Both time identical pointers were returned.

## 5.36 cyg_mbox_tryput

**Description**
This function puts a message in the message box if the message box isn't full.

1. Checks if the message box is created if it isn't the error handler is called

2. Checks if the message box is full, a message box is full if it's not empty and queue_front and queue_rear are at the same index, if the message box is full, NULL is returned.

3. If the message box isn't full a number of bytes corresponding to the MAX_MESSAGE_SIZE are copied to the message queue, starting with the byte pointed out by the in parameter item. Note that MAX_MESSAGE_SIZE bytes are always copied regardless of the length of the message.

4. The status of the message box is set to not empty.

5. queue_rear is advanced one position as this is a circular queue implemented in an array- queue_rear is set to 0 when it reaches the value CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE.

6. TRUE is returned

**Source code**

```
cyg_bool_t cyg_mbox_tryput(cyg_handle_t  mbox, void *item)
{
int index;
char *message=(char*) item;
cyg_scheduler_lock();
if(mbox_list[mbox].free==TRUE)
           SIERRA_ERROR(mbox_tryput);
/*If the message box is full return FALSE*/
if((mbox_list[mbox].empty==FALSE&&(mbox_list[mbox].queue_front==mbox_list[mbox
].queue_rear))){
           cyg_scheduler_unlock();
           return FALSE;
           }
/*If there is a slot left for the message copy the message to the messagebox*/
else {
           for(index=0; index<MAX_MESSAGE_SIZE; index++) {
                      mbox_list[mbox].queue[mbox_list[mbox].queue_rear]
                      [index]=message[index];
                      }
           mbox_list[mbox].empty=FALSE;
           mbox_list[mbox].queue_rear++;
/*Circular queue implemented in array*/
           if(mbox_list[mbox].queue_rear==CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE)
                      mbox_list[mbox].queue_rear=0;
           cyg_scheduler_unlock();
           return TRUE;
           }
}
```

**Testing**
As the test of cyg_mbox_tryget was successful, it was obviously possible to put messages in the queue. What needed to be verified was that NULL is returned when an attempt to put a message in a full queue was made. The test was successful

## 5.37 cyg_mbox_peek

**Description**
This function returns the number of messages in the message box

**Source code**
```
cyg_ucount32 cyg_mbox_peek(cyg_handle_t mbox)
{
unsigned int number;
cyg_scheduler_lock();
if(mbox_list[mbox].free==TRUE)
           SIERRA_ERROR(mbox_peek);
/*Calculate number of messages in queue*/
if(mbox_list[mbox].queue_rear>=mbox_list[mbox].queue_front){
           number=mbox_list[mbox].queue_rear-mbox_list[mbox].queue_front;
           if(number==0) {
                      if(mbox_list[mbox].empty==FALSE)
                                 number=CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE;
                      }
}
else
           number=CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE-mbox_list[mbox].queue_fr
ont+mbox_list[mbox].queue_rear;
cyg_scheduler_unlock();
return number;
}
```

**Testing**
First the function was tested with an empty queue the result was 0

Then the function was tested with a full queue (i.e. queue_front=queue_rear and empty =FALSE). The result was CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE as was expected.

To test the situation were queue_rear >= queue_front 1 message was put in the queue. Then this function was called the result was as expected 1

To test the situation were queue_rear<queue_front CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE messages were put in the queue. The first message was removed and a new message added, (queue rear is now at index 0 and queue front is now at index 1.) then this function was called. The result was as expected 1.

# References

[1] http://www.idt.mdh.se/kurser/ct1550/kursinfo/sumo/doc/sierra-api.pdf

[2] http://www.redhat.com/eCos

# Appendix A

## Test case 1

```
#include"SECOSinterface.h"

/*This code tests the functions cyg_mbox_create, cyg_mbox_delete,
cyg_mbox_peek, cyg_mbox_peek_item, cyg_mbox_tryget,cyg_mbox_tryput.
The default message box queue size of 5 is assumed for this test*/

/* The expected out put of the test is
Messagebox is full
1
2
3
4
5
Messagebox is empty
ERROR 15
*/

char testmessages[6][4]={{"1\r\n\0"},{"2\r\n\0"},{"3\r\n\0"},{"4\r\n\0"},{"5\r\n\0"},{"6\r\n\0"}};

void MTest (void) {
int index;
cyg_handle_t mbox_handle;
cyg_mbox mbox;

/*Code used to compensate for broken display :)*/
print("\n\n\n\n\n");
/*Create a messagebox*/
cyg_mbox_create(&mbox_handle,&mbox);

/*put message 6,1-4, in the messagebox*/
cyg_mbox_tryput(mbox_handle,(void *) testmessages[5]);
for(index=0;indexqueue_front. The result should be 1*if(cyg_mbox_peek(mbox_handle)!=1)
                print("ERROR\r\n");

/*Test cyg_mbox peek on full queue i.e. queue_rear=queue_front and empty=false the result
should be 5*/
for(index=0;index queue rear ,the expected result should still be 5 as one message is removed and
one message is added*/

cyg_mbox_tryget(mbox_handle);
cyg_mbox_tryput(mbox_handle,(void *) testmessages[0]);

if(cyg_mbox_peek(mbox_handle)!=5)
                print("ERROR\r\n");
/*Finally cyg_mbox_delete is tested*/

cyg_mbox_delete(mbox_handle);
/*Trying to remove something from an none existent messagebox should cause the error handler
to be called resulting in an out put of ERROR 15*/
cyg_mbox_tryget(mbox_handle);
}
```

