10/10/2020

NexOS User Manual

Table of Contents

1.0 Introduction

The goal of this document is to provide an overview and understanding of the NexOS. The kernel, each mandatory and optional module, kernel tasks, and callbacks will be explained. For a brief guide on how to get a project with the NexOS up and running refer to the document NexOS Quick Start Guide. This document will go into the NexOS in more detail as compared to the NexOS Quick Start Guide. This document does not go over the individual methods available in the NexOS and what they perform. For that information please refer to the corresponding header file which contains the documentation on each method.

1.1 Release Notes

September 7, 2020 – Pre release.
October 10, 2020 – Original Release.

1.2 License

Below is the license associated with use the of NexOS.

2.0 Source Code Overview

The current version of the NexOS at the time of this document is v1.00.00. This is the original release of the NexOS to the outside world. The NexOS is comprised of mandatory modules and optional modules.

2.1 Mandatory Source Code Files

The below is a list of the mandatory source code files required to be present to compile the NexOS and a brief description of each.

- **Kernel.c/h:** The logic for the CPU scheduler, OS initialization and OS helper functions.
- **KernelTasks.c/h:** Idle Task, Maintenance Task, and other OS task code.
- **Memory.c/h:** OS heap management and monitoring.
- **ContexSwitch.S:** Swapping the context of a task and starting the first task (port specific).
- **CriticalSection.c/h:** Enables and disables interrupts at OS_PRIORITY.
- **Port.c/h:** Functions for abstracting the OS timer and task stack initialization (port specific).
- **Task.c/h:** Task manipulation (resume, suspend, restart, delete, delay, etc.).
- **DoubleLinkedList.c/h:** This is a simple library for manipulating a double linked list. Each TASK has at least 1 DOUBLE_LINKED_LIST_NODE that the kernel uses to keep a handle on the TASK.
- **TaskObject.h:** This holds the data structure for a TASK.
- **RTOSConfig.h:** This has all the configurations for the OS that the user must define.

2.2 Optional Source Code Files

Below is a list of all the optional modules which can be used in the NexOS. These do not have to be present to compile the NexOS.

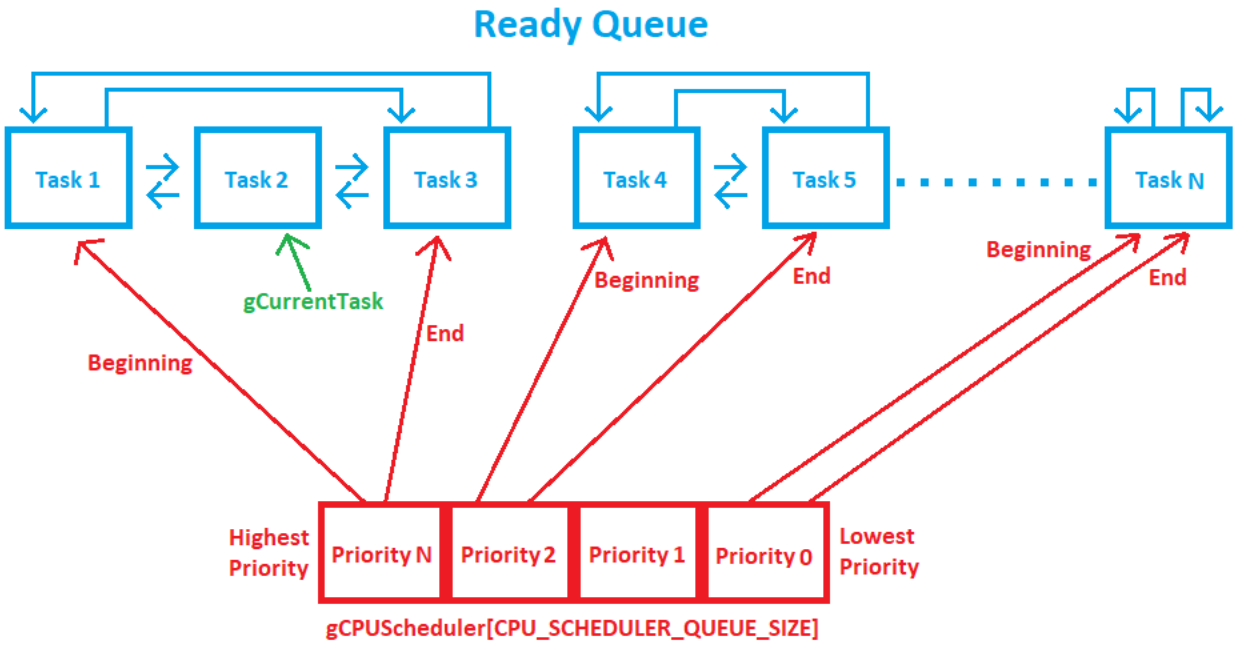
- **Event.c/h:** This allows hardware and user events to be raised. A task can block pending an event to be raised.
- **Interrupt.S:** This is a list of interrupt handlers that dispatch the interrupt to the appropriate method in InterruptHandlers.c
- **InterruptHandler.c/h:** This holds the C code which is called when an interrupt occurs. This area will raise the appropriate events and callback functions will be called.
- **MessageQueue.c/h:** This allows tasks or ISR functions to write/read data in an organized fashion to/from a queue. The queue uses the OS heap to grow in size.
- **OS_Callbacks.c/h:** This holds callbacks that are called periodically by the NexOS.
- **OS_EventCallbacks.c/h:** This holds callbacks that are called when a particular event is raised.
- **Pipe.c/h:** This allows tasks or ISR functions to write/read data in an organized fashion to/from a fixed sized buffer.
- **OS_BinarySemaphore.c/h:** This is the base object that all semaphores use.
- **BinarySemaphore.c/h:** This is a simple semaphore that is used for synchronization.

