OpenComRTOS
Layer L0

A next generation distributed and scalable RTOS supporting a coherent and unified system development methodology, based on Interacting Entities

WP1 Architecture and design

Version: 1.0.13

Review list

Name	Organization
Eric Verhulst	Open License Society
Gjalt de Jong	Open License Society

Signature list

Name	Signature	Date

Document History

Date	Version	Author	Change
30.09.2005	1.0.0	Valeriy Kamyshniy	
06.10.2005	1.0.1	Valeriy Kamyshniy	REVIEW iteration 1
11.10.2005	1.0.2	Valeriy Kamyshniy	REVIEW iteration 2
12.10.2005	1.0.3	Valeriy Kamyshniy	REVIEW iteration 2
01.11.2005	1.0.4	Valeriy Kamsyhniy	Functional Design additions
11.11.2005	1.0.5	Eric Verhulst Gjalt de Jong	Rewrite
15.11.2005	1.0.6	Gjalt de Jong	Updated diagrams
16.11.2005	1.0.7	Eric Verhulst	Clean up
23.11.2005	1.0.10	Gjalt de Jong	Added Task management services
11.01.2006	1.0.11	Eric Verhulst	Update
23.01.2006	1.0.12	Eric Verhulst	Update from working meetings
21.02.2006	1.0.13	Gjalt de Jong	Updated diagrams (synchronous services only)

Release information

Version 1.0.13 is the first stable release in line with the actual software. It covers only the single phase services (L0_XXX_W) although some elements of the two-phase services are mentioned in view of future kernel releases.

Table of Contents

1. Introduction

1.1. Purpose

This document describes the architecture and design of Layer L0 (further called L0) of **OpenComRTOS**. It considers possible alternative solutions, argues the chosen approach and explains the involved trade-offs. The ultimate goal of this document is to provide information that is sufficient for modeling and implementing L0.

1.2. Scope

This document is being developed in a number of iterations, gradually increasing the level of details in the OpenComRTOS L0 architectural description. The current iteration of the design is indicated by the last position (xx) in the document version, e.g. 1.0.xx.

The design decisions in this document are based on the requirements for the **OpenComRTOS** specified in [1].

1.3. Definitions, Acronyms and Abbreviations

Cluster	An ensemble of **Nodes**
Context switch	The process of swapping **Task**-specific information usually associated with CPU registers during **Task** scheduling
Inter-node link	Point to point communication system between two nodes. It can be virtualised when the communication medium is shared.
Flood-fill booting	Booting strategy whereby every booted **Node** activates the booting of **Node**s immediately connected with it with the same boot code
Group booting	Booting strategy whereby a selected group of **Node**s is booted
HAL	Hardware Abstraction Layer
PAL	Platform Abstraction Layer
Hub	Access point in a network to a local cluster of **Node**s
Interrupt latency	The time interval between the hardware interrupt signal and the first application level instruction of the interrupt service routine entry
Interrupt-to-task latency	The time interval between an interrupt service routine entry and the first application level instruction of an associated **Task**
ISR	Interrupt Service Routine
Network booting	Process whereby a network of processors is booted through the network connections
Node	A processing device in a network containing at least a CPU and its local memory
Orthogonal	Independent, non-overlapping
Packet	TBD
TCB	Task Control Block – data structure which is used by operating system to manage individual **Task**
Platform	Hardware system with CPU, specific peripherals and development support
Port	A L0 level kernel object used to synchronise and communicate between Tasks using Packets
Round Robin scheduling	Non-pre-emptive scheduling following a policy of “first come – first served”. **Attention:** often Round Robin means pre-emptive time slicing scheduling – this notion is not used in this document
RTOS	Real-time Operating System
Site	An ensemble of **Clusters**
System object	Operating system specific object required for **Task** management and communication (e.g. mutexes, queues)
Task	Active RTOS object: a function with its private workspace
(binary) image	The binary code that is booted
De-scheduling	Releasing the CPU to another **Task**
Re-scheduling	Process of scheduling a **Task** that was active but not running

1.4. References

1. OpenComRTOS: Requirements specifications, Open License Society vzw, 2005.

2. General concepts

2.1. Background of OpenComRTOS

The main purpose of **OpenComRTOS** is to provide a software runtime environment supporting a coherent and unified systems development methodology, based on **Communicating Objects**, more generally called "**Interacting Entities**".