## Test case 2

```
#include "sierrainterface.h"
#include "SECOSinterface.h"

/*This code is used to verify the correctness of:

cyg_thread_create
cyg_thread_resume
cyg_thread_delay
cyg_thread_get priority
cyg_thread_suspend
cyg_interrupt_create
cyg_interrupt_attach
cyg_interrupt_delet and implicitly cyg_thread_kill as that function is called as a part of
cyg_interrupt_delete.
cyg_semaphore_init
cyg_semaphore_wait
cyg_semaphore_post
cyg_semaphore_try_wait (and implicitly cyg_semaphore_peek) <-- In theory due to a bugg in
Sierra it currently cannot be tested
cyg_semaphore_destroy
cyg_mutex_init
cyg_mutex_lock
cyg_mutex_unlock <-- Not really due to a bug in Sierra cyg_semaphore_post has to be used to
emulate this function.

cyg_scheduler_lock
cyg_scheduler_unlock
cyg_interrupt_acknowledge
cyg_interrupt_detach
and implicitly cyg_scheduler start (As the system wouldn't run if this call didn't work as
supposed)

This code was constructed by starting from the test file ex1.c and replacing all the Sierra calls
with their eCos counterparts. The example were extended to test some more functions
*/
```

```
/*Note any output from any task verifies the correctness of cyg_thread_create and,
cyg_thread_resume and SECOSmain. A fault in any of those functions would result in no output
at all*/

#define THREAD_PRIO1 2
#define THREAD_PRIO2 3
#define IRQ_THREAD_PRIO  0
#define TIME_THREAD_PRIO 4
#define YIELD_THREAD_PRIO 5
#define IRQ_THREAD_PRIO2 1
#define IRQ_LEVEL  1
#define IRQ_LEVEL2  2
#define THREAD_STACK_SIZE 256

char thread_stack1[THREAD_STACK_SIZE];
char thread_stack2[THREAD_STACK_SIZE];
char irq_thread_stack[THREAD_STACK_SIZE];
char time_thread_stack[THREAD_STACK_SIZE];
char yield_thread_stack[THREAD_STACK_SIZE];
char irq_thread_stack2[THREAD_STACK_SIZE];
char yield_thread_stack2[THREAD_STACK_SIZE];

/* VT100 Escape sequence to clear a terminal screen. */
static char clrscr[] =
  { 0x1B, '[', '2', 'J', 0x1B, '[', '1', ';', '1', 'H', 0 };

/* CR LF to get a newline, use \r\n in ordinary strings */
static char crlf[] = { 13, 10, 0 };

cyg_mutex_t mutex1;

/**
 * Thread that competes for a shared semaphore.
 *
 * This is a thread that competes with #thread2()# for a shared
 * semaphore, sem1 in #ex1()#. The semaphore ID is transferred to
 * #thread1()# by temporarly casting the #int# value to a #void *#
 * pointer at thread creation. This all because all threads in RealFastOS
 * must have the same style prototype.

 * @param arg Semaphore ID disguised as a pointer.
 * @see thread2()

This function together with thread2 verifies cyg_semaphore_create, cyg_semaphore_twait,
cyg_semaphore_lock, cyg_mutext_init, cyg_mutex_lock and thread delay. The expected out put
is that thread1 should not run about twice as often as thread 2, rather it should run about equally
as often. This is because about half of the times thread1 wants to run thread 2 will hold the
semaphore. (Provided the semaphore management functions and cyg_thread delay are correctly
implemented. The use of a mutex is to guarantee that the output wont be garbled. One some, but
not all terminals the print function can be interrupted. An interrupted print functions is proof of
an error in the mutex managing functions
*/