- **CountingSemaphore.c/h:** This is the same as a binary semaphore but it can be obtained multiple times. To release it, the semaphore must be released the same amount of times it was obtained.
- **Mutex.c/h:** This is the same as a binary semaphore with one exception. The task which owns the mutex will take on a priority of any task waiting for the mutex which has a higher priority.
- **SoftwareTimer.c/h:** This allows the passing of OS ticks to be tabulated. This is also the base object for all other timers.
- **CallbackTimer.c/h:** This periodically calls a user specified method at a user specified periodicity.
- **EventTimer.c/h:** This allows the number of OS ticks which have elapsed between events to be monitored without the use of a task.

3.0 Kernel

3.1 CPU Scheduler

The NexOS CPU scheduler runs by one policy and one policy only. The CPU scheduler will choose the highest priority task that is ready to run. If there are multiple tasks of the same highest priority all those priority tasks will be run in a round robin fashion. The time to swap from one task to another of the same priority is a fixed time. This is due to the design of the CPU scheduler where it assembles ready tasks as a list of list based off of priority. The following is a diagram which shows how the CPU scheduler is constructed.



Notes
Priority N has 3 tasks present in it and they will be run in a round robin fashion.
Priority 2 has 2 tasks present in it.
Priority 1 has no tasks present in it.
Priority 0 has 1 task present in it. This would have to be the Idle task.
The current task pointer need only iterate to find the next task to run.
Tasks of the same priority are interconnected with the DOUBLE_LINKED_LIST_NODE data structure.
The gCPUScheduler[] is an array of type DOUBLE_LINKED_LIST_HEAD.

Each task gets one OS tick in time to execute before the OS timer interrupt will fire and the CPU scheduler will check which task to run next. The OS tick is determined by the define OS_TICK_RATE_IN_HZ inside of RTOSConfig.h. For example, if this define is 1000 each OS tick will be 1 millisecond in length.

If there are no tasks eligible to run the Idle task will run. The CPU scheduler runs on the assumption that there is always one task ready to run. For this reason the Idle task callback function cannot call any method which may potentially block it.

3.2 Memory Management

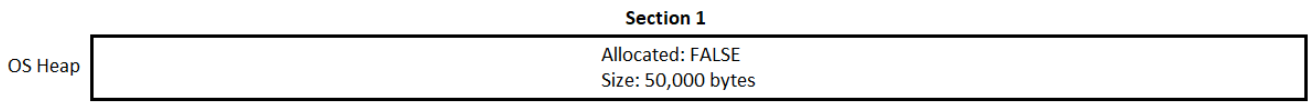
The OS heap is controlled with the files Memory.c/h. The size of the OS heap is determined by the define OS_HEAP_SIZE_IN_BYTES inside of RTOSConfig.h. The OS declares an array which holds OS_HEAP_SIZE_IN_BYTES plus OS_MEMORY_BLOCK_HEADER_SIZE_IN_BYTES. This the area that the OS uses to allocate and free memory through the calls AllocateMemory() and ReleaseMemory().

The OS heap is managed through using headers which declare how large their section is and if it is allocated. At startup there is only one section and it is unallocated. The largest size heap due to this allocation scheme is 2^{31} bytes large. As the user requests space from the OS heap the sections will be partitioned into smaller sections and marked as allocated. When a user releases memory back to the OS heap the OS will also join adjacent free sections to the current section being released.