In **OpenComRTOS** a communicating object is represented by a running SW entity, called a "**Task**".

A **Task** is running on a computing device (CPU + RAM + Peripherals + etc.), called a "**Node**".

There may be many **Tasks** that run on a single **Node**. These **Tasks** may be independent or synchronising and communicating with each other. In other words, it is possible to build a network of communicating Entities using only one **Node**, every Task virtualising a complete CPU instance.

OpenComRTOS however is a distributed RTOS and contains a build-in router and communication layer. While hidden from the application programmer, this allows **Tasks** to synchronise and communicate transparently across a network of processing nodes. This support for a transparent distributed operation however is an option that does not prevent using OpenComRTOS on a single CPU.

For the application programmer, there is no logical difference between **Tasks** running on the same **Node** or on multiple **Node**s. He programs in a network topology independent and transparent way, except when physical differences dictate otherwise.

2.2. Physical structure of target computation system

The following figure represents the physical structure of a generic and distributed computing system from the point of view of **OpenComRTOS**:

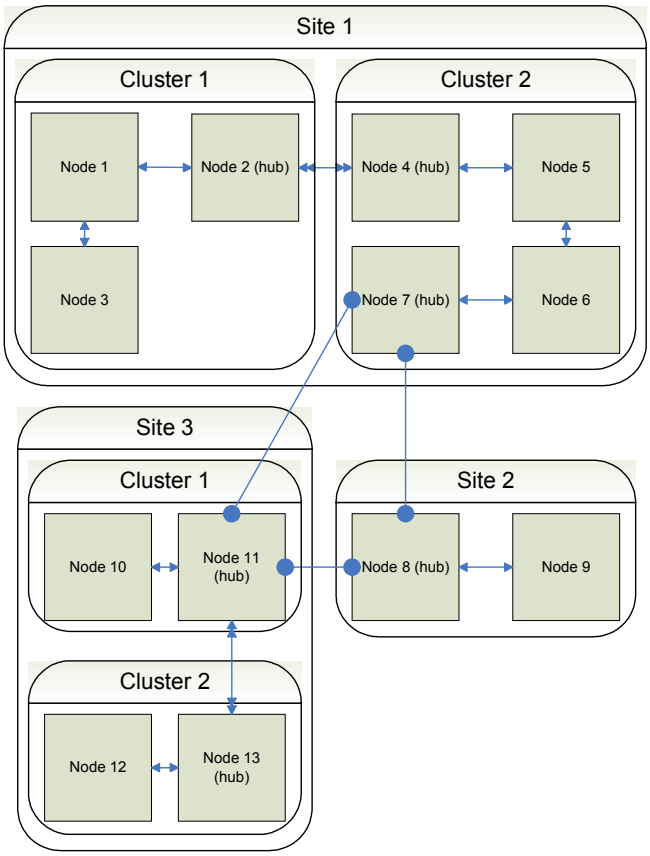


Figure 1 Generic structure of a distributed computing system

A target system is hierarchically composed of the following three layers:

- **Sites,** consisting of
- **Clusters**, consisting of
- **Nodes**, hosting
- **Entities (e.g. Tasks, Ports, …)**

The **Nodes** communicate with each other via various physical channels (IO buses, networks, IO channels, etc). There are special **Nodes** that fulfil the role of communication hubs providing communication between different clusters in the network. Note that these three layers will often correspond with three domains where the physical parameters of the communication layer will differ in performance, bandwidth and communication latency. Form a logical point of view however there is no difference at the application level. Only timing will differ.

