JenOS
User Guide

JN-UG-3075
Revision 1.4
19 December 2012

Contents

About this Manual

This manual provides a single point of reference for information relating to the Jennic Operating System, referred to as JenOS. The manual provides both conceptual and practical information concerning JenOS, and provides guidance on use of the JenOS Application Programming Interfaces (APIs). The API resources (functions and structures) are fully detailed.

JenOS is designed to be used with the NXP ZigBee PRO stack on the NXP JN51xx wireless microcontrollers. This manual should be used throughout ZigBee PRO wireless network application development, along with the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

Note: This manual is supplied in a ZIP file which also contains example software that is used in the tutorials in Chapter 14. The examples are detailed in Appendix A.

Organisation

This manual is divided into four parts:

- Part I: Concept and Operational Information comprises six chapters:
  - Chapter 1 introduces JenOS, including its five modules and APIs
  - Chapter 2 describes how to use the Real-time Operating System (RTOS)
  - Chapter 3 describes how to use the Persistent Data Manager (PDM)
  - Chapter 4 describes how to use the Power Manager (PWRM)
  - Chapter 5 describes how to use the Protocol Data Unit Manager (PDUM)
  - Chapter 6 describes how to use the Debug (DBG) module
- Part II: Reference Information comprises six chapters:
  - Chapter 7 describes the functions of the RTOS API
  - Chapter 8 describes the functions of the PDM API
  - Chapter 9 describes the functions of the PWRM API
  - Chapter 10 describes the functions of the PDUM API
  - Chapter 11 describes the functions of the DBG API
  - Chapter 12 details the structures used by the JenOS APIs
- Part III: Configuration Information comprises two chapters:
  - Chapter 13 outlines the static configuration required to use JenOS and its resources
  - Chapter 14 describes how to use the JenOS Configuration Editor

- Part IV: Appendices comprises three appendices that describe the example applications supplied with this manual, the use of hardware counters and clearing interrupts.

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the `Courier New` typeface.

This is a **Tip**. It indicates useful or practical information.

This is a **Note**. It highlights important additional information.

*This is a **Caution**. It warns of situations that may result in equipment malfunction or damage.*

Acronyms and Abbreviations

APDU	Application Protocol Data Unit
API	Application Programming Interface
DBG	Debug
ISR	Interrupt Service Routine
JenOS	Jennic Operating System
MAC	Media Access Control
PAN	Personal Area Network
NPDU	Network Protocol Data Unit
NVM	Non-Volatile Memory
OS	Operating System
PDU	Protocol Data Unit

PDUM	Protocol Data Unit Manager
PDM	Persistent Data Manager
PIC	Programmable Interrupt Controller
PWRM	Power Manager
RTOS	Real-Time Operating System
SDK	Software Developer's Kit
UART	Universal Asynchronous Receiver-Transmitter
ZPS	ZigBee PRO Stack

Related Documents

JN-UG-3048	ZigBee PRO Stack User Guide
JN-UG-3064	SDK Installation and User Guide
JN-UG-3066	JN51xx Integrated Peripherals API User Guide
JN-AN-1122	ZigBee PRO Home Sensor Demo Application Note
JN-AN-1123	ZigBee PRO Application Template Application Note
JN-DS-JN5148	JN5148 Data Sheet
JN-DS-JN516x	JN516x Data Sheet

Trademarks

Chip Compatibility

The software described in this manual can be used on the following NXP wireless microcontrollers:

- JN516x (currently only JN5168)
- JN5148 (variants JN5148-001 and JN5148-Z01)

Where described functionality is applicable to all the supported microcontrollers, the device may be referred to in this manual as the JN51xx.

Part I:
Concept and Operational Information

1. Introduction

The Jennic Operating System (JenOS) is designed for use in wireless network applications for the NXP JN51xx device, providing an interface which simplifies the programming of a range of operations that are not specific to wireless networking.

JenOS is primarily intended for use with the NXP ZigBee PRO stack in order to develop wireless network applications based on the ZigBee standard. Therefore, this manual should be studied in conjunction with the *ZigBee PRO Stack User Guide (JN-UG-3048)* - other reference resources are detailed in Section 1.3.

1.1 Modules and Architecture

JenOS is organised into five modules, each with a dedicated Application Programming Interface (API) to facilitate easy interaction between the application and the JenOS module. Each API consists of a set of C functions and associated resources.

In addition, a configuration editor is provided which allows the graphical configuration of the JenOS resources used by the application. This tool is known as the JenOS Configuration Editor and is a plug-in for the Eclipse IDE (Integrated Development Environment) - the tool is described in Chapter 13 and Chapter 14.

1.1.1 JenOS Modules

The JenOS modules are briefly described below:

- **Real-time Operating System (RTOS):** This module provides a mechanism for reacting to real-time events in a way that optimises the efficiency and reliability of the system. The RTOS module is described in Chapter 2.
- **Persistent Data Manager (PDM):** This module handles the storage of context and application data in Non-Volatile Memory (NVM), and the retrieval of this data. It provides a mechanism by which the JN51xx device can resume operation without loss of continuity following a power loss. The PDM module is described in Chapter 3.
- **Power Manager (PWRM):** This module manages the transitions of the JN51xx device into and out of low-power modes, such as sleep mode. The PWRM module is described in Chapter 4.
- **Protocol Data Unit Manager (PDUM):** This module is concerned with managing memory, as well as inserting data into messages to be transmitted and extracting data from messages that have been received. The PDUM module is described in Chapter 5.
- **Debug module (DBG):** This module allows diagnostic messages to be output when the application runs, as an aid to debugging the application code. The DBG module is described in Chapter 6.

1.1.2 Software Architecture

On a JN51xx-based node in a ZigBee PRO wireless network, JenOS interacts with the following software blocks:

- User application (through use of the JenOS APIs in the application code)
- ZigBee PRO stack
- JN51xx integrated peripherals

JenOS can be envisaged as sitting alongside the ZigBee PRO stack and the JN51xx Integrated Peripherals API, as depicted in the diagram below.

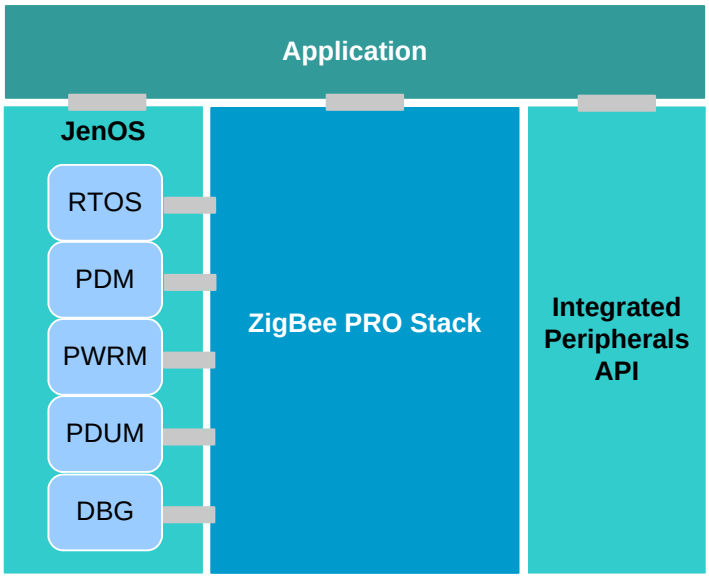


Figure 1: Basic Software Architecture

1.2 Installation

JenOS and related components are supplied in the NXP Software Developer's Kit (SDK) installers, as follows:

- **JN514x:** JenOS is installed as part of the *JN514x SDK Libraries (JN-SW-4040)*. The JenOS Configuration Editor (NXP plug-in for the Eclipse IDE) is provided separately.
- **JN516x:** JenOS is installed as part of the 'JN516x SDK Libraries' for the ZigBee application profiles (JN-SW-4062 for ZigBee Light Link, JN-SW-4064 for Smart Energy). The JenOS Configuration Editor (NXP plug-in for the Eclipse IDE) is also provided in these installers.

The JenOS Persistent Data Manager (PDM) is also provided as a standalone feature (without the rest of JenOS) as part of the *JN514x JenNet-IP SDK (JN-SW-4051)* and *JN516x JenNet-IP SDK (JN-SW-4065)*.

The above installers are available from **www.nxp.com/jennic/support**. Installation instructions are provided in the *SDK Installation and User Guide (JN-UG-3064)*.

1.3 Reference Resources

In addition to this manual, a number of other resources are available from **www.nxp.com/jennic/support** to aid the development of application code which uses JenOS:

- *SDK Installation and User Guide (JN-UG-3064)* describes the installation of JenOS as part of the Software Developer's Kits.
- *ZigBee PRO Stack User Guide (JN-UG-3048)* provides some guidance on the use of JenOS API functions in ZigBee PRO application code.
- *ZigBee PRO Home Sensor Demo (JN-AN-1122 for JN5148 and JN-AN-1183 for JN516x)* provides an example ZigBee PRO application which uses JenOS.
- *ZigBee PRO Application Template (JN-AN-1123 for JN5148 and JN-AN-1184 for JN516x)* provides a starting point for application development using the ZigBee PRO and JenOS APIs.

2. Real-time Operating System (RTOS)

This chapter introduces the concept of a Real-time Operating System (RTOS) and introduces the main features of the RTOS which is a module of JenOS.

An RTOS allows the host system to react to multiple real-time events and schedule processing to meet the deadlines of these events. For example:

- An application that controls a set of traffic lights at a crossroads, where pedestrians can press buttons to request crossing any of the roads
- An application that handles data sampling, compression and storage in a digital sound-recording system

The above applications each have a number of tasks with different urgencies competing for CPU time.

2.1 RTOS Fundamentals

An operating system reacts to events, such as interrupts, and allocates CPU usage to the processing of tasks that arise from these events. The simplest operating system works on a ‘first come, first served’ basis. Thus, the tasks are handled in the order that they occur and the processing for one task is allowed to finish before processing for the next task starts.

Most operating systems allow multiple tasks to seemingly run concurrently. In fact, the CPU may be able to process only one task at a time, but shares its time between the tasks by switching between them, giving the illusion of concurrency - this is called multi-tasking. A conventional operating system may use a simple scheduling algorithm to allocate CPU time to multiple tasks. A common scheduler is the round-robin algorithm, which allocates a time-slice to each task in cyclic order. In such a ‘time slicing’ scheme, all tasks are allocated equal slices of time.

In a real-time application, some tasks may have strict deadlines and must not be delayed by other tasks. For example, in a digital sound-recording application, a sampling task must be completed within a certain time of it starting. This task will be more important than a simultaneous task to compress the previous sample. This suggests the need for precedence to ensure that the sampling task is allowed to meet its deadline.

An RTOS accommodates this idea of precedence, as follows:

- Uses priorities assigned to the different tasks to be performed
- Schedules CPU usage such that higher priority tasks are handled before lower priority tasks
- May allow the processing of a task to be temporarily suspended while a newer, higher priority task is handled (pre-emptive RTOS only)

Therefore, an RTOS multi-tasks without time-slicing, but instead allocates CPU time according to the priorities of the current tasks.

2.2 Introduction to the JenOS RTOS

The RTOS within JenOS offers both immediacy and flexibility in scheduling real-time tasks with different priorities. It is a pre-emptive RTOS, so switches CPU usage to the highest priority task, but also provides a 'mutex' feature that allows execution of a critical section of the currently running task to be completed before any switch occurs.

The RTOS identifies the following set of high-level priorities which apply to four general classes of event:

1. Uncontrolled interrupts corresponding to events external to the operating system (not handled by RTOS)
2. Operating system housekeeping tasks
3. Controlled interrupts corresponding to events internal to the operating system (handled by RTOS)
4. User tasks (handled by RTOS)

Therefore, use of the RTOS in the user application is only concerned with the last two categories - controlled interrupts and user tasks. There is a separate set of user-defined priorities within each of these categories, but a controlled interrupt will always take priority over a user task.

The JenOS RTOS provides an 'idle task', which is executed when there are no other tasks to be run.

The RTOS is started using the function **OS_vStart()**. Following a warm start with memory held, it can be re-started using the function **OS_vRestart()**.

2.3 RTOS Configuration

The JenOS RTOS is configured at build time. This ensures that provision can be made for all types of application.

RTOS configuration is performed during application development using the JenOS Configuration Editor. This tool allows OS resources to be easily assigned and configured through a graphical interface. The tool then generates the necessary OS configuration files to feed into the application build process. The JenOS Configuration Editor and the build process are outlined in Chapter 13 before a more detailed account of the JenOS Configuration Editor is presented in Chapter 14.

2.4 RTOS Concepts and Features

This section details the following concepts and features of the JenOS RTOS:

- **User tasks:** The user application must be divided into user tasks, described in Section 2.4.1.
- **Interrupt Service Routines (ISRs):** The user application must define ISRs to deal with controlled interrupts (those handled by the RTOS) - see Section 2.4.2.
- **Priorities and scheduling:** Priorities must be assigned to tasks and ISRs statically by the system developer, using the method described in Section 2.4.3.
- **States and transitions:** The possible states of user tasks and ISRs, and the transitions between these states, are described in Section 2.4.5.
- **Scheduled activities:** Individual activities within tasks/ISRs can be scheduled to start at certain times, as described in Section 2.4.6.
- **Mutually exclusive access (mutex):** The mutex feature allows the priority system to be effectively over-ridden when tasks are competing for a shared resource and task switching should be avoided - see Section 2.4.7.
- **Inter-task communication:** Communications between tasks can be managed using message queues, as described in Section 2.4.8.

2.4.1 User Tasks

A user application can be conveniently subdivided into sections that are executed according to their real-time requirements. These sections can be implemented as tasks, where a task provides the framework for the execution of functions. The JenOS RTOS defines a task as consisting of a main C function and all its sub-functions. The main function of a task can only be invoked by the operating system (no user function is allowed to call it).

A task is defined using the RTOS macro **OS_TASK()**. The task must be given a reference handle, which is assigned in the RTOS configuration. The handle of the currently running task can be obtained using the function **OS_eGetCurrentTask()**.

The set of tasks in an application must be assigned priorities, used to determine the allocation of CPU time in a multi-tasking environment. This is described further in Section 2.4.3.

Caution: *To allow the RTOS to operate correctly, user tasks must be designed so that they do not block.*

2.4.2 Interrupt Service Routines (ISRs)

The user application may define Interrupt Service Routines (ISRs) to handle controlled interrupts. These are interrupts that are managed by the RTOS via its control mechanisms (e.g. mutex) and API calls. Each interrupt is assigned an ISR, which is invoked by the operating system when the interrupt occurs.

An ISR is defined using the RTOS macro **OS_ISR()**. The ISR must be given a reference handle, which is assigned in the RTOS configuration.

The set of ISRs in an application must be assigned priorities, used to determine the allocation of CPU time when multiple interrupts are pending. This is described further in Section 2.4.3.

Note 1: The RTOS API contains functions to enable/ disable interrupts. You can enable/disable all interrupts or just the controlled interrupts that the RTOS handles. For details of these functions, refer to Section 7.2.3.

Note 2: The RTOS cannot clear interrupts and it is the responsibility of the application to do this - refer to Appendix C.

***Caution:** The RTOS uses the Programmable Interrupt Controller (PIC) of the JN51xx device. When using the RTOS, you must therefore not use the PIC directly.*

2.4.3 Priorities and Scheduling

The user tasks and ISRs can both be prioritised for CPU allocation, each being assigned its own set of priorities. These priorities are statically set in the RTOS configuration by the system developer at build time. The sections below describe the priority system and the related topic of CPU scheduling.

Assigning Priorities

Within each of the two task categories (user task and ISR), the following method should be used to assign priorities:

1. Rank the tasks in order of importance with respect to their deadlines (i.e. deadline-monotonic analysis).
2. Assign priorities to the tasks as integer values, starting with 1 for the lowest priority task and incrementing the assigned value until all tasks have a priority.

Therefore, the assigned priorities are 1, 2,... n, where the higher the value, the higher the priority. The priority assigned to an ISR is referred to as its Interrupt Priority Level (IPL).

CPU Scheduling

In terms of scheduling, among the user tasks/ISRs waiting for CPU time, the one with the highest priority will be handled next. However, an ISR will always take precedence over a user task (the lowest ISR priority takes precedence over the highest user task priority). If an event then occurs which has a corresponding user task/ISR with higher priority than the one currently being executed, processing will switch to the higher priority process, with the existing process being temporarily suspended. In this case, the higher priority process is said to pre-empt the lower priority process.

Co-operative Tasks

Tasks can be grouped as co-operative tasks. A task in a Co-operative Task Group will not pre-empt another task from the same group, irrespective of their relative priorities. A running co-operative task temporarily takes the highest priority level assigned to the members of its group. However, any task from within the group can pre-empt or be pre-empted by a task from outside the group.

2.4.4 Task/ISR States

At any one time, a user task or ISR can be in one of three states:

- **Running:** In the running state, the CPU is executing the functions/instructions that make up the task/ISR. Only one task/ISR can be in this state at any one time.
- **Pending:** In the pending state, the task/ISR has been activated (and may have been previously run), and is waiting to become the highest priority task/ISR. When it does, it will be switched into the running state.
- **Dormant:** In the dormant state, a task/ISR is awaiting activation (which will put it in the pending state) or has already been run.

Transitions between these states are further described in Section 2.4.5.

A user task can be activated (moved from the dormant state to pending state) in the application code. The task is subsequently managed automatically by the RTOS. Activation of a user task is performed using the **OS_eActivateTask()** function. When this function is called, the activation counter for the task is incremented. It is possible to call this function even when the user task is already in the pending state - the activation counter keeps a record of the number of times the user task must be run before it can return to the dormant state.

Note: A user task must initially be activated from the main task but, once in the running state, can activate itself using **OS_eActivateTask()**, in which case its activation counter will be incremented.

An ISR is automatically activated when the corresponding interrupt is generated.

Once activated, a user task/ISR will be executed when it becomes the highest priority pending user task/ISR - see Section 2.4.3.

2.4.5 State Transitions

As detailed in Section 2.4.4, at any one time, a user task or ISR can be in the running, pending or dormant state. The possible transitions between these states are described below.

1. **Activate:** A user task must first be activated (using **eOS_ActivateTask()**) to move it from the dormant state to the pending state. Note that ISRs do not need to be explicitly activated.
2. **Start:** When a pending task/ISR becomes the highest priority task/ISR, it is automatically started by the RTOS, which moves it to the running state.
3. **Pre-empt:** A running task/ISR can be pre-empted by a higher priority task/ISR by suspending execution and moving it back to the pending state. Here it will stay until it becomes the highest priority task/ISR again. It will then be moved back in the running state and execution will be resumed from where it left off.
4. **Complete:** Once execution of a task/ISR has completed, it is automatically moved from the running state to the dormant state.

The possible state transitions are illustrated in the figure below.

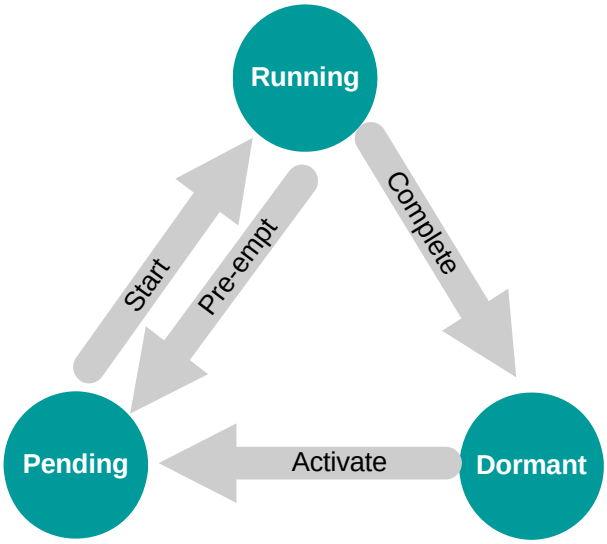


Figure 2: State Transitions for User Task or ISR

Note: An ISR always takes precedence over a user task. Therefore, a user task can never pre-empt an ISR.

2.4.6 Activity Scheduling (using Software Timers)

Within a user task or ISR, it may be necessary to schedule an activity to occur in the future or on a regular basis (i.e. periodically). For example, during a 'button detect' task, we may need to sample the state of a button every 10 ms. For such scheduled activities, the RTOS provides software timers and a number of timer functions.

A software timer is derived from a source counter, which can be either of:

- A hardware timer, such as the on-chip tick timer or an external timer
- Another software timer

The software timers and source counter required by the application are pre-defined in the RTOS configuration using the JenOS Configuration Editor. Each software timer has a handle, which is also assigned in the RTOS configuration.

A software timer is started using the function **OS_eStartSWTimer()**. As part of this function call, you must specify the number of ticks (of the source counter) before the software timer expires. This value is entered into a 'compare register' for the source counter. The counter increments and when the count reaches the value in the compare register, an interrupt may be generated and an ISR invoked, where this ISR is pre-defined in the RTOS configuration for the application. Data for this ISR can be specified through the function **OS_eStartSWTimer()**. The ISR must call the function **OS_ExpireSWTimers()**.

Note: Several software timers may be based on the same source counter. Their different expiration times are handled in a relay, by passing ownership of the source counter from one timer to the next - when one timer expires, the source counter's compare register is updated with the number of ticks until the next timer expires, and so on.

The function **OS_ExpireSWTimers()** sets the software timer status to 'expired' and checks whether there are any other pending software timers for the same source counter:

- If there are no pending software timers, the function disables the source counter.
- If there is at least one pending software timer, the function updates the source counter's compare register with the required number of ticks (until the next software timer expires).

When a software timer expires, if the same timer is to be re-started immediately (possibly with a different timed period), the function **OS_eContinueSWTimer()** can be used to re-start the timer without loss of continuity - there will be no break between the last expiry point and the new timer run, provided that this function is called before the next counter period starts.

Note: If the tick timer is used as the source counter and the maximum count of the tick timer is *T* (before the tick timer wraps around), there must be no more than *T/2* ticks between consecutive software timer expiry events (e.g. if the tick timer wraps around every 60 seconds, a software timer must expire every 30 seconds or less).

Once started, a software timer can be prematurely stopped at any time using the function **OS_eStopSWTimer()**.

Caution: *To allow the JN51xx device to enter sleep mode, no software timers should be active. Any running software timers must first be stopped and any expired timers must be de-activated. Both can be achieved using the function* ***OS_eStopSWTimer()****, which must be called individually for each running and expired timer.*

Callback Functions and Macros

The software timer functions mentioned above use callback functions that are user-defined, e.g. to enable/disable the hardware counter. The callback functions must be defined using macros provided in the RTOS module. These software timer functions are listed below with their associated callback functions and macros:

- **OS_eStartSWTimer():**
  - Calls the user-defined 'hardware counter enable' callback function, defined using the macro **OS_HWCOUNTER_ENABLE_CALLBACK()**
  - Calls the user-defined 'hardware counter get' callback function, defined using the macro **OS_HWCOUNTER_GET_CALLBACK()**
  - Calls the user-defined 'hardware counter set' callback function, defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**
- **OS_eStopSWTimer():**
  - Calls the user-defined 'hardware counter disable' callback function, defined using the macro **OS_HWCOUNTER_DISABLE_CALLBACK()**
- **OS_eContinueSWTimer():**
  - Calls the user-defined 'hardware counter set' callback function, defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**
- **OS_eExpireSWTimers():**
  - Calls the user-defined 'software timer expired' callback function, defined using the macro **OS_SWTIMER_CALLBACK()**
  - Calls the user-defined 'hardware counter set' callback function, defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**

For full details of these macros and callback functions, refer to Chapter 7. Implementations of these callback functions are provided in the Application Note *ZigBee PRO Application Template (JN-AN-1123)*.

2.4.7 Mutual Exclusion (Mutex)

The pre-emptive task scheduling of an RTOS can be problematic when two competing tasks need to access the same resource. In this case, switching the running task to a higher priority task, where both tasks can control a shared resource, may lead to undesirable results. For example, if both tasks need to access the same memory block and the running task is already writing data to memory, it is not desirable for this process to be suspended and for another task to start writing to the memory block. In such circumstances, it is important to allow the (lower priority) running task to complete its access before the higher priority task starts.

The JenOS RTOS provides a mutual exclusion (mutex) feature to prevent task switching during execution of a critical part of a user task or ISR. The critical section of code within the user task/ISR must be delimited with the functions **OS_eEnterCriticalSection()** and **OS_eExitCriticalSection()**.

For this feature, RTOS implements a Priority Inheritance Protocol, which works by temporarily raising the priority of the running task during the critical section of code (the task's priority is returned to its normal value outside the critical section). A running task and pending task can only be managed in this way if they belong to the same mutex group. Each mutex group is given a unique handle and the user tasks/ISRs in a given mutex group are pre-defined in the RTOS configuration for the application. During execution of the critical section, the priority of the running task is changed to the highest possible priority of the tasks within the same mutex group.

Use of the mutex feature in handling a critical section of code is illustrated in Figure 3.

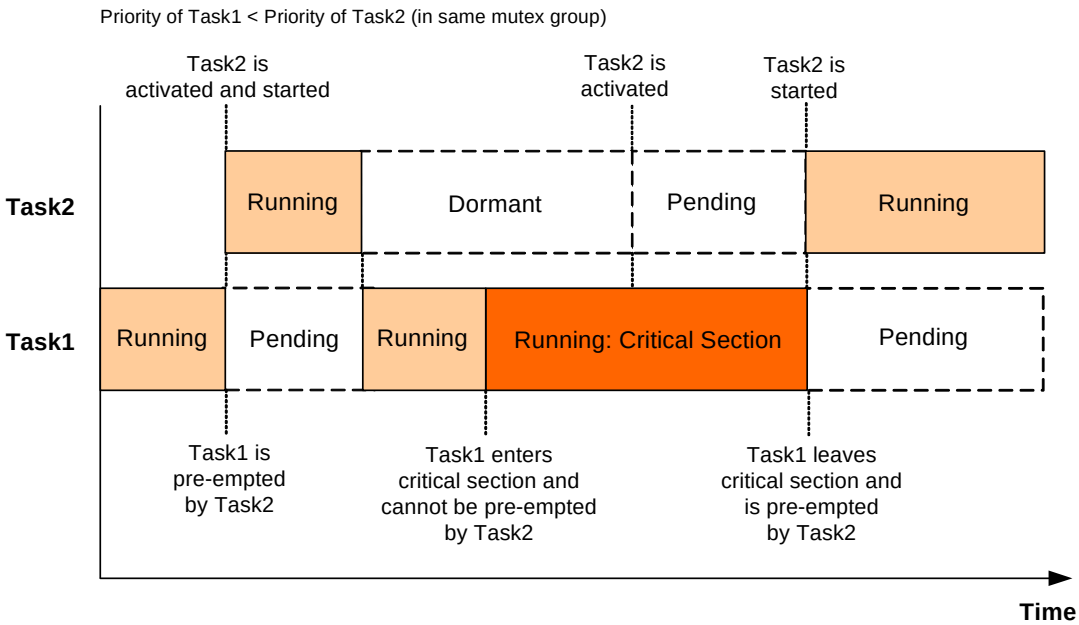


Figure 3: Mutex Handling of Critical Section

2.4.8 Inter-task Communication (using Messages)

A user task may often need to communicate with another user task, or an ISR may need to communicate with a user task. This communication can be implemented using messages. The message types required by the application are pre-defined in the header file **os_msg_types.h**. The identity of the destination task or ISR for a message type, and whether the message type is queued, are specified in the RTOS configuration for the application.

The function **OS_ePostMessage()** is used to send a message from a user task/ISR. While this user task/ISR is running, the destination task will be dormant or pending, and the message cannot be delivered immediately.

- If the message has associated user data, the message can be placed in a queue - this queue is specifically associated with the message type and the destination task, and is set up in the RTOS configuration. Alternatively, the message may be unqueued.
- If the message has no associated user data, the message is not queued.

If a message is not queued, it replaces any previous message of the same type that was waiting to be delivered to the destination task.

The destination task will run as soon as it becomes the highest priority task. Other messages may arrive in the message queue before the task runs. Once run, the task will collect the messages waiting in its queue. A message is collected using the function **OS_eCollectMessage()**, which can be called repeatedly until the queue is empty.

The function **OS_eGetMessageStatus()** is also provided which allows the destination task to determine if there are valid messages in its queue.

The above process of message sending and collection, for a queued message type, is illustrated in Figure 4 below.

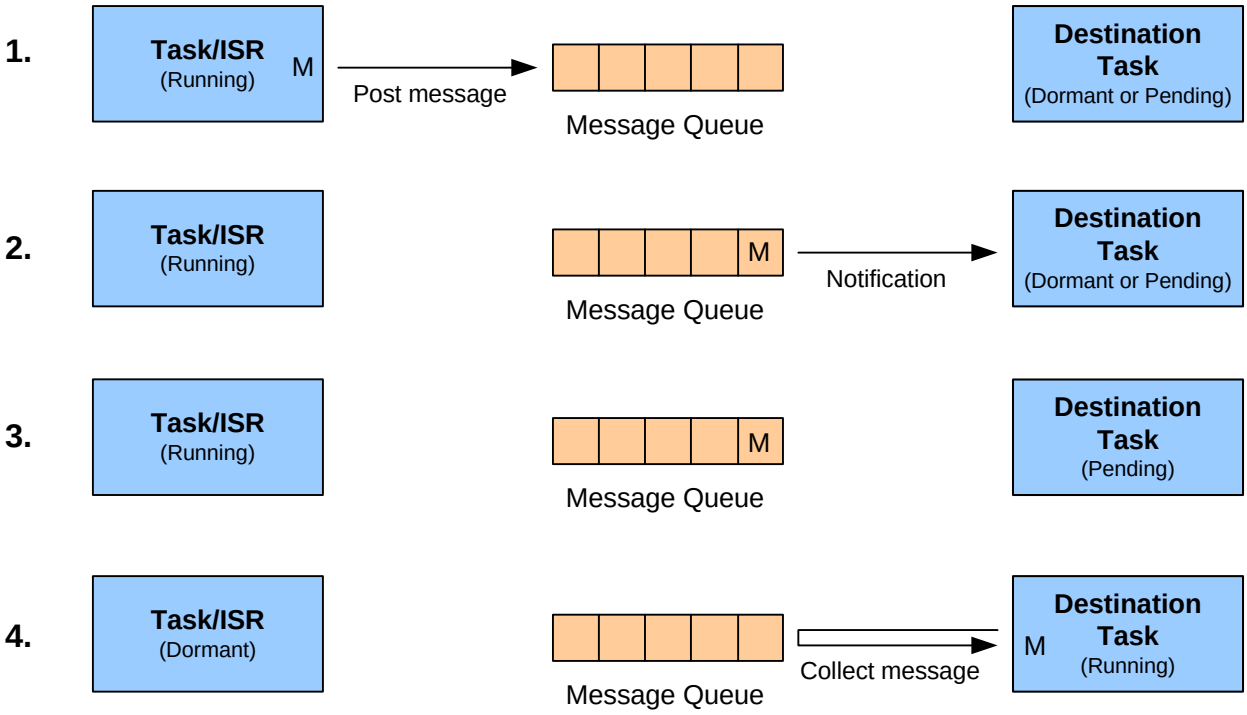


Figure 4: Message Queuing

When sending a message of a given type using **OS_ePostMessage()**, two alternative options are available:

- Invoke a callback function which performs user actions
- Activate the associated destination task through a notification - that is, to increment the task's activation counter and, if the task is not already pending, move it from the dormant to pending state.

The required send option is set in the RTOS configuration.

Data structures are generated at compile time detailing message senders, receivers and queue sizes, to ensure only the minimum resources necessary are allocated. Each task lists the message types that it can transmit and receive.

Note: The above messaging system is used by the NXP ZigBee PRO software to send and receive responses, notifications and events. These messages are automatically generated and sent by the stack software, but the application is responsible for collecting them using **OS_eCollectMessage()**.

2.5 Overlays (JN514x only)

Application code is stored on a network node as a binary file in NVM (Non-Volatile Memory) which is external to the JN5148 microcontroller, normally Flash memory. In order to be executed, the application is loaded from NVM to JN5148 RAM by the boot loader. The size of RAM therefore places a limit on the application size. JenOS provides an overlay feature which permits a larger application size than can normally be fitted into RAM.

The JN516x devices execute code from internal program Flash memory, so the overlay feature is not applicable to this family of microcontrollers.

2.5.1 Overlay Mechanism

In the overlay mechanism, part of the application program remains in NVM and is loaded into RAM only when it is needed (and is then discarded from RAM when it is no longer required). This part of the program is divided into pages or overlays. Only one overlay is loaded from NVM to RAM and executed at any one time.

An application program which uses overlays is normally organised such that the code for a set of functions (e.g. corresponding to a software library) is contained in an overlay. When a function is called by the application, the relevant overlay is loaded from NVM into RAM for execution of the function. The maximum overlay size for the NXP-supplied software libraries is 4 Kbytes, which is the recommended maximum to benefit from the overlay mechanism. An overlay typically loads from NVM to JN5148 RAM at a rate of 2 Kbytes per millisecond. If required, the overlay feature must be enabled for the application as described in Section 2.5.2. It is also possible to write the main application code to be handled in overlays but this option is beyond the scope of this document.

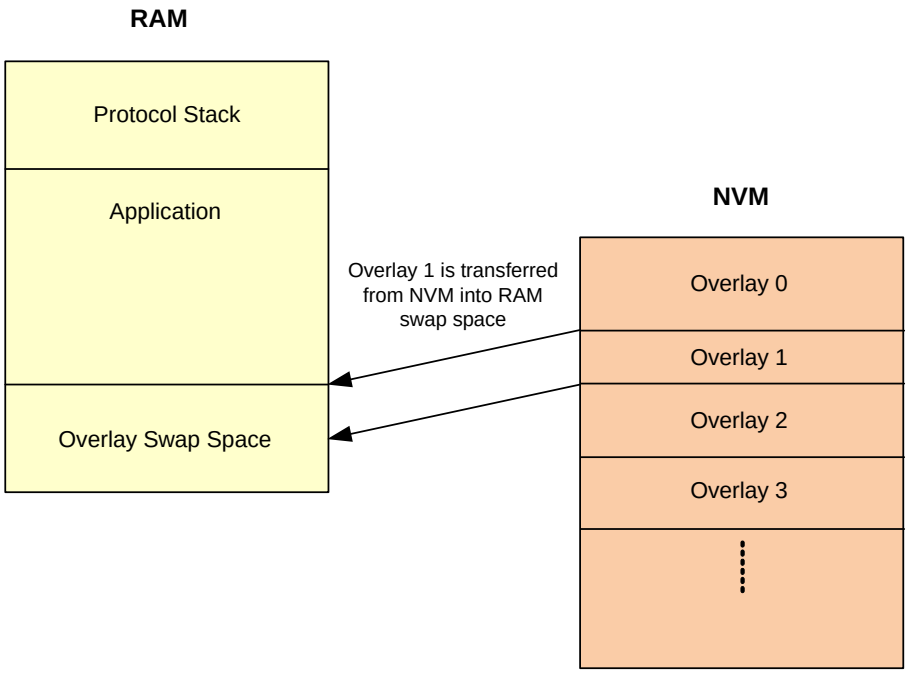


Figure 5: Overlay Mechanism

Note 1: The overlay mechanism is concerned with application code (no data), which is only ever read from NVM. Application code is never written to NVM as part of this process.

Note 2: The NXP ZigBee PRO stack and profile software is organised into overlays. You are recommended to use this software with overlays enabled, as tests show that overlays can be used without significantly impacting the performance of this software and will maximise the RAM space available for your application code.

2.5.2 Enabling Overlays

If required for an application, overlays must be enabled in three places:

- application code
- makefile
- RTOS configuration

Enabling overlays in these places is described below.

Enabling Overlays in Application Code

If overlays are to be implemented in an application, the function **OVLY_bInit()** must first be called to initialise the overlay mechanism - this must be done before calling any library functions that are contained in overlays. As part of this function call, initialisation data must be provided in a structure (see Section 12.6). This data includes pointers to three user-defined callback functions with the following purposes:

- Obtain a mutex to protect access to the SPI bus (to which NVM is connected)
- Release a mutex which protects access to the SPI bus
- Handle overlay-related events

Overlay-related events are normally only needed when debugging application code that uses overlays - see Section 2.5.3.

Default initialisation data values are available through null settings. For example, if you do not wish to use a particular callback function, you can give it a null pointer in the structure.

Important: The overlay feature and the JenOS PDM module (see Chapter 3) both need to access NVM via the SPI bus, but must be prevented from attempting accesses at the same time. This is achieved using a mutex. The user-defined callback functions to get and release a mutex for overlays must be designed such that overlays are in the same mutex group as the PDM module (see Section 2.4.7). If the mutex is not properly implemented, unpredictable behaviour may result.

Enabling Overlays in Makefile

If an application uses a software library which is to be implemented in an overlay, the overlay feature must be enabled for this library at build-time so that the resulting binary file can be internally organised into overlays. This is done through the makefile for the application.

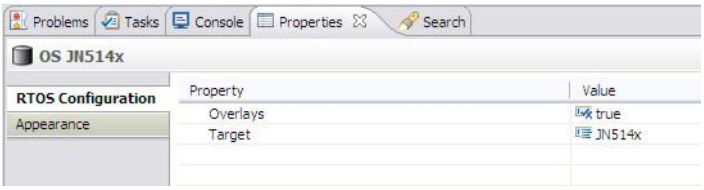
Example:

The NXP ZigBee PRO stack software is organised into multiple overlays. To enable overlays for this software, add the following lines to the application makefile:

```
ZBPRO_OVERLAYS = 1
INCFLAGS += -I$(COMPONENTS_BASE_DIR)/OVLY/Include
APPLIBS += OVLY
```

Enabling Overlays in RTOS Configuration

Overlays must also be enabled in the RTOS configuration. In the JenOS Configuration Editor, go to the **Properties** tab, select **RTOS Configuration** and set the **Overlays** property to 'true' (see screenshot below).



2.5.3 Overlays During Debug

This section describes how to track overlay activity when debugging application code.

Overlay Events

When debugging code for which overlays are enabled, you may need to be aware of the overlay operations that are in progress - for example, when an overlay is being loaded from NVM into RAM.

A number of events are available to indicate when certain overlay operations and errors occur. These events are enumerated in the `OVLY_teEvent` structure detailed in Section 12.7, which also describes the circumstances that trigger the events.

A user-defined callback function must be provided to deal with overlay-related events (and profiling - see Section 2.5.3). The prototype for this function is shown below:

```
void (*OVLY_prEvent)(OVLY_teEvent eEventType, OVLY_tuEventData *psData);
```

where:

- `eEventType` is the type of overlay event from those detailed in Section 12.7
- `psData` is a pointer to the event data, as described in Section 12.8

This callback function is registered via the initialisation function **OVLY_bInit()** - a pointer to the callback function is provided in the structure `OVLY_tsInitData` (see Section 12.6) which is passed into the initialisation function. If event handling is not needed, a null pointer should be provided for the callback function in the structure.