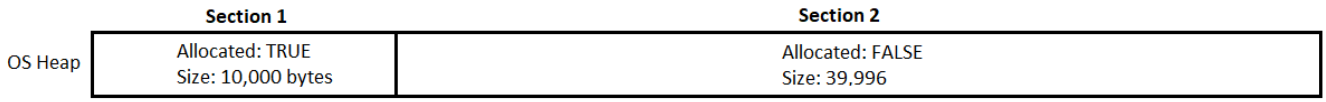
Memory is allocated on a first fit basis. This means that when the user requests memory from the OS heap the OS will use the first block which is big enough to allocate the requested space. The request for space is also rounded up to the nearest OS_WORD alignment in bytes. That is to say if the OS_WORD is 4 bytes large and the user asks for 13 bytes, the request will be rounded up to 16 bytes.

The following is an example of the user allocating and releasing memory and how the OS heap looks during the process. At the end of the example the OS heap is back to full size due to the OS joining adjacent empty sections when the user calls ReleaseMemory().

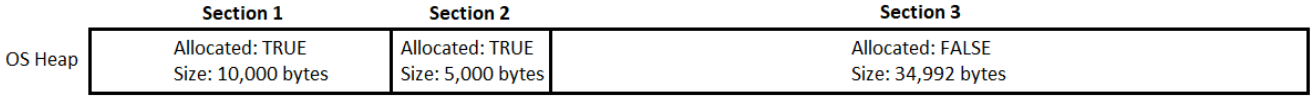
Step 1: At startup the OS Heap is all one section which is unallocated.



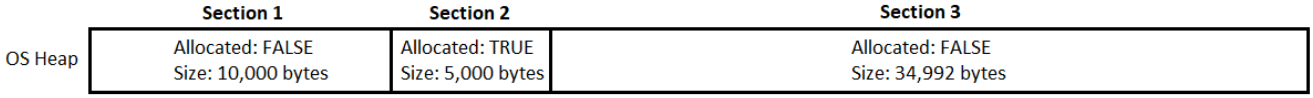
Step 2: The user asks for 10,000 bytes from the OS heap by calling the method AllocateMemory(10000). You will notice section 2 does not have the full remainder of the 40,000 bytes that seemingly should have been left over. This is because 4 bytes of the remainder are used for the memory block header.



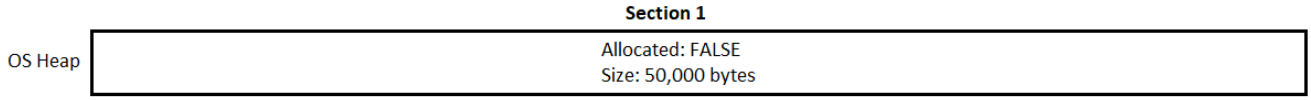
Step 3: The user asks for 5,000 bytes from the OS heap by calling the method AllocateMemory(5000).



Step 4: The user releases the 10,000 byte section back with a call to ReleaseMemory().



Step 5: The user releases the 5,000 byte section back a with a call to ReleaseMemory(). Since section 1 and 3 were empty they are joined with section 2.



Unless malloc and free are also going to be used, the heap size in the linker configuration of the project should be set to zero. Setting the heap size in the linker configuration has nothing to do with the OS heap and will only needlessly use up RAM.

3.3 Kernel Tasks

As of now there are a total of two kernel tasks. These are the Idle task, and the Maintenance task. The Idle task is always present in the NexOS. This is so that the scheduler can be designed with the assumption that one task is always available to run. For this reason the IdleTaskUserCallback cannot call any method which may block the Idle task. The Idle task creation parameters can be found in RTOSConfig.h if they are required to be modified. By default the Idle task has the lowest priority in the OS and will only run when no other task is ready to run.

The Maintenance task is present under certain configurations. If the restart task or delete task are enabled in RTOSConfig.h, the Maintenance Task will be present in the OS. The maintenance task is setup to run at a user specified amount of time to collect any tasks which need to be deleted or restarted. This specified time is in RTOSConfig.h and is the define as MAINTENANCE_TASK_DELAY_TICKS. All other configuration parameters for the Maintenance task can be found in RTOSConfig.h. By default the Maintenance task priority is HIGHEST_USER_TASK_PRIORITY. What this ensures is that when the delay for MAINTENANCE_TASK_DELAY_TICKS is finished, the Maintenance task will more than likely run right after being placed in the ready queue.