2.3. Layered architecture of OpenComRTOS

OpenComRTOS is being developed using a scalable architecture. Each higher level layer builds on the lower layers and provides a specific support:

- L0. The lowest layer. Provides the basic primitives services, such as task scheduling, task synchronisation and communication, routing and inter-Node communication.
- L1. The next layer. Will most often provide more complex and flexible services and can be used to emulate existing third party RTOS
- L2. The highest layer. This layer can support user-defined services, often supporting dynamic behaviour.

OpenComRTOS operates at the L0 layer by using Ports and Packets. The Ports are used to exchange Packets between Tasks and synchronise a send-receive pair of services. The L0-Packets are atomic units

containing a header and a zone for payload data and the kernel mostly operates by shuffling the packets around will updating or using the header field information. To implement L1 and L2 layer services additional Packets of fixed size are used, most often just containing data payloads. In this document we focus on the L0 layer only.

2.4. Logical view of Layer 0

The distributed environment, described in the Sections above is based on the existence of a fast and unified communication layer. The **OpenComRTOS *Layer 0*** therefore is defined as providing the following functionalities:

1. a Packet-switched communication layer using **InterNode Links** and inter-node communication **Routers**
2. a **Kernel** to provide functional services and operating resources to **Tasks**
3. a **Task Scheduler** to schedule the **Tasks** according to a real-time scheduling policy

The logical structure of **OpenComRTOS** on a single node is shown in *Figure 2.*