Overlay Profiling

Overlay profiling is a feature which allows overlay usage to be monitored during application execution and can therefore be useful during debug. Overlay profiling information is maintained in the `OVLY_tsProfiling` structure, detailed in Section 12.9. This structure is automatically updated as program execution proceeds and can be read by the application to extract information about overlay usage. The structure contains the following information:

- Total number of bytes of code loaded (so far) from NVM to RAM
- Number of checksum failures that have occurred (so far) in transferring code from NVM to RAM
- Array in which each element is a `OVLY_tsProfilingEntry` structure containing the profiling data for an individual overlay:
  - Size, in bytes, of the block of code in the overlay
  - Number of times the overlay has (so far) been loaded from NVM to RAM

If overlay profiling is to be used, a profiling (and event handling) callback function can be provided in the initialisation data passed to **OVLY_bInit()** or, alternatively, profiling can be enabled in the application by calling the function **OVLY_psProfilingInit()**. This function returns a pointer to the `OVLY_tsProfiling` structure and will over-ride any profiling/event handling callback function that was provided in the initialisation data.

2.6 OS Error Callback Function

Caution: *Errors affecting the OS can cause system instability. It is recommended that all applications register an OS error handler and enable strict error checks.*

Errors can be detected by testing the return codes from the calls to OS functions. However, this requires application code to test the return code from every OS call.

Registering an error callback function provides a robust alternative to checking the return codes. This function is invoked whenever an OS function returns an error. The function is registered as a parameter of **OS_vStart()**. The error callback option must also be enabled in the JenOS Configuration Editor.

Many OS errors leave the scheduler in an undefined state. For example, nesting a mutex by calling **OS_eEnterCriticalSection()** twice before calling **OS_eExitCriticalSection()** causes a OS_E_BAD_NESTING error. Once an error of this nature has occurred, the OS scheduler is in an undefined state.

2.6.1 Strict Error Checks

The OS scheduler will enter an undefined state if there are inconsistencies between the OS configuration diagram (in the JenOS Configuration Editor) and the application code. A strict error check option can be enabled in the JenOS Configuration Editor to check for inconsistencies between the OS configuration diagram and the software. The strict mode has a slight overhead in code space and execution time but it is good practice to enable strict checking where possible. For example, calling **OS_eEnterCriticalSection()** from a task which is not in the group for the mutex will generate OS_E_CURRENT_TASK_NOT_A_MUTEX_MEMBER with strict checking enabled. If strict checks were not enabled, the scheduler operation would be undefined and the system may become unstable.

2.6.2 Handling OS Errors

During testing, an application's error callback function should stop the application with a stack dump and the error should be fixed. The OS passes two parameters to the error callback - the status code of the error and a pointer to the handle which caused the error. These parameters should be printed out to help determine the cause of the error.

In production code, the device must be re-started from cold by calling **vAHI_SwReset()**. Data in the PDM module does not normally need to be erased, so the device can rejoin a ZigBee PRO network with existing security material.

The error callback function will be called on some non-fatal errors. Depending on the application design, the following errors can be ignored by the error callback function:

- OS_E_QUEUE_EMPTY
- OS_E_SWTIMER_STOPPED
- OS_E_SWTIMER_EXPIRED
- OS_E_SWTIMER_RUNNING

3. Persistent Data Manager (PDM)

This chapter describes the Persistent Data Manager (PDM) module which handles the storage of context and application data in Non-Volatile Memory (NVM).

Note: The PDM module is provided as a part of JenOS in the JN514x *SDK Libraries (JN-SW-4040)* and the application profile installer for ZigBee applications on the JN516x device. PDM is provided as a standalone feature in the *JN514x JenNet-IP SDK (JN-SW-4051)* and *JN516x JenNet-IP SDK (JN-SW-4065)*.

Tip: In this chapter, a cold start refers to either a first-time start or a re-start without memory (RAM) held. A warm start refers to a re-start with memory held (for example following sleep with memory held).

3.1 Overview

Data needed for the operation of a network node is normally stored in on-chip RAM. This includes data that may evolve during node operation, e.g. context data for the network stack and application data. This data is only maintained in memory while the node is powered and will normally be lost during an interruption to the power supply (e.g. power failure or battery replacement).

In order for the node to recover from a power interruption with continuity of service, provision must be made for storing a back-up of the operational data in NVM, normally Flash memory. This data can then be recovered during a re-boot following power loss, allowing the node to resume its role in the network.

Note: For the JN514x device, persisted data is held in an external SPI Flash memory. For the JN516x device, persisted data may be stored in either on-chip EEPROM or external SPI Flash memory.

The storage and recovery of operational data can be handled using the Persistent Data Manager (PDM) module, as described in the rest of this chapter. An overview of the use of the PDM API functions in application code is presented in Figure 6 below.

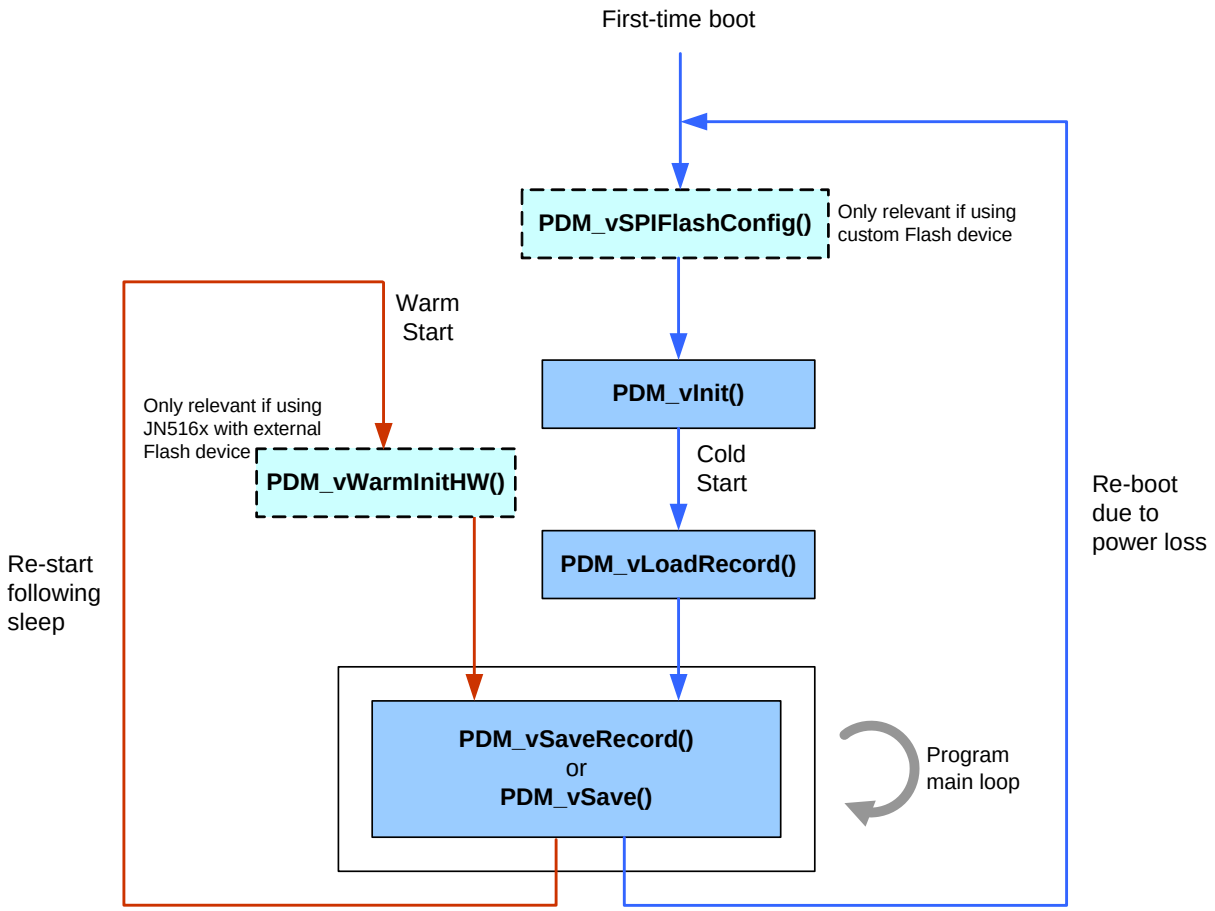


Figure 6: PDM Overview

3.2 Initialising the PDM

The initialisation of the PDM depends on the type of NVM device used. For a cold start (a first-time start or a re-start following power loss), the initialisation is as follows:

1. If NVM is a custom device, call the function **PDM_vSPIFlashConfig()**. This function requires you to specify a pointer to a set of custom functions that will be used by the PDM to interact with the NVM device (e.g. read and write functions). There is no need to call this function if using a supported NVM device.
2. Call the function **PDM_vInit()**. If using a supported NVM device, this function will auto-detect the device type. If required, optional mutexes can be specified through this function - for information on these mutexes, refer to Section 3.8. A security key can also be specified which will be used by the PDM module to encrypt data saved to NVM and to decrypt the data when read from NVM (the key can be specified by the application or obtained from eFuse) - this security will be automatically applied to stack context data but must be explicitly enabled for application data (see Section 3.4).

Both of the above functions require you to identify and specify the size of the NVM sectors to be managed by the PDM.

For a warm start (following sleep), there is no need to call **PDM_vSPIFlashConfig()** or **PDM_vInit()**. However, if an external SPI Flash device is used for a JN516x device, the function **PDM_vWarmInitHW()** must be called.

3.3 Data Storage in NVM

Data is stored in NVM in terms of 'records', where a record is an area of NVM used to store data and associated housekeeping information. Each NVM record corresponds to a data buffer in RAM, where the NVM record is used to back up the RAM buffer. Any number of NVM records of different lengths can be created, provided that they do not exceed the NVM capacity.

NVM records are created automatically for stack context data and by the application (as indicated in Section 3.4) for application data.

The stack context data which is stored in NVM includes the following:

- Application layer data:
  - AIB members, such as the EPID and ZDO state
  - Group Address table
  - Binding table
  - Application key-pair descriptor
  - Trust Centre device table
- Network layer data:
  - NIB members, such as PAN ID and radio channel
  - Neighbour table
  - Network keys
  - Address Map table

3.4 Recovering Data from NVM

Application data and stack context data are loaded from NVM to RAM as described below.

Application Data

During a cold start, after the PDM module has been initialised (see Section 3.2), the function **PDM_eLoadRecord()** must be called for each record of application data in NVM and its corresponding data buffer in RAM. This function requires a descriptor to be specified for the NVM record, where this record descriptor is held in RAM.

Note: The function **PDM_eLoadRecord()** must be called before calling any function which automatically saves application data to NVM - for example, before calling **ZPS_eAplAfInit()** to initialise the ZigBee PRO stack and **ZPS_vAplSecSetInitialSecurityState()** to initialise the ZigBee security state on the node.

Depending on the type of cold start, **PDM_eLoadRecord()** will do one of two things:

- During a first-time cold start, the function creates an NVM record that corresponds to the specified record descriptor (but stores no data in NVM). This NVM record will be used to back up application data from the specified buffer in RAM (see Section 3.5). Initial data values must be provided in this RAM buffer.
- During any subsequent cold start, the function loads data from the specified NVM record to the associated RAM buffer, thus recovering application data previously saved to NVM.

***Caution: PDM_eLoadRecord()** must not be called during a warm start, otherwise data held in RAM will be overwritten by possibly older data from NVM.*

PDM_eLoadRecord() also allows security to be enabled for the application data record. If security is enabled, data saved to the NVM record will be encrypted using the key specified through **PDM_vInit()** and will therefore be decrypted using the same key when read from the record.

Stack Context Data

The function **PDM_eLoadRecord()**, described above, is not used for records of stack context data. Loading this data from NVM to RAM is handled automatically by the stack (provided that the PDM has been initialised). When saved to NVM, the stack context data is automatically encrypted using the security key specified through **PDM_vInit()** and is therefore decrypted using the same key when read from NVM.

3.5 Saving Data to NVM

Application data and stack context data are saved from RAM to NVM as described below.

Note: If, during a data save, NVM needs to be defragmented and purged, this will be performed automatically resulting in all records being re-saved.

Application Data

Once you have application data in RAM to be backed up, you can save the RAM data associated with an individual record in NVM using the function **PDM_vSaveRecord()**. Alternatively, you can save the RAM data corresponding to all records in NVM using the function **PDM_vSave()**. In both cases, the saved application data will be encrypted if security was enabled for the corresponding NVM record through the function **PDM_eLoadRecord()** (see Section 3.4). You should save data to NVM when important changes have been made to the data in RAM.

Stack Context Data

The function **PDM_vSave()** can be used to save all records of stack context data as well as all records of application data. The saved stack context data is encrypted using the security key specified when the PDM module was initialised using **PDM_vInit()** (see Section 3.2). Note that the NXP ZigBee PRO stack automatically saves its own context data from RAM to NVM (encrypted as detailed above) when certain data items change.

3.6 Deleting Data in NVM

An individual record of application data in NVM can be deleted using the function **PDM_vDeleteRecord()**. Alternatively, all records (application data and stack context data) in NVM can be deleted using the function **PDM_vDelete()**. Note that when a record is deleted in NVM, the corresponding data buffer in RAM is not deleted.

***Caution:** You are not recommended to delete records of ZigBee PRO stack context data by calling **PDM_vDelete()** before a rejoin of the same secured network. If these records are deleted, data sent by the node after the rejoin will be rejected by the destination node since the frame counter has been reset on the source node. For more information and advice, refer to the "Application Design Notes" appendix in the ZigBee PRO Stack User Guide (JN-UG-3048).*

3.7 Ensuring Consistency of PDM Records

The data in the PDM may differ in structure from that expected by the application. The structures stored by the ZigBee PRO libraries can change due to altering table sizes in the ZPS Configuration Editor, as well as between releases of the ZigBee PRO stack libraries. Inconsistency can occur under the following circumstances:

- The internal EEPROM on a JN516x device is not erased when programming an application with the JN51xx Flash Programmer. If multiple applications are run on the same hardware, it is unlikely that the structures will be consistent between the applications.
- When an Over-The-Air (OTA) software update is performed, the PDM data is not erased. This is normally a benefit because it allows the application to rejoin the network. However, if any of the PDM structures change, a factory reset must be performed by calling **PDM_vDelete()**

Applications normally contain a way to perform a factory reset of the PDM module - for example, by calling **PDM_vDelete()** if a button is held down during reset.

The application can automatically check for PDM consistency by storing an application-specific 'magic number' in a record. A new magic number should be used if the application software or ZigBee PRO libraries PDM usage is inconsistent with the previous version of the software. Immediately after calling **PDM_vInit()**, the application should call **PDM_eLoadRecord()**. If the magic number does not match, the application should call **PDM_vDelete()** to erase all records before attempting to start the ZigBee PRO stack. If the call to **PDM_eLoadRecord()** indicates that the record has not been found, the application should also call **PDM_vDelete()** because another application may have been running that does not use the same record ID but has written inconsistent ZigBee PRO records to the PDM module.

3.8 Mutexes in PDM

The PDM module can use optional mutexes, as follows:

- PDM functions are not re-entrant and a mutex can be optionally used to prevent concurrent PDM function calls - if enabled, the mutex is applied automatically during a PDM function call.
- External NVM is accessed via the SPI bus of the JN51xx device, but this bus may also be used to access other resources. The PDM module can optionally use a mutex to prevent concurrent accesses to the SPI bus - if enabled, the mutex is applied automatically during a PDM access to NVM. The SPI bus mutex is not required when using the EEPROM on a JN516x device, as PDM has exclusive use of this memory.

If required, the above mutexes can be specified when the PDM module is initialised using the function **PDM_vInit()** - see Section 3.2. The user task must also be linked to the relevant mutex in the JenOS Configuration Editor - see Section 14.7.

The principles of a mutex are described in Section 2.4.7. All tasks that use a mutex must be connected to the mutex in the OS configuration diagram. The ZigBee PRO

stack makes PDM calls. Therefore, any task that makes ZigBee PRO function calls must be connected to the PDM mutexes.

Note: If using the overlay feature on a JN514x device (see Section 2.5) in addition to the PDM feature, it is important to use a mutex to prevent concurrent SPI bus accesses by the two features. In this case, the PDM module and the overlay feature must be in the same mutex group. If the mutex is not properly implemented, unpredictable behaviour may result.

3.9 Selecting Internal or External Storage (JN516x only)

On a JN516x device, the PDM storage mechanism is determined by setting the build flag PDM_BUILD_TYPE to either PDM_BUILD_TYPE=_EEPROM or PDM_BUILD_TYPE=_EXTERNAL_FLASH.

If the build type is set to _EEPROM, the pre-processor macro PDM_EEPROM is defined. This allows application software to be written to work with either internal or external storage. Applications would normally use the internal EEPROM. However, on a JN5168 device, the internal storage is 4 Kbytes in size. If larger records are required, an external SPI Flash device can be used.

***Caution:** When using internal EEPROM storage, the space available for user data is less than the capacity of the storage due to overheads in storing the PDM data structures. PDM also needs space to write the changed sectors of a new version of a record before erasing the previous copy of that record.*

3.10 Registering an Error Handler (JN516x EEPROM only)

The internal PDM library allows an error handler to be called to alert the application of error conditions. The error handler is registered by calling the function **PDM_vRegisterSystemCallback().**

An application must trap E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_SPACE and E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED callback errors during testing. The ZigBee PRO stack uses multiple records. Once an ‘out of space’ error has occurred, the records will be in an inconsistent state. The software must be altered to use smaller record sizes or an external SPI Flash device. The PDM record sizes for the ZigBee PRO stack are dependent on table sizes set in the ZPS Configuration Editor.

3.11 EEPROM Capacity (JN516x EEPROM only)

The JN516x internal EEPROM consists of multiple small sectors. On a JN5168 device, there are 64 sectors of 64 bytes each. The internal PDM library can only store data for one record in each sector. The PDM library needs to store some system information in each sector, so in practice each sector can hold only up to 48 bytes of record data. This means that a PDM record that has a single byte of information will need the same space as a 48-byte record and that a 49-byte record will need two sectors (the same as a 96-byte record).

The function **u8PDM_CalculateFileSystemCapacity()** returns the number of sectors that are free for PDM. The function **u8PDM_GetFileSystemOccupancy()** returns the number of sectors that are in use.

One of these functions may be called after all the records have been created and saved (including records in the ZigBee PRO stack). The return value may be printed to determine the free space. PDM writes the data for a new record before erasing the old record. To allow for the worst-case scenario, the value returned by **u8PDM_CalculateFileSystemCapacity()** must be greater than the number of sectors required to store the largest record.

4. Power Manager (PWRM)

This chapter describes the Power Manager (PWRM) module, which manages the transitions of the JN51xx device into and out of low-power modes.

Low-power modes are typically used to prolong the battery life of a node by reducing the power consumption of the device during periods when the node does not need to receive, transmit or perform any other activities. Thus, low-power modes only apply to End Devices, as the Co-ordinator and Routers always need to remain fully alert for routing purposes.

4.1 Low-Power Modes

A number of low-power modes are available on the JN51xx device. In descending order of power consumption, the modes are:

- Doze mode
- Sleep modes:
  - Sleep with memory held
  - Sleep without memory held
- Deep Sleep mode

When the node is inactive, the Power Manager will put the device into the lowest power mode possible.

The above low-power modes are described in the sub-sections below. For further information on the low-power modes of the JN51xx device, refer to the *JN51xx Data Sheet (JN-DS-JN5148 or JN-DS-516x)*.

4.1.1 Doze Mode

In Doze mode, the CPU of the chip pauses (the CPU clock is stopped) but all other parts of the JN51xx device continue to run. Any interrupt will cause Doze mode to terminate and the application program will continue running from the next instruction. To prevent the Watchdog firing when in Doze mode, the application should ensure that a timer is running at a higher frequency than the Watchdog expiry period.

4.1.2 Sleep Mode with Memory Held

During Sleep with memory held, the contents of on-chip RAM are maintained, including stack context data and application data. Thus, on waking, the device can recover from sleep very quickly to continue normal operation from the next instruction.

In this mode, all power domains are powered down except those for the on-chip RAM and VDD supply. In addition, the 32-kHz on-chip oscillator can optionally be left running, which allows the device to be woken from sleep using wake timers.

Otherwise, the device can only be woken by changes on the DIO pins or the comparator input, or by a pulse counter expiry.

Although the contents of memory are held, on waking it is still necessary to re-configure the IEEE 802.15.4 stack layers and to re-initialise most of the on-chip peripherals. Wake callback functions can be registered for this purpose:

- You DO NOT have to re-initialise the DIOs, wake timers and comparator.
- You DO have to re-initialise everything else, including all other on-chip peripherals, the IEEE 802.15.4 MAC layer and, if using callbacks, the Programmable Interrupt Controller (PIC) - the callback functions must be re-registered. On the JN516x device, the SPI hardware must also be re-initialised.

4.1.3 Sleep Mode without Memory Held

During Sleep without memory held, on-chip RAM is powered down, and therefore stack context data and application data are not preserved on-chip. Normally, this data must be saved to external NVM (Non-Volatile Memory) before the chip enters sleep mode, and then recovered from NVM on waking (see Chapter 3).

In this mode, all power domains are powered down except the VDD power supply domain. Again, the 32-kHz on-chip oscillator can optionally be left running, which allows the device to be woken from sleep using wake timers. Otherwise, the device can only be woken by changes on the DIO pins or the comparator input, or by a pulse counter expiry.

On waking, the application program must be re-loaded from external NVM before the node can resume operation. All variables and peripherals must be re-initialised, except those used as wake sources and the DIO lines.

4.1.4 Deep Sleep Mode

In Deep Sleep mode, all switchable power domains are powered down and the 32-kHz oscillator is stopped. The device can be woken from deep sleep either via a hardware reset (by taking the RESETN pin low or by power cycling the device) or a change on the DIO pins.

On waking, the application program must be re-loaded from external NVM before the node can resume operation. All variables and peripherals must be re-initialised, including the DIO lines.

4.2 Callback Functions for Power Manager

If you intend to use the Power Manager, a number of callback functions must be available for the Power Manager to call in order to:

- start the application (see Section 4.2.1)
- perform housekeeping tasks when entering and leaving low-power mode (see Section 4.2.2)
- handle interrupts from Wake Timer 1 (see Section 4.2.3)

4.2.1 Essential Callback Function

When your application uses the Power Manager, you must define and use the callback function **vAppMain()** in your code. The main task of your application must be included in this function (which must never return).

4.2.2 Pre-sleep and Post-sleep Callback Functions

In order to implement low-power modes, you must provide the Power Manager with user-defined callback functions to perform housekeeping tasks when the node enters and leaves low-power mode. Registration functions are provided for these callback functions, where the registration functions must be called in the user-defined callback function **vAppRegisterPWRMCallbacks()**.

- The pre-sleep callback function is called by the Power Manager just before putting the device into low-power mode. This callback function is registered in your code through the API function **PWRM_vRegisterPreSleepCallback()**.
- The post-sleep callback function is called by the Power Manager just after the device leaves low-power mode (irrespective of how the device was woken from sleep). This callback function is registered in your code through the API function **PWRM_vRegisterWakeupCallback()**.

vAppRegisterPWRMCallbacks() is called by the stack as part of a cold start.

The pre- and post-sleep callback function themselves must each be declared in the code using the macro

PWRM_CALLBACK(*fn_name***)**

where *fn_name* is the name of the callback function.

Each of these callback functions must also have a descriptor. This is a structure that is used in the above registering functions to specify the callback function to register.

The callback descriptor must be declared using the macro

PWRM_DECLARE_CALLBACK_DESCRIPTOR(*desc_name***,** *fn_name***)**

where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vPreSleepCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(pscb1_desc, vPreSleepCB1);
```

4.2.3 Wake Timer Callback Function

If you intend to use wake events, derived from Wake Timer 1, your interrupt handler must call the pre-defined callback function **PWRM_WakeInterruptCallback()**. This function maintains the list of scheduled wake events - if required, it will re-start the wake timer for the next wake point. It also calls the user-defined callback function specified through **PWRM_vScheduleActivity()**.

For further information on waking the device using scheduled wake events, refer to Section 4.6.

4.3 Initialising and Starting the Power Manager

The Power Manager is initialised and started using the function **PWRM_vInit()**. This function requires one of five possible low-power configurations to be specified:

- Sleep with 32-kHz oscillator running and memory held
- Sleep with 32-kHz oscillator running and memory not held
- Sleep with 32-kHz oscillator not running and memory held
- Sleep with 32-kHz oscillator not running and memory not held
- Deep sleep (oscillator not running and memory not held)

The specified configuration is the low-power mode in which the Power Manager will attempt to put the device during inactive periods. Note that Doze mode cannot be explicitly specified, but the Power Manager may put the device into Doze mode at times when the specified mode cannot be entered (see Section 4.8.1).

The criteria for selecting a sleep mode are as follows:

- **Oscillator setting:**
  - If the 32-kHz oscillator is left running during sleep, a wake point can be scheduled using **PWRM_vScheduleActivity()** - see Section 4.6.
  - Otherwise, the device must be woken by an external event (a change on a DIO line or comparator input, a pulse counter expiry or a reset).
- **Memory setting:**
  - If memory is held during sleep, stack context data and application data will be preserved in memory, allowing the device to quickly resume operation through a warm re-start following sleep.
  - Sleep without memory held provides a greater power saving. However, stack context data and application data must be saved to external NVM (Non-Volatile Memory) before entering sleep mode and restored into on-chip memory during a cold re-start on exiting sleep (see Chapter 3).

4.4 Enabling Power-Saving

To enable the Power Manager to put the JN51xx device into low-power mode at appropriate times, you must call the function **PWRM_vManagePower()**, normally called from the OS idle task. Once this function has been called, the Power Manager will, whenever possible, put the JN51xx device into the sleep mode specified through **PWRM_vInit()** (or, alternatively, into Doze mode - see Section 4.8.1).

Note: Sleep mode cannot be entered while there are software timers active (in running or expired states). You must therefore de-activate any such timers to allow the Power Manager to put the JN51xx device into sleep mode - refer to the Caution in Section 2.4.6.

4.5 Non-interruptible Activities

In order to enter sleep mode, no activities must be running that must not be interrupted by sleep. This condition for entering sleep mode is monitored using an activity counter - sleep mode can only be entered when this counter is zero. The application is responsible for maintaining the activity counter, as follows:

- Whenever an activity is started that must not be interrupted by sleep, the application must notify the Power Manager using the function **PWRM_eStartActivity()**, which increments the activity counter.
- Whenever such an activity is completed, the application must notify the Power Manager using the function **PWRM_eFinishActivity()**, which decrements the activity counter.

Caution: PWRM_eFinishActivity() *must only be called by an application following a matching call to* ***PWRM_eStartActivity()****. The OS and ZigBee PRO stack both use the activity counter, so calling* ***PWRM_eFinishActivity()*** *inappropriately can leave the OS or ZigBee PRO stack in an inconsistent state.*

You can obtain the current value of the activity counter using the function **PWRM_u16GetActivityCount()**.

4.6 Terminating Low-Power Mode

Low-power modes can be terminated in a number of ways:

- **Any Interrupt:** When in Doze mode, the device can be woken by any interrupt.
- **Wake Timer:** When in Sleep mode in which the 32-kHz oscillator runs, the device can be woken by a scheduled wake event configured using the function **PWRM_vScheduleActivity()**. For more information on scheduled wake events, refer to Section 4.6.
- **External Wake Event:** The following external wake events are available:
  - **DIO:** When in Sleep and Deep Sleep modes, the device can be woken by a change of state of a DIO line.
  - **Comparator input:** When in Sleep mode, the device can be woken by a change of state of the comparator input.
  - **Pulse counter:** When in Sleep mode, the device can be woken on expiry of the pulse counter, which counts pulses on an external input.

  The above external wake events can be controlled by functions of the JN51xx Integrated Peripherals API, described in the *Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)*. In addition, the applicable external wake events must be configured in the RTOS configuration for the application.
- **Hardware Reset:** When in Deep Sleep mode, the device can be woken by a hardware reset.

The valid wake sources for the different low-power modes are summarised in Table 1 below.

On leaving low-power mode, the Power Manager will call the user-defined callback function that has been registered using **PWRM_vRegisterWakeupCallback()**.

Low-Power Mode	Wake Source					
	Any Interrupt	Wake Timer	DIO	Comparator	Pulse Counter	Hardware Reset
Doze mode	✓	×	×	×	×	×
Sleep mode with oscillator running and memory held	×	✓	✓	✓	✓	×
Other Sleep modes	×	×	✓	✓	✓	×
Deep Sleep mode	×	×	✓	×	×	✓

Table 1: Valid Wake Sources for Low-Power Modes

4.7 Scheduling Wake Events

Note: This section is only applicable to the sleep mode in which the 32-kHz oscillator is left running and memory is held.

In **PWRM_vInit()**, if you have selected the Sleep mode with the 32-kHz oscillator running and memory held, you can schedule wake events which ensure that the device will be awake at certain times - that is, if the device is sleeping, it will be woken at the scheduled time. This scheduling uses Wake Timer 1 of the JN51xx device, which operates at 32 kHz.

A wake event can be scheduled using the function **PWRM_eScheduleActivity()**. This function requires you to specify the number of ticks of the wake timer until the wake event. You must also specify the user-defined callback function that must be called when the wake event occurs.

When the wake timer expires for a scheduled wake event, an interrupt is generated. The application's interrupt handler then calls the pre-defined callback function **PWRM_WakeInterruptCallback()**. This function maintains the list of scheduled wake events and, if necessary, re-starts the wake timer for the next scheduled wake event. The function also calls the user-defined callback function specified through **PWRM_eScheduleActivity()**.

Note: In addition, when the device wakes from sleep, the user-defined callback function registered through **PWRM_vRegisterWakeupCallback()** will also be called. However, this is a general-purpose wake-up function which is called irrespective of how the device was woken (it is not unique to scheduled wake events, but also called for external wake events).

4.8 Doze Mode

Doze mode is a lighter power-saving mode than the sleep modes, as all elements of the JN51xx device remain powered but the CPU is paused (CPU clock is stopped).

This low-power mode cannot be explicitly selected in **PWRM_vInit()**. The Power Manager will put the JN51xx device into Doze mode only in certain circumstances, described in Section 4.8.1 below. However, to enter Doze mode, the Power Manager must have been initialised using **PWRM_vInit()** and the power-saving modes must have been enabled using **PWRM_vManagePower()**.

4.8.1 Circumstances that Lead to Doze Mode

Although Sleep and Deep Sleep modes cannot be entered while there are activities running that must not be interrupted by sleep (see Section 4.5), the Power Manager can put the device into Doze mode while the activity counter is non-zero.

Even when the activity counter is zero, if a sleep mode has been configured with the 32-kHz oscillator running (see Section 4.3) but no wake event has been scheduled (see Section 4.6), the Power Manager will put the device into Doze mode instead of Sleep mode.

The decision to put a device into a Sleep mode or Doze mode is illustrated in the flowchart in Figure 7 below.

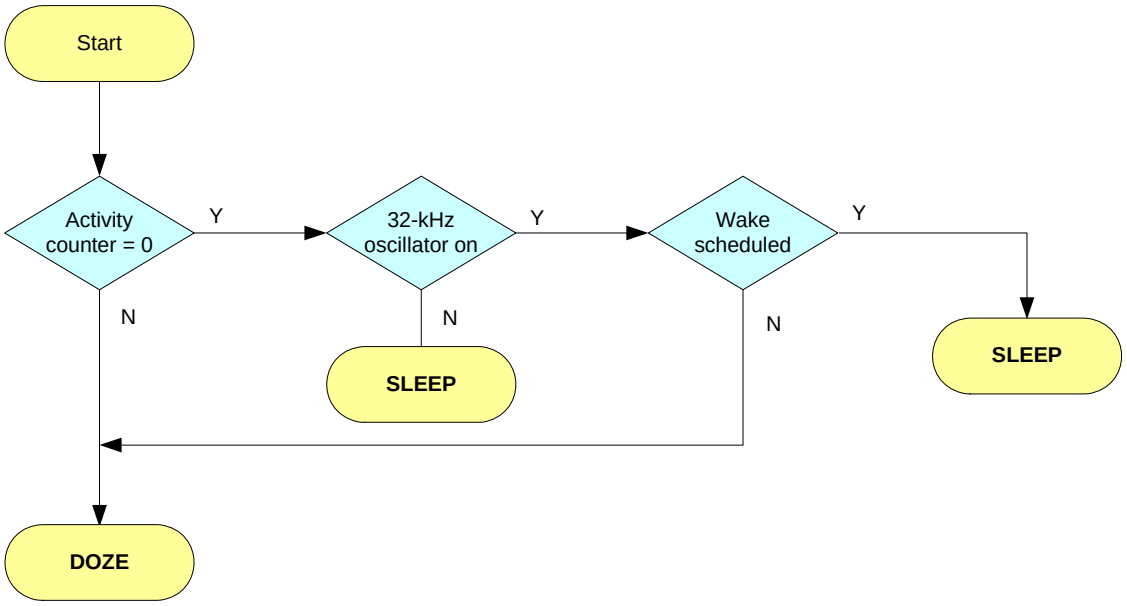


Figure 7: Flowchart of Decision to Enter Doze Mode

4.8.2 Doze Mode Monitoring During Development

Depending on the circumstances described in Section 4.8.1, the JN51xx device may spend a significant proportion of its time in Doze mode. The Power Manager API provides functions that can be used to investigate the fraction of time that the JN51xx device typically spends in Doze mode for a given application. These functions can be used when the application is running in debug mode.

The function **PWRM_vSetupDozeMonitor()** must first be called to configure the Doze mode monitor and start a monitoring session. This function allows two timers to be enabled:

- **Elapsed timer:** This timer measures the total elapsed time of the current monitoring session.
- **Doze timer:** This timer measures the amount of time the device spends in Doze mode during the current monitoring session.

The timers can be configured to have a clock cycle of 1 µs or 32 µs. This value determines the time resolution of the results.

To obtain sensible results, the doze monitor must then be allowed to run for some time before the following two functions are called:

- **PWRM_u32GetDozeElapsedTime()** obtains the current value of the elapsed timer (the amount of time since the current monitoring session began).
- **PWRM_u32GetDozeTime()** obtains the current value of the doze timer (the amount of time spent in doze mode during the current monitoring session).

The fraction of time that the JN51xx device spends in Doze mode can then be estimated by dividing the return value of **PWRM_u32GetDozeTime()** by the return value of **PWRM_u32GetDozeElapsedTime()**.

The use of the above functions in doze monitoring is illustrated in Figure 8.

Note: The two timers can be reset at any time using the function **PWRM_vResetDozeTimers()**.

Note: When **PWRM_vSetupDozeMonitor()** is called, you can optionally enable output of the doze state on DIO1 pin of the JN51xx device - that is, the state of the pin will reflect the doze state of the device. This allows you to make doze state measurements externally.

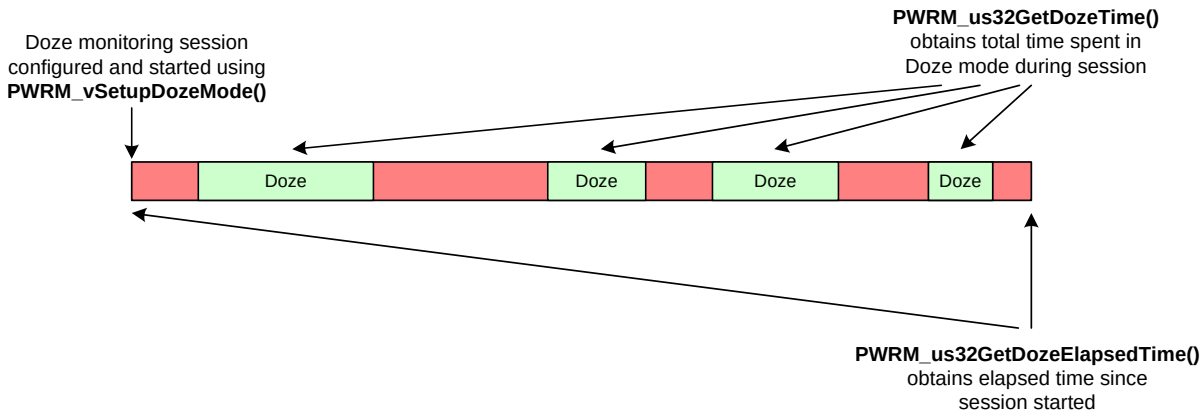


Figure 8: Doze Monitoring

5. Protocol Data Unit Manager (PDUM)

Communication between nodes in a wireless network is implemented using messages which contain application data. The part of a message which contains this data is called the Application Protocol Data Unit (APDU). The Protocol Data Unit Manager (PDUM) is concerned with APDU memory management, and assembling and disassembling APDUs - that is, inserting data into APDUs to be transmitted and extracting data from received APDUs.

5.1 Message Assembly and Disassembly

A message travels over a wireless network as a packet (usually an 802.15.4 packet) containing application data surrounded by header and footer information relating to the different layers of the protocol stack.

A message to be sent is prepared at the application level, at the top of the protocol stack, by creating an Application Protocol Data Unit (APDU) containing the application data to be included in the message. This APDU is then passed down the layers of the stack, with each layer adding its own protocol information to the header and footer. On reaching the 'physical' layer at the bottom of the stack, the message is complete and ready to be transmitted.

For transmission, the message is converted to an NPDU (Network Protocol Data Unit). If the length of the message is greater than the packet size used in network communication (e.g. 802.15.4 packet size), the message is divided up and transmitted in multiple NPDUs (Network Protocol Data Units). You will need to be aware of this if using a sniffer to detect transmitted packets.

Note: Data is stored in memory in the JN51xx device in big-endian byte order but is transmitted over the network in little-endian byte order.

A received message is passed up the protocol stack, with each stack layer stripping out the corresponding protocol information from the header and footer. On reaching the application level, only the APDU remains. The application data can then be extracted from this APDU.

The assembly and disassembly of a message, described above, are illustrated in the figure below, in which the lower stack layers (MAC and Physical) are provided by the IEEE 802.15.4 protocol.

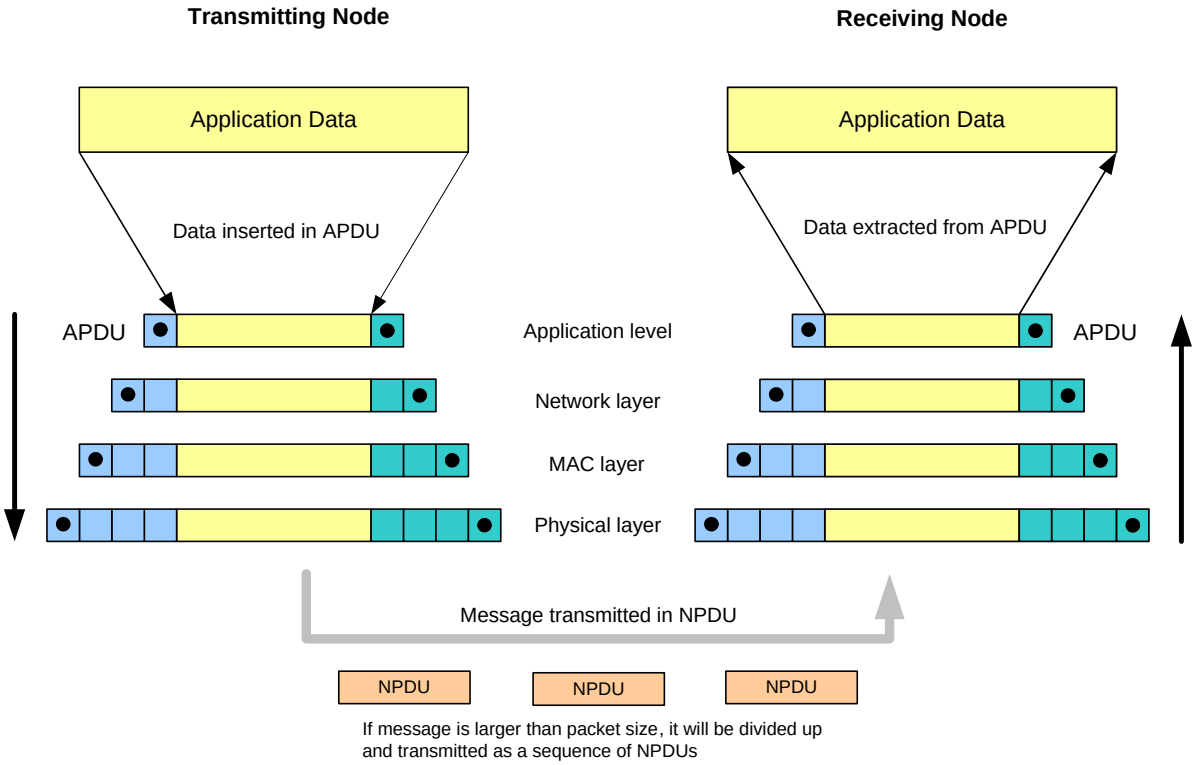


Figure 9: Message Assembly and Disassembly

5.2 Preparing the PDU Manager

In order to use the PDU Manager:

- You must statically define the required APDUs using the ZPS Configuration Editor (described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*). Each APDU is given a unique handle. While the data payload of an APDU can be of arbitrary length, a maximum length is set for an APDU.
- Before calling any other PDUM functions in your code, you must call the function **PDUM_vInit()** to initialise the PDU Manager.

5.3 Inserting Data into Outgoing Message

When sending a message to another node, you must first create an APDU containing the application data to be sent. To do this, first allocate an APDU instance by calling the function **PDUM_hAPduAllocateAPduInstance()** and then populate the APDU instance with data using **PDUM_u16APduInstanceWriteNBO()**, in which you must specify:

- the handle of the APDU instance in which data is to be inserted (this is the handle returned by **PDUM_hAPduAllocateAPduInstance()**)
- the starting position of the data in the APDU - that is, the position of the least significant data byte
- the format of the data payload - the data can be made up of a sequence of data values of different types
- the data values to be inserted in the data payload

Alternatively, the function **PDUM_u16APduInstanceWriteStrNBO()** can be used to populate the APDU instance - this function allows a data structure to be inserted into the APDU.

You must then use the relevant ZigBee PRO API function to send the message - refer to the *ZigBee PRO Stack User Guide (JN-UG-3048)*. Once the message has been sent, the ZigBee PRO stack automatically de-allocates the memory-space used for the APDU instance.

Note that **PDUM_u16APduInstanceWriteNBO()** performs the necessary data conversion from big-endian byte order to little-endian byte order for transmission.

Alternatively, you can produce your own code to insert data into the payload of an APDU. To help you, two functions are provided:

- **PDUM_pvAPduInstanceGetPayload()**: This function returns a pointer to the start of the payload section of the APDU instance.
- **PDUM_eAPduInstanceSetPayloadSize()**: This function sets the size, in bytes, of the data payload.

Caution: *Data must be stored in memory in big-endian order but is transmitted over the network in little-endian byte order. Therefore, if you use your own code to insert data into an APDU, you must reverse the byte order of the data before inserting it. Failure to change the endianness of the data will result in an alignment exception.*

5.4 Extracting Data from Incoming Message

The function **PDUM_u16APduInstanceReadNBO()** provides an easy way of extracting the data payload from an incoming message, which is received using the RTOS function **OS_eCollectMessage()** - refer to Section 2.4.8. The **PDUM_u16APduInstanceReadNBO()** function requires the following to be specified:

- the handle of the APDU instance containing the data to be extracted (this is the handle contained in the ZPS_EVENT_AF_DATA_INDICATION stack event which notified the application of the arrival of the data message)
- the starting position of the data in the APDU - that is, the position of the least significant data byte
- the format of the data payload - the data can be made up of a sequence of data values of different types
- a pointer to a structure in which the extracted data will be stored

Once the data has been extracted, you should de-allocate the memory space used for the APDU instance by calling the function **PDUM_eAPduFreeAPduInstance()**.

Note that **PDUM_u16APduInstanceReadNBO()** performs the necessary data conversion from little-endian byte order to big-endian byte order for storage.

Alternatively, you can produce your own code to extract the payload data from an APDU. To help you, two functions are provided:

- **PDUM_pvAPduInstanceGetPayload()**: This function returns a pointer to the start of the payload data in the APDU instance.
- **PDUM_u16APduInstanceGetPayloadSize()**: This function returns the size, in bytes, of the data payload.

Caution: *Data is received from the network in little-endian byte order, but must be stored in memory in big-endian order. Therefore, if you use your own code to extract data from an APDU, you must reverse the byte order of the data before storing it. Failure to change the endianness of the data will result in an alignment exception.*

6. Debug (DBG) Module

This chapter describes the Debug (DBG) module which allows application code to be debugged by means of diagnostic messages that are output to a display device.

6.1 Overview

The Debug module comprises an API containing diagnostic functions that can be embedded in your application code. Application debugging using the Debug module requires the JN51xx device to be connected to a display device (such as a PC) via an IO interface (such as one of the on-chip UARTs). The display device must provide a dumb terminal through which output from the JN51xx device can be viewed. A typical implementation is illustrated in the figure below.

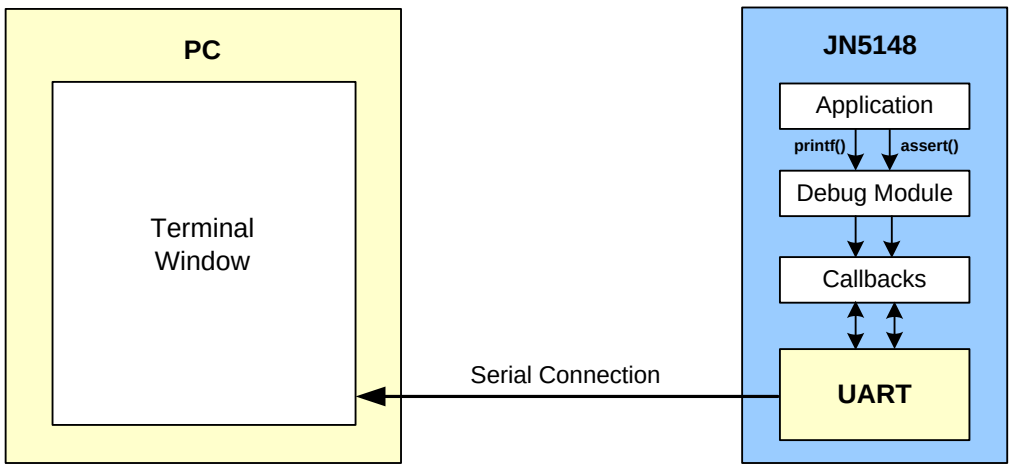


Figure 10: Typical Hardware Set-Up for Debugging

The API provides the essential printf- and assert-style debug functions, which can be strategically placed in your code:

- **DBG_vPrintf()** is used to output formatted strings and data values at an appropriate point during program execution, in order to indicate progress.
- **DBG_vAssert()** is used to test a logical condition, and to stop program execution if the test fails (condition is FALSE).

User-specified callback functions are used by the Debug module to control the IO interface (see Section 6.3).

6.2 Enabling the Debug Module

The Debug module API is defined in the header file **DBG.h**, which must be included in your code.

In order to use the Debug module, it must be explicitly enabled at build time by defining DBG_ENABLE in the build - for example, by adding -DDBG_ENABLE to the compiler. If the module is not enabled in this way, all the Debug functions embedded in your code will be ignored.

In addition, the functions **DBG_vPrintf()** and **DBG_vAssert()** each include a Boolean parameter which can be used to enable/disable individual instances of these functions. Two or more instances of these functions can be grouped to form a 'stream' for which this Boolean parameter is a common constant used to enable/disable the whole function group. This constant can be set at build time (see Section 6.4).

Tip: By default, the Debug module will display each 'printf line' as passed. However, if DBG_VERBOSE is defined at build time then each line displayed will be prefixed with the file name and line number of the debug statement.

6.3 Initialising and Configuring the Debug Module

The way that the Debug module is configured and initialised depends on the serial IO interface which is to be used to output debug information from the JN51xx device to the attached PC:

- If a JN51xx UART is to be used for output, the required initialisation/ configuration is as described in Section 6.3.1. This option will be taken by most users.
- If any other serial IO interface is to be used for output, the required initialisation/ configuration is as described in Section 6.3.2.

6.3.1 Using JN51xx UART Output

When a JN51xx UART is to be used for the output of debug information, the configuration and initialisation of the Debug module is accomplished with a single call to the function **DBG_vUartInit()**, which allows selection of the UART (0 or 1) and the baud-rate to be used. This function is used both during a cold start of the JN51xx device and during a warm start (where the latter is a device re-start with memory contents retained).

6.3.2 Using Alternative Serial Output

When an alternative to an on-chip UART is to be used for the output of debug information, the required IO interface must first be configured and enabled (using the relevant functions from the JN51xx Integrated Peripherals API).

The Debug module must then be initialised using the function **DBG_vInit()**. This function is used both during a cold start of the JN51xx device and during a warm start (where the latter is a device re-start with memory contents retained). The function takes as input a structure which contains pointers to four callback functions needed for debugging:

```
typedef struct
  {
   void (*prInitHardwareCb)(void);
   void (*prPutchCb)       (char c);
   void (*prFlushCb)       (void);
   void (*prFailedAssertCb)(void);
} tsDBG_FunctionTbl;
```

The callback functions are user-defined and are described in the table below.

Pointer	Callback Function
**prInitHardwareCb*	Function which re-initialises the IO interface after a warm start, e.g. when JN51xx device wakes from sleep.
**prPutchCb*	Function used by **DBG_vPrintf()** to output a single character to the IO interface.
**prFlushCb*	Function used by **DBG_vPrintf()** to flush the IO interface buffer to allow buffered output characters to be displayed. If the output is unbuffered, this function should do nothing or wait for the last character output using the **putch()** function to be made available. Note that the function should not append a newline character, as this should be handled by the formatting string passed to **DBG_vPrintf()**.
**prFailedAssertCb*	Function which is called when **DBG_vAssert()** fails. The function should stop execution and may reset the device.

Table 2: Callback Functions Specified in DBG_vInit()

6.4 Example Diagnostic Code

The following code fragment illustrates use of the Debug module API. The JN51xx UART 0 is used. Two debug 'streams' (1 and 2) are used to separately enable/disable two groups of debug lines.

```
#include <jendefs.h>
#include "DBG.h"
#include "DBG_Uart.h"

#ifndef DBG_STREAM_1
#define DBG_STREAM_1 FALSE
#endif

#ifndef DBG_STREAM_2
#define DBG_STREAM_2 FALSE
#endif

void appColdStart(void)
{
int i = 0;

/* Initialise the standard UART hardware */
DBG_vUartInit(DBG_E_UART_0, DBG_E_UART_BAUD_RATE_115200);

/* Now we can use DBG_vPrintf() and DBG_vAssert() to output characters
   to the UART device */
DBG_vPrintf(DBG_STREAM_1, "Printing to stream 1\n");
DBG_vPrintf(DBG_STREAM_2, "Printing an integer %i to stream 2\n", 10);
DBG_vAssert(DBG_STREAM_1, i == 1);
}
```

When building this application, you have following options:

- Debug disabled (the default)
- Debug enabled only for stream 1 - build with:

  `-DDBG_ENABLE -DDBG_STREAM_1=TRUE`
- Debug enabled only for stream 2 - build with:

  `-DDBG_ENABLE -DDBG_STREAM_2=TRUE`
- DBG enabled for both streams - build with:

  `-DDBG_ENABLE -DDBG_STREAM_1=TRUE -DDBG_STREAM_2=TRUE`

Part II:
Reference Information

7. Real-time Operating System (RTOS) API

The chapter contains descriptions of the macros and functions of the JenOS RTOS (Real-Time Operating System) API. The API is defined in the header files **os.h** and **os_lib.h**. The Overlay functions are defined separately in the header file **ovly.h**.

- The RTOS macros are described in Section 7.1.
- The RTOS functions are described in Section 7.2.
- The Overlay functions are described in Section 7.3.

Caution: *To control interrupts, you should only use the functions supplied as part of the RTOS.*

7.1 RTOS Macros

The section contains descriptions of the macros of the RTOS API. These macros are used to define user tasks, ISRs (Interrupt Service Routines) and callback functions (related to the hardware counter and associated software timers), and are listed below.

OS_TASK

OS_TASK(*taskname***)**

Description

This macro is used in the application code to define a user task. The name of the user task must be specified, where this task name has been declared in advance using the JenOS Configuration Editor (see Section 14.4.1).

The macro is followed by the task body, as illustrated in the code fragment below:

```
OS_TASK(TaskA) {
    task body …
}

OS_TASK(TaskB) {
    task body …
}
```

Parameters

taskname	Name of user task

OS_ISR

OS_ISR(*ISRname***)**

Description

This macro is used in the application code to define an ISR (Interrupt Service Routine) which will be invoked when the corresponding interrupt occurs. The name of the ISR must be specified, where this ISR name has been declared in advance using the JenOS Configuration Editor (see Section 14.4.2). The association between this ISR and an interrupt source is also configured in this file.

The macro is followed by the ISR body, as illustrated in the code fragment below:

```
OS_ISR(TickTimerISR) {
    ISR body …
}
```

Parameters

ISRname	Name of ISR

OS_SWTIMER_CALLBACK

OS_SWTIMER_CALLBACK(*CBname***,** *pData***)**

Description

This macro is used to define a 'software timer expiry' callback function. This user-defined function will be automatically invoked when a software timer expires which has been started using the function **OS_eStartSWTimer()** - that is, when the timer reaches the number of ticks specified in the start function. The callback function is called by the function **OS_eExpireSWTimers()** (as an alternative to activating a user task on expiry of the software timer).

The name of the callback function must be specified in the macro, where this name has been declared in advance using the JenOS Configuration Editor (see Section 14.5.1). A pointer to a buffer can optionally be specified, where this buffer will contain any data to be processed by the callback function. This data is passed to the callback function through a parameter of **OS_eStartSWTimer()**.

The macro is followed by the callback function definition, as illustrated in the code fragment below:

```
OS_SWTIMER_CALLBACK(MyCallback, pvMyData)
{
   Callback definition...
}
```

Parameters

CBname	Name of callback function
pData	Pointer to buffer containing data to be used by callback function (if no data is needed, specify a valid dummy pointer)

OS_HWCOUNTER_ENABLE_CALLBACK

OS_HWCOUNTER_ENABLE_CALLBACK(*CBname***)**

Description

This macro is used to define a callback function to enable and start the hardware counter which may be used to drive one or more software timers. This user-defined function will be automatically invoked when a software timer is started using the function **OS_eStartSWTimer()**, provided that there are no other software timers already active that use the hardware counter.

The name of the callback function must be specified in the macro, where this name has been declared in advance using the JenOS Configuration Editor (see Section 14.5.1).

The macro is followed by the callback function definition, as illustrated in the code fragment below:

```
OS_HWCOUNTER_ENABLE_CALLBACK(EnableTickTimer)
{
   Callback definition...
}
```

Parameters

CBname	Name of callback function

OS_HWCOUNTER_DISABLE_CALLBACK

OS_HWCOUNTER_DISABLE_CALLBACK(*CBname***)**

Description

This macro is used to define a callback function to stop and disable the hardware counter which may be used to drive one or more software timers. This user-defined function will be automatically invoked:

- when a software timer is stopped using the function **OS_eStopSWTimer()**
- by the function **OS_eExpireSWTimers()** when a software timer expires

provided that there are no other software timers still active that use the hardware counter.

The name of the callback function must be specified in the macro, where this name has been declared in advance using the JenOS Configuration Editor (see Section 14.5.1).

The macro is followed by the callback function definition, as illustrated in the code fragment below:

```
OS_HWCOUNTER_DISABLE_CALLBACK(DisableTickTimer)
{
   Callback definition...
}
```

Parameters

CBname	Name of callback function

OS_HWCOUNTER_SET_CALLBACK

OS_HWCOUNTER_SET_CALLBACK(*CBname***,** *CompValue***)**

Description

This macro is used to define a callback function to set the value in the compare register for the hardware counter which may be used to drive one or more software timers. The compare register contains a value with which the incrementing counter value can be compared - action may be taken when the two values match.

This user-defined callback function is normally used to set a compare register value for the next software timer expiry point. Since the hardware counter increments continuously (and loops around), the compare register must be set to a value equal to the current counter value plus the number of ticks until the next expiry point.

The callback function will be automatically invoked when a software timer is started using the function **OS_eStartSWTimer()** or re-started using the function **OS_eContinueSWTimer()**. It is also called by **OS_eExpireSWTimers()** when a software timer expires and the compare register must be updated for the next software timer to expire (if any).

The name of the callback function must be specified in the macro, where this name has been declared in advance using the JenOS Configuration Editor (see Section 14.5.1). The required compare register value must also be specified.

The callback definition should follow the format below:

```
OS_HWCOUNTER_SET_CALLBACK(SetTickTimerCompare, u32CompVal)
{
     if (u32CompVal is in the future) {
         /* set the tick timer compare register */
         return TRUE;
     } else { /* compare value is in the past */
         /* expire timer immediately */
         return FALSE;
     }
}
```

Parameters

CBname	Name of callback function
CompValue	Value to set in the compare register for the hardware counter

OS_HWCOUNTER_GET_CALLBACK

OS_HWCOUNTER_GET_CALLBACK(*CBname***)**

Description

This macro is used to define a callback function to obtain the current value of the hardware counter which may be used to drive one or more software timers. The function must return a **uint32** containing the hardware counter value.

The name of the callback function must be specified in the macro, where this name has been declared in advance using the JenOS Configuration Editor (see Section 14.5.1).

The macro is followed by the callback function definition, as illustrated in the code fragment below:

```
OS_HWCOUNTER_GET_CALLBACK(GetCurrentCount)
{
   Callback definition...
}
```

Parameters

CBname	Name of callback function

7.2 RTOS Functions

This section contains descriptions of the functions of the RTOS API. The API functions are sub-divided into the following areas, which are covered in separate sub-sections:

Functional Area	Section Reference
Initialisation	Section 7.2.1
User Tasks	Section 7.2.2
Interrupts	Section 7.2.3
Mutex	Section 7.2.4
Messaging	Section 7.2.5
Software Timers	Section 7.2.6
Overlays	Section 7.3

7.2.1 Initialisation Functions

This section describes the RTOS initialisation functions.

The functions are listed below, along with their page references:

OS_vStart

```
void OS_vStart(void (*prvInitFunction)(void),
               void (*prvUnclaimedIRQ)(void),
void (*prvErrorHook)(OS_teStatus, void *osHandle));
```

Description

This function is used to start the RTOS.

Three optional user-defined functions can be specified:

- An initialisation function for the hardware and user application, e.g. initialises CPU peripherals and enables interrupt generation flags. This function is called with all controlled interrupts disabled. On completion of this function, all controlled interrupts are enabled.
- A function to capture any unclaimed interrupts that occur but do not have an ISR configured for them.
- A function to handle OS errors, as described in Section 2.6.

On returning from **OS_vStart()**, the idle task is entered.

Caution: *The user-defined initialisation function must not perform any operations that require the use of interrupts, since interrupts are not enabled until the RTOS has started, which occurs just before the function* ***OS_vStart()*** *returns.*

Parameters

***prvInitFunction**	Optional pointer to a user-defined initialisation function. If not used, this parameter must be set to NULL.
***prvUnclaimedIRQ**	Optional pointer to user-defined function to capture unclaimed interrupts. If not used, this parameter must be set to NULL.
***prvErrorHook**	Optional pointer to user-defined callback function to handle OS errors. Only required if error hook is set to true in the JenOS configuration editor.

Returns

None

OS_vRestart

```
void OS_vRestart(void);
```

Description

This function is used to restart the RTOS following a warm start with memory held. The function re-initiates the interrupt hardware.

Parameters

None

Returns

None

7.2.2 User Task Functions

This section provides descriptions of the RTOS API functions concerned with user tasks.

The functions are listed below, along with their page references:

OS_eActivateTask

OS_teStatus OS_eActivateTask(OS_thTask *hTask***);**

Description

This function is used to activate the specified user task - that is, to move the task from the dormant state to the pending state (i.e. ready to run). When it becomes the highest priority pending task, it will then execute.

If the task is already pending then its activation count is incremented. The activation count keeps track of the number of times the task must be executed. Once the task has been executed, the activation count is decremented. If this count reaches zero, the task is moved back to the dormant state, otherwise the task returns to the pending state.

A handle for the task is pre-defined using the JenOS Configuration Editor.

Parameters

hTask	Handle of the user task to be activated

Returns

OS_E_OK (successful)

OS_E_BADTASK (invalid task handle used)

OS_E_OVERACTIVATION (maximum number of activations exceeded: 65535)

OS_eGetCurrentTask

OS_teStatus OS_eGetCurrentTask(OS_thTask ***phTask*);**

Description

This function is used to obtain the handle of the user task that is currently in the running state.

Parameters

**phTask*	Pointer to place to store task handle obtained

Returns

OS_E_OK (successful)

OS_BADVALUE (invalid pointer specified)

7.2.3 Interrupt Functions

This section provides descriptions of the RTOS API functions concerned with interrupts.

The functions are listed below, along with their page references:

Caution: *These interrupt functions must* ***not*** *be called from within an Interrupt Service Routine (ISR).*

Note: The RTOS cannot clear interrupts and it is the responsibility of the application to do this - refer to Appendix C.

OS_eDisableAllInterrupts

```
OS_teStatus OS_eDisableAllInterrupts(void);
```

Description

This function is used to disable all CPU interrupts - that is, both controlled and uncontrolled interrupts.

Note that nested calls to this function are not permitted and that the function cannot be called in an ISR.

These interrupts can be re-enabled using the function **OS_eEnableAllInterrupts()**.

Parameters

None

Returns

OS_E_OK (successful)

OS_eEnableAllInterrupts

```
OS_teStatus OS_eEnableAllInterrupts(void);
```

Description

This function is used to enable all CPU interrupts - that is, both controlled and uncontrolled interrupts.

Note that nested calls to this function are not permitted and that the function cannot be called in an ISR.

These interrupts can be disabled using the function **OS_eDisableAllInterrupts()**.

Parameters

None

Returns

OS_E_OK (successful)

OS_eSuspendOSInterrupts

```
OS_teStatus OS_eSuspendOSInterrupts(void);
```

Description

This function is used to disable interrupts managed by the RTOS - that is, controlled interrupts.

Note that nested calls to this function are permitted.

These interrupts can be subsequently re-enabled using the function **OS_eResumeOSInterrupts()**.

Parameters

None

Returns

OS_E_OK (successful)

OS_E_OSINTOVERFLOW (maximum nesting level exceeded: 0xFFFFFFFF)

OS_eResumeOSInterrupts

```
OS_teStatus OS_eResumeOSInterrupts(void)
```

Description

This function is used to re-enable interrupts managed by the RTOS - that is, controlled interrupts.

Note that nested calls to this function are permitted.

These interrupts can be disabled using the function **eSuspendOSInterrupts()**.

Parameters

None

Returns

OS_E_OK (successful)

OS_E_OSINTUNDERFLOW (too many resumes - unbalanced suspend/resumes)

7.2.4 Mutex Functions

This section provides descriptions of the RTOS API functions concerned with the mutex (Mutually Exclusive Activities) feature.

The functions are listed below, along with their page references:

OS_eEnterCriticalSection

OS_teStatus OS_eEnterCriticalSection(OS_thMutex *hMutex***);**

Description

This function is used at the start of a critical section of code, during which the currently running user task or ISR must not be pre-empted by another user task/ISR within the same “mutex” group (see below).

The function is paired with the function **OS_eExitCriticalSection()**, which is placed at the end of the critical section. Execution of the critical section will be allowed to complete before processing is switched to a higher priority task/ISR that may be pending within the same “mutex” group.

This mechanism, known as a “mutex”, applies to a group of user tasks or ISRs. Effectively, the mutex feature allows pre-set user task/ISR priorities to be over-ridden by temporarily assigning the highest priority in the mutex group to the task/ISR containing the critical section. The mutex feature is useful in allowing exclusive access to shared resources (e.g. hardware peripherals and shared memory). It allows the currently running user task/ISR to finish accessing a shared resource before processing is switched to a higher priority user task/ISR (from the same mutex group) that also needs to access the resource. In this way, access to a shared resource can be serialised.

Function calls to enter/exit the same critical section should not be nested. Nested calls to enter/exit different critical sections must be strictly nested. The critical section must be exited before termination of the task that uses it.

A handle for the mutex group is pre-defined using the JenOS Configuration Editor.

Parameters

hMutex	Handle of mutex group for critical section of code

Returns

OS_E_OK (successful)

OS_E_BADMUTEX (invalid mutex handle used)

OS_E_BAD_NESTING (bad nesting)

OS_eExitCriticalSection

OS_teStatus OS_eExitCriticalSection(OS_thMutex *hMutex***);**

Description

This function must be used at the end of a critical section of code during which the current user task or ISR cannot be pre-empted by another user task/ISR from the same mutex group.

The function is paired with the function **OS_eEnterCriticalSection()**, which is placed at the start of the critical section. Execution of the critical section will be allowed to complete before processing is switched to a higher priority user task/ISR that may be pending within the same mutex group.

The handle of the mutex group to which the critical section belongs must be specified in this function and must be the same as the handle used in the matching call to **OS_eEnterCriticalSection()**.

Function calls to enter/exit the same critical section should not be nested. Nested calls to enter/exit different critical sections must be strictly nested. The critical section must be exited before termination of the task that uses it.

Parameters

hMutex	Handle of mutex group for critical section of code

Returns

OS_E_OK (successful)

OS_E_BADMUTEX (invalid mutex handle used)

OS_E_BAD_NESTING (bad nesting)

7.2.5 Messaging Functions

This section provides descriptions of the RTOS API functions concerned with sending and collecting inter-task messages.

The functions are listed below, along with their page references:

OS_ePostMessage

OS_teStatus OS_ePostMessage(OS_thMessage *hMessage*,
**void ****pvData*);

Description

This function sends a message of the specified type, possibly containing user data, to another user task or ISR.

The message type must be defined in the header file **os_msg_types.h.** The identity of the destination task or ISR, and whether the message type is queued, are pre-configured using the JenOS Configuration Editor (see Section 14.6.2). The data content for a message type can be of arbitrary size, including zero length (no data).

- If the data content has non-zero length, the sent message may possibly be queued with other messages of the same type (depending on how the message type has been configured).
- If the data content has zero length, the sent message is not queued (regardless of how the message type has been configured).

If a sent message is not queued, the message overwrites any previous, uncollected message of the same type.

This function supports one-to-one and many-to-one communications, but does not support many-to-many communications.

Messages posted with this function are collected by the destination task using the function **OS_eCollectMessage()**.

Note: A message sent with this function can alternatively activate the destination task or invoke a callback function to perform user actions. These options are configured using the JenOS Configuration Editor.

Parameters

hMessage	Handle of message type
***pvData**	Pointer to data to be sent in message. Set to NULL for a message with no data

Returns

OS_E_OK (successful)

OS_E_BADMESSAGE (invalid message type handle specified)

OS_E_QUEUE_FULL (message queue is full)

OS_eCollectMessage

```
OS_teStatus OS_eCollectMessage(
                    OS_thMessage hMessage,
                    void *pvData);
```

Description

This function is used to collect a posted message of the specified message type. Any extracted message data is placed in the specified location.

The function is used to collect a message sent using the function **OS_ePostMessage()**.

If the message is unqueued, it will always be successfully collected.

Parameters

hMessage	Handle of message type of message to collect
***pvData*	Pointer to place where extracted message data will be stored. Set to NULL for a message with no data

Returns

OS_E_OK (successful)

OS_E_BADMESSAGE (invalid message handle specified)

OS_E_BADVALUE (invalid data pointer specified)

OS_E_QUEUE_EMPTY (message queue contains no messages)

OS_eGetMessageStatus

```
OS_teStatus OS_eGetMessageStatus(
                    OS_thMessage hMessage);
```

Description

This function is used to obtain the status of the incoming data message queue for the specified message type. The returned value indicates whether the queue contains messages or is empty.

Parameters

hMessage	Handle of message type for message queue to check

Returns

OS_E_BADMESSAGE (invalid message handle specified)

OS_E_QUEUE_EMPTY (message queue contains no messages)

OS_E_QUEUE_FULL (message queue contains uncollected messages)

OS_E_UNQUEUED (message type is not queued)

7.2.6 Software Timer Functions

This section provides descriptions of the RTOS API functions concerned with the software timers that can be used to schedule the start of an activity within a user task or ISR.

The functions are listed below, along with their page references:

Note 1: Some of the above software timer functions call user-defined callback functions that must be defined using macros described in Section 7.1. The required callback functions and associated macros are mentioned in the function descriptions in this section.

Note 2: If the tick timer is used as the source counter and the maximum count of the tick timer is *T* (before the tick timer wraps around), there must be no more than *T/2* ticks between consecutive software timer expiry events (e.g. if the tick timer wraps around every 60 seconds, a software timer must expire every 30 seconds or less).

Caution: *To allow the JN51xx device to enter sleep mode, no software timers should be active. Any running software timers must first be stopped and any expired timers must be de-activated. Both can be achieved using the function* ***OS_eStopSWTimer()****, which must be called individually for each running and expired timer.*

OS_eStartSWTimer

OS_teStatus OS_eStartSWTimer(OS_thSWTimer *hSWTimer***,**
uint32 *u32Ticks***,**
void * *pvData***);**

Description

This function is used to start the specified software timer and configure it to expire after the specified number of ticks. The software timer is defined and given a handle using the JenOS Configuration Editor (see Section 14.5.2). It is derived from a source counter - this could be a hardware counter, such as the on-chip tick timer, or another software timer. The source counter used determines the tick period and therefore the timed period.

In this function, you must specify the number of ticks of the source counter until the software timer expires. If the on-chip tick timer is used as the source counter, the specified value must not be greater than half the maximum count of the tick timer (when the tick timer wraps around) - since the tick timer is a 32-bit counter, you must not specify a value greater than 2147483647 ticks (0x7FFFFFFF ticks).

Using the JenOS Configuration Editor, it is possible to configure a task or callback function to be executed on expiry of the software timer. A parameter is provided which allows a pointer to data to be specified, which will be used by a callback function on expiry of the timer.

This function calls the user-defined 'hardware counter enable' callback function, defined using the macro **OS_HWCOUNTER_ENABLE_CALLBACK()**, provided that there are no other software timers already active that use the hardware counter. The function also calls the following user-defined callback functions:

- 'hardware counter get' function, defined using the macro **OS_HWCOUNTER_GET_CALLBACK()**, which obtains the current value of the hardware counter
- 'hardware counter set' function, defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**, which sets the hardware counter's compare register to the appropriate value for the timed duration (equal to the specified number of ticks added to the current value of the hardware counter previously obtained)

Parameters

hSWTimer	Handle of software timer to start
u32Ticks	The number of ticks before software timer expires (if using on-chip tick timer, refer to above description)
**pvData*	Pointer to data to be passed to optional callback function on expiry of timer (NULL if not required). Ignored for tasks

Returns

OS_E_OK (successful)

OS_E_BADSWTIMER (invalid software timer handle)

OS_E_SWTIMER_RUNNING (software timer already running)

OS_eStopSWTimer

OS_teStatus OS_eStopSWTimer(OS_thSWTimer *hSWTimer***);**

Description

This function is used to de-activate the specified software timer - this can be a running timer or an expired timer.

This function calls the user-defined 'hardware counter disable' callback function, defined using the macro **OS_HWCOUNTER_DISABLE_CALLBACK()**, provided that there are no other software timers still active that use the hardware counter.

Parameters

hSWTimer	Handle of software timer to de-activate

Returns

OS_E_OK (successful)

OS_E_BADSWTIMER (invalid software timer handle)

OS_E_SWTIMER_STOPPED (software timer already de-activated)

OS_eExpireSWTimers

```
OS_teStatus OS_eExpireSWTimers(
                    OS_thHWCounter hHWCounter);
```

Description

This function is used to expire scheduled software timers associated with the specified source counter, where these timers have previously reached their target counts (source counter has matched the value in the counter's compare register).

The function should be called in the source counter's ISR that is invoked when the above match occurs. The function:

- sets the status of the software timer to 'expired' by calling the user-defined callback function that is defined using the macro **OS_SWTIMER_CALLBACK()**
- checks whether there are any other pending software timers and:
  - if there is at least one pending software timer, the function updates the source counter's compare register with the required number of ticks (until the next software timer expires) by calling the user-defined callback function that is defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**
  - if there are no pending software timers, the function does nothing to the source counter (leaves it running)

Before this function returns, it deals with any software timers that expire in quick succession (without the need to re-call the function for the individual software timers).

For more information on the software timers, refer to Section 2.4.6.

Parameters

hHWCounter	Handle of counter

Returns

OS_E_OK (successful)

OS_E_BADHWCOUNTER (invalid counter handle)

OS_E_NOTHINGTOEXPIRE (no software timers to expire)

OS_E_OVERACTIVATION (associated task activated too many times)

OS_E_BADTASK (invalid task handle used)

OS_E_HWCOUNTERIDLE (no active timers)

OS_eContinueSWTimer

```
OS_teStatus OS_eContinueSWTimer(
                    OS_thSWTimer hSWTimer,
                    uint32 u32Ticks,
                    void *pvData);
```

Description

This function is used to re-start the specified software timer, if it has previously expired, and configure it to expire again after the specified number of ticks. The timer will be re-started without any discontinuity between the previous expiry point and the new run, provided that the function is executed before the next timer cycle is complete.

The function is designed to be used immediately following the expiry of the timer. It should not be called a long time after the previous expiry (0x7FFFFFFF ticks). **OS_eStartSWTimer()** should be used in these circumstances.

In this function, you must specify the number of ticks of the source counter until the software timer expires. If the on-chip tick timer is used as the source counter, the specified value must not be greater than half the maximum count of the tick timer (when the tick timer wraps around) - since the tick timer is a 32-bit counter, you must not specify a value greater than 2147483647 ticks (0x7FFFFFFF ticks).

It is possible to specify a pointer to data which will be used if the timer is configured to invoke a callback function on expiry (the callback function is specified using the JenOS Configuration Editor).

If the specified software timer will be the next to expire, this function calls the user-defined 'hardware counter set' callback function, defined using the macro **OS_HWCOUNTER_SET_CALLBACK()**, which sets the hardware counter's compare register to the appropriate value for the remainder of the timed duration.

Parameters

hSWTimer	Handle of software timer to re-start
u32Ticks	The number of ticks before software timer expires
**pvData*	Pointer to data to be passed to optional callback function on expiry of timer (NULL if not required)

Returns

OS_E_OK (successful)

OS_E_BADSWTIMER (invalid software timer handle)

OS_E_SWTIMER_RUNNING (software timer already running)

OS_eGetSWTimerStatus

```
OS_teStatus OS_eGetSWTimerStatus(
                    OS_thSWTimer hSWTimer);
```

Description

This function is used to obtain the status of the specified software timer. The timer is reported as running, stopped or expired.

Parameters

hSWTimer	Handle of software timer

Returns

OS_E_BADSWTIMER (invalid software timer handle)

OS_E_SWTIMER_RUNNING (software timer is running)

OS_E_SWTIMER_STOPPED (software timer has been stopped)

OS_E_SWTIMER_EXPIRED (software timer has expired)

7.3 Overlay Functions (JN514x only)

This section provides descriptions of the RTOS API functions concerned with the use of overlays, introduced in Section 2.5. Overlays are an optional feature which, if required, must be explicitly enabled as described in Section 2.5.2. They are currently designed for use with the NXP ZigBee PRO stack and profile software.

The functions are listed below, along with their page references:

OVLY_bInit

bool_t OVLY_bInit(OVLY_tsInitData ***psInitData*);**

Description

This function initialises the overlay feature and must be called before any functions that are present in overlays are called, e.g. before the ZigBee PRO stack is initialised. The function installs the memory bus exception handler and initialises the NVM device. Initialisation data can be provided via a structure or default values can be used.

The initialisation data includes pointers to three user-defined callback functions.

The two callback functions for obtaining and releasing a mutex to protect the SPI bus (to which the NVM is connected) both have the same prototype:

```
void (*OVLY_prMutex)(void);
```

In particular, these mutex callback functions must be designed such that overlays are in the same mutex group as the PDM module (if used), in order to prevent the overlay feature and the PDM module from attempting to access the SPI bus at the same time.

The callback function for handling overlay-related events has the following prototype:

```
void (*OVLY_prEvent)(OVLY_teEvent eEventType, OVLY_tuEventData *psData);
```

where:

- `eEventType` is the type of overlay event from those detailed in Section 12.7
- `psData` is a pointer to the event data, as described in Section 12.8

Note that the overlay feature must also be enabled in the makefile for the application and in the RTOS configuration, as described in Section 2.5.2.

Parameters

psInitData	Pointer to a structure containing overlay initialisation data (see Section 12.6). To use default values, set to NULL

Returns

Outcome of overlay initialisation, one of:

TRUE - initialisation successful
FALSE - initialisation unsuccessful

OVLY_vReInit

```
void OVLY_vReInit(void);
```

Description

This function can be used to re-install the overlay exception handler, if it has been overwritten.

Parameters

None

Returns

None

OVLY_psProfilingInit

```
OVLY_tsProfiling *OVLY_psProfilingInit(void);
```

Description

This function initiates the overlay profiling feature which allows the use of overlays to be assessed during application execution - this is useful during application debug. The function returns a pointer to a structure containing:

- total number of bytes loaded from NVM (Flash memory) into RAM
- number of checksum failures
- array containing number of loads of each overlay

This structure will then be maintained and can be read by the application as and when required.

Note that this function will override any profiling callback function that was specified in the initialisation data when **OVLY_bInit()** was called.

Overlay profiling is described further in Section 2.5.3.

Parameters

None

Returns

Pointer to overlay profiling structure (for details of the structure, see Section 12.9)

8. Persistent Data Manager (PDM) API

This chapter describes the functions of the JenOS Persistent Data Manager (PDM) API. The API is defined in the header file **pdm.h**.

The PDM API functions are listed below, along with their page references:

Caution: *When using the Persistent Data Manager, do not use the JN51xx Integrated Peripherals API to interact with the Flash memory device connected to the JN51xx chip.*

PDM_vInit