If you do not want to add the Maintenance task to the OS, but still need the ability to delete and restart tasks, the define IDLE_TASK_PERFORM_DELETE_TASK can be used. This will configure the system to use the Idle task to delete and restart tasks. However, since the idle task priority by default is zero, it is potentially not going to run as often or predictably as the Maintenance task.

4.0 Tasks

Tasks hold all the information necessary for executing code. Each task has its own priority, state, stack and other associated information for proper execution. The smallest size a task can be on a 32-bit wide architecture is 32 bytes plus the size of the tasks stack. The task with the highest priority in the ready queue is what will be executed by the CPU. The OS enforces no limit to the number of tasks a user can create as long as there is enough RAM to create the task. As a task is manipulated its state will change to reflect this. Below is the enum for different task states.

enum TASK_STATE

READY – This is state means the task is ready for execution and is in the ready queue.

SUSPENDED – This state means that the task is in the suspended queue and will not execute.

BLOCKED – This state means that the task is blocked waiting for an event or resource to become available. While blocked the task will not execute.

RESTARTING – This state means that the task has been marked for restarting but the maintenance task has yet to actually restart the task.

DELETING – This state means that the task has been marked for deletion but the maintenance task has yet to actually delete the task.

HIBERNATING – This state means that the task is in hibernation mode. This is essentially an alternate version of the suspended state. While in hibernation the task will not execute.

Some of the functionality that the OS offers for tasks are the following.

- Suspending and resuming a task
- Hibernating and waking a task
- Delaying a task for a specified number of OS ticks
- Deleting a task
- Calling a custom task exit method when it is deleted
- Restarting a task
- Individually configurable software watchdog timer for each task
- Task priority setting and getting

- Task names
- Blocking on signals waiting to be set
- Local storage pointers

The methods for using these features are all outlined in the file Task.h.

5.0 Semaphores

Semaphores are used to synchronize resources and tasks. The NexOS offers various semaphore functionality to control access to resources. The following sections go over these modules and how to use them.

5.1 OS Semaphores

The OS semaphore is the base object for all semaphore types in the NexOS. The files OS_BinarySemaphore.c and OS_BinarySemaphore.h are required to use any other semaphore type in the NexOS. The user should not call any method in the OS_BinarySemaphore.c/h files.

If there are constantly multiple tasks on waiting on a semaphore the lowest priority task can be starved of ownership of the semaphore. For this scenario the NexOS provides the option of starvation prevention for semaphores. Take the following example of starvation protect being used. There are 3 tasks with the following priorities going after the same semaphore.

- Task 1 Priority 3
- Task 2 Priority 3
- Task 3 Priority 1

Task 1 obtains the semaphore and then task 2 and 3 try to get the semaphore but end up on the waiting list of the semaphore because task 1 owns it. Once task 1 gives up the semaphore it will go to the task on the semaphore waiting list with the highest priority. In this case that is task 2 with priority 3. No task 3 will be bumped up to a priority of 2 because it was starved of the semaphore from Task 2. The next event to happen is task 1 goes for the semaphore again but since task 2 currently owns it, task 1 goes on the semaphore waiting list. Now task 2 releases the semaphore. The semaphore will go to the highest priority task in the waiting list. In this case it would be task 1 with a priority of 3. Since task 3 go passed over again for the semaphore its priority will be increased to 3. Now once task 1 gives up the semaphore task 3 will get it. Once task 3 gets the semaphore its priority returns back to normal which is 1.

This feature prevents a lower priority task from being starved of the semaphore by other higher priority tasks.

5.2 Binary Semaphores

A binary semaphore is the most simple semaphore. Once a task calls the method to obtain the semaphore it does so until it calls the method to release the semaphore. This can be used to control access to a resource or critical section. Multiple calls to own a semaphore cause no side effects or errors. If a task owns a binary semaphore any task that tires to get it goes on the semaphore waiting

list. Once the binary semaphore is released by the owner it goes to the task on the waiting list with the highest priority.

A binary semaphore can be dynamically created and deleted. When this is done the OS heap is used to allocate memory for the semaphore. To delete the binary semaphore and reclaim the memory in the OS heap the semaphore must not be in use. That is to say, if a semaphore needs to be deleted no task can currently own the semaphore.

5.2.1 Binary Semaphore Requirements

To enable the use of counting semaphores the files BinarySemaphore.c/h, and OS_BinarySemaphore.c/h must be included in the project.