cyg_thread_entry_t thread1 (void *arg)
{
 cyg_sem_t semID = (cyg_sem_t)arg;  /* Convert the semID back to an integer */

 while (42)
   {
     cyg_mutex_lock(&mutex1);
/*      print (" THREAD1, cyg_semaphore_trywait()\n\r");*/
               print (" THREAD1, cyg_semaphore_wait()\n\r");
     cyg_semaphore_post(&mutex1);
/*      if(cyg_semaphore_trywait(&semID)) {*/
               cyg_semaphore_wait(&semID);
               cyg_thread_delay(250);
               cyg_thread_delay(250);
      cyg_semaphore_post(&semID);
      cyg_mutex_lock(&mutex1);
/*             print(" THREAD1 got the semaphore\r\n");*/
      print (" THREAD1 cyg_semaphore_post ()\n\r");
cyg_semaphore_post(&mutex1);
/*                              }*/
/*      else
               cyg_thread_delay(125);*/


               }
}




/**
 * Thread that competes for a shared semaphore.
 *
 * This is another thread that competes with #thread1()# for a shared
 * semaphore, sem1 in #ex1()#. The semaphore ID is transferred to #thread1()#
 * by temporarily casting the #int# value to a #void *# pointer at thread creation.
 * This all because all threads in RealFastOS must have the same style prototype.
 *
 * @param arg Semaphore ID disguised as a pointer.
 * @see thread2()
 */

cyg_thread_entry_t thread2 (void *arg)
```

```
{
  cyg_sem_t semID = (cyg_sem_t)arg; /* Convert the semID back to an integer */

  while (42)
    {

      cyg_mutex_lock(&mutex1);
      print ("THREAD2; new period, cyg_semaphore_wait()\n\r");
      cyg_semaphore_post(&mutex1);
      cyg_semaphore_wait(&semID);
      cyg_thread_delay(250);
      cyg_thread_delay(250);
      cyg_thread_delay(250);
      cyg_thread_delay(250);
      cyg_semaphore_post(&semID);
      cyg_mutex_lock(&mutex1);
      print ("cyg_semaphore_post (thread no 2)\n\r");
      cyg_semaphore_post(&mutex1);
      }
}

/*Output from this function after an interrupt has occurred verifies cyg_interrupt_create and
cyg_interrupt attach. If those functions were not correctly implemented we wouldn't get any
output. The absence of output when no interrupt has occurred verifies
cyg_interrupt_acknowledge, if that function were incorrectly implemented there would be output
regardless of if the interrupt had occurred or not When the interrupt occurs for the second time
cyg_interrupt delete is called deleting the interrupt, therefore this interrupt can only be called
twice calling it more than twice results in an error. The occurrence of the error verifies
cyg_interrupt_delete*/

cyg_thread_entry_t irq_thread (void *arg)
{
  int value;
  int my_irq = (int)arg;
  char buffer[40];
  static int access=0;

  while (42)
    {
      cyg_interrupt_acknowledge (my_irq);
      cyg_mutex_lock(&mutex1);
      print ("\n\rISR 1\n\r");
      cyg_semaphore_post(&mutex1);
      if(access==0) {
                                cyg_mutex_lock(&mutex1);
                                print("One time left\r\n");
                                cyg_semaphore_post(&mutex1);
                                access++;
                                }
     else {
                                print("ISR 1 is NO MORE !\r\n");
                                cyg_interrupt_delete(cyg_thread_self());
                                }
   }
}
```