```
void PDM_vInit(uint8 u8StartSector,
               uint8 u8NumSectors,
               uint32 u32SectorSize,
               OS_thMutex hPdmMutex,
               OS_thMutex hPdmMediaMutex,
               PDM_tsHwFncTable *psHwFuncTable,
               const tsReg128 *psKey);
```

Description

This function initialises the PDM module, and must be called during a cold start.

The function takes as input details of the sectors of the NVM (Non-Volatile Memory) device to be managed by the module, as well as a set of functions that the PDM will use to interact with the NVM device. These functions are specified in the structure detailed in Section 12.1.

Optional mutexes can be specified in order to:

- Serialise PDM function calls - if specified, this mutex is automatically applied during a PDM function call to prevent concurrent PDM function calls
- Serialise SPI bus access - if specified, this mutex is automatically applied during an access to NVM via the SPI bus to prevent concurrent accesses to the SPI bus (useful if other resources are also accessible via the SPI bus)

If a mutex is used, the user task must be linked to the relevant mutex in the JenOS Configuration Editor - refer to Section 14.7. A mutex is required if using both the PDM module and the overlay feature (see Section 2.5.1), as both features use the SPI bus - in this case, the PDM module and the overlay feature must be in the same mutex group.

The function also allows you to specify a key to be used by the PDM module to encrypt and decrypt saved data. An option is available to use a key stored in eFuse. Security based on this encryption key is applied to the context data that is automatically stored/restored by the stack, but security must be enabled for individual application data records when **PDM_eLoadRecord()** is called.

PDM_vInit() will auto-detect the NVM device type (manufacturer/model), unless you have already called **PDM_vSPIFlashConfig()** which allows you to specify a specific or custom SPI Flash device type. Note that if you have specified a set of NVM device functions in a call to **PDM_vSPIFlashConfig()**, you can set a NULL pointer to these functions in **PDM_vInit()**.

Parameters

u8StartSector	Number of the first sector of NVM to be managed by PDM module
u8NumSectors	Number of contiguous sectors of NVM to be managed by PDM module
u32SectorSize	Size of each sector, in bytes
hPdmMutex	Optional handle of the mutex to be used to serialise PDM function calls

hPdmMediaMutex	Optional handle of the mutex to be used to serialise access to NVM via the SPI bus. This mutex is not required when using internal EEPROM on JN516x
***psHwFuncTable*	Pointer to set of custom functions to be used with NVM device (see Section 12.1). Set to NULL if not needed
***psKey*	Pointer to structure containing the 128-bit encryption key to be used by the PDM module (see Section 12.5). A NULL pointer indicates that a key stored in eFuse is to be used (if no key is stored, a zero value will be used). The parameter should be set to NULL when using the internal EEPROM on JN516x. The internal EEPROM is physically protected inside the JN516x to the same degree as the unencrypted RAM record, so encryption is not required

Note: On the JN516x device, different parameter values must be used for internal EEPROM and external SPI Flash memory. The application can support both by having two **PDM_vInit()** calls selected by #ifdef PDM_EEPROM, as described in Section 3.9.

Returns

None

PDM_vSPIFlashConfig

```
void PDM_vSPIFlashConfig(
                    teFlashChipType eFlashType,
                    tSPIflashFncTable *psFlashFuncTable);
```

Description

This function should be used if the NVM device is an SPI Flash device and you do not wish to auto-detect the type of Flash device (manufacturer/model) - for example, if you are using an unsupported or custom Flash device.

If you wish to auto-detect the device type, you do not need to call this function, as auto-detection is implemented by default.

If required, this function must be called before **PDM_vInit()**.

The function requires you to specify the Flash device type. If a custom Flash device is selected (E_FL_CHIP_CUSTOM), you also need to specify a table of custom functions for the device. The structure through which these functions are specified are described in Section 12.2.

Parameters

eFlashType	Flash device type, one of: E_FL_CHIP_ST_M25P10_A E_FL_CHIP_SST_25VF010 E_FL_CHIP_ATMEL_AT25F512 E_FL_CHIP_CUSTOM E_FL_CHIP_AUTO
***psFlashFuncTable**	Pointer to custom function table (see Section 12.2), for case when a custom Flash device has been selected (E_FL_CHIP_CUSTOM). If a supported device has been selected, set this pointer to NULL.

Returns

None

PDM_eLoadRecord (JN514x only)

```
PDM_teStatus PDM_eLoadRecord(
                    PDM_tsRecordDescriptor *psDesc,
                    const char acName[PDM_NAME_SIZE],
                    void *pvData,
                    uint32 u32DataSize,
                    bool_t bSecure);
```

Description

This function is used during a cold start to load an individual record of application data from NVM into RAM, or to create an application data record in NVM:

- During a first-time cold start, the function defines a record to be created in NVM.
- During any subsequent cold start, the function restores (into RAM) application data previously stored in the NVM record.

The function must be called before calling any function which automatically saves application data to NVM - for example, before calling **ZPS_eAplAfInit()** to initialise the ZigBee PRO stack and before calling **ZPS_vAplSecSetInitialSecurityState()** to initialise the ZigBee security state on the node.

A pointer to a record descriptor and a unique name for the record must be specified. In addition, a pointer must be provided to the start of the corresponding data buffer in RAM and the data size must also be specified.

When called, the function first checks whether the specified record already exists in NVM.

- If the record does exist, the data from the NVM record is loaded into RAM.
- If the record does not exist, the record will be created in NVM the next time **PDM_vSaveRecord()** or **PDM_vSave()** is called. In this case, the specified RAM buffer contents remain unchanged and are saved to the NVM record.

The function also allows the record to be secured. If this option is enabled, data saved to the NVM record will be encrypted using the key specified in **PDM_vInit()** and the data will be decrypted using the same key when it is read from the record.

Note that this function must not be called after **PDM_vSaveRecord()** or **PDM_vSave()**, and must not be called during a warm start (for example, following sleep with memory held). Otherwise, the latest data in RAM will be overwritten.

Parameters

***psDesc**	Pointer to record descriptor (you do not need to be concerned with the contents of this descriptor)
acName	Character array containing name of record (maximum of 16 characters)
**pvData*	Pointer to start of RAM buffer in which to store data (in the case of record creation, initial data must be provided in the buffer)
u32DataSize	Size of record, in bytes

bSecure	Enable/disable data encryption for record: TRUE: Enable encryption FALSE: Disable encryption

Returns

One of:

PDM_E_STATUS_OK (success)

PDM_E_STATUS_INVLD_PARAM (an invalid parameter value was supplied)

PDM_eLoadRecord (JN516x only)

```
PDM_teStatus PDM_eLoadRecord(
                    PDM_tsRecordDescriptor *psDesc,
                    uint16 u16IdValue,
                    void *pvData,
                    uint32 u32DataSize,
                    bool_t bSecure);
```

Description

This function is used during a cold start to load an individual record of application data from NVM into RAM, or to create an application data record in NVM:

- During a first-time cold start, the function defines a record to be created in NVM.
- During any subsequent cold start, the function restores (into RAM) application data previously stored in the NVM record.

The function must be called before calling any function which automatically saves application data to NVM - for example, before calling **ZPS_eAplAfInit()** to initialise the ZigBee PRO stack and before calling **ZPS_vAplSecSetInitialSecurityState()** to initialise the ZigBee security state on the node.

A pointer to a record descriptor and a unique ID value for the record must be specified. In addition, a pointer must be provided to the start of the corresponding data buffer in RAM and the data size must also be specified.

When called, the function first checks whether the specified record already exists in NVM.

- If the record does exist, the data from the NVM record is loaded into RAM.
- If the record does not exist, the record will be created in NVM the next time **PDM_vSaveRecord()** or **PDM_vSave()** is called. In this case, the specified RAM buffer contents remain unchanged and are saved to the NVM record.

The function also allows the record to be secured when saving to external SPI Flash memory. If this option is enabled, data saved to the NVM record will be encrypted using the key specified in **PDM_vInit()** and the data will be decrypted using the same key when it is read from the record.

Note that this function must not be called after **PDM_vSaveRecord()** or **PDM_vSave()**, and must not be called during a warm start (for example, following sleep with memory held). Otherwise, the latest data in RAM will be overwritten.

Parameters

***psDesc**	Pointer to record descriptor (you do not need to be concerned with the contents of this descriptor)
u16IdValue	Unique 16-bit record identifier. Application software must not use values that would clash with the NXP libraries being used with that application. The ZigBee PRO stack libraries use values above 0x8000. The JenNet-IP libraries use values between 0x3000 and 0x3007

**pvData*	Pointer to start of RAM buffer in which to store data (in the case of record creation, initial data must be provided in the buffer)
u32DataSize	Size of record, in bytes
bSecure	Enable/disable data encryption for record: TRUE: Enable encryption FALSE: Disable encryption This parameter must be set FALSE when using EEPROM

Returns

One of:

PDM_E_STATUS_OK (success)

PDM_E_STATUS_INVLD_PARAM (an invalid parameter value was supplied)

EEPROM only:

PDM_E_STATUS_PDM_FULL (there is no space to store this record)

PDM_E_STATUS_RECOVERED (this record was recovered from a previous save)

PDM_vSaveRecord

void PDM_vSaveRecord(PDM_tsRecordDescriptor ***psDesc*);**

Description

This function saves the specified application data record from RAM to NVM.

Following a cold start, the function should only be called after all records have been created or loaded using **PDM_eLoadRecord()**.

The application data will be saved encrypted if security was enabled for the NVM record when the function **PDM_eLoadRecord()** was called.

Alternatively, you can save all records in RAM to NVM using the function **PDM_vSave()**.

On a JN516x device that is using internal EEPROM, the save may fail due to insufficient space for safely writing the change in the record before deleting the old record. An error callback will be made in this case.

Parameters

***psDesc**	Pointer to descriptor of record to be saved to NVM

Returns

None

PDM_vSave

```
void PDM_vSave(void);
```

Description

This function saves all records, including both application data and stack context data, from RAM to NVM.

Following a cold start, the function should only be called after all application data records have been created or loaded using **PDM_eLoadRecord()**.

The stack context data is saved encrypted using the security key specified when **PDM_vInit()** was called. The application data will be saved encrypted using this key only if security was enabled for the corresponding NVM record when **PDM_eLoadRecord()** was called.

Alternatively, an individual application data record can be saved to NVM using the function **PDM_vSaveRecord()**.

On a JN516x device that is using internal EEPROM, the save may fail due to insufficient space for safely writing the change in the record before deleting the old record. An error callback will be made in this case.

Parameters

None

Returns

None

PDM_vDeleteRecord

```
void PDM_vDeleteRecord(
                    PDM_tsRecordDescriptor *psDesc);
```

Description

This function deletes the specified record of application data in NVM.

Alternatively, all records in NVM can be deleted using the function **PDM_vDelete()**.

Parameters

***psDesc**	Pointer to descriptor of record to be deleted

Returns

None

PDM_vDelete

```
void PDM_vDelete(void);
```

Description

This function deletes all records in NVM, including both application data and stack context data.

Caution: *You are not recommended to delete records of stack context data before a rejoin of the same secured network. If these records are deleted, data sent by the node after the rejoin will be rejected by the destination node since the frame counter has been reset on the source node. For more details, refer to "Application Design Notes" appendix in the ZigBee PRO Stack User Guide (JN-UG-3048).*

Alternatively, an individual record of application data can be deleted using the function **PDM_vDeleteRecord()**.

Parameters

None

Returns

None

PDM_vWarmInitHw (JN516x only)

```
void PDM_vWarmHwInit(void);
```

Description

This function initialises the SPI Flash device following a warm start. The function must be called immediately following a warm start, before calling **OS_vRestart()**. Otherwise the ZigBee PRO stack may attempt to save records before the SPI hardware is ready.

Parameters

None

Returns

None

PDM_ vRegisterSystemCallback

```
void PDM_vRegisterSystemCallback(
                    PDM_tpfvSystemEventCallback
                            fpvPDM_SystemEventCallback);
```

Description

This function registers a user-defined callback function to handle PDM errors and events.

Parameters

fpvPDM_SystemEventCallback	Pointer to the application callback function. The function type **PDM_tpfvSystemEventCallback** is documented in Section 12.11. The events generated by the PDM library are documented in Section 12.12

Returns

None

u8PDM_CalculateFileSystemCapacity (JN516x only)

```
uint8 u8PDM_CalculateFileSystemCapacity(void);
```

Description

This function returns the remaining capacity of the JN516x internal EEPROM.

Parameters

None

Returns

Number of free sectors that are available to store new PDM records

u8PDM_GetFileSystemOccupancy (JN516x only)

```
uint8 u8PDM_GetFileSystemOccupancy(void);
```

Description

This function returns the number of sectors in the JN516x internal EEPROM that are occupied by PDM records.

Parameters

None

Returns

Number of sectors that are occupied by PDM records

9. Power Manager (PWRM) API

This chapter describes the functions of the JenOS Power Manager (PWRM) API. The API is defined in the header file **pwrm.h**.

Caution: *The Power Manager uses Wake Timer 1 of the JN51xx device if scheduled wake events are configured. In this case, do not use this wake timer for any other purpose in your application.*

The PWRM API functions are divided into the following categories:

- 'Core' functions, described in Section 9.1
- 'Callback Set-up' functions, described in Section 9.2
- 'Debugging' functions, described in Section 9.3

9.1 Core Functions

The PWRM core functions are listed below, along with their page references:

PWRM_vInit

void PWRM_vInit(PWRM_tePowerMode *ePowerMode***);**

Description

This function is used to initialise the Power Manager and specify the low-power mode in which the JN51xx device should be put when inactive.

There are five possible low-power modes that can be specified:

- Sleep with 32-kHz oscillator running and memory held
- Sleep with 32-kHz oscillator running and memory not held
- Sleep with 32-kHz oscillator not running and memory held
- Sleep with 32-kHz oscillator not running and memory not held
- Deep Sleep (32-kHz oscillator not running and memory not held)

The enumerations for the above power modes are listed below and described in Section 12.3. For further information on these low-power modes and how to wake from them, refer to Section 4.1.

Note that if the Power Manager is unable to put the JN51xx device into the specified low-power mode, it will put the device into Doze mode instead - see description of **PWRM_vManagePower()**.

If the 32-kHz oscillator is run, the JN51xx device's Wake Timer 1 is calibrated and made available (and then must not be used for any other purpose).

Parameters

ePowerMode The power mode to be used during sleep, one of:
PWRM_E_SLEEP_OSCON_RAMON
PWRM_E_SLEEP_OSCON_RAMOFF
PWRM_E_SLEEP_OSCOFF_RAMON
PWRM_E_SLEEP_OSCOFF_RAMOFF
PWRM_E_SLEEP_DEEP

Returns

None

PWRM_eStartActivity

```
PWRM_teStatus PWRM_eStartActivity(void);
```

Description

This function is used to notify the Power Manager that an activity has been started which must not be interrupted by sleep. Thus, while such an activity is running, the JN51xx device will not enter sleep mode.

The function **PWRM_eFinishActivity()** must then be called when the activity has completed. However, if **PWRM_eStartActivity()** has also been called for other activities that have not yet finished, the device will not be able to enter sleep mode until **PWRM_eFinishActivity()** has been called for all such activities.

The activity for which **PWRM_eStartActivity()** is called does not need to be identified, since the function simply increments a counter of running activities that must not be interrupted by sleep. There is an upper limit of 64K to the value of this counter. If this limit is exceeded, an overflow error is returned.

Parameters

None

Returns

PWRM_E_OK (success)

PWRM_E_ACTIVITY_OVERFLOW (activity counter limit exceeded)

PWRM_eFinishActivity

```
PWRM_teStatus PWRM_eFinishActivity(void);
```

Description

This function is used to notify the Power Manager that an activity has completed which was not to be interrupted by sleep.

The function call must be paired with a previous call to **PWRM_eStartActivity()**. Sleep mode cannot be entered until **PWRM_eFinishActivity()** has been called for all activities for which **PWRM_eStartActivity()** has been previously called.

The activity for which **PWRM_eFinishActivity()** is called does not need to be identified, since the function simply decrements a counter of running activities that must not be interrupted by sleep. Sleep mode must not be entered until this counter reaches zero. If this function is called when the counter is already zero, an underflow error is returned.

Parameters

None

Returns

PWRM_E_OK (success)

PWRM_E_ACTIVITY_UNDERFLOW (activity counter already zero)

PWRM_u16GetActivityCount

```
uint16 PWRM_u16GetActivityCount(void);
```

Description

This function obtains the current value of the activity counter which indicates the number of activities currently running that must not be interrupted by sleep. Sleep mode cannot be entered until the value of this counter is zero.

Parameters

None

Returns

Current value of activity counter

PWRM_eScheduleActivity

```
PWRM_teStatus PWRM_eScheduleActivity(
                    pwrm_tsWakeTimerEvent *psWake,
                    uint32 u32Ticks,
                    void (*prCallbackfn)(void));
```

Description

This function can be used to add a wake point and associated callback function to a list of scheduled wake points and callbacks. The new wake point is linked to an exclusive 32-kHz software wake timer, through the specified structure.

The function takes as input the number of ticks of the wake timer until the scheduled wake point. When the wake timer expires, the JN51xx device will be woken from sleep and the specified callback function will be called.

To use this function, the Power Manager must be configured through **PWRM_vInit()** to implement a low-power mode in which the 32-kHz oscillator is running and memory is held (otherwise, the list of scheduled wake points will be lost when the device enters sleep mode).

The function will return an error (see below) if the 32-kHz oscillator has not been configured to run during sleep or the software wake timer is already running for another wake point.

Parameters

***psWake**	Pointer to a structure to be populated with the wake point and callback function (see below)
u32Ticks	The number of ticks of the 32-kHz wake timer until wake point
***prCallbackfn**	Pointer to callback function associated with wake point

Returns

PWRM_E_OK (wake timer started successfully)

PWRM_E_TIMER_RUNNING (wake timer already running for another wake point)

PWRM_E_TIMER_INVALID (oscillator not configured to run during sleep)

PWRM_vManagePower

```
void PWRM_vManagePower(void);
```

Description

This function instructs the Power Manager to manage the power state of the JN51xx device. The device must be idle when this function is called, i.e. the function is typically called from the OS idle task.

Once this function has been called, whenever appropriate, the Power Manager will put the device into the low-power mode specified through the function **PWRM_vInit()**. To allow the device to enter sleep mode:

- No activities that are uninterruptable by sleep must be running - that is, the activity counter must be zero.
- If the 32-kHz oscillator will run during sleep, a wake point must have been scheduled using **PWRM_vScheduleActivity()** (this condition does not apply when the oscillator is not used)

If the above two conditions are not satisfied, the function will put the device into Doze mode instead of sleep mode. Doze mode simply pauses the on-chip CPU, leaving all components powered (e.g. radio), and requires an interrupt to be configured to wake the device.

Before putting the device into sleep mode, this function calls any user-defined callback functions that have been registered using the function **PWRM_vRegisterPreSleepCallback()**.

Parameters

None

Returns

None

9.2 Callback Set-up Functions

The PWRM callback set-up functions are used to introduce user-defined callback functions that must be defined when using the Power Manager.

The functions are listed below, along with their page references:

vAppMain

```
void vAppMain(void);
```

Description

This is a user-defined callback function which is the application entry point when using the Power Manager. This function should never return.

Parameters

None

Returns

None

PWRM_vRegisterPreSleepCallback

```
void PWRM_vRegisterPreSleepCallback(
                    tsCallbackDescriptor *psCBDesc);
```

Description

This function is used to register a user-defined callback function that will be called by the Power Manager before the JN51xx device enters sleep mode. You must specify a pointer to a structure containing a descriptor for your callback function.

The callback function must have been declared using the macro **PWRM_CALLBACK(***fn_name***)**, where *fn_name* is the name of the callback function.

The callback descriptor must have been declared using the macro **PWRM_DECLARE_CALLBACK_DESCRIPTOR(***desc_name***,** *fn_name***)**, where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vPreSleepCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(pscb1_desc, vPreSleepCB1);
```

The callback function should perform any housekeeping tasks that are necessary before the device enters sleep mode.

Note that this registration function is normally called within the user-defined function **vAppRegisterPWRMCallbacks()**. This ensures that the callback is registered during a cold start.

Parameters

***psCBDesc**	Pointer to callback descriptor structure

Returns

None

PWRM_vRegisterWakeupCallback

```
void PWRM_vRegisterWakeupCallback(
                    tsCallbackDescriptor *psCBDesc);
```

Description

This function is used to register a user-defined callback function that will be called by the Power Manager when the JN51xx device wakes from sleep (this may be due to a change on a DIO line or comparator input, or the expiry of a wake timer). You must specify a pointer to a structure containing a descriptor for your callback function.

The callback function must have been declared using the macro **PWRM_CALLBACK(***fn_name***)**, where *fn_name* is the name of the callback function.

The callback descriptor must have been declared using the macro **PWRM_DECLARE_CALLBACK_DESCRIPTOR(***desc_name***,** *fn_name***)**, where *desc_name* is the descriptor name and *fn_name* is the callback function name.

For example:

```
PWRM_CALLBACK(vWakeUpCB1);
PWRM_DECLARE_CALLBACK_DESCRIPTOR(wucb1_desc, vWakeUpCB1);
```

The callback function should perform any housekeeping tasks that are necessary after the device wakes from sleep.

Note that this registration function is normally called within the user-defined function **vAppRegisterPWRMCallbacks()**. This ensures that the callback is registered during a cold start.

Parameters

***psCBDesc**	Pointer to callback descriptor structure

Returns

None

vAppRegisterPWRMCallbacks

```
void vAppRegisterPWRMCallbacks(void);
```

Description

This is a user-defined function to register pre- and post-sleep callback functions, if required.

The function definition must itself use **PWRM_vRegisterPreSleepCallback()** and **PWRM_vRegisterWakeupCallback()** to register the required callbacks.

Parameters

None

Returns

None

PWRM_vWakeInterruptCallback

```
void PWRM_vWakeInterruptCallback(void);
```

Description

This function is a pre-defined callback function which must be called from the application's interrupt handler to deal with interrupts from Wake Timer 1 on the JN51xx device.

The function is needed to maintain the scheduled wake points list, by restarting the wake timer for the next wake-up event (if any) when the previous one has just completed. The function also calls the user-defined callback function specified through **PWRM_vScheduleActivity()**.

Parameters

None

Returns

None

9.3 Debugging Functions

The PWRM debugging functions can be used to investigate how long the JN51xx device spends in Doze mode. Either of the following two approaches to Doze mode monitoring can be taken:

- In a given monitoring session, two timers can be enabled which allow you to measure:
  - The elapsed time of the current session
  - The amount of time the device spends in Doze mode in the current session

  From the above results, you can then calculate the proportion of time that the device typically spends in Doze mode.
- The Doze state can be output on the pin DIO1 for external monitoring.

The functions are listed below, along with their page references:

PWRM_vSetupDozeMonitor

```
void PWRM_vSetupDozeMonitor(bool_t bUseIO,
                            bool_t bUseTimers,
                            bool_t b32uSec);
```

Description

This function can be used during debug to configure and start a Doze mode monitoring session on the JN51xx device - that is, to investigate the proportion of the time that the device typically spends in Doze mode.

The Doze state of the device can be optionally output on the pin DIO1. This allows the times spent in and out of Doze mode to be measured externally.

An internal Doze mode monitor can also be optionally enabled. This includes two timers, which can be configured and started by this function:

- The 'elapsed timer' is used to measure the time that has elapsed since the monitoring session was started using this function
- The 'doze timer' is used to measure the amount of time the device spends in Doze mode during the monitoring session started by this function

The clock period of the timers is configured using this function as 1 µs or 32 µs.

Parameters

bUseIO	Boolean which determines whether the Doze state will be output on DIO1: TRUE: Enable output FALSE: Disable output
bUseTimers	Boolean which determines whether the Doze mode monitoring timers will be enabled: TRUE: Enable timers FALSE: Disable timers
b32uSec	Boolean which determines the clock period of the Doze mode monitoring timers (1 or 32 µs), if they are enabled: TRUE: 32 µs FALSE: 1 µs

Returns

None

PWRM_u32GetDozeTime

```
uint32 PWRM_u32GetDozeTime(void);
```

Description

This function can be used to obtain the current value of the ‘doze timer’, which records the amount of the time spent by the JN51xx device in Doze mode during the current monitoring session. The function can only be used if the Doze mode monitoring timers have been enabled in **PWRM_vSetupDozeMonitor()**.

The value returned is the number of timer clock periods that the device has so far spent in Doze mode. To obtain the actual time spent in Doze mode, multiply the returned value by either 1 µs or 32 µs, depending on the clock period set in **PWRM_vSetupDozeMonitor()**.

This result can be used with the result of **PWRM_u32GetDozeElapsedTime()** to estimate the fraction of time spent in Doze mode.

Parameters

None

Returns

Current ‘doze timer’ value as number of clock periods

PWRM_u32GetDozeElapsedTime

```
uint32 PWRM_u32GetDozeElapsedTime(void);
```

Description

This function can be used to obtain the current value of the 'elapsed timer' which records the elapsed time of the current monitoring session. The function can only be used if the Doze mode monitoring timers have been enabled in **PWRM_vSetupDozeMonitor()**.

The value returned is the number of timer clock periods that have elapsed since Doze mode monitoring was started using **PWRM_vSetupDozeMonitor()**. To obtain the actual time spent monitoring, so far, multiply the returned value by either 1 µs or 32 µs, depending on the clock period set in **PWRM_vSetupDozeMonitor()**.

This result can be used with the result of **PWRM_u32GetDozeTime()** to estimate the fraction of time spent in Doze mode.

Parameters

None

Returns

Current 'elapsed timer' value as number of clock periods

PWRM_vResetDozeTimers

```
void PWRM_vResetDozeTimers(void);
```

Description

This function resets to zero both the ‘doze timer’ and ‘elapsed timer’ of the doze monitor. The function can only be used if the Doze mode monitoring timers have been enabled in **PWRM_vSetupDozeMonitor()**.

Parameters

None

Returns

None

10. Protocol Data Unit Manager (PDUM) API

This chapter describes the functions of the JenOS Protocol Data Unit Manager (PDUM) API. The API is defined in the header file **pdum.h**.

The PDUM API functions are listed below, along with their page references:

Note: In ZigBee PRO, the APDUs used by the application must be pre-defined (before building the application) using the ZPS Configuration Editor. This tool is detailed in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

PDUM_vInit

```
void PDUM_vInit();
```

Description

This function initialises the PDU Manager and must therefore be the first PDUM function called.

Parameters

None

Returns

None

PDUM_hAPduAllocateAPduInstance

```
PDUM_thAPduInstance
PDUM_hAPduAllocateAPduInstance(
                    PDUM_thAPdu hAPdu);
```

Description

This function allocates an instance of an Application Protocol Data Unit (APDU) - that is, memory space is allocated to the APDU instance.

The available APDUs (types and their handles) are pre-defined using the ZPS Configuration Editor (refer to the *ZigBee PRO Stack User Guide (JN-UG-3048)*).

The allocated APDU instance can subsequently be populated with data and sent to another node.

Parameters

hAPdu	Handle of APDU (type)

Returns

Handle of allocated APDU instance

PDUM_INVALID_HANDLE if no APDU instances free

PDUM_eAPduFreeAPduInstance

```
PDUM_teStatus PDUM_eAPduFreeAPduInstance(
                    PDUM_thAPduInstance hAPduInst);
```

Description

This function de-allocates the specified APDU instance, thus freeing the associated memory space.

Parameters

hAPduInstance Handle of APDU instance

Returns

PDUM_E_INTERNAL_ERROR

PDUM_u16APduInstanceReadNBO

```
uint16 PDUM_u16APduInstanceReadNBO(
                PDUM_thAPduInstance hAPduInst,
                uint16 u16Pos,
                const char *szFormat,
                void *pvStruct);
```

Description

This function reads data from the specified APDU instance and inserts the data into a C structure. The byte position of the start (least significant byte) of the data in the APDU instance must be specified, as well as the format of the data.

Data is read from the APDU instance in packed network byte order (little-endian) and translated into unpacked host byte order for the C structure (big-endian for the JN51xx device).

Parameters

hAPduInst	Handle of APDU instance to read the data from
u32Pos	The starting position (least significant byte) of the data within the APDU
**szFormat*	Format string of the data: b 8-bit byte h 16-bit half-word (short integer) w 32-bit word l 64-bit long-word (long integer) a\xnn nn (hex) bytes of data (array) p\xnn nn (hex) bytes of packing
***pvStruct*	Pointer to C structure to receive the data

Note that the compiler will not correctly interpret the format string “a\xnnb” for a data array followed by a single byte, e.g. “a\x0ab”. In this case, to ensure that the ‘b’ (for byte) is not interpreted as a hex value, use the format “a\xnn” “b”, e.g. “a\x0a” “b”.

Returns

Total number of data bytes read from the APDU instance

PDUM_u16APduInstanceWriteNBO

```
uint16 PDUM_u16APduInstanceWriteNBO(
                PDUM_thAPduInstance hAPduInst,
                uint16 u16Pos,
                const char *szFormat, ...);
```

Description

This function writes the specified data values into the specified APDU instance. The byte position of the start of the data (least significant byte) in the APDU instance must be specified, as well as the format of the data.

The data values are written into the APDU instance at the specified position in packed network byte order (little-endian). The input data values should be in host byte order (big-endian for the JN51xx device).

Parameters

hAPduInst	Handle of the APDU instance to write the data into
u32Pos	The starting position (least significant byte) of the data within the APDU instance
**szFormat*	Format string of the data: b 8-bit byte h 16-bit half-word (short integer) w 32-bit word l 64-bit long-word (long integer) a\xnn nn (hex) bytes of data (array) p\xnnnn (hex) bytes of packing
...	Variable list of data values described by the format string

Note that the compiler will not correctly interpret the format string “a\xnnb” for a data array followed by a single byte, e.g. “a\x0ab”. In this case, to ensure that the ‘b’ (for byte) is not interpreted as a hex value, use the format “a\xnn” “b”, e.g. “a\x0a” “b”.

Returns

Total number of bytes written to the APDU instance

PDUM_u16APduInstanceWriteStrNBO

```
uint16 PDUM_u16APduInstanceWriteStrNBO(
                PDUM_thAPduInstance hAPduInst,
                uint16 u16Pos,
                const char *szFormat,
                void *pvStruct);
```

Description

This function writes data from the specified structure into the specified APDU instance. The byte position of the start of the data (least significant byte) in the APDU instance must be specified, as well as the format of the data.

The data values are written into the APDU instance at the specified position in packed network byte order (little-endian). The input data values should be in host byte order (big-endian for the JN51xx device).

Parameters

hAPduInst	Handle of the APDU instance to write the data into
u32Pos	The starting position (least significant byte) of the data within the APDU instance
**szFormat*	Format string of the data: b 8-bit byte h 16-bit half-word (short integer) w 32-bit word l 64-bit long-word (long integer) a\xnn nn (hex) bytes of data (array) p\xnnnn (hex) bytes of packing
***pvStruct*	Pointer to C structure to containing data

Note that the compiler will not correctly interpret the format string “a\xnnb” for a data array followed by a single byte, e.g. “a\x0ab”. In this case, to ensure that the ‘b’ (for byte) is not interpreted as a hex value, use the format “a\xnn” “b”, e.g. “a\x0a” “b”.

Returns

Total number of bytes written to the APDU instance

PDUM_u16SizeNBO

uint16 PDUM_u16SizeNBO(const char ****szFormat*);**

Description

This function obtains the size, in bytes, of an APDU data payload, given the format of the data.

Parameters

**szFormat*	Format string of the data: b 8-bit byte h 16-bit half-word (short integer) w 32-bit word l 64-bit long-word (long integer) a\xnn nn (hex) bytes of data (array) p\xnnnn (hex) bytes of packing

Note that the compiler will not correctly interpret the format string “a\xnnb” for a data array followed by a single byte, e.g. “a\x0ab”. In this case, to ensure that the ‘b’ (for byte) is not interpreted as a hex value, use the format “a\xnn” “b”, e.g. “a\x0a” “b”.

Returns

Number of bytes in data payload

PDUM_u16APduGetSize

uint16 PDUM_u16APduGetSize(PDUM_thAPdu *hAPdu***);**

Description

This function obtains the maximum size, in bytes, of the specified APDU (type).

Parameters

hAPdu	Handle of APDU

Returns

Number of bytes in APDU

PDUM_pvAPduInstanceGetPayload

```
void * PDUM_pvAPduInstanceGetPayload(
                    PDUM_thAPduInstance hAPduInst);
```

Description

This function obtains a pointer to the payload data of the specified APDU instance.

Parameters

hAPduInst	Handle of APDU instance to access

Returns

Pointer to data as an array of bytes

PDUM_u16APduInstanceGetPayloadSize

```
uint16 PDUM_u16APduInstanceGetPayloadSize(
                    PDUM_thAPduInstance hAPduInst);
```

Description

This function obtains the size, in bytes, of the payload data of the specified APDU instance.

Parameters

hAPduInst	Handle of APDU instance to access

Returns

Size of the payload data, in bytes

PDUM_eAPduInstanceSetPayloadSize

```
PDUM_teStatus PDUM_eAPduInstanceSetPayloadSize(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16Size);
```

Description

This function sets the size, in bytes, of the payload of the specified APDU instance.

Parameters

hAPduInst	Handle of APDU instance
u16Size	Size of payload to set, in bytes

Returns

PDUM_OK
PDUM_E_APDU_INSTANCE_TOO_BIG

PDUM_vDBGPrintAPduInstance

```
void PDUM_vDBGPrintAPduInstance(
                    PDUM_thAPduInstance hAPduInst);
```

Description

This function can be used to output the specified APDU instance via the Debug (DBG) module.

For details of the DBG functions, refer to Chapter 10.

Parameters

hAPdu	Handle of APDU instance to output

Returns

None

11. Debug Module API

The chapter describes the functions of the JenOS Debug (DBG) module API. The API is defined in the header file **dbg.h**.

To use the Debug module, it must be enabled at build-time by defining DBG_ENABLE in the build - for example, by adding the -DDBG_ENABLE option to the compiler.

By default, the Debug module will just display each line as passed. However, if DBG_VERBOSE is defined at build-time then each line displayed will be prefixed with the file name and line number of the debug statement.

Note: Compiling with the DBG option results in a larger application size, requiring a lot more space in RAM.

The DBG API functions are listed below, along with their page references:

DBG_vInit

void DBG_vInit(tsDBG_FunctionTbl ****psFunctionTbl*);**

Description

This function is used to initialise the Debug module.

Note: If a JN51xx UART is to be used as the debug output interface, **DBG_vUartInit()** must be called instead. Thus, **DBG_vInit()** will not be needed by most users, since a UART will normally be used for debug output.

The function can be used during a cold start or a warm start (with memory held). Its parameter accepts a structure containing pointers to four user-defined callback functions concerned with the output interface:

```
typedef struct
  {
   void (*prInitHardwareCb)(void);
   void (*prPutchCb)       (char c);
   void (*prFlushCb)       (void);
   void (*prFailedAssertCb)(void);
  } tsDBG_FunctionTbl;
```

The callback functions pointed to by this structure are as follows:

***prInitHardwareCb**	Points to function which re-initialises the interface after a warm start, e.g. when JN51xx device wakes from sleep
***prPutchCb**	Points to function used by **DBG_vPrintf()** to output a single character to the interface
***prFlushCb**	Points to function used by **DBG_vPrintf()** to flush the interface buffer to allow buffered output characters to be displayed. If the output is unbuffered, this function should do nothing or wait for the last character output using the **putch()** function to be made available. Note that the function should not append a newline character, as this should be handled by the formatting string passed to **DBG_vPrintf()**
***prAssertFailedCb**	Points to function which is called when **DBG_vAssert()** fails. The function should stop execution and may reset the device

Parameters

***psFunctionTbl**	Pointer to structure containing list of callback functions.

Returns

None

DBG_vUartInit

```
void DBG_vUartInit(DBG_teUart eUart,
                   DBG_teUartBaudRate eBaudRate);
```

Description

This function is used to initialise the Debug module when one of the JN51xx on-chip UARTs is to be used as the output interface. In this case, this function should be called instead of **DBG_vInit()**. This will be the case for most users, as a UART will normally be used for debug output.

The function can be used during a cold start or a warm start (with memory held). It is necessary to specify the UART (0 or 1) and the required baud rate.

Note that the callback functions required by **DBG_vInit()** are not needed for **DBG_vUartInit()**, since they are pre-defined by NXP for the on-chip UARTs.

Parameters

eUart	UART to use as output interface, one of: DBG_E_UART_0 (UART0) DBG_E_UART_1 (UART1)
eBaudRate	Baud rate of UART, one of: DBG_E_UART_BAUD_RATE_4800 (4800 bps) DBG_E_UART_BAUD_RATE_9600 (9600 bps) DBG_E_UART_BAUD_RATE_19200 (19200 bps) DBG_E_UART_BAUD_RATE_38400 (38400 bps) DBG_E_UART_BAUD_RATE_76800 (76800 bps) DBG_E_UART_BAUD_RATE_115200 (115200 bps)

Returns

None

DBG_vPrintf

```
void DBG_vPrintf(bool_t bStreamEnabled,
                 const char *pcFormat, ...);
```

Description

This function is an adapted **printf()** function, allowing a formatted string to be output (e.g. via the UART) for display.

The function contains a parameter which allows the output of the string to be enabled or disabled - the value of this Boolean parameter must be a literal. If disabled, the compiler will optimise out this function, but its parameters will still be evaluated.

Parameters

bStreamEnabled	Boolean which determines whether string will be output: TRUE: Output string FALSE: Do not output string (compile out function)
***pcFormat**	Pointer to printf-style formatting string
...	As for the standard **printf()** function

Returns

None

DBG_vAssert

```
void DBG_vAssert(bool_t bStreamEnabled,
                 bool_t bAssertion);
```

Description

This function is an adapted **assert()** function, allowing a Boolean condition to be tested.

The function contains a parameter which allows the test to be enabled or disabled - the value of this Boolean parameter must be a literal. If disabled, the compiler will optimise out this function.

The Boolean condition to be tested is specified as a parameter:

- If the condition is TRUE, program execution continues.
- If the condition is FALSE, an error message is output and execution is passed to a callback function, which stops execution. This callback function is specified when **DGB_vInit()** is called for a cold start.

Parameters

bStreamEnabled	Boolean which determines whether test will be performed: TRUE: Perform test FALSE: Do not perform test
bAssertion	Boolean expression to be tested

Returns

None

DBG_vDumpStack

```
void DBG_vDumpStack(void);
```

Description

This function outputs the contents of the CPU stack (e.g. via the UART) for display.