5.3 Counting Semaphores

A counting semaphore is very similar to a binary semaphore with one exception. The counting semaphore can be obtained by the same task multiple times. In order to release a counting semaphore the owning task must release the counting semaphore as many times as it obtained the counting semaphore. For example, if a task obtains the counting semaphore 5 times it must release the counting semaphore 5 times to release total ownership of the counting semaphore. This allows behavior allows a task to recursively obtain a counting semaphore and release it without having to keep track of how many times it actually obtained it. The count of a counting semaphore must not exceed 2^{32}. If the counting semaphore is obtained 2^{32} times, then the next time it is obtained the count will rollover to zero.

5.3.1 Counting Semaphore Requirements

To enable the use of counting semaphores the files CountingSemaphore.c/h, and OS_BinarySemaphore.c/h must be included in the project.

5.4 Mutexes

A mutex is very similar to a binary semaphore with one exception. The owning tasks priority can be dynamically changed depending on tasks waiting for the mutex. If a task with a higher priority than the owner is on the waiting list of the mutex, the priority of the mutex owner gets bumped up to the higher priority. For example if a task with priority 1 owns a mutex and a task with priority 5 and another task with priority 3 try to get the mutex they will go on the waiting list of the mutex. At this time the mutex owner will be bumped up to a priority of 5 while it owns the mutex. Once the mutex owner task releases the mutex its priority will return back to normal and be a 1 and the task with the highest priority will inherit the mutex.

5.4.1 Mutex Requirements

To enable the use of mutexes the files Mutex.c/h, and OS_BinarySemaphore.c/h must be included in the project. The define USING_MUTEXES inside of RTOSConfig.h must be set to a 1 to use mutexes.

6.0 Pipes

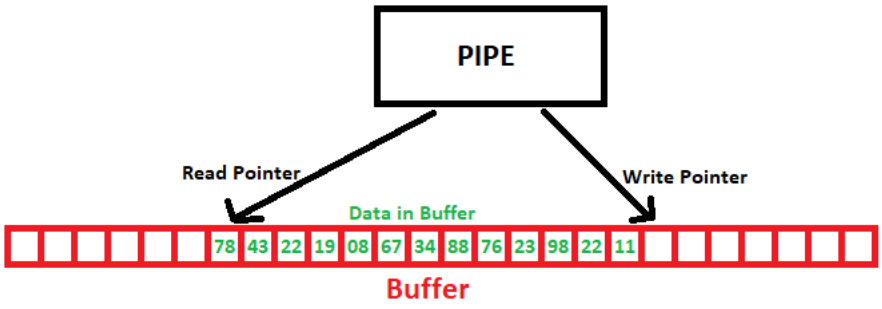
Pipes allow multiple tasks and or ISR functions to read/write data to/from a pipe. The buffer size of a pipe is of fixed size and determined at creation time. The user can specify that the pipe have its buffer allocated in the OS heap or statically allocated and passed in during creation. Pipes also have the ability to use starvation protection for a task to access them.

6.0.1 Pipe Requirements

To enable the use of pipes the files Pipe.c/h must be included in the project.

6.1 Pipe Operation

When data is written to or read from a pipe it is done so on a byte to byte basis and of no fixed size. For instance, one task could write 20 bytes to a pipe while another task reads 10 bytes from the pipe twice. The pipe maintains a set of read and write pointers to the buffer for read and write operations (see below).



The user should not worry about the alignment of data in a pipe or if it wraps around from the end of the buffer to the beginning. The data being transferred through a pipe should always be seen as a stream of bytes.

There is no set relationship between how many tasks can read/write data to/from a pipe. That is to say data being exchanged through a pipe can be done so in a one to one, many to one, one to many, or many to many relationship between tasks and ISR functions. A task can be blocked when reading or writing to a pipe. If the pipes buffer is full and a task writes to the pipe, the task will block until bytes are read from the pipe. At this point the task will be placed back in the ready queue of the CPU scheduler. Once the task runs it will write as many bytes as is permitted based on the empty space in the buffer. That is to say if the task had to write 20 bytes but only 10 bytes were available in the buffer, 10 bytes would be written to the pipe and then the task would be blocked. For this reason partial messages can be interrupted by other messages that are written to by higher priority tasks. It is the users responsibility to maintain atomicity when reading and writing data to a pipe. To ensure this either

a one to one relationship exists with the tasks and a pipe, or messages are of fixed size. For variable length messages please reference the message queue in section 7.0 Message Queues.