/*Runs once and then detaches itself, the fact that this interrupt causes an error if it occurs more than once verifies cyg_interrupt_detach*/

```
cyg_thread_entry_t irq_thread2 (void *arg)
{
int threadid3=(int) arg;
while(1) {
                cyg_interrupt_acknowledge(IRQ_LEVEL2);
                print("\r\nISR 2\r\n");
                cyg_interrupt_detach(self);
                }
}
```

/*This taks print it's priority every time its run, if the out put is TIME_THREAD_PRIO cyg_thread_get priority must be correct. The first 50 times this thread is run it locks the scheduler only to immediately unlock it. the 50 time it doesn't unlock the scheduler causing the system to halt upon the completion of the threads execution. This behaviour verifies cyg_scheduler_lock and cyg_scheduler unlock. An error in cyg_scheduler_lock would prevent the system from halting after 50 executions of this thread; an error in cyg_scheduler_unlock would casuse the system to halt after only 1 execution of this thread. When the system has halted cyg_semaphore_destroy is called, after that the value of semID is printed if cyg_semphore_destroy is correct this should be -1 */

```
cyg_thread_entry_t time_thread (void *arg)
{
static short lifetime=0;
char buf[40];
cyg_sem_t semID = (cyg_sem_t)arg; /* Convert the semID back to an integer */
  while (42)
    {
     cyg_thread_delay (200);
     cyg_thread_delay (200);
     cyg_thread_delay (200);
     cyg_thread_delay (200);
     cyg_mutex_lock(&mutex1);
     print ("\n\rTIME_THREAD1 prio ");
     print (itoa(cyg_thread_get_priority(cyg_thread_self()),buf,10));
     print ("\r\n");
     cyg_semaphore_post(&mutex1);
     cyg_scheduler_lock();
```

```
                  if (lifetime2)
                  cyg_thread_exit();
    }
}

cyg_thread_entry_t yield_thread2 (void *arg)
{
static int count=0;
  while (42)
    {
    print ("YIELDThread2\r\n");
    count++;
    cyg_thread_yield();
    if(count>2)
                  cyg_thread_exit();
    }
}


/**
 * Initiates the thread stacks in this example.
 * Sets all positions in all stacks to #value#
 * @param value Value to write to all positions in stack memory
 */

void init_thread_stacks (char value)
{
  int i;

  for (i = 0; i < THREAD_STACK_SIZE; i++)
    thread_stack1[i] = value;
  for (i = 0; i < THREAD_STACK_SIZE; i++)
    thread_stack2[i] = value;
  for (i = 0; i < THREAD_STACK_SIZE; i++)
    irq_thread_stack[i] = value;
  for (i = 0; i < THREAD_STACK_SIZE; i++)
    time_thread_stack[i] = value;
  for (i = 0; i < THREAD_STACK_SIZE; i++)
    yield_thread_stack[i] = value;
 for (i = 0; i < THREAD_STACK_SIZE; i++)
    irq_thread_stack2[i] = value;
for (i = 0; i < THREAD_STACK_SIZE; i++)
    yield_thread_stack2[i] = value;
}


/*This function creates and initialises the threads, semaphores, interrupts and mutexes */
```

```
int test(void)
{
 cyg_sem_t sem1;  /* Used by thread1 and thread2 */

 cyg_handle_t threadid1;
 cyg_handle_t threadid2;
 cyg_handle_t threadid3;
 cyg_handle_t threadid4;
 cyg_handle_t threadid5;
 cyg_handle_t threadid6;
 cyg_handle_t threadid7;

 print (clrscr);
 print ("Example #1\n\r");

 /* Initiate all stacks to contain 0xEE. This helps if we want to locate
  * stack overflows due to too small stacks for threads.
  */
 init_thread_stacks (0xEE);

 /* Start by creating all the shared resources. Semaphores, shared memory, etc. */

 /* Create a binary semaphore to be used by thread1 & thread2 */
 cyg_semaphore_init(&sem1,1);
 if (ERROR == sem1)
   {
    print ("sem1 error\n\r");

    while (42)
      {
       ;
      }
   }
 else
   {
    print ("\n\rsem1 OK\n\r");
    print ("Creating semaphore threads ...\r\n");
              cyg_thread_create(THREAD_PRIO1,thread1,(void *) sem1,NULL,
                            thread_stack1, THREAD_STACK_SIZE,&threadid1,NULL);
    if (ERROR == threadid1)
              {
               print ("Could not create thread1\n\r");
               print ("Aborting semaphore & periodic thread test.\n\r");
              }
    else
              {

                cyg_thread_create(THREAD_PRIO2,thread2,(void *)sem1,NULL,
                                thread_stack2, THREAD_STACK_SIZE,&threadid2,NULL);

                if (ERROR == threadid2)
                  {
                   print ("Could not create thread2\n\r");
                   print ("Only thread1 will \"compete\" for sem1 ... \n\r");
                  }
               }
   }

   cyg_interrupt_create(IRQ_LEVEL,IRQ_THREAD_PRIO,(void *)
IRQ_LEVEL,irq_thread,NULL,&threadid3,irq_thread_stack);


   cyg_thread_create(TIME_THREAD_PRIO,time_thread,(void *) sem1,NULL,
                                time_thread_stack, THREAD_STACK_SIZE,&threadid4,NULL);


   cyg_thread_create(YIELD_THREAD_PRIO,yield_thread,NULL,NULL,
                                yield_thread_stack, THREAD_STACK_SIZE,&threadid5,NULL);


cyg_interrupt_create(IRQ_LEVEL2,IRQ_THREAD_PRIO2,(void *)
threadid3,irq_thread2,NULL,&threadid6,irq_thread_stack2);

cyg_thread_create(YIELD_THREAD_PRIO,yield_thread2,NULL,NULL,
                                yield_thread_stack2,
THREAD_STACK_SIZE,&threadid7,NULL);

  cyg_thread_resume (threadid1);
  cyg_thread_resume (threadid2);
  cyg_interrupt_attach(threadid3);
  cyg_thread_resume (threadid4);
  cyg_thread_resume (threadid5);
  cyg_interrupt_attach(threadid6);
  cyg_thread_resume(threadid7);
  cyg_mutex_init(&mutex1);
  return OK;
}
```