Parameters

None

Returns

None

12. JenOS Structures

This chapter describes the structures used by the JenOS APIs.

The structures are listed below along with their page references.

12.1 PDM_tsHwFncTable

This structure is used in the function **PDM_vInit()** to specify a set of user-defined functions used to interact with a custom NVM device.

```
typedef struct
{
    /* This function is called after a cold or warm start */
    void (*prInitHwCb)(void);

    /* This function is called to erase the given sector */
    void (*prEraseCb) (uint8 u8Sector);

    /*This function is called to write data to an address
     * within a given sector. Address zero is the start of the
     * given sector */
    void (*prWriteCb) (uint8 u8Sector,
                       uint16 u16Addr,
                       uint16 u16Len,
                       uint8 *pu8Data);

    /* This function is called to read data from an address
     * within a given sector. Address zero is the start of the
     * given sector */
    void (*prReadCb)  (uint8 u8Sector,
```

```
                          uint16 u16Addr,
                          uint16 u16Len,
                          uint8 *pu8Data);
} PDM_tsHwFncTable;
```

12.2 tSPIflashFncTable

This structure is used in the function **PDM_vSPIFlashConfig()** to specify a set of user-defined functions used to interact with a custom SPI Flash device.

```
typedef struct tagSPIflashFncTable {
  uint32                  u32Signature; //always set to 0x1234678
  uint16                  u16FlashId; //(u8ManufactureId<<8)|u8DeviceId
  uint16                  u16Reserved; //Reserved
  tpfvZSPIflashInit       vZSPIflashInit; //see below
  tpfvZSPIflashSetSlaveSel vZSPIflashSetSlaveSel; //see below
  tpfvZSPIflashWREN       vZSPIflashWREN; //see below
  tpfvZSPIflashEWRSR      vZSPIflashEWRSR; //see below
  tpfu8ZSPIflashRDSR      u8ZSPIflashRDSR; //see below
  tpfu16ZSPIflashRDID     u16ZSPIflashRDID; //see below
  tpfvZSPIflashWRSR       vZSPIflashWRSR; //see below
  tpfvZSPIflashPP         vZSPIflashPP; //see below
  tpfvZSPIflashRead       vZSPIflashRead; //see below
  tpfvZSPIflashBE         vZSPIflashBE; //see below
  tpfvZSPIflashSE         vZSPIflashSE; //see below
} tSPIflashFncTable;
```

The custom functions specified in this structure are outlined in Table 1 below.

Function Prototype	Description
void vZSPIflashInit(int *iDivisor***, uint8** *u8SlaveSel***);**	Initialises variables for Flash access. • *iDivisor* Clock divisor • *u8SlaveSel* Byte used for slave select
void vZSPIflashSetSlaveSel(uint8 *u8SlaveSel***);**	
void vZSPIflashWREN(void);	Enables writes to Flash. Called before erasing or programming Flash.
vZSPIflashEWRSR(void);	Enables writes to Flash Status Register. Called before writing to the Flash Status Register.
uint8 u8ZSPIflashRDSR(void);	Reads Flash Status Register and returns Status Register data
uint16 u16ZSPIflashRDID(void);	Reads Flash ID Register and returns ID Register data, 0 on error or 2 bytes [ManufacturerId, DeviceId]
void vZSPIflashWRSR(uint8 *u8Data***);**	Writes data to Flash Status Register • *u8Data* Status Register data
void vZSPIflashPP(**uint32** *u32Addr***,** **uint16** *u16Len***,** **uint8*** *pu8Data***);**	Writes data to Flash. • *u32Addr* Address • *u16Len* Length, in bytes • *pu8Data* Data to write
void vZSPIflashRead(**uint32** *u32Addr***,** **uint16** *u16Len***,** **uint8*** *pu8Data***);**	Reads data from Flash. • *u32Addr* Address • *u16Len* Length, in bytes • *pu8Data* Data read
void vZSPIflashBE(void);	Performs a bulk erase of Flash
void vZSPIflashSE(uint8 *u8Sector***);**	Performs a sector erase of Flash • *u8Sector* Sector number

Table 1: Functions of tSPIflashFncTable Structure

12.3 PWRM_teSleepMode

This structure contains the enumerations used to set the power mode of the JN5148 device during sleep.

```
typedef enum
{
    PWRM_E_SLEEP_OSCON_RAMON,   /*32-kHz Osc on and RAM on*/
    PWRM_E_SLEEP_OSCON_RAMOFF,  /*32-kHz Osc on and RAM off*/
    PWRM_E_SLEEP_OSCOFF_RAMON,  /*32-kHz Osc off and RAM on*/
    PWRM_E_SLEEP_OSCOFF_RAMOFF, /*32-kHz Osc off and RAM off*/
    PWRM_E_SLEEP_DEEP,          /*Deep Sleep*/
} PWRM_teSleepMode;
```

12.4 DBG_tsFunctionTbl

This structure contains callback functions used by the Debug (DBG) module to interact with the output interface.

```
typedef struct
   {
    void (*prInitHardwareCb)(void);
    void (*prPutchCb)       (char c);
    void (*prFlushCb)       (void);
    void (*prFailedAssertCb)(void);
   } DBG_tsFunctionTbl;
```

For details of the callback functions, refer to the description of DBG_vInit on page 150.

12.5 tsReg128

This is a constant structure which contains a 128-bit encryption key used by the PDM module - the key is passed into the module via the **PDM_vInit()** function.

```
typedef struct
{
    uint32 u32register0;
    uint32 u32register1;
    uint32 u32register2;
    uint32 u32register3;
} tsReg128;
```

In the above structure, `u32register0` contains the 32 least signifacant bits and `u32register3` contains the 32 most significant bits of the key.

12.6 OVLY_tsInitData

This structure contains initialisation data for the overlay feature and is used by the function **OVLY_bInit()**.

```
typedef struct
{
    uint32          u32ImageOffset;
    OVLY_prMutex    prGetMutex;
    OVLY_prMutex    prReleaseMutex;
    OVLY_prEvent    prOverlayEvent;
    uint8           u8ReadAttempts;
    uint16          u16TransferBlocks;
} OVLY_tsInitData;
```

where:

- `u32ImageOffset` is the offset of the application image from the beginning of NVM (Flash memory) - normally this is zero
- `prGetMutex` is a pointer to a user-defined callback function to get the SPI mutex
- `prReleaseMutex` is a pointer to a user-defined callback function to release the SPI mutex
- `prOverlayEvent` is a pointer to a user-defined callback function for handling overlay-related events and profiling overlay operation
- `u8ReadAttempts` is the number of times to attempt to load an overlay in the case of a checksum failure
- `u16TransferBlocks` is the maximum number of 16-byte blocks to read in one transfer

For the callback function prototypes, refer to the description of the function **OVLY_bInit()** on page 98.

12.7 OVLY_teEvent

This structure contains enumerations for the events related to overlays.

```
typedef enum
{
    OVLY_E_EVENT_LOAD,
    OVLY_E_EVENT_READ,
    OVLY_E_EVENT_INTERRUPTED,
    OVLY_E_EVENT_ERROR_INDEX,
    OVLY_E_EVENT_ERROR_SIZE,
    OVLY_E_EVENT_ERROR_CHECKSUM,
    OVLY_E_EVENT_FAILED
} OVLY_teEvent;
```

The events are outlined in the table below.

Event Enumeration	Description
OVLY_E_EVENT_LOAD	Generated once at the beginning of each overlay load
OVLY_E_EVENT_READ	Generated for each chunk transferred from NVM, where a 'chunk' is defined by `u16TransferBlocks` in the initialisation data (see Section 12.6)
OVLY_E_EVENT_INTERRUPTED	Generated if load has been interrupted by another load and is therefore being restarted
OVLY_E_EVENT_ERROR_INDEX	Generated if an overlay table entry read from NVM fails its checksum
OVLY_E_EVENT_ERROR_SIZE	Not used
OVLY_E_EVENT_ERROR_CHECKSUM	Generated if the page loaded fails its checksum - load will be retried
OVLY_E_EVENT_FAILED	Generated if the page load fails more than `u8ReadAttempts` times (defined in the initialisation data - see Section 12.6)

Table 2: RTOS Overlay Events

12.8 OVLY_tuEventData

This structure contains a union of overlay-related event data.

```
typedef union
{
    struct
    {
        uint32              u32TargetAddress;
        uint32              u32ReturnAddress;
        uint16              u16TargetPage;
        uint16              u16ReturnPage;
    } sLoad;

    struct
    {
        uint32              u32Offset;
        uint16              u16Length;
    } sRead;
} OVLY_tuEventData;
```

where:

- `u32TargetAddress` is the address in memory of the invoked function which is contained in an overlay
- `u32ReturnAddress` is the address in memory of the point to which program execution must return on completion of the invoked function
- `u16TargetPage` is the page number of the overlay which contains the invoked function
- `u16ReturnPage` is the page number of the overlay to which program execution must return on completion of the invoked function
- `u32Offset` is the address offset from the start of the application of the block of code read from NVM (and copied into RAM)
- `u16Length` is the size, in bytes, of the block of code read from NVM (and copied into RAM)

12.9 OVLY_tsProfiling

This structure is used to hold profiling information on overlay operation - the structure is updated during application execution. Overlay profiling can be implemented using **OVLY_psProfilingInit()** or a callback function registered through **OVLY_bInit()**.

```
typedef struct
{
    uint32                  u32TotalBytesLoaded;
    uint32                  u32ChecksumFailures;
    OVLY_tsProfilingEntry   asOverlay[0];
} OVLY_tsProfiling;
```

where:

- `u32TotalBytesLoaded` is the total number of bytes of code loaded (so far) from NVM to RAM
- `u32ChecksumFailures` is the number of checksum failures that have occurred (so far) in transferring code from NVM to RAM
- `asOverlay[0]` is an array in which each element is a structure containing the profiling data for an individual overlay (this structure is detailed below in Section 12.10)

12.10 OVLY_tsProfilingEntry

This structure contains the profiling information on an individual overlay and is used in the `OVLY_tsProfiling` structure, described in Section 12.9.

```
typedef struct
{
    uint16                  u16Size;
    uint32                  u32Loads;
} OVLY_tsProfilingEntry;
```

where:

- `u16Size` is the size, in bytes, of the block of code in the overlay
- `u32Loads` is the number of times the overlay has (so far) been loaded from NVM to RAM

12.11 PDM_tpfvSystemEventCallback

This type defines the callback function that receives PDM events.

```
typedef void (*PDM_tpfvSystemEventCallback) (
    uint32                          u32eventNumber,
    PDM_eSystemEventCode            eSystemEventCode);
```

where:

- `u32eventNumber` gives further information about the event depending on the event code, as detailed in Section 12.12
- `eSystemEventCode` identifies the type of event that triggered the callback.

12.12 PDM_eSystemEventCode

This structure contains enumerations for the events generated by the PDM library.

```
typedef enum
{
    E_PDM_SYSTEM_EVENT_WEAR_COUNT_TRIGGER_VALUE_REACHED=0,
    E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED,
    E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_SPACE,
    E_PDM_SYSTEM_EVENT_EEPROM_SEGMENT_HEADER_REPAIRED,
    E_PDM_SYSTEM_EVENT_SYSTEM_INTERNAL_BUFFER_WEAR_COUNT_SWAP,
    E_PDM_SYSTEM_EVENT_SYSTEM_DUPLICATE_FILE_SEGMENT_DETECTED
} PDM_eSystemEventCode;
```

The events are outlined in Table 3 below.

Event Enumeration	Description
E_PDM_SYSTEM_EVENT_WEAR_COUNT_TRIGGER_VALUE_REACHED	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_DESCRIPTOR_SAVE_FAILED	A save has failed. u32eventNumber contains the u16IdValue of the record that failed to save. This is a fatal error as the ZigBee PRO stack records may be inconsistent. Test software should log this error and halt. Production software may need to perform a factory reset.
E_PDM_SYSTEM_EVENT_PDM_NOT_ENOUGH_SPACE	There is not enough space to hold all the PDM records. u32eventNumber contains the u16IdValue of the record that was being processed. This is a fatal error as the ZigBee PRO stack records may be inconsistent. Test software should log this error and halt. Production software may need to perform a factory reset.
E_PDM_SYSTEM_EVENT_EEPROM_SEGMENT_HEADER_REPAIRED	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_SYSTEM_INTERNAL_BUFFER_WEAR_COUNT_SWAP	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.
E_PDM_SYSTEM_EVENT_SYSTEM_DUPLICATE_FILE_SEGMENT_DETECTED	This code can be ignored by the application software and only needs to be logged if requested by NXP Technical Support.

Table 3: PDM Event Codes

12.13 OS_teStatus

This structure contains enumerations for the JenOS status codes generated by the core OS library.

```
typedef enum {
    OS_E_OK                                 = 0,
    OS_E_BADTASK                            = 1,
    OS_E_BADMUTEX                           = 2,
    OS_E_BADMESSAGE                         = 3,
    OS_E_BADVALUE                           = 4,
    OS_E_OVERACTIVATION                     = 5,
    OS_E_QUEUE_EMPTY                        = 6,
    OS_E_QUEUE_FULL                         = 7,
    OS_E_UNQUEUED                           = 8,
    OS_E_OSINTOVERFLOW                      = 9,
    OS_E_OSINTUNDERFLOW                     = 10,
    OS_E_BADSWTIMER                         = 11,
    OS_E_BADHWCOUNTER                       = 12,
    OS_E_SWTIMER_STOPPED                    = 13,
    OS_E_SWTIMER_EXPIRED                    = 14,
    OS_E_SWTIMER_RUNNING                    = 15,
    OS_E_HWCOUNTERIDLE                      = 16,
    OS_E_NOTHINGTOEXPIRE                    = 17,
    OS_E_PRIORITY_ERROR                     = 18,
    OS_E_BAD_NESTING                        = 19,
    OS_E_TICKS_TOO_BIG                      = 20,
    OS_E_CURRENT_TASK_NOT_A_MUTEX_MEMBER    = 21,
    OS_E_TASK_NOT_A_MESSAGE_POSTER          = 22,
    OS_E_TASK_NOT_A_MESSAGE_COLLECTOR       = 23
} OS_teStatus;
```

The status codes are described in Table 4 below. Fatal errors must be handled using the mechanism described in Section 2.6.

Event Enumeration	Description
OS_E_OK	The function completed without error.
OS_E_BADTASK	A bad task handle has been passed to a function. This is a fatal error.
OS_E_BADMUTEX	A bad mutex handle has been passed to a function. This is a fatal error.
OS_E_BADMESSAGE	A bad message handle has been passed to a function. This is a fatal error.
OS_E_BADVALUE	An out-of-range value has been passed to a function. This is a fatal error.
OS_E_OVERACTIVATION	A task has been activated too many times. This is a fatal error.
OS_E_QUEUE_EMPTY	An attempt has been made to read from an empty queue.
OS_E_QUEUE_FULL	An attempt has been made to post to a queue that is full. Whist the OS can recover from this situation, this error should normally be treated as fatal. If a ZigBee PRO stack queue overflows, the stack can be left in an inconsistent state.
OS_E_UNQUEUED	Returned by **OS_eGetMessageStatus** when the queue is not empty or full.
OS_E_OSINTOVERFLOW	There have been too many nested interrupts. This is a fatal error.
OS_E_OSINTUNDERFLOW	A resume from interrupts has failed due to there being no matching interrupt. This is a fatal error.
OS_E_BADSWTIMER	A bad timer handle has been passed to a function. This is a fatal error.
OS_E_BADHWCOUNTER	A bad hardware counter handle has been passed to a function. This is a fatal error.
OS_E_SWTIMER_STOPPED	An attempt has been made to stop a timer that is already stopped. This is not necessarily a fatal error.
OS_E_SWTIMER_EXPIRED	The software timer has expired. This is not a fatal error.
OS_E_SWTIMER_RUNNING	The software timer is running. This is not normally a fatal error. However, when calling **OS_eContinueSWTimer()** on a timer that is already running, the expiry time of the timer will remain at the time previously set and will not be changed by this call.
OS_E_HWCOUNTERIDLE	A code used internally that is not presented to application software.
OS_E_NOTHINGTOEXPIRE	The hardware timer has expired but no software timers are due to expire. This is a fatal error.

Table 4: OS Event Codes

Event Enumeration	Description
OS_E_PRIORITY_ERROR	The priority levels internal to the OS are not consistent. This is a fatal error.
OS_E_BAD_NESTING	There have been two calls to enter a critical section without a leave call. This is a fatal error.
OS_E_TICKS_TOO_BIG	An attempt has been made to start a timer too far into the future. This is a fatal error.
OS_E_CURRENT_TASK_NOT_A_MUTEX_MEMBER	A task has attempted to take a mutex when the task is not in the mutex group in the OS configuration diagram. This is a fatal error.
OS_E_TASK_NOT_A_MESSAGE_POSTER	A task has attempted to post a message to a queue when the task is not connected to the queue in the OS configuration diagram. This is a fatal error.
OS_E_TASK_NOT_A_MESSAGE_COLLECTOR	A task has attempted to collect a message from a queue when the task is not connected to the queue in the OS configuration diagram. This is a fatal error.

Table 4: OS Event Codes

Part III:
Configuration Information

13. JenOS Configuration

The use of certain JenOS resources must be statically configured before building an application, particularly resources relating to the RTOS and PDU Manager, such as timers, mutexes and ISRs. This chapter introduces the JenOS Configuration Editor, the graphical tool used to perform this configuration.

The JenOS Configuration Editor is an NXP-devised plug-in for the Eclipse IDE. The Eclipse platform is provided as part of the *SDK Toolchain (JN-SW-4041)*.

In developing a ZigBee PRO application, ZigBee network parameters must be pre-configured using the ZPS Configuration Editor, which is also an NXP-devised plug-in for the Eclipse IDE.

Note: The plug-ins are available separately from the SDK but installation of the plug-ins is described in the *SDK Installation Guide and User (JN-UG-3064)*.

The principles of the build-time configuration for a ZigBee PRO application are described in Section 13.1. The JenOS Configuration Editor is then outlined in Section 13.2 but is described more fully in Chapter 14. The ZPS Configuration Editor is described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

13.1 Configuration Principles

The build process for a ZigBee PRO application takes a number of configuration files, in addition to the application source file and header file. The following files are generated from Eclipse to feed into the build process:

- ZigBee PRO Stack files:
  - **zps_gen.c**
  - **zps_gen.h**
- PDU Manager files:
  - **pdum_gen.c**
  - **pdum_gen.h**
- RTOS files:
  - **os_gen.c**
  - **os_gen.h**
  - **os_irq.s**

All of the above files are produced according to the same basic principles. The NXP plug-ins in Eclipse are used to edit the configuration data and output this data as XML files (the XML files can be coded manually, outside of Eclipse, but this is not

recommended). As part of the build process, the application's makefile invokes command line utilities that use the XML files to generate the files listed above.

The full build process is illustrated in Figure 1.

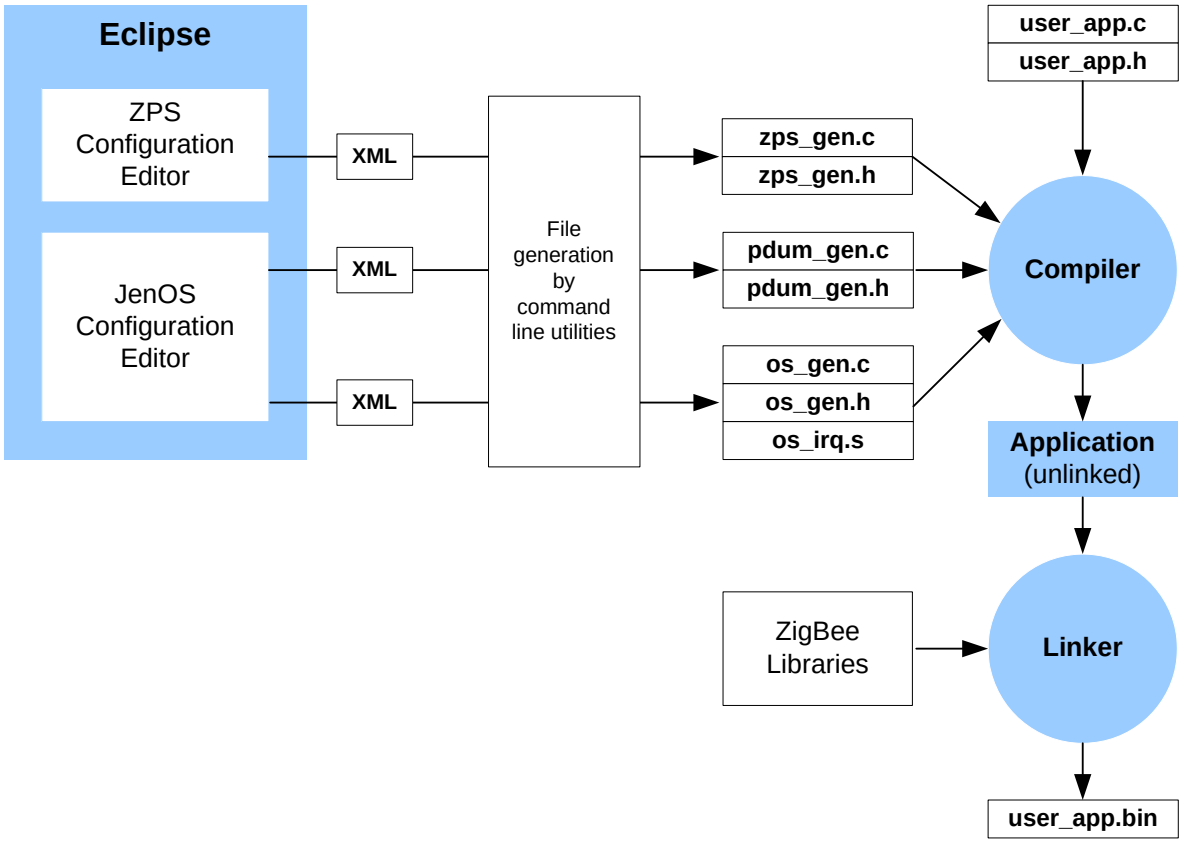


Figure 1: Application Build Process

13.2 Configuring JenOS Resources

The JenOS Configuration Editor is outlined below. In addition, operational information is provided in Chapter 14.

The JenOS configuration is closely linked to application coding and can be performed before, during or after coding. This configuration is largely concerned with the management of task/ISR execution and scheduling. Typical settings include:

- The execution priorities of individual user tasks and ISRs
- A user task's membership of a particular mutex group
- The interrupt which 'stimulates' execution of a particular ISR
- The software timers derived from a particular hardware counter
- The callback functions associated with a particular hardware counter
- The user task that is activated when a particular software timer expires
- The message types that can be sent and received by a particular user task
- The length of a user task's message queue for a particular message type

This information is represented in the editor in graphical form. The editor's graphical window is divided into separate regions for the application, ZigBee PRO stack and CPU exceptions. An example of this window is shown below.

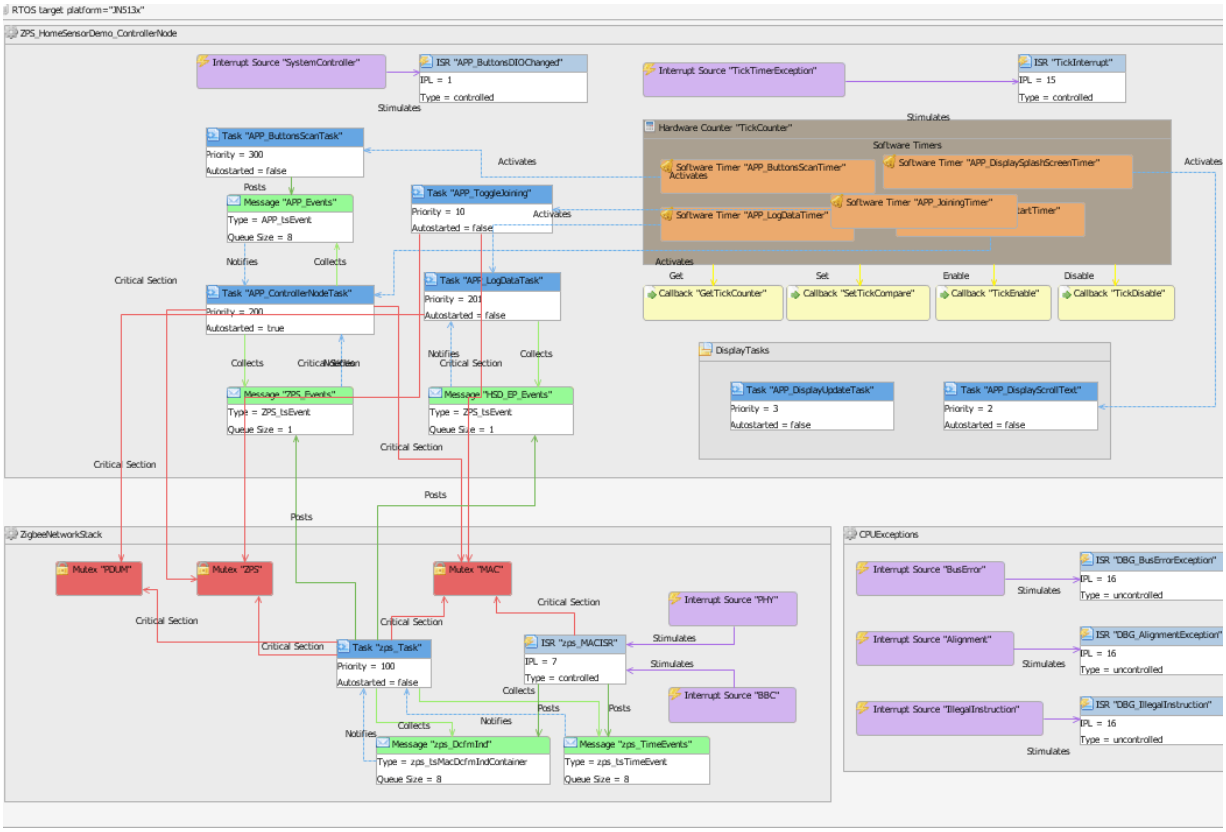


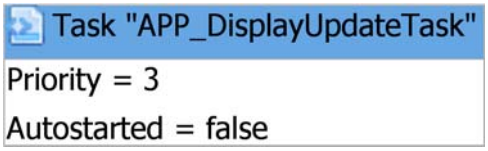
Figure 2: Graphical Configuration Window

Generally, the stack region of the diagram is the same for all applications and can be taken from the template provided in the Application Note *ZigBee PRO Application Template (JN-AN-1123)*. Thus, most of the new configuration is required in the application region of the diagram.

In the diagram, the following objects are represented by colour-coded boxes:

- Task (blue)
- ISR (light blue)
- Callback (yellow)
- Message queue (green)
- Mutex group (red)
- Interrupt source (purple)
- Hardware counter (brown)
- Software timer (orange)

For example, the following box represents a task:



The blue banner indicates a user task and the name of the task is included in quotes ("APP_UpdateDisplayTask" here). The white area of the box contains configurable attributes of the task (here, the execution priority and autostart status).

Relationships between the objects are indicated using colour-coded lines that run between them. For example:

- A red line between a task/ISR and a mutex group indicates that the task/ISR belongs to the mutex group.
- A dark-green arrowed line between a task and a message queue indicates that the task is allowed to post messages in the queue.
- A light-green arrowed line between a task and a message queue indicates that the task is allowed to collect messages from the queue.
- A blue arrowed line between a software timer and a task indicates that expiry of the timer will activate the task.
- A purple arrowed line shows the interrupt source that stimulates a certain ISR.

The graphic below defines "TickTimerException" as the interrupt source that triggers execution of the ISR "TickInterrupt".

14. JenOS Configuration Editor

The JenOS Configuration Editor is a graphical editor which runs as a plug-in on the Eclipse platform. It is used to construct diagrams which show the JenOS services that the application program will use and the relationships between them. Typically the diagrams include instantiations of tasks, messages and interrupt service routines (ISRs), and the connections between them - for example, showing how two tasks use a particular message queue to communicate or which hardware interrupts start a task running.

Once a diagram has been produced, it is converted into C code for compilation as part of the application program (see Section 13.1).

To support the tutorials in this chapter, a number of example application files are provided in the ZIP file for this manual. These applications are summarised in Appendix A.

Important: The ZIP file for this manual includes the ZIP file **OS-Tutorial.zip** which contains the software for the tutorials in this chapter. **OS-Tutorial.zip** must be extracted to **<JN51xx_SDK_ROOT>\Application**, where **<JN51xx_SDK_ROOT>** is the path into which the SDK is installed (by default, this is **C:\Jennic**). You should then have a project directory for the tutorials - e.g. **C:\Jennic\Application\OS-Tutorial**.

14.1 Getting Started

Before we can start to create a JenOS configuration diagram, we need to have the JenOS Configuration Editor plug-in installed in the Eclipse IDE.

To check if this is the case, start Eclipse and select **File > New > Other** from the main menu. Check that a Jennic option exists in the **Select a wizard** dialogue box - expanding the Jennic option should show the "Jennic RTOS Configuration Diagram", as shown in the screenshot below. If this is not present, install the JenOS Configuration Editor by referring to the chapter on installing configuration plug-ins in the *SDK Installation and User Guide (JN-UG-3064)*.

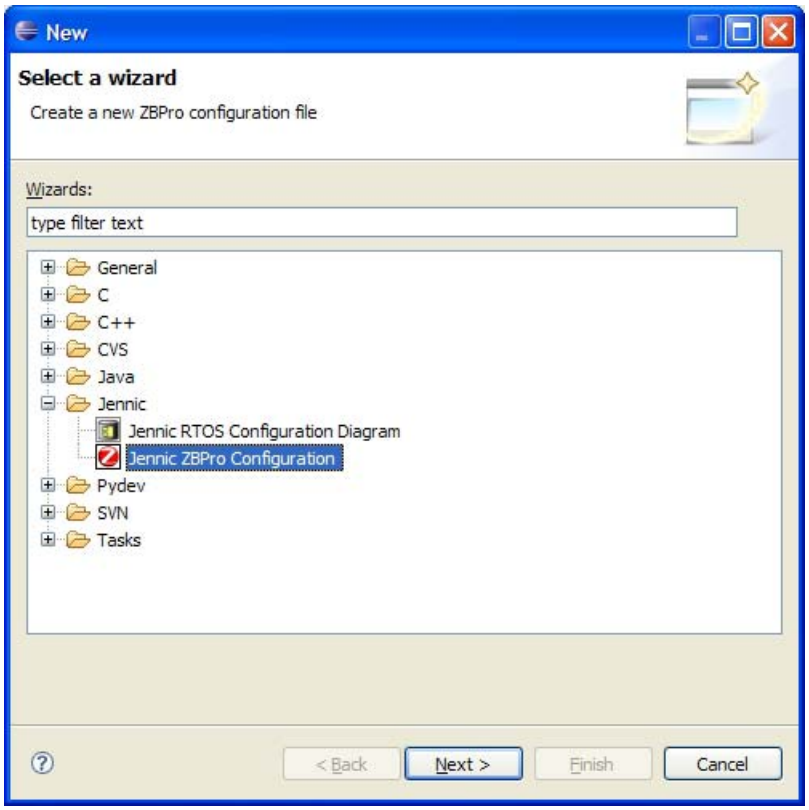


Figure 3: Select a Wizard

Using the wizard shown in the screenshot above, we can start to create a new configuration diagram.

14.2 Creating an RTOS Configuration Diagram

Step 1 In the Eclipse **Select a wizard** box shown in Table 3 on page 176, select **Jennic RTOS Configuration Diagram** and click **Next**. The following screen appears:

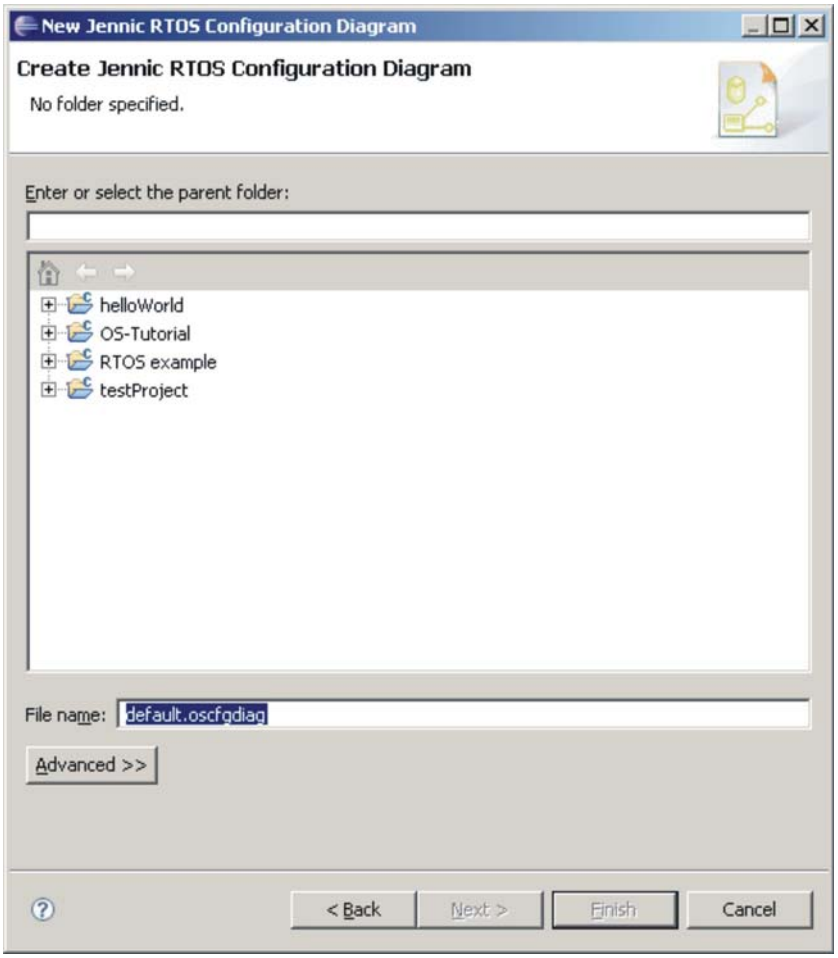


Figure 4: Create New RTOS Configuration Diagram

Step 2 Select the project to which you want to add the diagram and give the diagram a name in the **File name** box, keeping the **.oscfgdiag** filetype (creating a new project is described in the *SDK Installation and User Guide (JN-UG-3064)*).

Step 3 Click **Finish**. The diagram editor now opens with a blank tab, as shown below. In this case, the diagram is called **OS-example.oscfgdiag**, as shown in the tab.

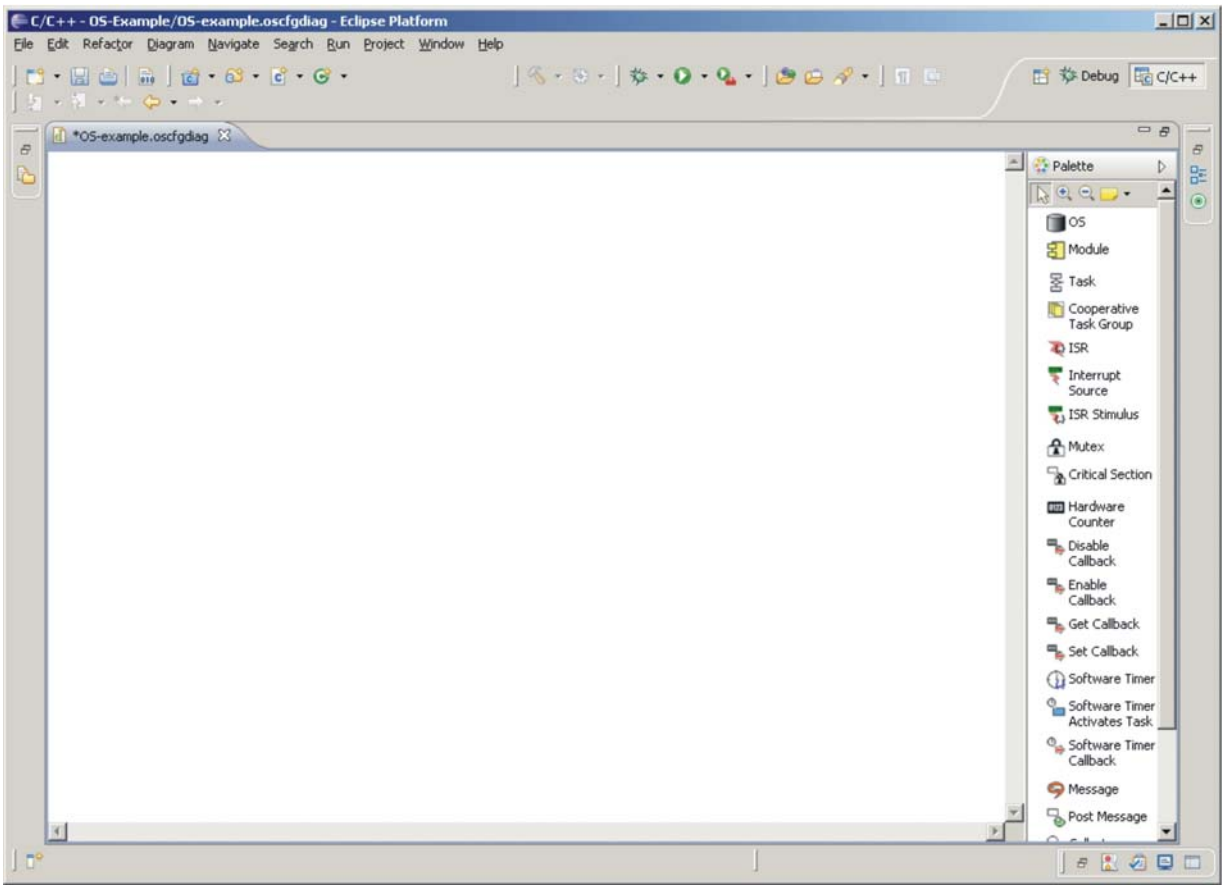


Figure 5: The Diagram Editor

The screen is divided into two parts:

- the **diagram sheet**, which uses tabs at the top of the area to differentiate between diagrams if more than one is open
- the **tool palette**, which contains all the entities and connection methods available for building a diagram

We will look at each of the icons on the palette and use them in a number of examples, gradually building up more functionality on the diagram.

14.3 Building Configuration Diagrams

In this section, we will start to add elements to the diagram and gradually build up the functionality we require. The first two additions to the diagram are generic to all diagrams, so we will use the next sections to introduce the methods for placing and manipulating elements on the diagram.

14.3.1 Starting the Diagram - the OS Icon

The configuration diagram is based around a hierarchy of elements. The highest level in the hierarchy is the OS, of which there is only one per diagram. This element encapsulates all the other elements which make up the diagram.

So the first thing we do on the diagram is add the overall OS. There are a number of ways of doing this. The first method uses the palette to explicitly select the item we want to place in the diagram - in this case, the OS.