6.2 Starvation Protection

Pipes as a whole can be setup at compile time to have a starvation protection mechanism enabled in them. To do this the option USING_PIPE_STARVATION_PROTECTION inside of RTOSConfig.h must be set to a 1. This allows tasks which are trying to read from an empty pipe or write to a full pipe the ability to do so if they keep getting passed over by a higher priority task reading/writing to a pipe. Each time a task gets passed over its priority is increased by 1. Once the task finished is operation with the pipe its priority returns back to whatever it was prior to reading/writing to the pipe. While this doesn't solve the problem of a task getting starved of the resource, the task has a higher chance of getting access to the resource.

7.0 Message Queues

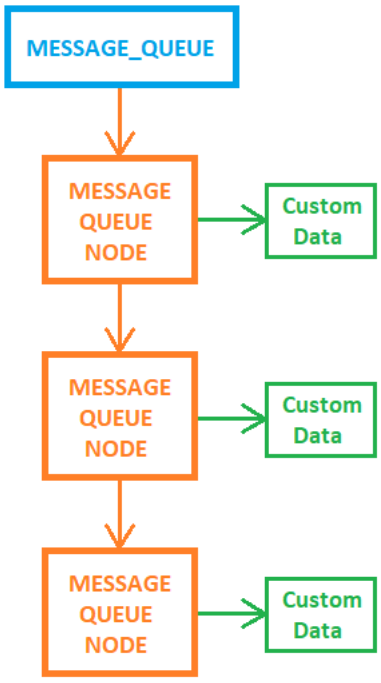
A message queue can be used to communicate data between tasks and/or ISR functions. A message queue relies on the OS heap to allocate space for the data. For a fixed size buffer that can be used to communicate data between tasks and/or ISR functions refer to section 6.0 Pipes.

Message queues were designed with the intent of having a producer consumer model of many to one or one to one, but this is not strictly enforced. It could be used in a one to many or a many to many relationship but this is highly inadvisable. The message queue handles the data being written/read to/from it in a FIFO fashion.

A task can block on an empty message queue to wait for data. However, the OS will only allow one task to block on an empty message queue at a time. If another task tries to read data from the message queue it will return back with an error code indicating that the message queue is in use.

Message queues have the ability to store simple UINT32 data in each message, and/or a void *pointer of data. This allows for a flexible mechanism to transfer data. The data that the void *pointer points to can be of variable size. For this reason a message queue can have a custom free method which the consumer may call to release allocated memory back to the OS heap. If the consumer is an ISR special care must be taken with the custom free method being called from the ISR.

The message queue consists of a head which points to nodes. Each node points to a set of data and the next node in the queue.



Each time the user adds data to the queue a new node is also allocated in the OS heap. Each time a message is read the node pointing to the data is released back into the OS heap. It is up to the user to release any memory back to the OS heap if it was allocated in the OS heap prior to being written to the message queue.

7.1 Message Queue Requirements

To enable the use of message queues the files MessageQueue.c/h must be included in the project.

8.0 Events

Events are mostly hardware related events tied to interrupts. For this reason they are also heavily dependent upon the level of support of the port of the NexOS. A task can block while waiting for an event to occur. The OS has a global array of size NUMBER_OF_EVENTS – 1. When a task blocks on an event it gets placed in this array at the appropriate spot for the event being blocked on. When an event is triggered the OS looks at the corresponding spot in the array and any task located at that index will be placed into the ready queue.

Each event also has a user callback which can be called when the event is triggered. For more information on user callbacks see section 10. User Callbacks. Events can also be used to start or stop event timers. This is useful for timing the number of OS ticks between events without a task monitoring the events. For more information on event timers refer to section 9.3 Event Timer.

A task blocking on an event affords the user the ability to not have to constantly monitor a hardware stimulus with a task.

8.0.1 Event Requirements

To enable the use of an event the files Event.c/h, Interrupt.S, InterruptHandler.c, OS_EventCallbacks.c, and PortInterruptHandler.h must be included in the project. USING_EVENTS must also be defined as a 1 in RTOSConfig.h. To enable a specific event it must be set to a 1 in RTOSConfig.h. This file also lists all events available for the NexOS.

8.1 Event List

The following is a list of all supported events with the NexOS. To enable the event set the corresponding define in RTOSConfig.h to a 1.

enum EVENT

USING_EXT_INT_x_EVENT – This will enable the external interrupt x event. Where x is a value 0 through 4. These events occur when an input changes state from high to low, or low to high depending on the configuration.

USING_CN_INT_EVENT – This will enable the change notification interrupt event. This occurs when pins change from either high to low or vise versa.

USING_TIMER_x_EVENT – This will enable the timer x event. Where x is 1 through 5. These events occur when a timer overflows on its compare value.