Step 1 Click on the **OS** icon in the palette, which is then highlighted, and move the mouse pointer to the diagram pane. Note that the pointer has a small box and plus sign attached to it, showing that it has the OS icon selected.

Step 2 Click to place the **OS** element on the diagram, and use the resizing handles around the edges and on the corners of the **OS** element to expand it. Since this element needs to encapsulate all the other elements of the diagram, stretch it to fill most of the diagram sheet, as shown in the figure below.

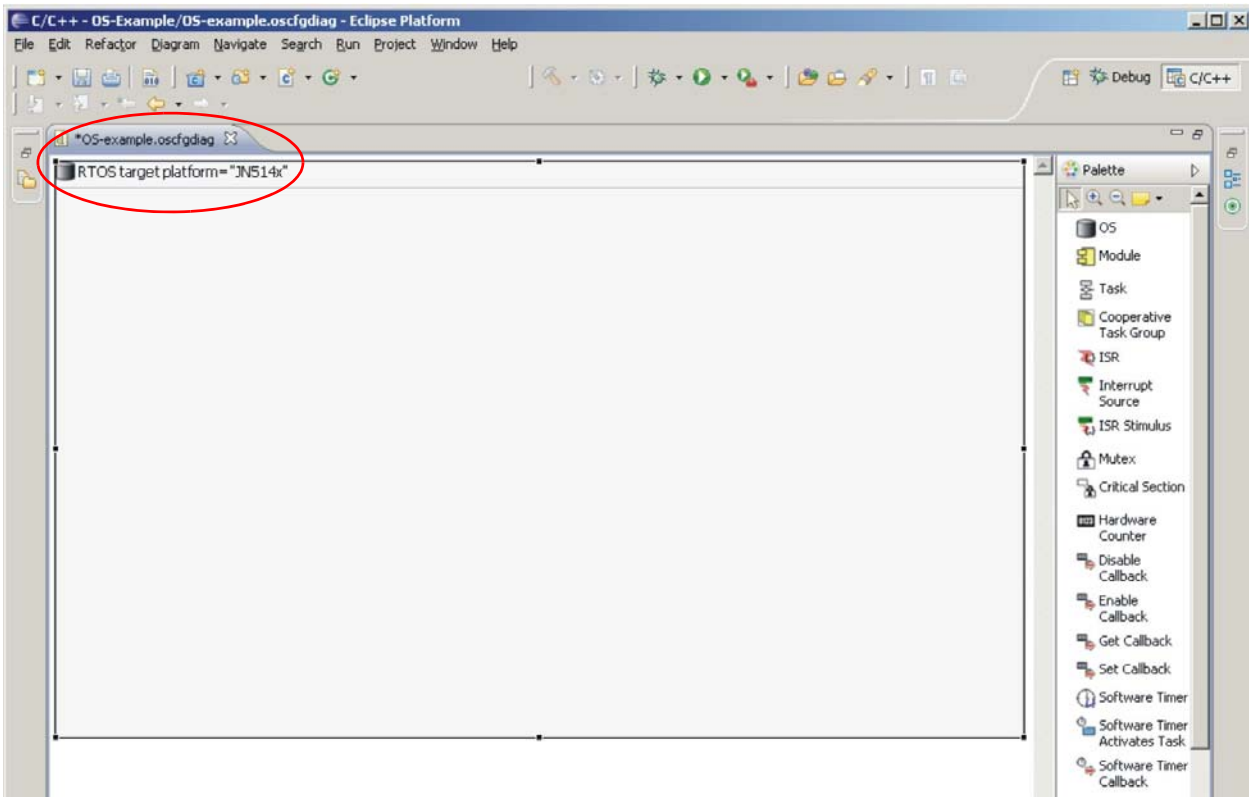


Figure 6: Placing the OS Element

The other method of adding elements to the diagram uses a context-sensitive menu which appears when the cursor is left over (hovered over) different parts of the diagram. We can show this by deleting the OS element that we have placed

Step 3 Click inside the element, so that the outline and sizing handles show, and then press delete, or alternatively right-click in the element and select **Delete from Model** from the menu.

Step 4 Leave the cursor in the diagram pane and, after a short delay, the OS element icon appears next to the cursor. The icon will disappear after a few moments, but moving the cursor slightly will bring it back. When the icon is displayed, move the cursor over it and click on it - this again places the OS element on the diagram.

Note that if the cursor is moved to another blank area of the diagram, it will again show the OS element icon. However, since the tool knows that there should be only one OS element on a diagram, attempts to place another OS element will not work. In the same way, if the OS element is selected on the tool palette and the cursor moved onto the diagram, the cursor changes to the shape below to indicate that the element cannot be placed.

Figure 7: 'Operation Unavailable' Cursor

Whenever you see this icon, it means that it is not possible to perform the operation - it is referred to as the "Operation Unavailable" cursor.

At the top of the OS element, there is a title 'RTOS target platform = "JN514x"'. This indicates the processor family that the diagram is to be built for - in this case, the JN514x family.

14.3.2 Editing the RTOS Properties

Right-clicking on the OS element and selecting "Show Properties View" will open a window at the bottom of the screen as shown in figure Figure 8.

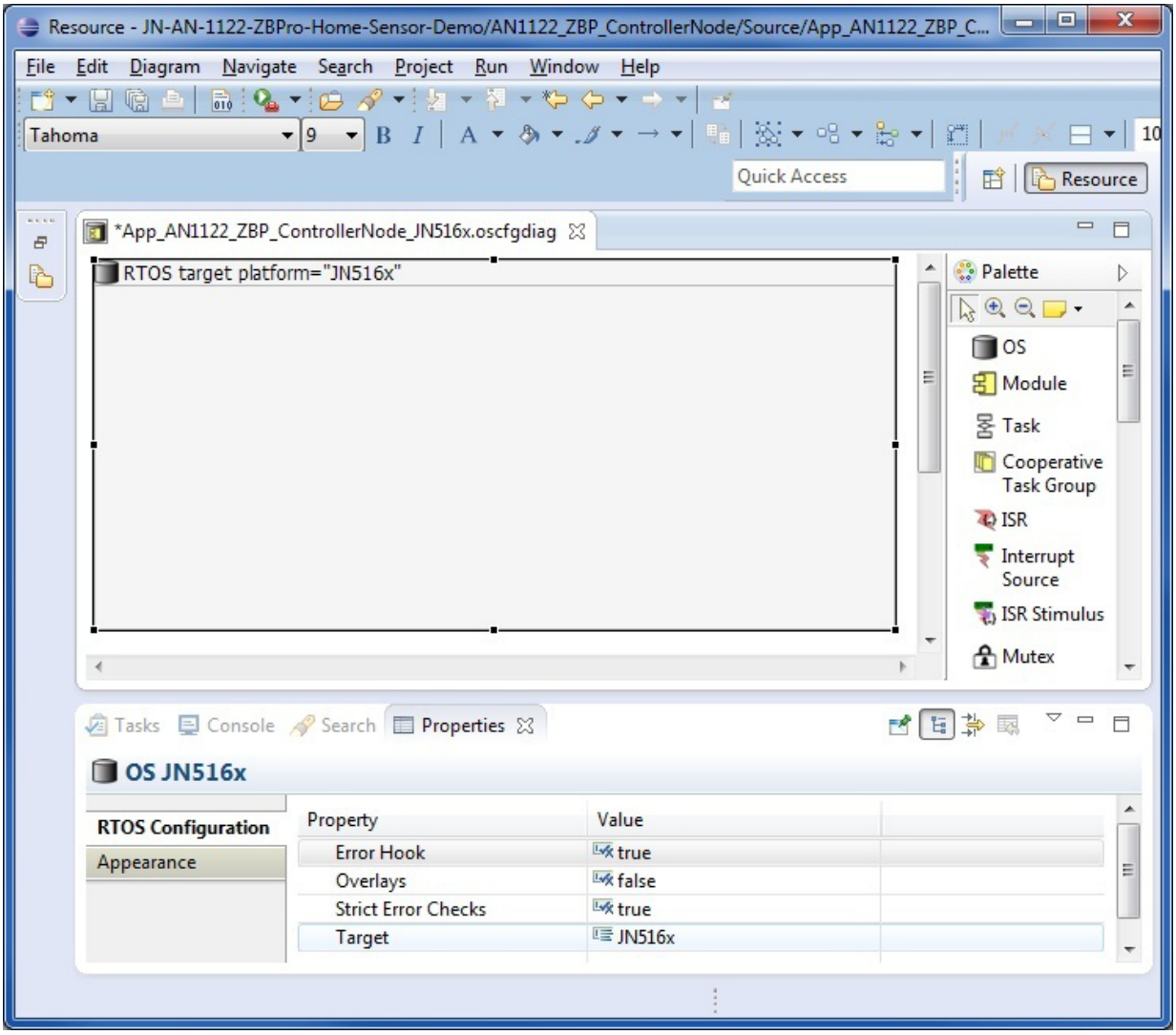


Figure 8: RTOS Properties

The “Error Hook” and “Strict Error Checks” properties control the error hook callback feature described in Section 2.6. Both of these options should normally be set to true.

The “Overlays” property controls the overlay feature described in Section 2.5. This should normally be enabled for a JN514x device and must never be enabled when using a JN516x device.

The “Target” property selects between a JN514x and JN516x device. To create an application that can run on either family of devices, two separate OS diagrams must be used.

14.3.3 Adding a Module

The next item that needs to be added to the diagram is a **Module**. A module is typically a collection of tasks and other OS entities which are logically connected - for example, the application sitting on top of a protocol stack would be one module on a diagram, while the ZigBee protocol stack would be represented by a separate module.

We can place a module inside the OS element by either selecting the **Module** icon in the tool palette or hovering the cursor within the OS element diagram (but not in the OS title box).

Step 5 Add a module using one of the methods already described, and adjust its size to approximately 2/3 of the height of the OS element sheet and half its width.

If you hover the cursor in the unoccupied part of the OS sheet level with the module element, the cursor will indicate another module may be placed in this space.

When a module is added, the title bar is shown empty and needs a name filling in - this name is used to refer to the module within the software that uses the JenOS data structures that are created from the diagram.

In our case, we are producing a module which will hold a simple application program, so it can be named Application. If you have placed a module and not named it, this is not a problem. It is possible to name or rename a module element by right-clicking on the element and selecting "Show Properties View", which will open a window at the bottom of the screen as shown in the figure below.

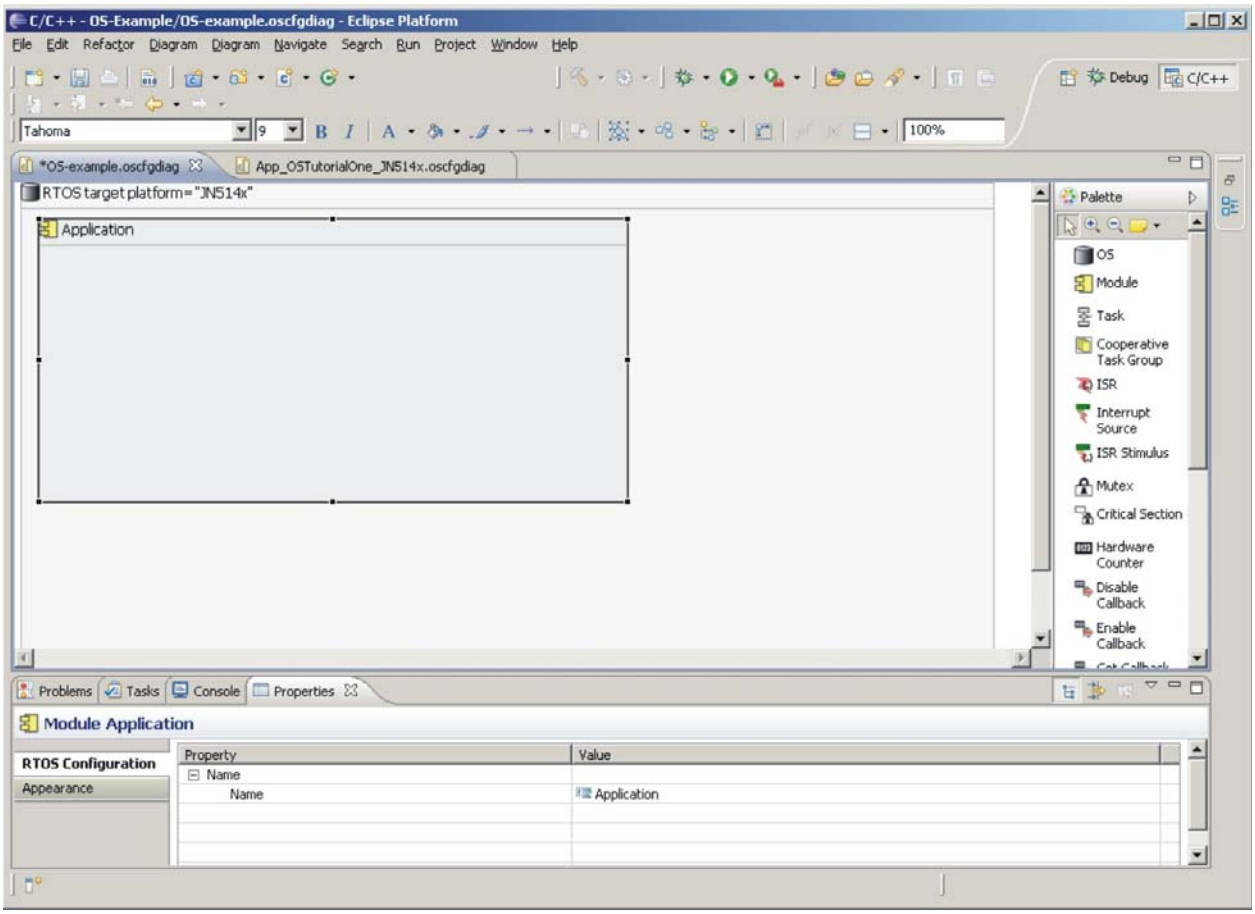


Figure 9: Module Element Properties

The name of the module can be changed by clicking on the **RTOS Configuration** tab and then the **Name** property. As shown in the diagram, the value of the **Name** property for this module is **Application**, but this can be changed by clicking on the **Name** property value field and editing it. You can also change the name of the module by clicking on the title box and typing in the new name.

14.4 Example 1 - Using a Task

In this example, we are going to show how to use a task from JenOS to continuously flash one of the LEDs on a Sensor board from a JN5148-EK010 Evaluation Kit or a Generic Expansion Board from a JN516x-EK001 Evaluation Kit. In the interests of simplicity and to introduce features to the diagram a few at a time, the example uses a simple delay loop to provide the timing for the ON and OFF times of the LED, rather than using one of the software or hardware timers which are available. For more information on tasks in JenOS, refer to Section 2.4.1.

14.4.1 Adding a Task to the Diagram

To add a task to the diagram, we use the same methods as in the previous sections for adding the OS and Module elements, namely selecting the **Task** icon on the tools palette and placing the icon in the Module, or by hovering the cursor inside the active area of the module and clicking on the task icon. However, now when we hover the cursor inside the module, the effect is different from when we were placing the module inside the OS element. In that case, the only icon that can be legally placed inside an OS element is a module, and so only a module icon is visible - now, because we are inside a module, the number of element types that are legal has increased and the module is replaced by a bar of 8 icons, one of which is the task.

Step 1 Add a task so that the diagram looks like the one shown in the figure below.

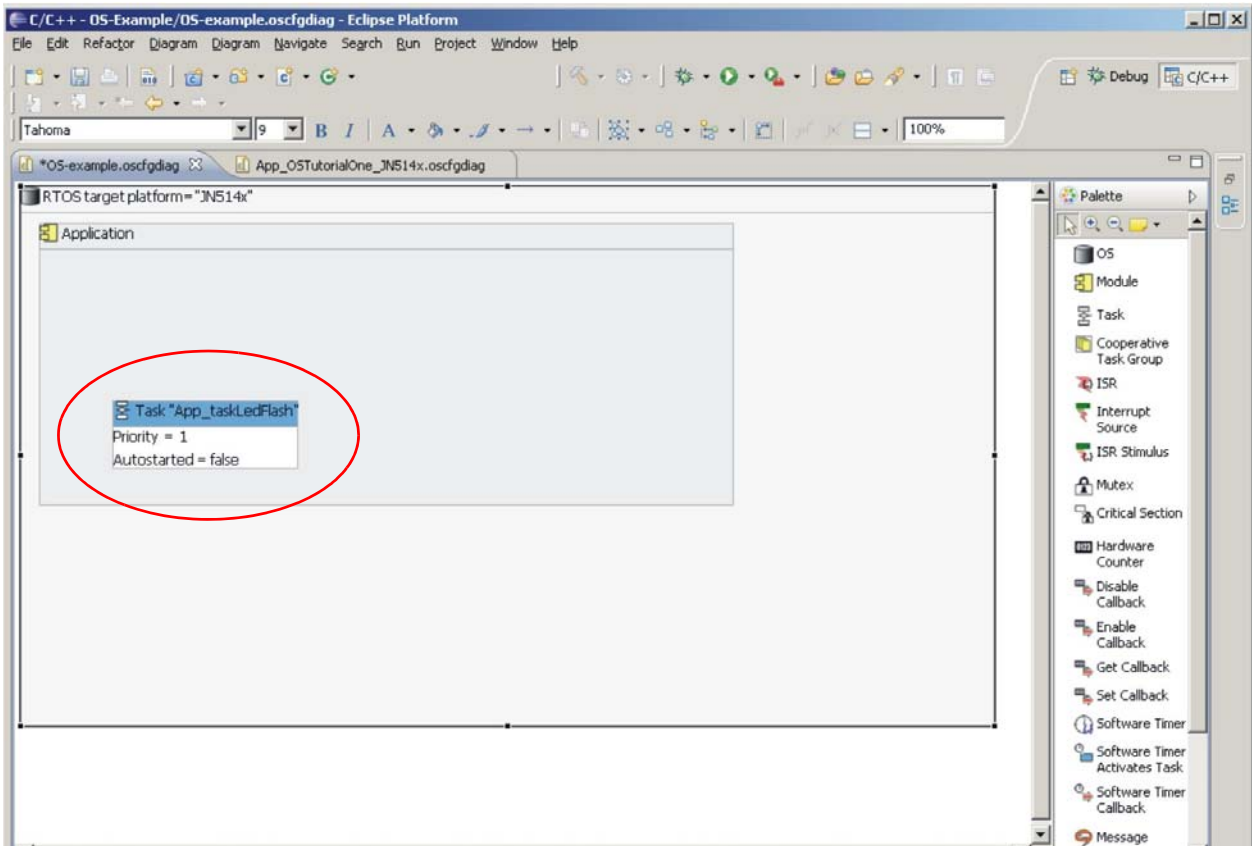


Figure 10: Task Element

Note that the name in the task title has been set to **App_taskLedFlash**. This can be done when the task is placed on the diagram, or by selecting the task title and typing the name, or by right-clicking on the task and selecting **Show Properties**.

The **Show Properties** tab for a task has more entries than that of the Module or OS element. As well as the **Name** field which we set to **App_taskLedFlash**, there is another entry called **Task Details**. This contains two elements, **Autostarted** and **Priority**.

If **Autostarted** is selected by clicking on the name, as well as the line being highlighted, an icon for a drop-down box (a downward-pointing arrow) appears to the right of the value field. Clicking on the arrow reveals two options, **false** and **true**.

- Setting **Autostarted** to **true** will cause the task to start when the JenOS start-up function **OS_vStart()** is called.
- Setting **Autostarted** to **false** means that the task will only start when function **OS_eActivateTask()** is called with the task name as a parameter.

Step 2 Set **Autostarted** set to **true** so that the LED task runs immediately.

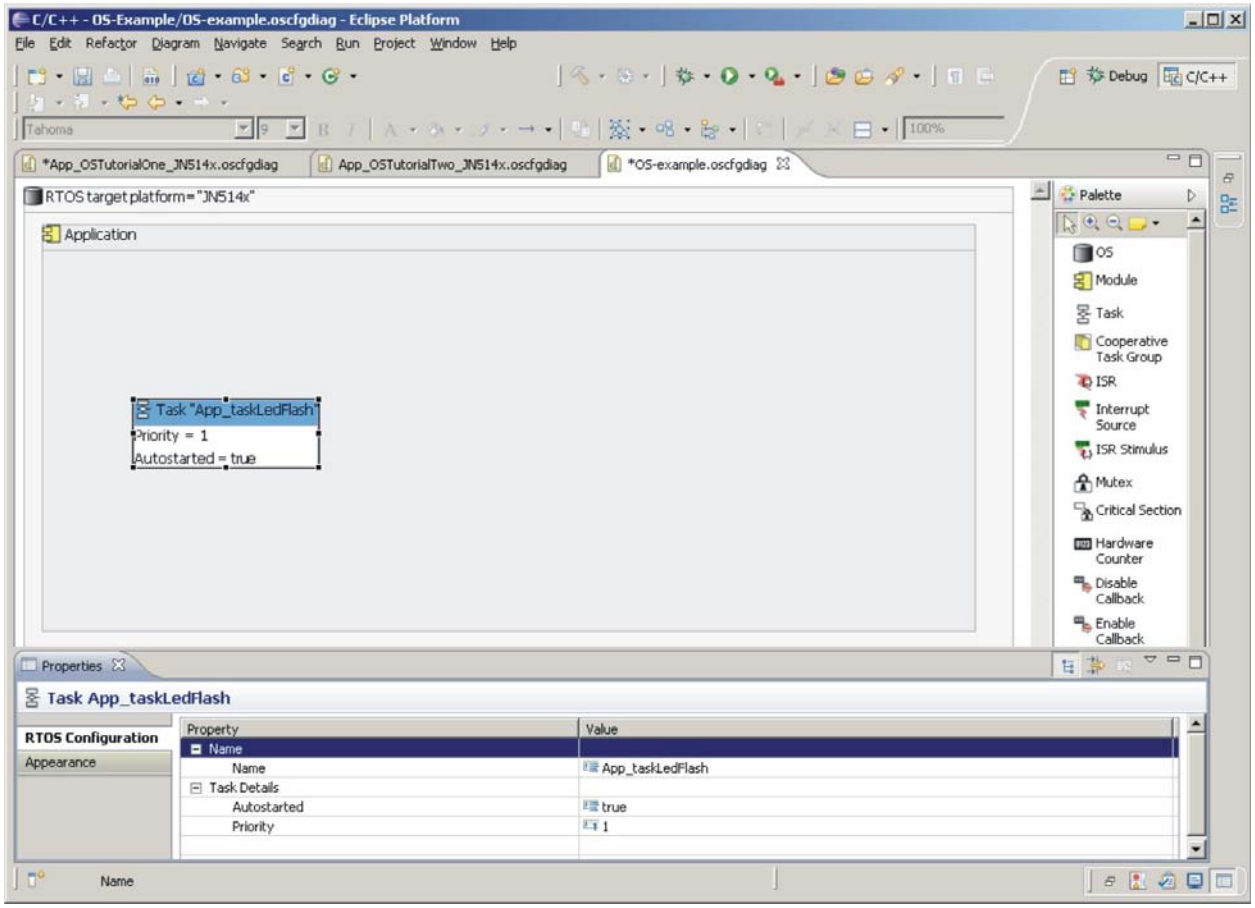


Figure 11: Task Element Properties

The **Priority** setting indicates the priority at which the task will run - the higher the number, the higher the priority. Tasks must not share the same priority. The lowest priority is 1 and the maximum value is 2^{31}-1. However, since there are only 32 tasks available, an upper limit of 32 would be sufficient to represent all tasks running at different priorities.

These properties can also be seen and modified in the body of the task element.

Note that the fact that the task is on the diagram only sets up its data structures - it does not include any code that the task actually runs. This is something that the user must supply - in this case, the code for flashing the LED will look something like:

```
OS_TASK(APP_taskLedFlash)
{
    static bool bLedState = OFF;

    bLedState = ~bLedState;
    SET_LED(LED0,bLedState);

    // Waste some time
    vDelayMsec(500);

    OS_eActivateTask(APP_taskLedFlash);
}
```

Note that the task is declared using the macro **OS_TASK** and the name corresponds to the name of the task on the diagram **App_taskLedFlash**. The task uses the **SET_LED** macro to control an LED on the evaluation kit. The task starts by setting a static variable *bLedState* which holds the state of the LED (ON or OFF) to its initial value of OFF. The state is then complemented and written to LED0, and next the delay routine **vDelayMsec()** is called to implement a 500-ms delay. Finally, the task activates itself again using **OS_eActivateTask(***App_taskLedFlash***)** before exiting.

At first sight, this last statement looks a little odd. We have activated the task again before it has completed - however, this will not cause a problem since the new invocation of the task is called at the same priority level as the one currently running. The consequence of this is that the new invocation is put in the pending state - a task will not pre-empt a running task of the same priority. Thus, the current invocation carries on running and terminates, and then the new instance is allowed to run. This time, the state of the LED will be complemented from ON to OFF and the LED appears to flash at a 1-second interval with 50% duty cycle.

Note: Co-operative tasks can be created by first adding a **Cooperative Task Group** to the diagram from the tool palette, assigning a name to the group and then adding tasks within the boundary of the group. For information on co-operative tasks, refer to Section 2.4.3.

14.4.2 Other Elements Needed on the Diagram

The configuration tool places a further requirement on the diagram - it requires interrupts to be handled in some way, even though they may not be used. To this end, we need to add another couple of entities to the diagram. The first is an **Interrupt Source**, represented by the icon below:

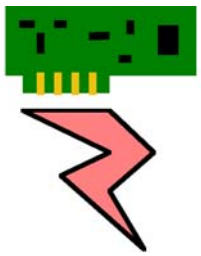


Figure 12: 'Interrupt Source' Icon

Step 3 Place an interrupt source element on the diagram, as shown in Figure 13 below.

The name in the title box of the element defaults to **SystemController** - this is actually one of a number of interrupt sources that are available on the JN51xx device. The options for the interrupt source can be viewed by displaying the properties of the icon (right-click and select "Show Properties View"). Selecting the **RTOS Configuration** tab and expanding the **Interrupt Stimulus** property shows the **Source** field - clicking on the value shown adds a drop-down menu to the right of the value field. Clicking on the drop-down icon reveals the interrupt sources that can be selected, which include hardware devices such as UARTs and timers, and also exceptions generated by the processor (such as alignment errors and stack overflow). In our case, we will select the SystemController as the interrupt source, which controls the interrupts generated from Wake Timers, the DIO pins and the Analogue Comparators.

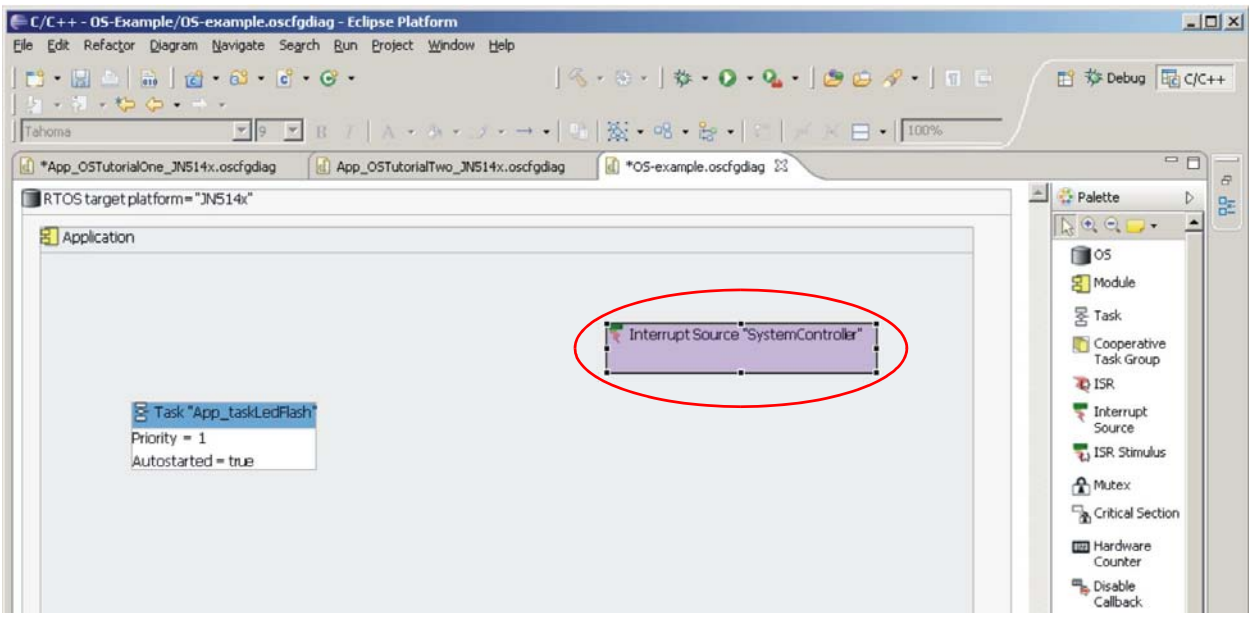


Figure 13: 'Interrupt Source' Element

Once an interrupt source has been added to the diagram, we need to deal with the interrupt using an interrupt handler or Interrupt Service Routine (ISR). The ISR icon is shown below.

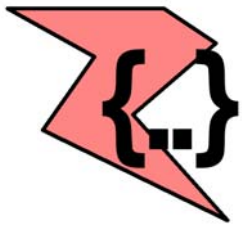


Figure 14: 'ISR Element' Icon

Step 4 Add an ISR to the diagram as shown in Figure 15 below.

The properties of the ISR include a name, which must correspond to the name of the ISR routine provided in the software - in our case, we will use **APP_isrSystemController**. They also include a couple of other properties under the **Interrupt Details** field. Expanding this field shows the properties **IPL** and **Type**:

- **IPL** stands for Interrupt Priority Level, which denotes the relative priority assigned to the interrupt, ranging from 1 to 16, where the higher the number the greater the priority. Thus, an interrupt at IPL1 can itself be interrupted by an interrupt at IPL2, and an interrupt at IPL3 will complete before an interrupt at IPL2 is honoured. Interrupts from any of the hardware peripherals of the JN51xx device can take IPL values from 1 to 15, while the IPL value 16 is reserved for exceptions such as Bus Error, DataPageFault, InstructionPageFault, TickTimerException, IllegalInstruction, D-TLBMiss, I-TLBMiss, Alignment Error, Range Error, System Call, FloatingPoint Error, Trap, UnimplementedModule and StackOverflow xxception. On the JN516x device, the Watchdog can be configured to generate a StackOverflowException rather than resetting the device. The functions to configure this are described in *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*.
- The **Type** field of an ISR can take two values, **Controlled** or **Uncontrolled**. Controlled indicates that the interrupt is being handled through JenOS, while Uncontrolled indicates that the interrupt is handled directly by the user's code. In most situations (as here), an interrupt will be Controlled, but there are circumstances where it is either necessary or more efficient to handle the interrupt outside of the RTOS services. However, when uncontrolled interrupts occur, no OS function calls can be made.

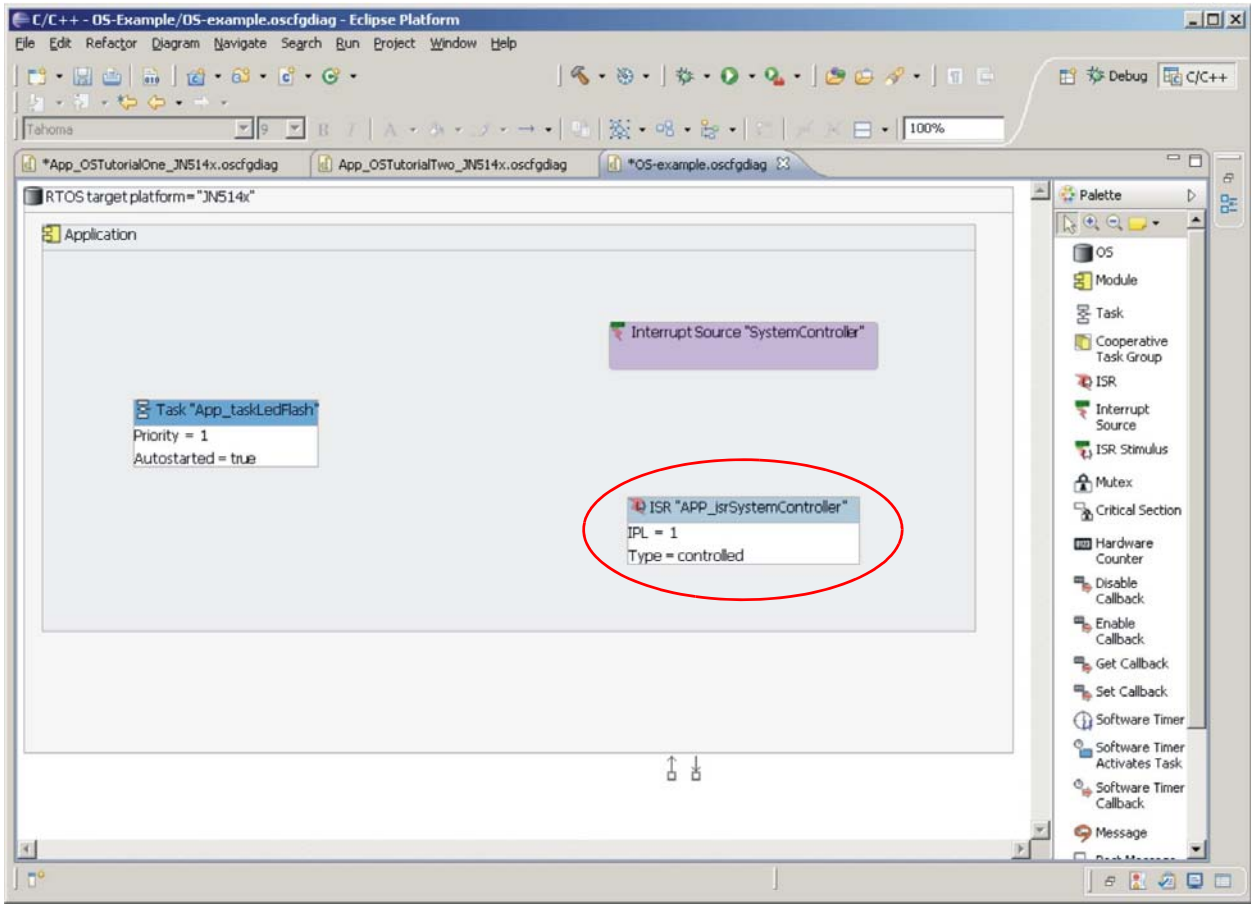


Figure 15: ISR Element

Finally, we need to connect the interrupt source to its handler. This is done using the **ISR Stimulus** icon.

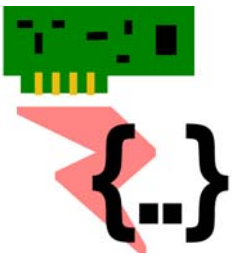


Figure 16: 'ISR Stimulus' Icon

This is a line on the diagram that will run from the interrupt source to the ISR.

Step 5 Click on the **ISR Stimulus** icon to select it, and hover the cursor on the diagram outside the Interrupt Source element. Note that the cursor is an arrow with the "Operation Unavailable" icon attached (see Figure 7), denoting that the stimulus cannot be placed.

Step 6 Move the cursor over the **Interrupt Source** element - the cursor will change to remove the **Operation Unavailable** icon, meaning that it is legal to place one end of the line in this element. Hold down the left mouse button and drag the cursor into the **ISR** element and release. There should now be an arrow connecting the two elements, as shown in the figure below.

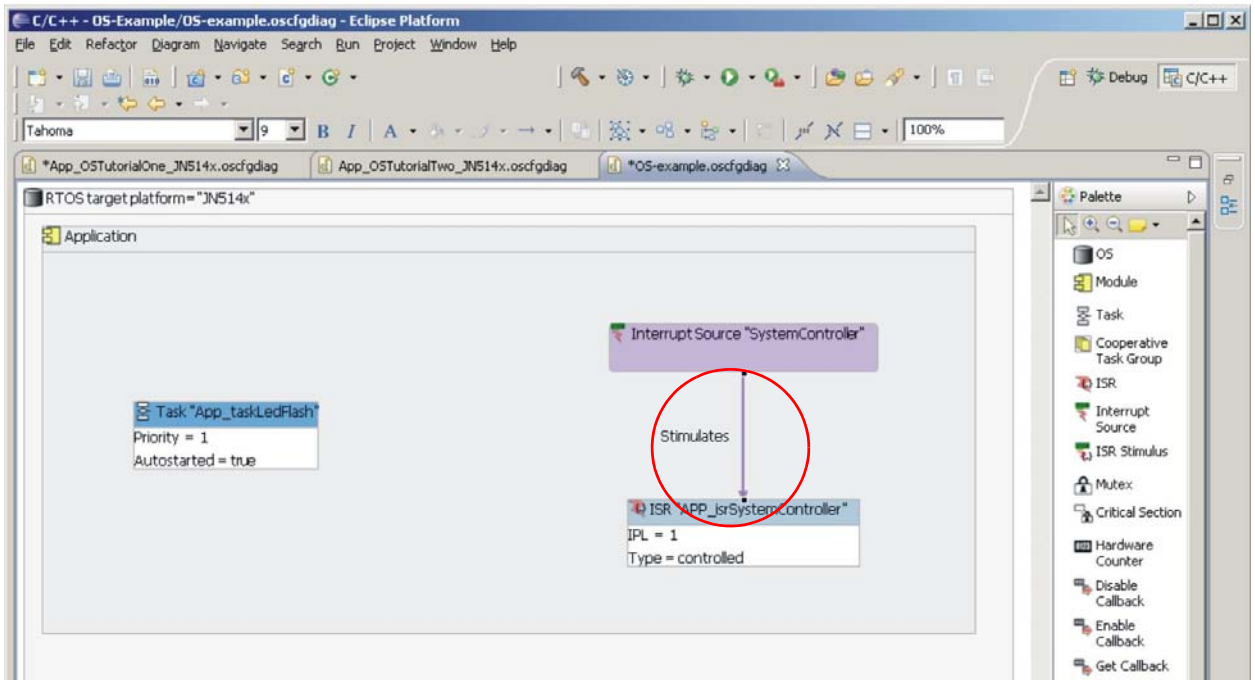


Figure 17: 'ISR Stimulus' Element

Notice the text next to the line "Stimulates" - this label can be re-positioned by selecting it and dragging, if the diagram becomes cluttered in a particular area. During the drag process, the text is connected by a line to the arrow with which it is associated.

The code that corresponds to the ISR on the diagram is shown below:

```
OS_ISR(APP_isrSystemController)
{
    /*
     * ISR handles Wake Timer, Comparator and DIO
     *  interrupts
     *
     * Clear pending interrupt flags by reading the status
     * registers
     */
    (void)u8AHI_WakeTimerFiredStatus();
    (void)u8AHI_ComparatorWakeStatus();
    (void)u32AHI_DioWakeStatus();
}
```


Note that the declaration of the ISR uses the macro **OS_ISR()**, and that the name of the routine corresponds with that used in the **ISR** entity on the diagram.

You can see a full implementation of the application code by looking in the **OS_tutorial/Source/OSTutorialOne** directory in the ZIP file for this manual. Here, you will find files **app_startOne.c** and **app_os_tutorialOne.c**:

- **app_startOne.c** contains all the start-up code for the application, initialising the hardware and starting JenOS, which in turn starts the LedFlash task
- **app_os_tutorialOne.c** contains the routines needed for the task and interrupt handler shown in the diagram.

Instructions on building applications and downloading the resulting binaries to JN51xx-based boards can be found in the *SDK Installation and User Guide (JN-UG-3064)*.

14.5 Example 2 - Hardware and Software Timers

While the code in Example 1 works, it is not a very elegant way of creating a periodic event such as switching a LED on and off, especially when the JN51xx device has hardware timers specifically for producing accurate timed intervals. This example shows how to use one of the timers on the device to trigger two tasks to flash two LEDs. The structure of the example also shows one task being pre-empted by another.

14.5.1 Adding the Hardware Counter

Step 1 Start a generic diagram by opening a blank **.oscfgdiag** document, as described in Section 14.2, and add the OS element and a Module called **Application**.

Step 2 Add a hardware counter to the diagram as shown in Figure 18 below and name it **APP_cntrTickTimer**.

The hardware counter is going to act as the timing source for two software timers, which will be used to start tasks.

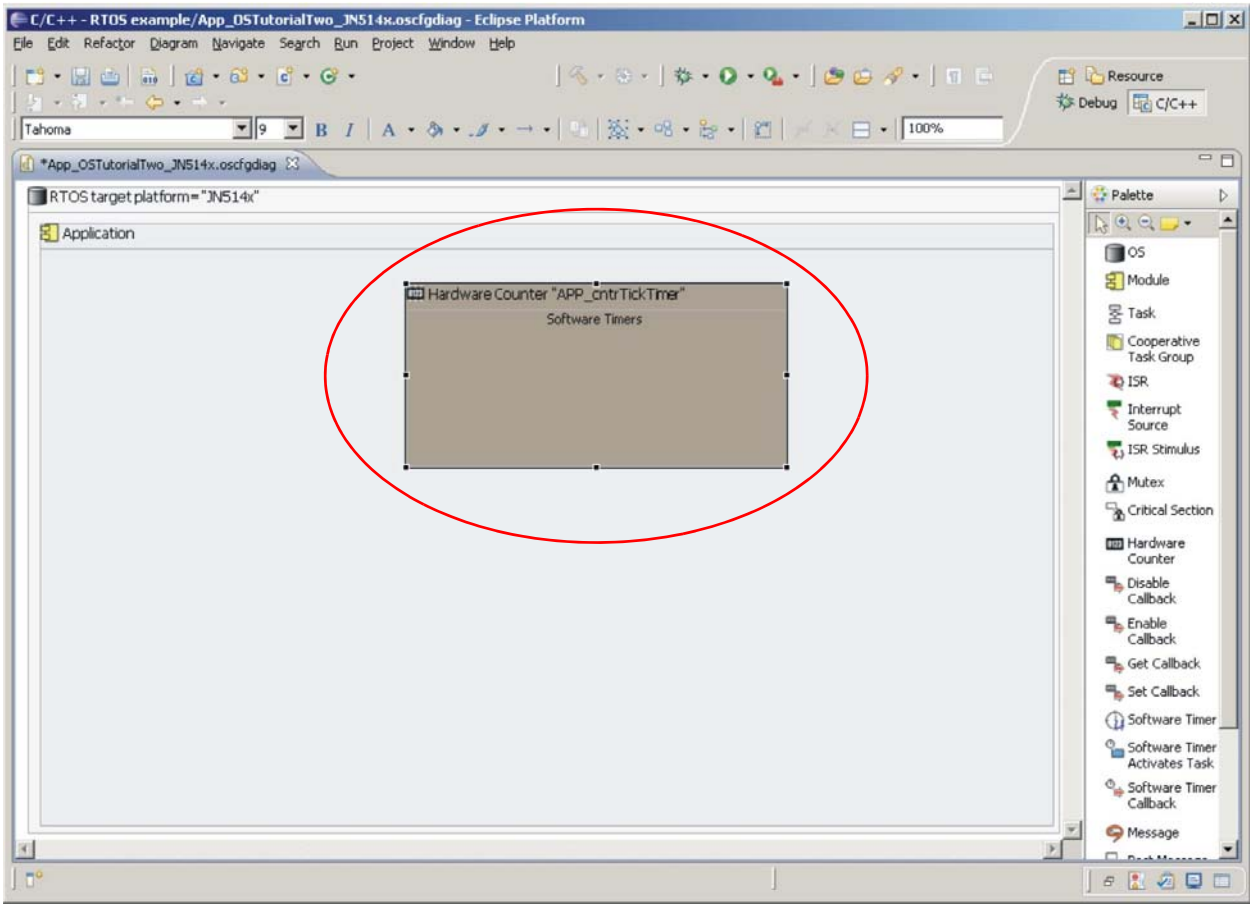


Figure 18: 'Hardware Counter' Element

The only property available on a hardware counter is its name - the name which we have used indicates that we are going to use the Tick Timer on the JN51xx device as the hardware timing element.

How the hardware counter works

We need to be able to control the hardware timer. There are four operations provided as built-in functions by JenOS - these are:

- **Enable**, used to start the timer
- **Disable**, used to stop the timer
- **Get**, used to read the current value of the timer
- **Set**, which writes a timeout value to the counter's compare register (this is the value at which the timer will expire and generate an interrupt)

These operations are provided as callback functions, which means they are user-supplied routines which are called by JenOS when needed - see Section 2.4.6. The callback functions are defined using JenOS macros, detailed in Section 7.1. More information on the use of hardware counters is provided in Appendix B.

Step 3 Add four callback routines to the diagram, as shown in Figure 19 below, naming them:

- **APP_cbEnableTickTimer**
- **APP_cbDisableTickTimer**
- **APP_cbGetTickTimer**
- **APP_cbSetTickTimerCompare**

Step 4 Link the callback functions to the hardware counter using the **Enable Callback**, **Disable Callback**, **Get Callback** and **Set Callback** connectors. These work in the same way as the **ISR Stimulus** connector was used to link the Interrupt Source and ISR in Section 14.4.2 - the cursor shape will indicate whether the cursor is over a part of the diagram where the ends of the connector may start of finish. Start by placing the connector at the hardwire counter and then drag to the appropriate callback element.

Note: The callback routines are in the SDK installation directory, in **components/utilities/source/app_timer_driver.c**. For an explanation of how the hardware counter works in general, see Appendix B.

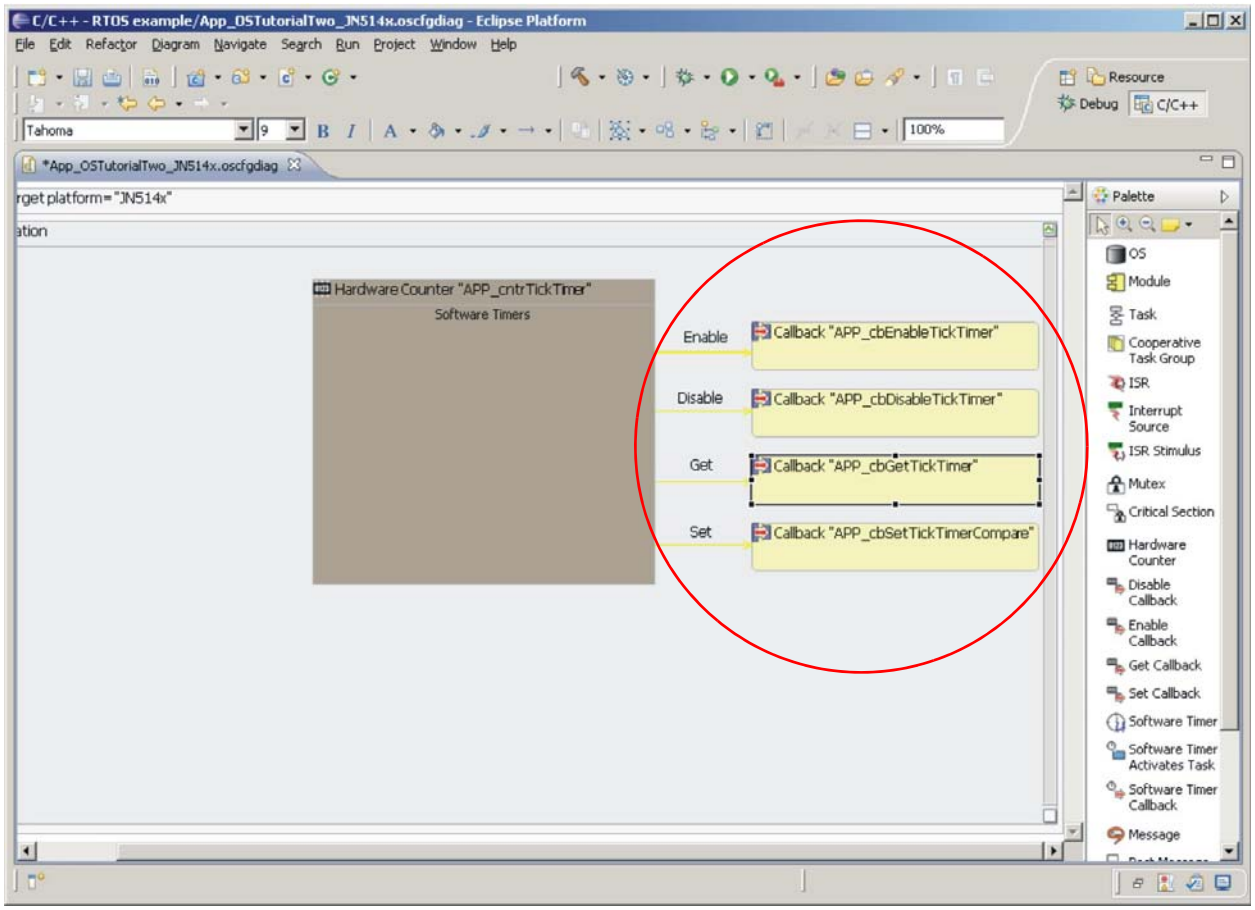


Figure 19: Callback Elements and Connectors

Step 5 Add the above interrupt source and ISR to the diagram, as shown in the figure below.

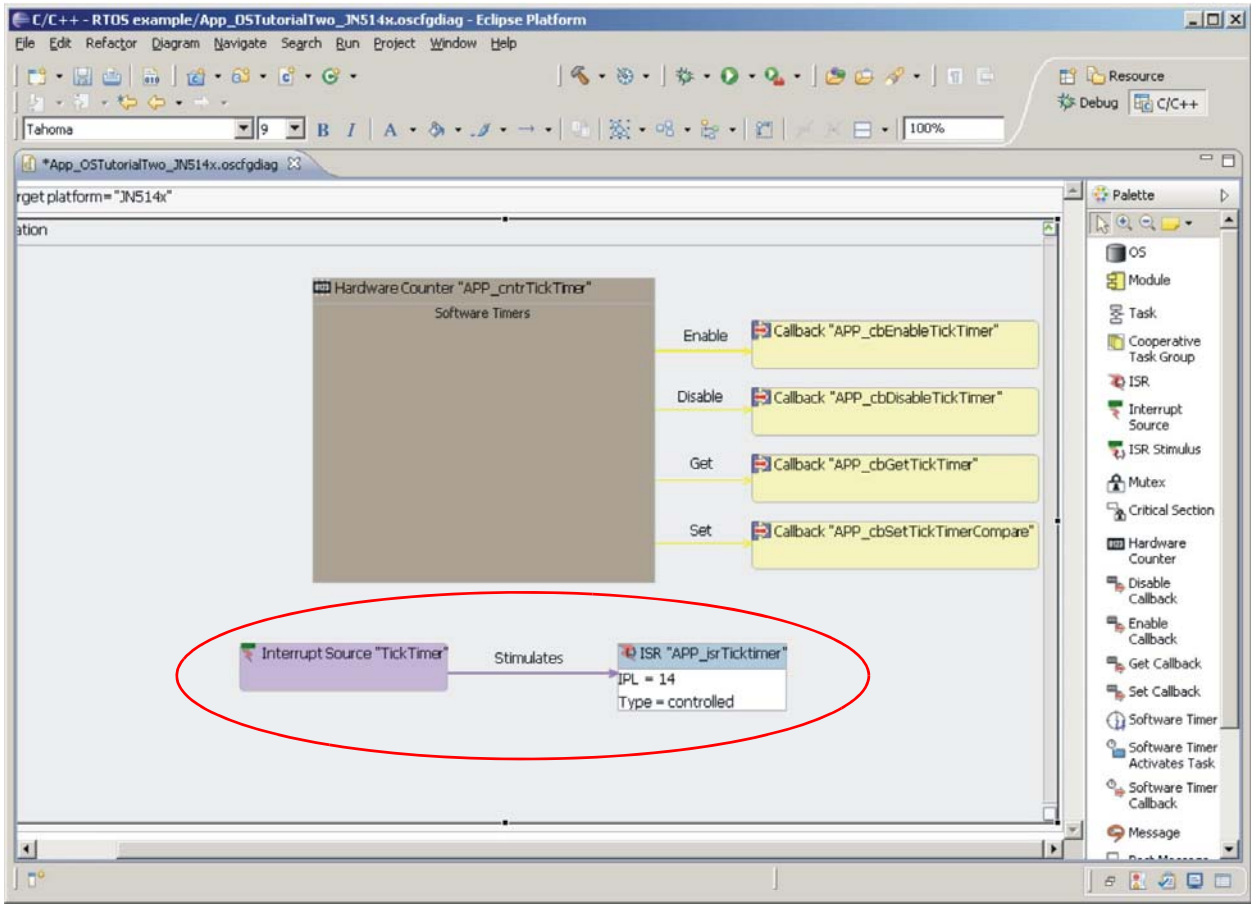


Figure 20: Interrupt Source and ISR

This time, the interrupt source is set to be the Tick Timer, using the **Property Edit** window, and the ISR name is **APP_isrTickTimer**. The Tick Timer interrupt priority has been set to 14, so that it will be serviced in preference to most other interrupts.

The code for the ISR is also found in the file **app_timer_driver.c**.

14.5.2 Adding the Software Timers and Tasks

As noted in Section 14.5.1, hardware counters are used as the timing source for software timers. In the diagram, you can see that the hardware counter has an area for software timers - we are going to instantiate two timers to run off the Tick Timer counter.

Step 6 Select the **Software Timer** icon from the tool palette and place a copy inside the **Hardware Counter** box, directly under the "Software Timers" title.

Step 7 Name the timer **APP_tmrFlashLed0** and then add another software timer next to it, named **APP_tmrFlashLed1**. Rearrange the diagram as shown in Figure 21 below.

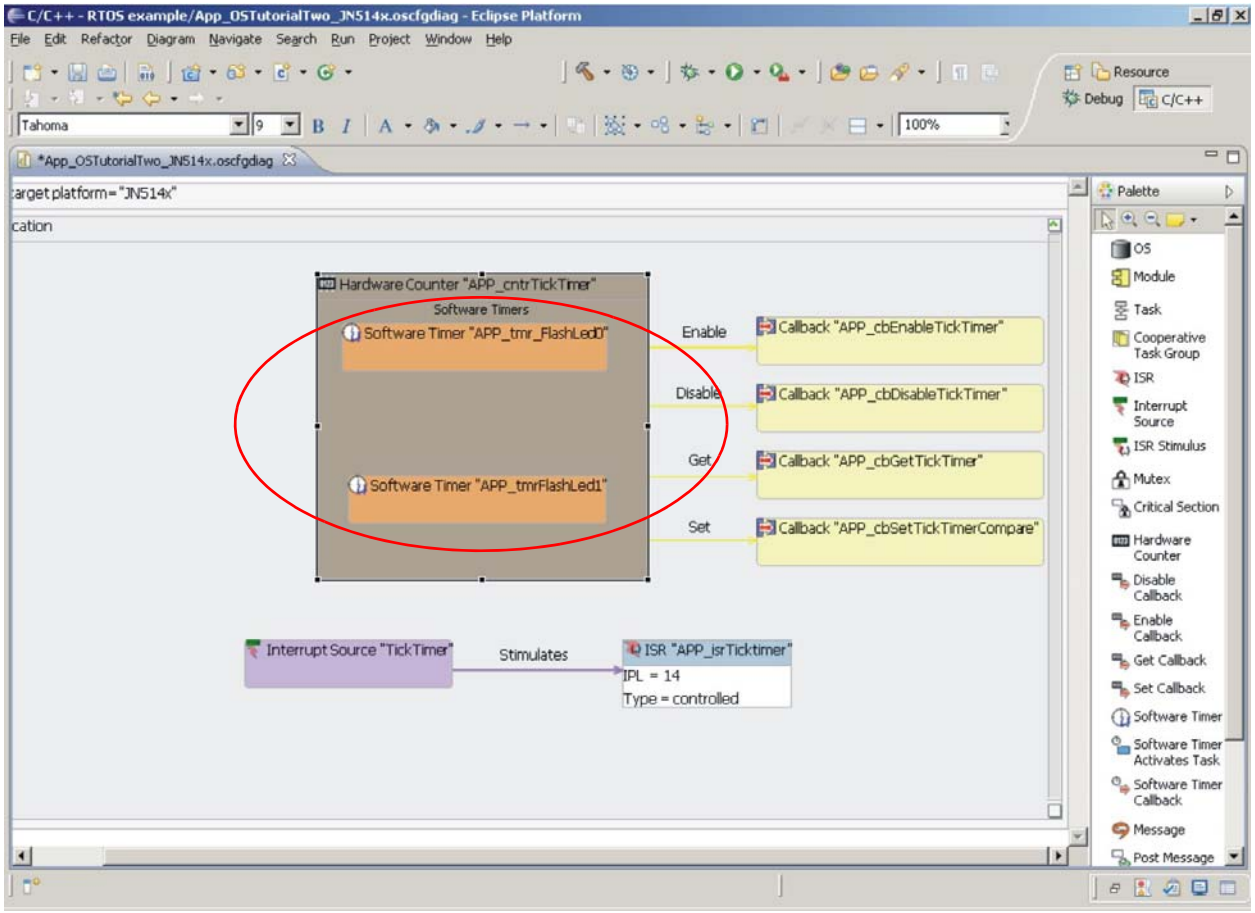


Figure 21: Software Timers

Placing the timers within the **Hardware Counter** box means that they will now be driven by the expiry (interrupt) from the Tick Timer and will be checked to see if they have also expired by the **OS_eExpireSWTimers()** call in the ISR - the expiry of a software timer can be used to generate an event which can be used to activate a task. We are going to add two tasks to the diagram, one driving LED0 and the other driving LED1, and use the timeout events generated from the software timers to activate them.

Step 8 Add two tasks called **APP_taskFlashLed0** and **APP_taskFlashLed1** to the diagram as before, and connect them to their respective software timers using the **Software Timer Activates Task** connector, as shown in Figure 22 below.

Note that the Led0 task has priority 10 and the Led1 task has priority 20. Their **Autostarted** state is set to **false**, since the timer events will activate them.

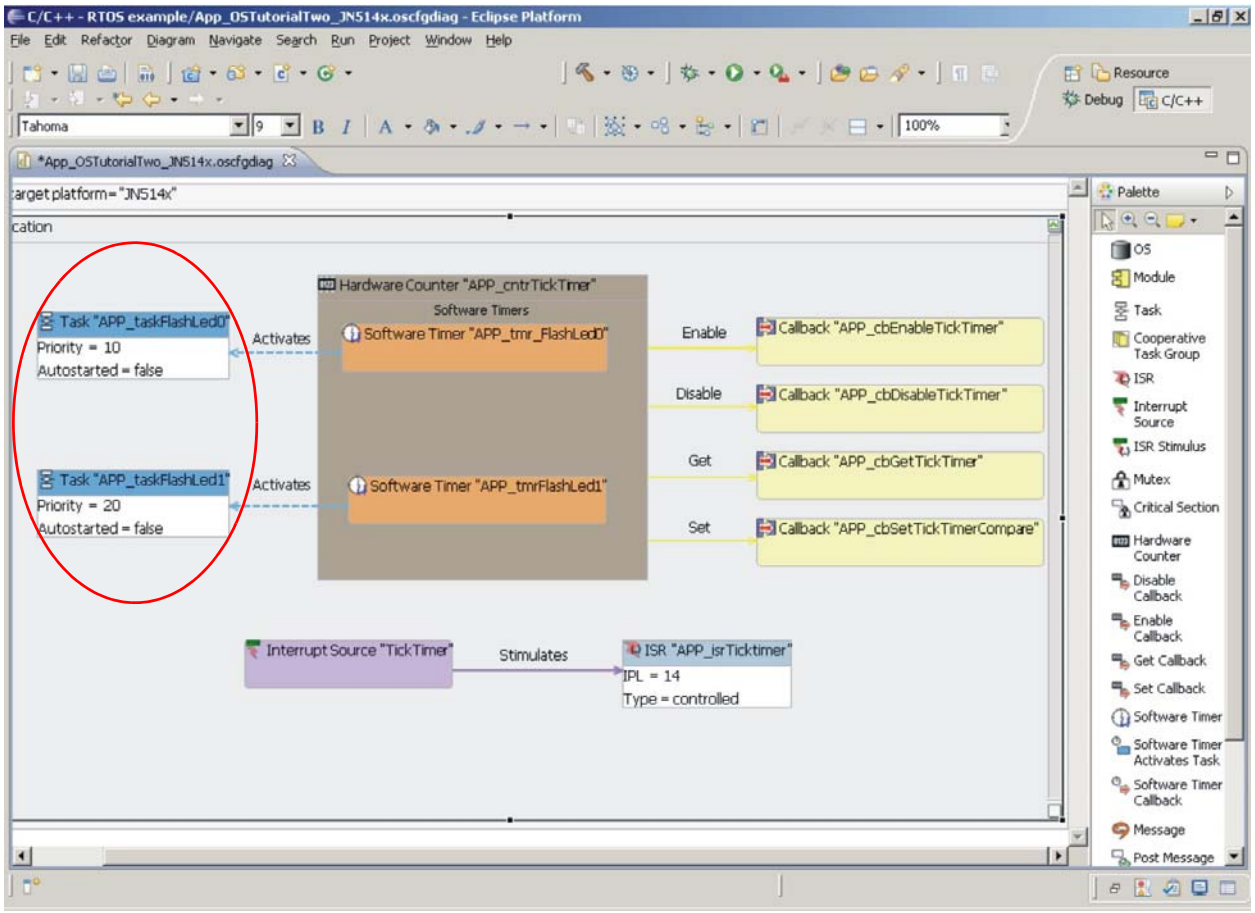


Figure 22: Flash LED Tasks

We are going to use the tasks to control the two LEDs on a Sensor board from a JN5148-EK010 evaluation kit, in a similar manner to the task in Example 1 in Section 14.4, except that instead of using a delay loop to schedule the re-activation of the task, the timer event will be used.

The code used to do this can be seen in the file **app_os_tutorialTwo.c**, supplied in the ZIP file for this manual. Each task essentially contains a static variable to hold the current state of the LED that it is controlling. On each activation, the LED state is complemented before it is written out to the LED, switching it from ON to OFF or vice versa. The task then resets its software timer to expire in the future and then terminates. Expiry of the timer after the specified period re-activates the task and the cycle repeats.

```
OS_TASK(APP_taskFlashLed1)
{
    static bool bLedState = OFF;

    /* Toggle the led state */
    bLedState = ~bLedState;

    /* Write the new state to the hardware */
    SET_LED(LED1,bLedState);

    /* Set up sw timer to reactivate task again */
    OS_eContinueSWTimer(APP_tmrFlashLed1,
                        FLASH_TIME_1,
                        NULL);
}
```

The flash rates that the timers use are set differently, so LED1 will flash faster than LED0 - their rates are also chosen to be unsynchronised, so that at various points in time both tasks will be activated at the same time. However, due to the Led1 task having higher priority than the Led0 task, it will pre-empt Led0 (if it is running) and cause Led0 to wait before its new state is written out, causing 'jitter'.

This is also reflected in the trace output that can be monitored using a terminal emulator connected to UART0 of the board. Whenever the tasks are started or end, they write out a message. Normally, one would expect to see matching task start/end message pairs in sequence, when neither task interferes with the other. However, at the points where the Led1 task pre-empts the Led0 task, you will see a sequence where the Led0 task starts and then the Led1 task starts and ends before Led0 completes.

The code to initialise the application is held in **app_startTwo.c**, where **vAppMain()** is the entry point and is used to set up the UART hardware to allow the trace output to be used, and then JenOS is started, calling routine **InitialiseApp()** in the process. Here, the LEDs are initially switched off, the Tick Timer is initialised and the two software timers are started.

14.6 Example 3 - Using Messages and Queues

In Example 2 in Section 14.5, we saw how to use hardware and software timers to implement periodic activations of tasks. Now we are going to extend that example to allow us to show how Message Queues can be used to allow tasks to communicate.

In this example, we are going to add another task which reads and debounces the state of the switches on the JN51xx Sensor board. The switch task passes the state of either switch (i.e. pressed or not pressed) to the task controlling LED1 through a message which connects the two tasks. When switch SW0 is pressed, LED1 begins flashing and when switch SW1 is pressed, LED1 stops flashing and is switched off.

14.6.1 Adding the Switch Scan Task and ISR

Step 1 Take a copy of the diagram in Example 2 and add a task **APP_taskButtonScan**.

This task is going to be activated on a regular basis to look at the state of the switches it is monitoring - we do this using a software timer.

Step 2 Add another software timer **APP_tmrButtonScan** to the Tick Timer hardware counter and connect the timer to the task using the **Software Timer Activates Task** connector, as shown in the figure below.

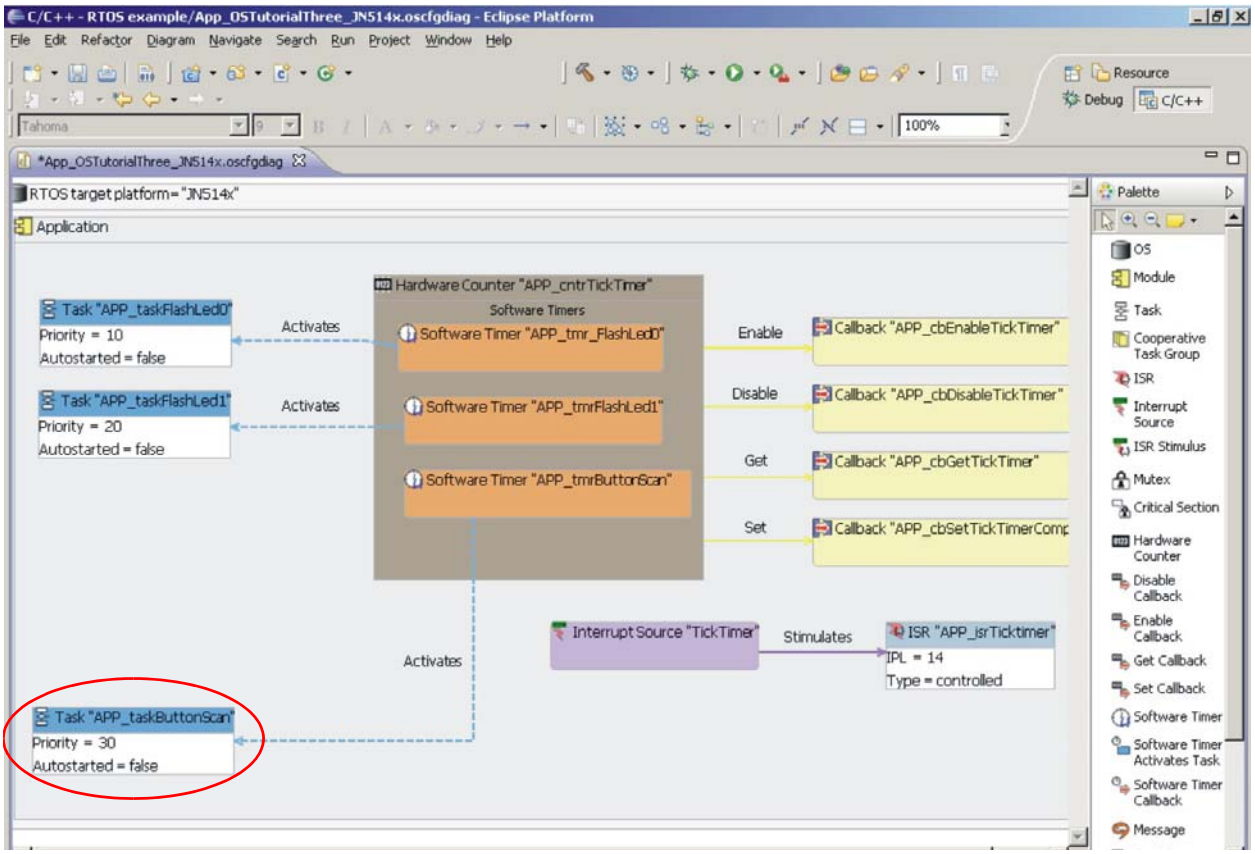


Figure 23: 'Button Scan' Task

Notice that the priority of the ButtonScan task is set to 30, meaning it is the highest priority task on the diagram.

We are going to use the interrupt generated when the switches connected to the DIO lines of device are pressed, so we need to add an interrupt source and ISR to deal with them. The DIO lines are controlled by the SystemController interrupt source.

Step 3 Add the above interrupt source and ISR (called **APP_isrSystemController**) to the diagram, as shown in the figure below.

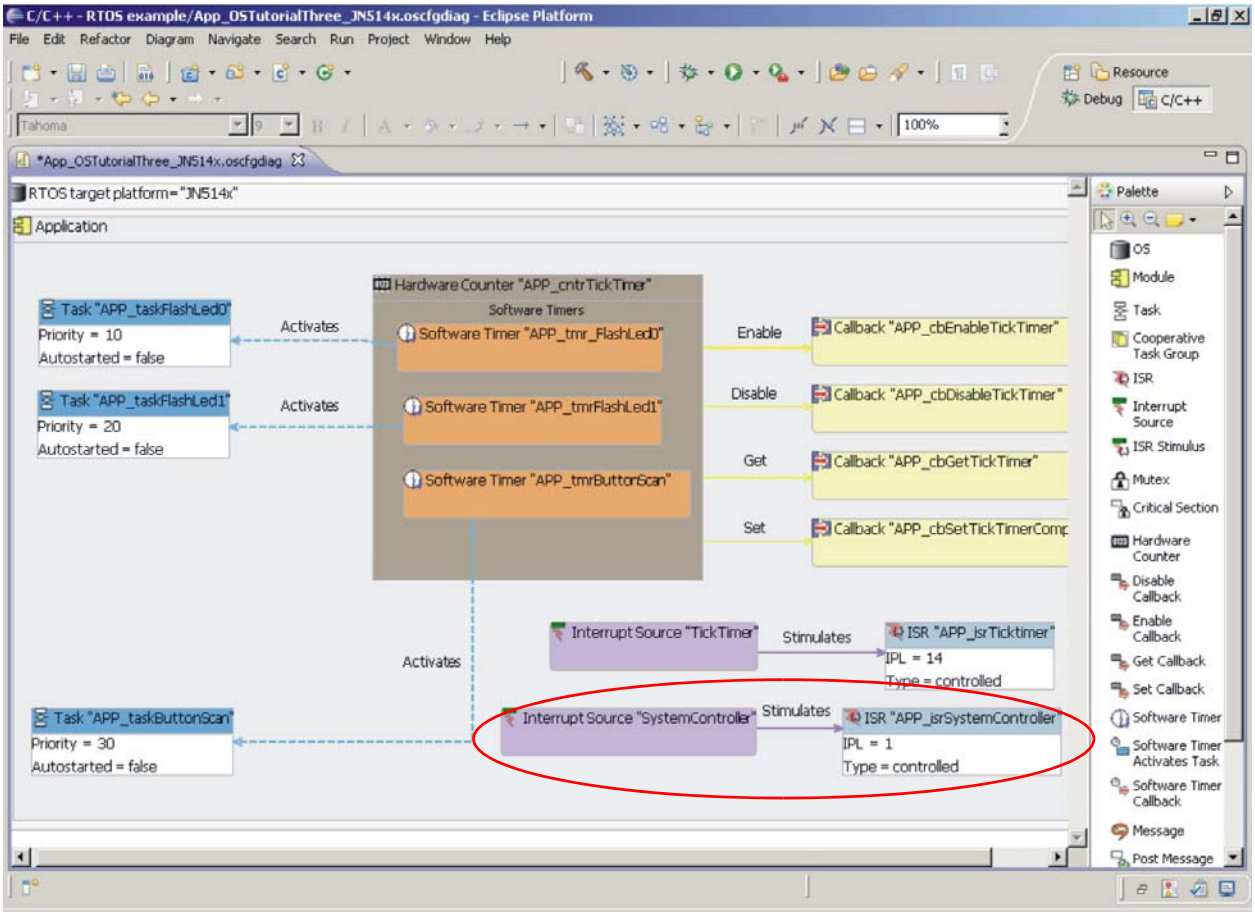


Figure 24: System Controller Interrupt and ISR

14.6.2 Adding the Message Queue

To complete the diagram, we need to provide a means for the ButtonScan task to send the states of the switches to the FlashLed1 task. This is done by means of a **Message Queue**.

Step 4 Select the **Message** icon to add a message element to the diagram, as shown in Figure 25 below.

Step 5 Using the **Property** view on the Message, name the message **APP_msgButtons** and set the queue size to 4, meaning that it can accommodate up to 4 messages in a queue.

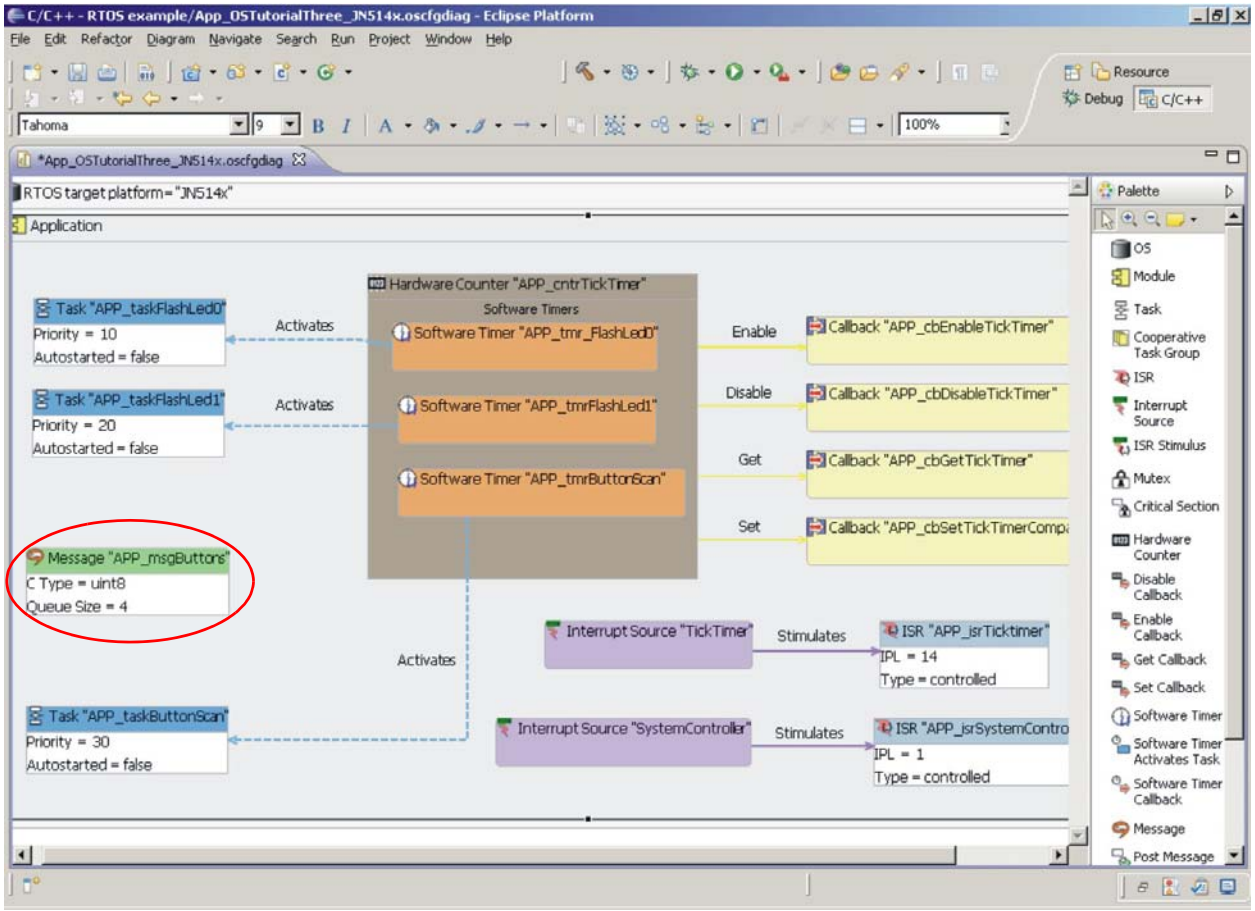


Figure 25: Message Element

Each message in the queue is of the same type, specified by the C Type property. In this case, we are posting information that is of type **uint8** (i.e. unsigned integer 8 bits wide). Single objects up to 32 bits in size can be posted directly to the queue. Values or objects posted to the queue are copied into the queue storage and so persist until they are delivered.

An object which is larger than 32 bits must be specified in a structure. All such structures must be defined in or included in the header file **os_msg_types.h**.

Step 6 To connect up the tasks and Message queue, make the following additions to the diagram (also refer to Figure 26):

a) To connect the ButtonScan task to the Message queue, use the **Post Message** connector to allow the task to send the switch press information to the queue.

b) To allow the FlashLed1 task to be activated when a message arrives in the queue, connect the Message queue to the FlashLed1 task using a **Message Notifies Task** connector.

c) To allow the FlashLed1 task to receive messages from the Message queue, connect the FlashLed1 task to the Message queue using the **Collect Message** connector.