USING_EXT_OSC_FAILED_EVENT – This will enable the external oscillator failed event. This occurs when the external oscillator fails to run.

USING_MEMORY_WARNING_EVENT – This will occur when the OS heap falls below a certain user predefined level. The level is defined in RTOSConfig.h and is called MEMORY_WARNING_LEVEL_IN_BYTES. This is a software event and is not tied to hardware.

USING_MEMORY_WARNING_CLEAR_EVENT – This will occur when the OS heap falls below MEMORY_WARNING_LEVEL_IN_BYTES after exceeding it. This is a software event and is not tied to hardware.

USING_CPU_EXCEPTION_RAISED_EVENT – This occurs when a CPU exception is generated. There are various CPU exceptions that can occur depending on the CPU architecture.

USING_USER_x_EVENT – This will enable user x event. Where x is a value 1 through 10. This is a software event and is not necessarily tied to hardware. To generate a user event the define **USING_RAISE_EVENT_METHOD** in RTOSConfig.h must be set to a 1 and the method OS_RESULT RaiseEvent(EVENT Event) must be called with the user x event passed to it.

9.0 Timers

Timers are used to count how many OS ticks have elapsed since the timer was enabled. There are multiple types of timers, but all use the software timer as a base object and deviate from there.

9.1 Software Timer

A software timer is the most simplistic timer offered by the NexOS. It is able to count the number of OS ticks that have elapsed since the timer was enabled. When a software timer is enabled it is placed on a global list. Each time the OS timer tick interrupt occurs the OS will review the software timer list and increment the tick count of each timer on the list. When a software timer is disabled it is removed from the list. This approach reduces overhead each time the tick count needs to be incremented for all active software timers.

Basic functionality for enabling, clearing, restarting, and stopping a software timer is provided. A user can get the number of OS ticks the software timer has kept tract of at any point in time. Meaning, you do not have to stop a software timer to read from it. The software timer module must be present for all other timers the NexOS offers to be used.

9.1.1 Software Timer Requirements

To enable software timers the files SoftwareTimer.c/h must be included in the project, and USING_SOFTWARE_TIMERS must be defined as a 1 in RTOSConfig.h.

9.2 Callback Timer

A callback timer is designed to execute a user specified method at a user defined periodicity. Some examples might be to toggle an output to an LED every 500ms to show a heartbeat of the system running. Callback timers can be started, stopped, and modified at any point in time. The callback which the callback timer executes cannot call any method which may block and should be as small as possible.

Each OS timer tick will cause the callback timer list to be reviewed, and this occurs after the software timer list was updated. Only enabled callback timers are on the callback timer list. If the software timer associated with the callback timer has the same tick count as the periodicity the callback timer was configured for, the callback method will then be execute and the software timer will have its count reset to zero. This process will repeat indefinitely.

9.2.1 Callback Timer Requirements

The requirements in section 9.1.1 Software Timer Requirements must be met. The files CallbackTimer.c/h must also be present in the project. USING_CALLBACK_TIMERS must also be defined as a 1 in RTOSConfig.h.

9.3 Event Timer

Event timers are used to count the number of OS ticks between two events. This allows events to be timed without the need for a task to monitor the events. Event timers use all valid events in the EVENT enumeration of Event.h. There is a global list the OS uses to keep track of enabled event timers. When a valid event is generated the OS will iterate through the list of event timers and see if any timer has the event for a start or stop event. If it does the OS will start/stop the event timer according to the start and stop policy.

When an event timer is created two events are specified. The starting event and the ending event. When the starting event occurs the event timer will begin counting the number of OS ticks that have elapsed. Once the ending event occurs the event timer will stop counting the number of OS ticks. This way you can see how much time was between the two events.

Event timers also have an optional start and stop callback that can be specified by the user. These callbacks are called when the start or stop event occurs and the event timer is enabled. The start callback must be specified if CALLBACK_ON_EACH_START_EVENT is used for the event timer start policy. The stop callback must be specified if CALLBACK_ON_EACH_STOP_EVENT is used for the event timer stop policy.

There are different policies for when the event timer should be started or stopped. Below are the enum's for these policies and their meanings.

enum EVENT_TIMER_START_POLICY

CALLBACK_ON_EACH_START_EVENT – This is used in conjunction with a START_EVENT_CALLBACK. The START_EVENT_CALLBACK must call EventTimerStartCallbackAction() with the policy that it wants to enforce for the start event.

ONLY_START_WHEN_NOT_RUNNING – This will only start the EVENT_TIMER if it is currently not running.

RESTART_TIMER_ON_EACH_START_EVENT – This will start the EVENT_TIMER and reset its current tick count to zero.

enum EVENT_TIMER_STOP_POLICY

CALLBACK_ON_EACH_STOP_EVENT – This is used in conjunction with a STOP_EVENT_CALLBACK. The STOP_EVENT_CALLBACK must call EventTimerStopCallbackAction() with the policy that it wants to enforce for the stop event.

STOP_TIMER – This will only stop the timer and preserves the tick count. The EVENT_TIMER remains enabled and will start on the next start event.

STOP_CLEAR_TIMER – This will stop the EVENT_TIMER and clear the tick count back to zero. The EVENT_TIMER remains enabled and will start on the next start event.

STOP_CLEAR_DISABLE_TIMER – This will clear the tick count back to zero and disable the EVENT_TIMER. The EVENT_TIMER will not start on the next start event.

STOP_DISABLE_TIMER – This will stop the EVENT_TIMER and disable it. The timer will not start on the next start event, and its tick count is preserved.

9.3.1 Event Timer Requirements

The requirements in section 9.1.1 Software Timer Requirements must be met. The requirements in section 8.0.1 Event Requirements must be met. The files EventTimer.c/h must be present in the project.

10. User Callbacks

The NexOS has the ability to call certain callbacks when software or hardware events are asserted. The events features and user callbacks utilize the same code base, but can be used individually. Information about each specific callback and when it is called can be found in RTOSConfig.h. When using a callback no blocking method can be called from the callback. These should be treated as if they are being called from an ISR.

The file OS_EventCallback.c holds all the callbacks which are related to hardware events. The file OS_Callback.c holds all the callbacks which are related to software events. The callback code can be placed inside of each method which will be utilized.

10.1 User Callback Requirements

Any hardware callback used also needs the files Interrupt.S, InterruptHandler.c, OS_EventCallbacks.c, and PortInterruptHandler.h present in the build. The corresponding define in RTOSConfig.h must also be set to a 1 to use the callback. For any of the software callbacks the files OS_Callback.c/h should be present in the build. Below is a list of all the hardware callbacks defined in RTOSConfig.h (the x is replaced with a number).

USING_EXT_INT_x_CALLBACK – This will enable the external interrupt x callback. Where x is a value 0 through 4. These callbacks occur when an input changes state from high to low, or low to high depending on the configuration.

USING_CN_INT_CALLBACK – This will enable the change notification interrupt callback. This occurs when pins change from either high to low or vise versa.

USING_TIMER_x_CALLBACK – This will enable the timer x callback. Where x is 1 through 5. These callbacks occur when a timer overflows on its compare value.

USING_EXT_OSC_FAILED_CALLBACK – This will enable the external oscillator failed callback. This occurs when the external oscillator fails to run.

Below is a list of all the software callbacks defined in RTOSConfig.h.

USING_MEMORY_WARNING_USER_CALLBACK - This callback will be called when the OS heap falls below a certain user predefined level. The level is defined in RTOSConfig.h and is called MEMORY_WARNING_LEVEL_IN_BYTES.

USING_MEMORY_WARNING_CLEAR_USER_CALLBACK - This callback will be called when the OS heap falls below MEMORY_WARNING_LEVEL_IN_BYTES after exceeding it. This is a software event and is not tied to hardware.

USING_IDLE_TASK_USER_CALLBACK – This callback will be called whenever the Idle task is run.

USING_MAINTENANCE_TASK_USER_CALLBACK – This callback will be called whenever the Maintenance task is run.

USING_TASK_CHECK_IN_USER_CALLBACK – This callback is called when a task exceeds its check in count and is then either deleted or restarted.

USING_OS_TICK_UPDATE_USER_CALLBACK – This callback is called each time the OS timer triggers and interrupt and increments the OS tick count.

USING_CONTEXT_SWITCH_USER_CALLBACK – This callback is called each time the currently executing task is swapped out for a new task to execute.

USING_CPU_EXCEPTION_RAISED_USER_CALLBACK - This callback will be called when a CPU exception is generated. There are various CPU exceptions that can occur depending on the CPU architecture.

USING_ENTER_DEVICE_SLEEP_MODE_USER_CALLBACK – This callback will be called right before the device enters sleep mode with a call to PortEnterSleepMode().

USING_EXIT_DEVICE_SLEEP_MODE_USER_CALLBACK – This callback will be called right after the device wakes up and finishes executing the method PortStartOSTickTimer().