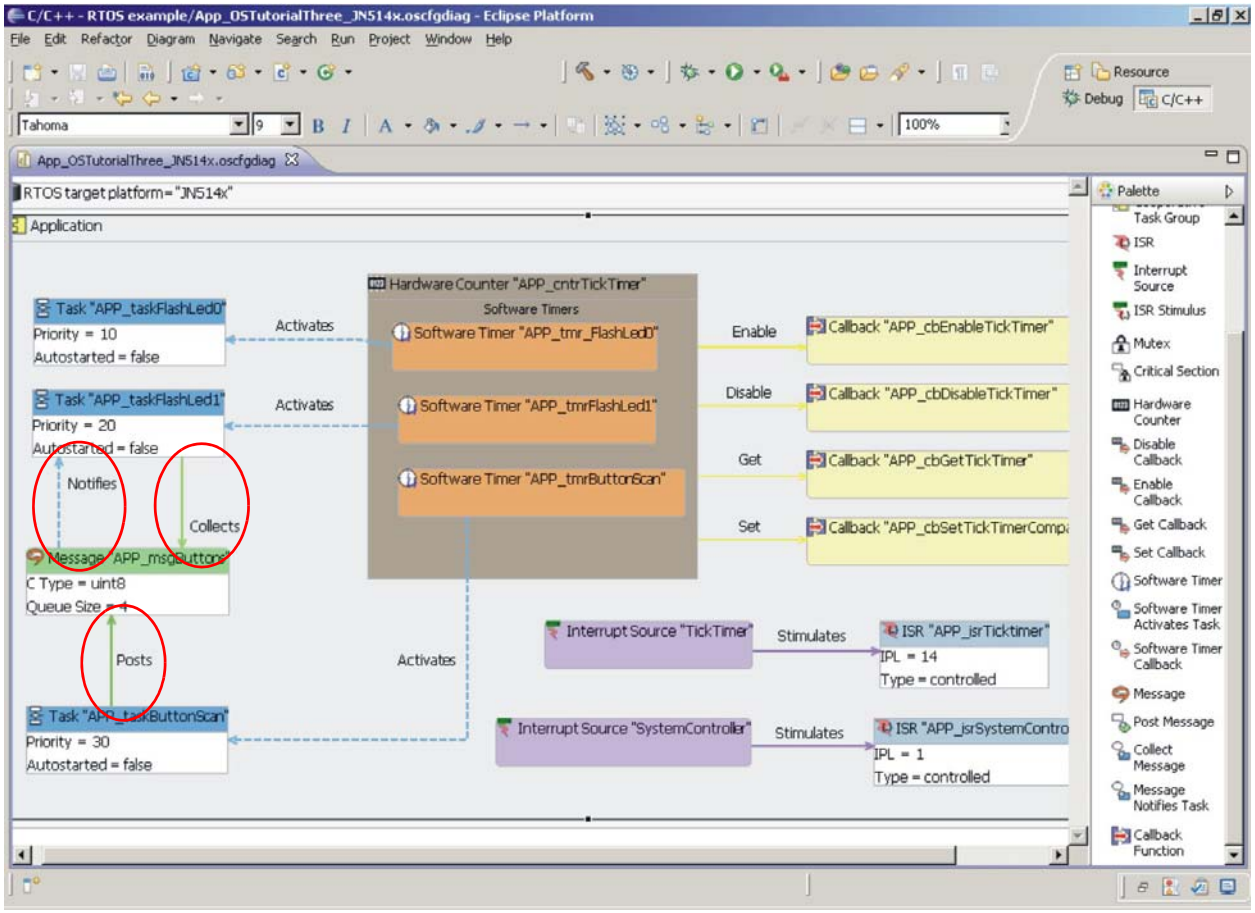


Figure 26: Connecting the Message Queue

14.6.3 How it Works

The ButtonScan task works as follows to debounce both switches on the board. The switches are each connected to separate DIO pins on the JN51xx device. The **SystemController** interrupt mechanism is set up to generate an interrupt when the IO lines show a button has been pressed - in other words, when a 1-to-0 transition is seen on one of the switch pins. This causes **APP_isrSystemController** to be called, and this reads the System Controller interrupt status register to determine if either of the pins corresponding to the switches caused the interrupt. If they were the source of the interrupt, the ISR disables the edge-detect interrupt in preparation for the debounce algorithm run by the ButtonScan task, and then sets up the software timer **APP_tmrButtonScan** to activate the ButtonScan task in 5 ms - this is the time interval between consecutive samples of the pins.

When the ButtonScan software timer expires, the ButtonScan task is activated and reads the status of the DIO pins connected to the switches. To simplify the description, we will look at how the debounce algorithm works for one switch only.

The algorithm uses a set of 8 samples of the switch state to determine if a switch value has stabilised, collected over 8 consecutive invocations of the task. The value of the DIO pin for the switch is isolated and then fed into the least significant bit of a byte-wide variable used to contain the 8 samples. Before storing the new DIO bit, the contents of the store are left-shifted one bit to make room for the new sample. When the store contains 0xff or 0x00, the switch is deemed to be in a stable state. The following code fragment would perform this set of operations:

```
OS_TASK(APP_taskButtonScan)
{
   /* Storage for 8 samples of switch */
   static uint8 u8Debounce = 0xff;


    /* Read the IO port that the button is connected to
     * Mask the IO line connected to the button and
     * shift it to the lsb of u8Button
     */
    uint32 u32DIOState =
        u32AHI_DioReadInput() & BUTTON_BIT_MASK;

    uint8 u8Button =
        (uint8)((u32DIOState >> BUTTON_BIT_POSITION) & 1);

    /* Now add the latest button state into the sample
     * store
     */
    u8Debounce <<= 1;
    u8Debounce |= u8Button;

    switch (u8Debounce & 0xff)
    {
    case 0:
        u8SwitchState = PRESSED;
        break;
    case 0xff:
        u8SwitchState = NOT_PRESSED;
        break;
    default:
        u8SwitchState = UNSTABLE;
    }
```

At this point, the state of the switch, including the current sample point, is known. On finding that a switch is in a stable pressed state and has transitioned from the unpressed state, the switch state is posted into the message queue.

```
    /* don't do anything if the switch is changing */
    if (UNSTABLE == u8SwitchState) exit;

    /* Only post something when the switch is pressed */
    if ((PRESSED == u8SwitchState) &&
       (u8SwitchState != u8LastSwitchState))

        /* Post a message to the queue if the switch has
         * changed state from NOT_PRESSED to PRESSED
         */
        OS_ePostMessage(APP_msgButtons,
                        (void*)&u8SwitchState);

    /* record the current switch state so we can
     * check if it has changed next time around
     */
    u8LastSwitchState = u8SwitchState;
}
```

In addition (but not shown in the code fragment above), if the debounce algorithm detects that both switches are in the stable unpressed state, it re-enables the edge detection for a button press transition to be ready to run the debounce process again if a press is detected. Otherwise, it will restart the **APP_tmrButtonScan** timer to perform another scan and debounce cycle in 5 ms time.

As stated earlier, this is a simplification of the code that is needed to send messages which show when SW0 or SW1 is pressed. The full code required can be found in the file **app_os_tutorialThree.c** in the directory **Source/OsTutorialThree** in the ZIP file for this manual.

The code used to implement the behaviour of **APP_taskFlashLed1** looks like the following:

```
OS_TASK(APP_taskLed1)
{
    static bool bLedState = OFF;
    static enum { E_FLASH,
                  E_NO_FLASH
                }eState = E_FLASH;

    uint8 u8Button = 0;

    if (OS_E_OK ==
            OS_eCollectMessage(APP_msgButtons,
                               (void*)&u8Button))
```

```
        {
            if (u8Button == 1)
                eState = E_FLASH;
            else
            if (u8Button == 2)
            {
                eState = E_NO_FLASH;
                SET_LED(LED1, OFF);
            }
        }
        if (E_FLASH == eState)
        {
            /* Toggle the led state... */
            bLedState = ~bLedState;

            /* ...and write it to the led */
            SET_LED(LED1,bLedState);
        }

        /* schedule repeat again in 1 second */
        OS_eContinueSWTimer(APP_tmrFlashLed1,
                            FLASH_TIME_1,
                            NULL);
    }
```

When the task is activated, it attempts to read a message from the queue - in this case, a switch value 1 for 'SW0 pressed' and 2 for 'SW1 pressed'. If SW0 is pressed, the flashing state variable is changed to flash, while if SW1 is pressed, the state changes to no flash, and the LED is switched OFF. If the flash status is FLASH, the current state of LED is toggled and is written out to the LED, causing the LED to change from ON to OFF or vice versa. Finally, the task is rescheduled to activate in the future by setting the software timer to time out.

14.7 Example 4 - Critical Sections

In Examples 2 and 3 (described in Section 14.5 and Section 14.6), we have two tasks for writing out values to the LEDs on the Sensor board. In reality, these LEDs are controlled through the same register in the JN51xx device. There is a small possibility that **taskFlashLed1** could pre-empt **taskFlashLed0** just at the point where it is updating the status of LED0 and overwrite the value (in fact, the routines that are being used protect against this eventuality). This is a common situation in RTOS systems, where one task needs guaranteed uninterrupted access to a resource for a short amount of time to ensure it performs an operation correctly. The mechanism employed to do this within JenOS is the Mutex (for Mutual Exclusion).

We can use a Mutex to safeguard the update of the LED register by both tasks by making them share a Mutex which defines which task owns uninterrupted access to the register, thus protecting the critical code sections or resources.

14.7.1 Adding the Mutex to the Diagram

Step 1 Add a Mutex to the diagram in Example 3 as shown in the figure below (you need to rearrange the position of the elements to make room for it):

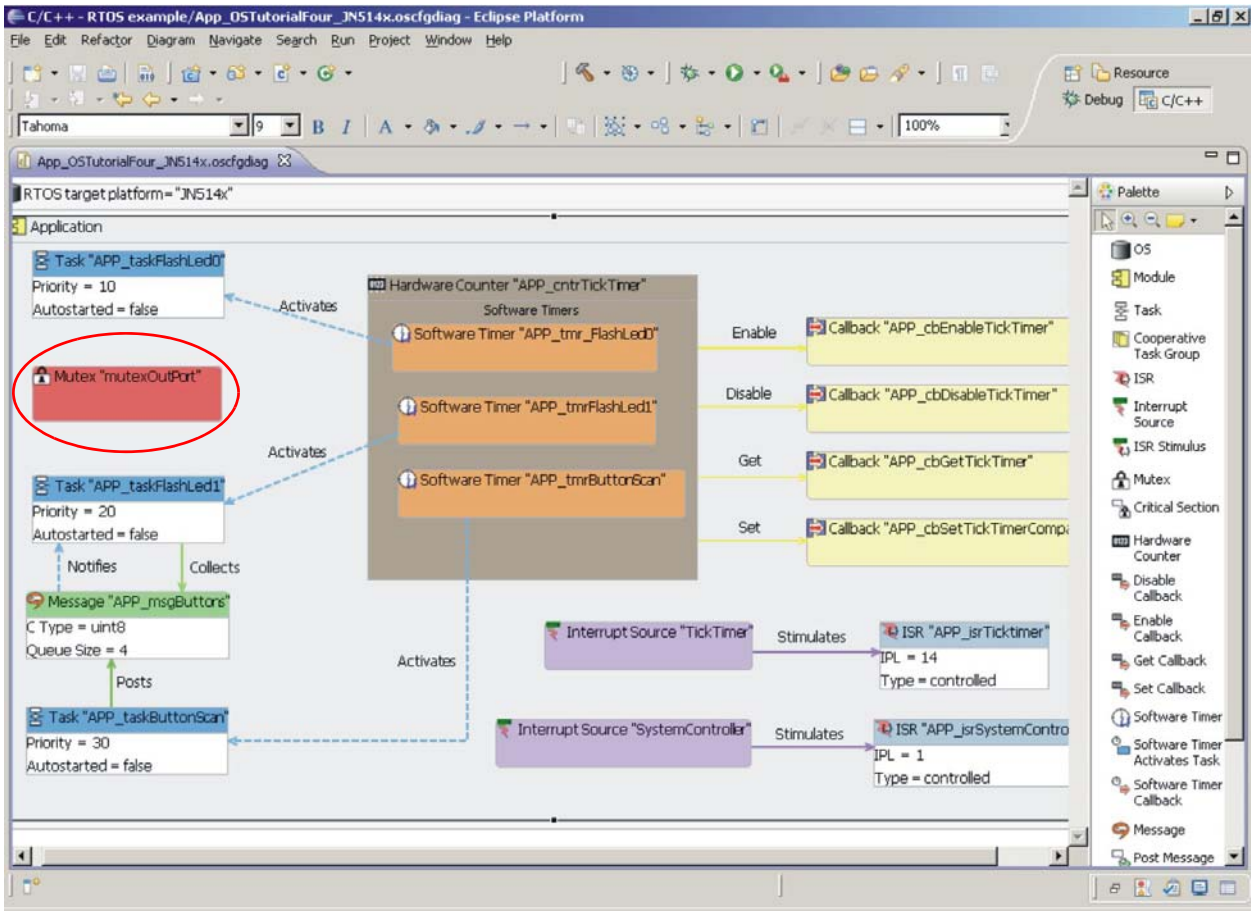


Figure 27: Adding a Mutex

Step 2 Give the Mutex the name **mutexOutPort**.

There are no other properties associated with a Mutex.

Step 3 Use the **Critical Section** connector to connect the Mutex to the tasks that share it, drawn from the task to the Mutex, as shown in the figure below.

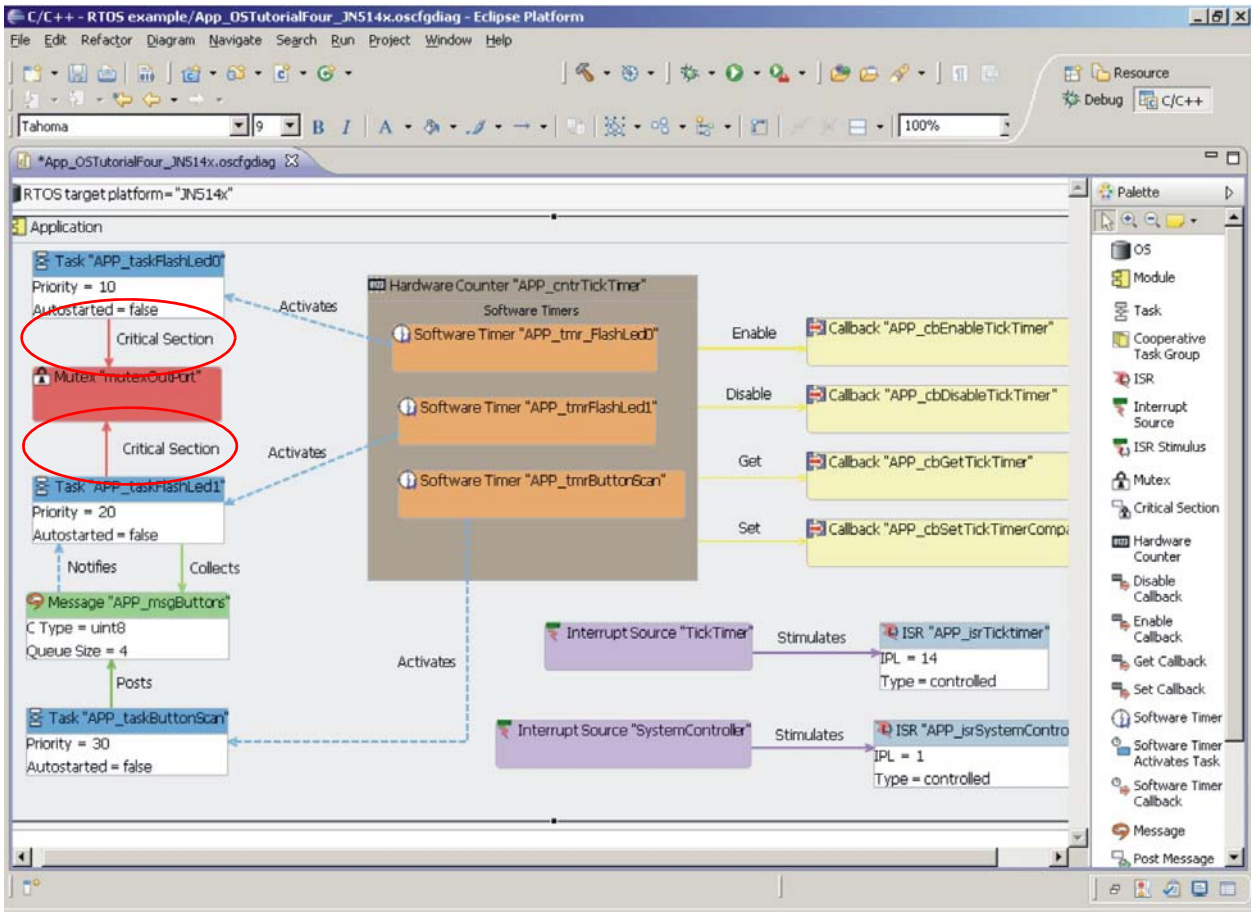


Figure 28: Critical Section Connector

14.7.2 How it Works

We need to change the code in the two FlashLed tasks to use the Mutex. It will be easier to look at the changes using taskFlashLed0, since this routine does not contain all the extra code required to implement the button control for LED flashing found in taskFlashLed1.

The modified version of taskFlashLed0 is shown below:

```
OS_TASK(APP_taskLed0)
{
    static bool bLedState = 0;

    /* Toggle the led state */
    bLedState = ~bLedState;

    /* Write the state to the led */
    OS_eEnterCriticalSection(mutexOutPort);
    vWriteToPortPin(LED_0_PIN, bLedState);
    OS_eExitCriticalSection(mutexOutPort);

    /* Schedule activation again in 1 second */
    OS_eContinueSWTimer(APP_tmrFlashLed0,
                        FLASH_TIME_0,
                        NULL);
}
```

The modification has been made at the point where the new ON/OFF state is written out to the LED. In this instance, we are using the routine **vWriteToPortPin()** to write to the IO pin directly instead of **SET_LED()**. It is this operation of writing to the port that we want to protect, so we surround it with a pair of **OS_eEnterCriticalSection()** and **OS_eExitCriticalSection()** calls. Calling **OS_eEnterCriticalSection()** means that the task will not be pre-empted by a higher priority task which is sharing the same Mutex that is specified in the call, until the corresponding **OS_eExitCriticalSection()** for the same Mutex is called.

The mechanism that the Mutex uses to ensure that other tasks will not interrupt a task in a critical section is explained in Section 2.4.7.

14.8 Example 5 - Using Callbacks

A callback function is a user-supplied routine which is run by JenOS when a specific event occurs - for example, the expiry of a timer. These functions are also used so that abstract devices, such as the hardware counter, can be adapted to a number of different implementations - for example, the actual counter used could be the Tick Timer or one of the user timers that are available on the chip. By providing a standardised set of interfaces to which the callback routines are written, it is easy for the user to provide a new implementation of the device.

We are going to modify the behaviour of Example 4 to show how a callback function can be used. In this case, we are going to break the link on the diagram between the software timer and task controlling the flashing of LED0 (i.e. the activation of the task directly by the expiry of the timer) and replace it by an explicit activation call within a callback routine which is run when the timer expires.

Step 1 Delete the connector between **APP_tmrFlashLed0** and **APP_taskFlashLed0**, and add a callback to the diagram called **APP_cbFlashLed0** connected to **APP_tmrFlashLed0** with the **Software Timer Callback** connector as shown in figure below:

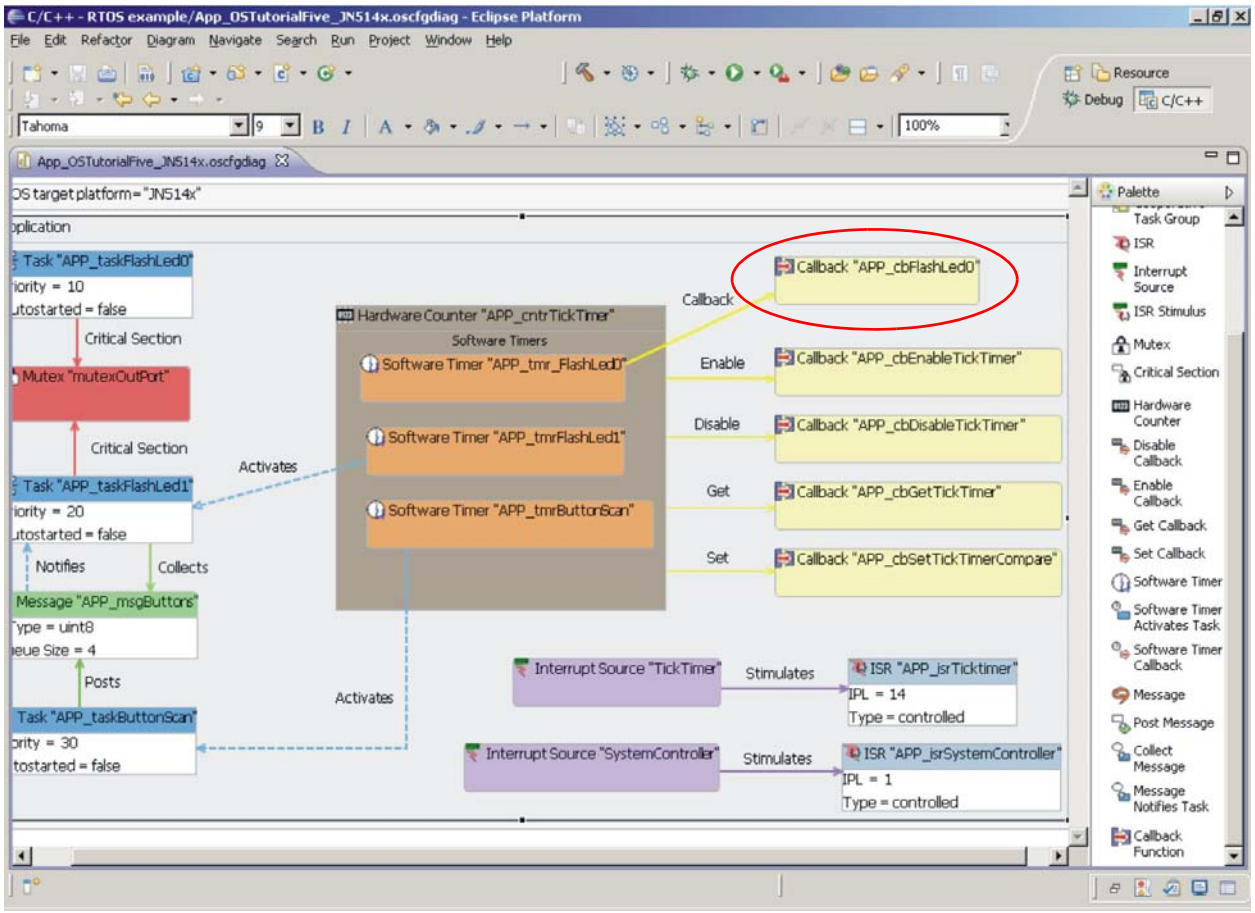


Figure 29: Software Timer Callback

The callback is activated when the software timer expires. When the callback is called, it is supplied with a typeless parameter (specified as **void***) that can be used for carrying information of any type up to 32 bits into the routine. Note that callbacks are executed at OS priority and so should be treated like ISRs (i.e. be as short as possible).

The timer callback for this example is defined as follows:

```
OS_SWTIMER_CALLBACK(APP_cbFlashLed0, pvParam)
{
    /* Explicitly activate the Led 0 task */
    OS_eActivateTask(APP_taskFlashLed0);
}
```

As can be seen, all that happens when the callback is run is that the task **APP_taskFlashLed0** is explicitly activated.

Part IV: Appendices

A. Example Applications for OS Configuration

This appendix summarises the example applications provided for use in the OS configuration tutorials in Chapter 14. The files for these applications are supplied in the ZIP file for this manual.

This ZIP file includes the ZIP file **OS-Tutorial.zip** which contains the software for the tutorials. **OS-Tutorial.zip** must be extracted to the directory

<JN51xx_SDK_ROOT>\Application

where **<JN51xx_SDK_ROOT>** is the path into which the SDK is installed (by default, this is **C:\Jennic**). You should then have a project directory for the tutorials - for example, **C:\Jennic\Application\OS-Tutorial**.

Five mini-projects are provided, which progressively add more features of JenOS and its associated configuration editor. The examples cover the following features:

- Tasks and task priorities
- Hardware and software timers
- Interrupts
- Inter-task communication (messages)
- Timer callbacks
- Mutex (mutual exclusions)

All the code, configuration diagrams, makefiles and project files required to build the examples are provided. Each example is located in its own sub-directory of the **Source** directory in the ZIP file. The makefile for each example is in a sub-directory of the **Build** directory in the ZIP file.

The tutorial is built within Eclipse. The build configuration selects the tutorial to build, numbered from one to five. The tutorials may be run on two platforms: a Sensor board (DK2) from a JN5148-EK010 Evaluation Kit (DK2) or a Generic Expansion Board (DK4) from a JN516x-EK001 Evaluation Kit. To select the chip and platform, right-click on the project, select Properties then C/C++ Build and edit the Build Command:

make JENNIC_CHIP=JN5168 PLATFORM=DK4 TRACE=1

The example project should always be built with debug (TRACE=1). Messages are sent from UART0 (115200, 8, N, 1) and these can be viewed in a terminal program such as HyperTerminal.

The examples are described below.

A.1 OS Tutorial One

This application is used in Example 1 in Section 14.4.

This example consists of a single task, **APP_taskLedFlash**, that flashes LED0 on the sensor board with 50% duty cycle and a 0.5-second on/off time. The file **app_startOne.c** contains the code to initialise the application, OS and hardware. The file **app_os_tutorialOne.c** contains the code to implement the task. The files

os_gen.c, **os_gen.h** and **os_irq.S** are automatically generated at build time from the elements in the OS configuration diagram - these files should not be modified. The OS configuration diagram is in **App_OSTutorialOne_JN514x.oscfgdiag** or **App_OSTutorialOne_JN516x.oscfgdiag**.

A.2 OS Tutorial Two

This application is used in Example 2 in Section 14.5.

This example consists of two tasks, **APP_taskFlashLed0** to flash LED0 and **APP_taskFlashLed1** to flash LED1 on the sensor board. The time that the LED0 task takes to execute has been extended by using an empty 'for loop' - this is solely for the purpose of demonstrating the pre-emption of the low priority task **APP_taskFlashLed0** by the higher priority task **APP_taskFlashLed1**. In order to control the times at which these tasks are activated, two software timers have been added, one for each task. In order to drive these software timers, a hardware timer using the Tick Timer has been added. When a hardware timer is added, the OS requires that four functions are provided, which the OS can call to manage the starting, stopping and expiry of software timers. These functions are **APP_cbEnableTickTimer()**, **APP_cbDisableTickTimer()**, **APP_cbGetTickTimer()** and **APP_cbSetTickTimer()**. It is also necessary to provide an ISR to handle interrupts from the hardware timer. These functions are in the file **app_timer_driver.c**.

A.3 OS Tutorial Three

This application is used in Example 3 in Section 14.6.

In this example, a third task, **APP_taskButtonScan**, has been added to the previous example. This task monitors the state of switches SW0 and SW1, and notifies the LED1 task when a switch is pressed. An interrupt handler for the system controller has been added, **APP_isrSystemController**, as well as a third software timer for the switch scan task. When a switch is pressed, the negative-going edge causes an interrupt. The handler checks that the correct DIO pin was the cause and starts the switch scan timer.

When the scan timer expires, it activates the scan task. The scan task debounces the switches and is activated at regular intervals by the software timer. When the scan task has detected a valid key press, it notifies the LED1 task by 'posting' a message with the value of the pressed switch into the message queue, **APP_msgButtons**, of the LED1 task. The LED1 task is then activated so that it can collect the message from the message queue.

When the switch scan task detects that all switches are in the 'up' state, it does not start the scan timer but re-enables the DIO interrupts. The scan task is then not activated again until the next change in DIO state is detected via an interrupt.

Switch SW1 is used to stop LED1 from flashing and switch SW0 is used to start LED1 flashing.

The addition of the message queue means that the file **os_msg_types.h** must be added. This makes the OS aware of any user-defined types that are posted to

message queues. Add `#include my_types.h` into the file if any non-standard types are passed to message queues.

A.4 OS Tutorial Four

This application is used in Example 4 in Section 14.7.

In this example, a Mutex (mutual exclusion) is added to the previous example in order to protect the writing to the output pins by the LED0 and LED1 tasks. Since the LED1 task has a higher priority than the LED0 task, it is possible for it to pre-empt the LED0 task, possibly at the point where the output port is being written to. If two or more tasks share a resource (hardware or software), a Mutex is used to stop tasks from being pre-empted while using that shared resource. Use of the shared resource is preceded by a call to **OS_eEnterCriticalSection()** and followed by a call to **OS_eExitCriticalSection()** - between these calls, the Mutex is in place.

A.5 OS Tutorial Five

This application is used in Example 5 in Section 14.8.

In this example, the method of activating **APP_taskFlashLed0** has been changed so that it is no longer directly activated by the expiry of the software timer. A callback function, **APP_cbFlashLed0()**, has been implemented which is executed when the software timer expires. When executed, this callback activates the LED0 task. The parameter passing capability of the timer callback is also demonstrated.

B. Hardware Counter Details

A hardware counter is an abstract device used as a source of timing events to drive a number of software timers which are associated with it (see Section 2.4.6). It requires four user-defined callback functions to implement the following set of operations: **Enable** and **Disable** the timer, **Get** the current count value, **Set** the compare value. These callback functions are used by JenOS to manage the software timers and are defined using JenOS macros (detailed in Section 7.1).

B.1 Hardware Counter Operation

JenOS maintains the expiry times of the software timers associated with a hardware counter as a list of delta values - in other words, the number of ticks between now and the point at which the software timer is to expire. When a software timer is started, an entry in this list is created for its expiry time, the delta time being calculated relative to its closest earlier neighbour. The nearest timer later than the one inserted will have its delta value adjusted so that its expiry time is now relative to the newly inserted entry. Timers expiring later than the adjusted timer entry will not need their delta values changing since they are still measured relative to the adjusted timer event.

This can be more easily understood with the following example. Timer A is set to expire 25 ms from now (time 0), Timer B is set to expire 50 ms from now and Timer C at 75 ms from now. The list delta values are thus A=25, B=25, C=25, since B will expire 25 ms after A and C will expire 25 ms after B. 10 ms later, Timer D is set to expire after 20 ms. Timer D entry will expire in-between Timer A and Timer B (i.e. 30 ms from time 0), so it is inserted between A and B in the list. Its delta value from timer A is 5 ms (i.e. Timer D expires 5 ms after Timer A expires). However, since Timer D is now in-between Timers A and B in the list, Timer B's delta value must be adjusted so that it is relative to Timer D's expiry time. Therefore, Timer B's adjusted delta value is now 20. However, since Timer C's delta is calculated relative to Timer B, it does not need to change. So after the insertion, the list appears as A=25, D=5, B=20, C=25.

To run the counter, the first element from the list is removed and JenOS uses the Get and Set functions to calculate and set the compare value for the next expiry point (current count + delta from the list). The counter runs to the compare value and then generates an interrupt. JenOS then expires all software timers that match this point, and any others that may have expired during the time it was processing those which caused the interrupt. After performing all the processing to expire the timers (e.g. activating tasks associated with the timers), the next value from the list is added to the current hardware counter value and becomes the new compare value. This is managed by the function **OS_eExpireSWTimers()**.

Any free running counter with the ability to generate an interrupt when the counter reaches a compare value can be used.

B.2 Use of Tick Timer as Hardware Counter

The Tick Timer of the JN51xx device is normally used as the hardware counter for the JenOS software timers:

- The Tick Timer is a free-running counter running at 16 MHz (62.5 ns). An interrupt occurs when the Tick Timer counter value matches the value in the Tick Timer compare register.
- The macro **APP_TIME_MS(*t*)** in **app_timer_driver.h** (in **Components/ Utilities/Include**) gives the number of tick timer ticks that will occur in *t* ms (1 ms = 62.5 ns x 16000).
- The **OS_eStartSWTimer()** and **OS_eContinueSWTimer()** functions must be called with the number of ticks less than 2147483647. This equates to approximately two minutes with the 16-MHz hardware timer. Application software can time a longer period by maintaining a count of timer expiries.
- Routines which implement the callbacks required by the hardware counter can be found in **components/utilities/source/app_timer_driver.c** and consist of the following functions:
  - **APP_cbEnableTickTimer()**: Clears pending tick interrupts, sets the timer for continuous running and then enables Tick Timer interrupts.
  - **APP_cbDisableTickTimer()**: Disables tick interrupts and stops the Tick Timer.
  - **APP_cbGetTickTimer()**: Returns the current value of the Tick Timer.
  - **APP_cbSetTickTimerCompare()**: Sets the compare value of the Tick Timer with the value calculated - when this matches the Tick Timer counter value, an interrupt is generated.
  - **APP_isrTickTimer**: ISR called when Tick Timer value matches compare value - calls the OS function that manages the OS software timers, **OS_eExpireSWTimers()**.

C. Clearing Interrupts

When using JenOS, it is the programmer's responsibility to clear down an interrupt source within the relevant Interrupt Service Routine (ISR). JenOS only manages the Programmable Interrupt Controller (PIC) to set interrupt priorities and does not clear interrupts.

Unless an interrupt source is cleared down, the corresponding interrupt status bits will remain set, resulting in continuous interrupts. For interrupt sources among the JN51xx peripherals, the JN51xx Integrated Peripherals API contains a number of functions for clearing down some but not all peripheral interrupt sources. Clearing down JN51xx peripheral interrupts is described below for the various peripheral blocks.

Where no API function exists for clearing a peripheral interrupt, a workaround is detailed. Callback functions registered through the Integrated Peripherals API are not permitted when using JenOS. Instead, you should link an interrupt source to an ISR in the JenOS Configuration Editor by adding the interrupt source (purple box) and connecting it to an ISR (using a purple arrow), as illustrated in the diagram below.

Figure 1: Linking an Interrupt Source to an ISR

Note: Where a dedicated function is referenced below for clearing down a peripheral interrupt source, the function is fully detailed in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)*.

System Controller

The following interrupt sources associated with the System Controller can be cleared using the function **vAHI_ClearSystemEventStatus()**:

- System clock
- Comparator
- Pulse counter
- Random number generator
- Brownout detector

For further information, refer to the function description in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x).*

Interrupts from the following System Controller sources must be cleared using the functions indicated:

- DIO interrupts may be cleared using **u32AHI_DioInterruptStatus()** or **vAHI_DioWakeStatus()**
- Wake timer interrupts are cleared using **u8AHI_WakeTimerFiredStatus()**

Analogue Peripherals

There are two analogue peripheral interrupts:

- ADC/DAC conversion complete (CAPT) - this is set when an ADC conversion and a DAC conversion have taken place (ADC and DAC running concurrently)
- ADC conversion complete in accumulation mode (ADCACC) - this is set when a number of digital samples have been accumulated (2, 4, 8 or 16)

There are no API functions to clear down these interrupts and therefore a workaround is needed:

- The relevant interrupt status bits can be read (bit 0 for CAPT, bit 1 for ADCACC) using the following function call:

```
u32Data=u32REG_AnaRead(REG_ANPER_IS);
```

- The bits can be written back using the following function call:

```
vREG_AnaWrite(REG_ANPER_IS, u32Data);
```

These register access functions are not described in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)* but are available in the header file **PeripheralRegs.h**, which is included in the SDK.

You should check both bits in the ISR and process accordingly, as there are situations in which both bits are set.

UARTs

The UART interrupts are set in response to a number of external conditions. The interrupt status of a UART (0 or 1) can be read using the function **u8AHI_UartReadInterruptStatus()**. For details of the returned value, refer to the function description in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)*.

The accessed register is read only. The only way to clear a UART interrupt is to remove the cause-condition.

Timers

A timer can generate interrupts on the rising and/or falling edges of the timer output and the function **u8AHI_TimerFired()** can be used to clear the interrupt source.

Tick Timer

Any pending Tick Timer interrupt can be cleared using the function **vAHI_TickTimerIntPendClr()**.

Serial Interface (2-wire)

A Serial Interface interrupt is asserted to indicate the status of a data transfer, such as a byte transfer having completed or loss of arbitration. The interrupt can be cleared using the function **bAHI_SiMasterSetCmdReg()** in the following call:

```
bAHI_SiMasterSetCmdReg(FALSE,FALSE,FALSE,FALSE,FALSE,TRUE);
```

SPI Master

If enabled, a SPI interrupt is generated when each transfer has completed. There is no API function to clear down this interrupt and therefore a workaround is needed:

- The relevant interrupt status bit can be read using the following function call (bit 0 is set if the interrupt is pending):

```
u32Data=u32REG_SpiRead(REG_SPIM_IS);
```

- The bit can be written back using the following function call:

```
u32REG_SpiWrite(REG_SPIM_IS,u32Data);
```

These register access functions are not described in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)* but are available in the header file **PeripheralRegs.h**, which is included in the SDK.

Intelligent Peripheral

A Serial Interface interrupt is asserted to indicate the status of a data transfer, such as transaction completed. There is no API function to clear down this interrupt and therefore a workaround is needed:

- The relevant interrupt status bit can be read using the following function call (bit 6 is set if the interrupt is pending):

```
u32Data= u32REG_SpiIpRead(REG_INTPER_CTRL);
```

- The bit can be written back using the following function call:

```
u32REG_SpiIpWrite(REG_INTPER_CTRL,u32Data);
```

These register access functions are not described in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)* but are available in the header file **PeripheralRegs.h**, which is included in the SDK.

Digital Audio Interface (DAI)

If enabled, a DAI interrupt is generated to indicate the completion of a data transfer. There is no API function to clear down this interrupt and therefore a workaround is needed:

- The relevant interrupt status bit can be read using the following function call (bit 0 is set if the interrupt is pending):

```
u32Data=u32REG_DaiRead(REG_DAI_INT);
```

- The bit can be written back using the following function call:

```
vREG_DaiWrite(REG_DAI_INT, u32Data);
```

These register access functions are not described in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)* but are available in the header file **PeripheralRegs.h**, which is included in the SDK.

Sample FIFO

The Sample FIFO interrupts indicate the status of the Transmit and Receiver buffers. There is no API function to clear down this interrupt and therefore a workaround is needed:

- The relevant interrupt status bit can be read using the following function call (bit 0 is set if the interrupt is pending):

```
u32Data=u32REG_SampleFifoRead(REG_SFF_INT);
```

- The bit can be written back using the following function call:

```
u32REG_SampleFifoWrite(REG_SFF_INT, u32Data);
```

These register access functions are not described in the *JN51xx Integrated Peripherals API User Guide (JN-UG-3066 for JN514x, JN-UG-3087 for JN516x)* but are available in the header file **PeripheralRegs.h**, which is included in the SDK.

However, the application must also remove the condition(s) that caused the interrupt, otherwise the just-cleared interrupt will be immediately set again.

Revision History

Version	Date	Comments
1.0	25-Nov-2010	First release containing JenOS information taken from *ZigBee PRO Stack User Guide (JN-UG-3048)*, former *ZigBee PRO APIs Reference Manual (JN-RM-2041)* and former *ZigBee PRO Configuration Guide (JN-UG-3065)*
1.1	03-Mar-2011	Overlays feature added, co-operative tasks described and other minor updates made. Accompanying software files also updated
1.2	20-Sept-2011	Appendix on clearing interrupts added. Advice on de-activating software timers before sleeping added. Software timer ‘Start’ and ‘Expire’ functions modified. Other minor updates/corrections also made
1.3	24-Aug-2012	Minor updates/corrections made
1.4	19-Dec-2012	Updated for JN516x

Important Notice

Limited warranty and liability - Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use - NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications - Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control - This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com/jennic