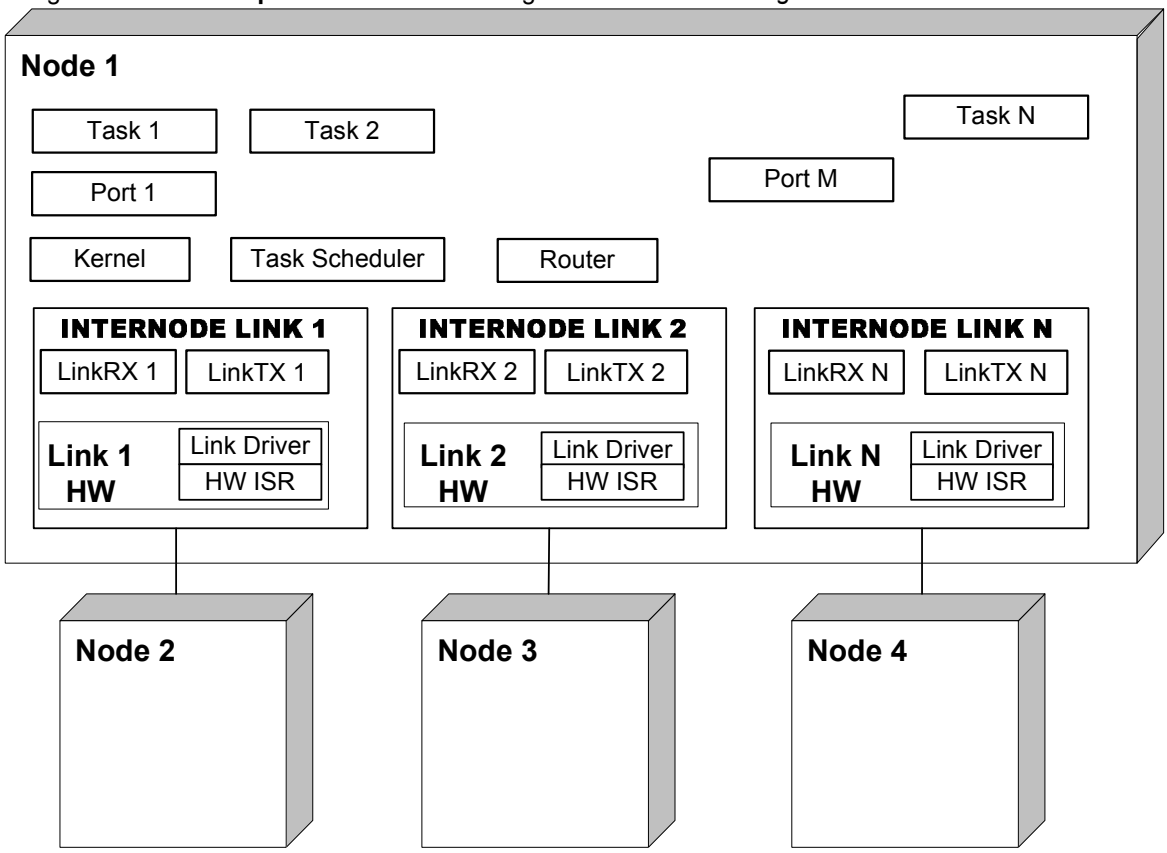


Figure 2. Logical structure of the distributed system

2.4.1. Principle of synchronization and communication

The distributed environment, described in Sections 2.1 and 2.2, is based on the existence of a fast and unified communication layer, independent of the underlying communication protocol or hardware. In terms of this communication layer, an abstraction of the physical inter-node communication channel is called by L0 a **Internode Link**.

Each **Node** can have a number of **Internode Links** to other **Nodes.**
Logically, every **Internode Link** is a point-to-point connection to another **Node**. It consists of a transmitting and receiving channel, called **LinkTX** and **LinkRX** respectively. Self-loops are allowed as well as multiple

Links between the Nodes. If there are no links, e.g. when there is only one Node in the system, the routing function is void and the system works in an identical way.

Tasks interact with the **Internode Links** via a standardized interface. The interaction to the related hardware is hardware specific and should not influence the interface.

OpenComRTOS is based on **Packet**-switched communication. This means that **Packets** of a fixed size are passed from one **Task** to another **Task**. As the **Communicating Tasks** may be located on different **Nodes**, the **Packet** may be passed from one **Node** to another. Coming from a source **Node** to a destination **Node**, the **Packets** may pass through a number of intermediate **Nodes**. For the application programmer however, **Packets** are sent to a "**Port**" object and received from a **Port** object**.** This effectively isolates **Tasks** from each other and increases the scalability of the system. At the higher levels of **OpenComRTOS** (L1 and L2), **Ports** and **Packets** might be invisible to the developer and encapsulated in the services provided.

To provide the routing of **Packets** from **Node** to **Node**, there are inter-node communication **Routers** in the distributed network.
The **Router** is a *function* present on every **Node**. This function provides a mapping between destination **Nodes** and **Internode Links** to be used by **OpenComRTOS** to reach the destination **Node**. The router itself is invisible to the application programmer. As all **OpenComRTOS** services are by default "distributed", the routing is void when routing between local **Tasks**.

2.4.2. Scheduling Tasks and Task synchronisation/communication through the RTOS kernel

To timely provide the **Tasks** with the required operating resources (RAM, CPU time, functional services, etc.), **OpenComRTOS** has a **Kernel** and a **Task scheduler**.

The **Kernel** is the logical entity that i) provides services to the **Tasks** and ii) also schedules the **Tasks** according to a real-time scheduling policy. Although the functions are logically separate, in practice they can be intertwined in **OpenComRTOS**.

From the point of view of the functional relationships between the above mentioned entities, the software runtime environment on a **Node** consists of:

- **A Task scheduler that switches the CPU context between Tasks**
- **(One or more) Tasks that request services from the Kernel** (using a Packet, but that may be hidden)
- **The Kernel that provides these services. When one of the Tasks is remote, it passes on the service request to remote Nodes**
- **When remote services and Entities are involved, Routers are used for passing on the Packets to Internode Links, respectively to receiving them from Internode Links**
- **Internode Links have Transmitting (LinkTX) and Receiving (LinkRX) logical channels**
- **LinkTX and LinkRX are provided by the Link hardware, managed by (hardware) specific link drivers and interrupt service routines (Link driver, HW ISR)**

The relations are represented in *Figure 3*.

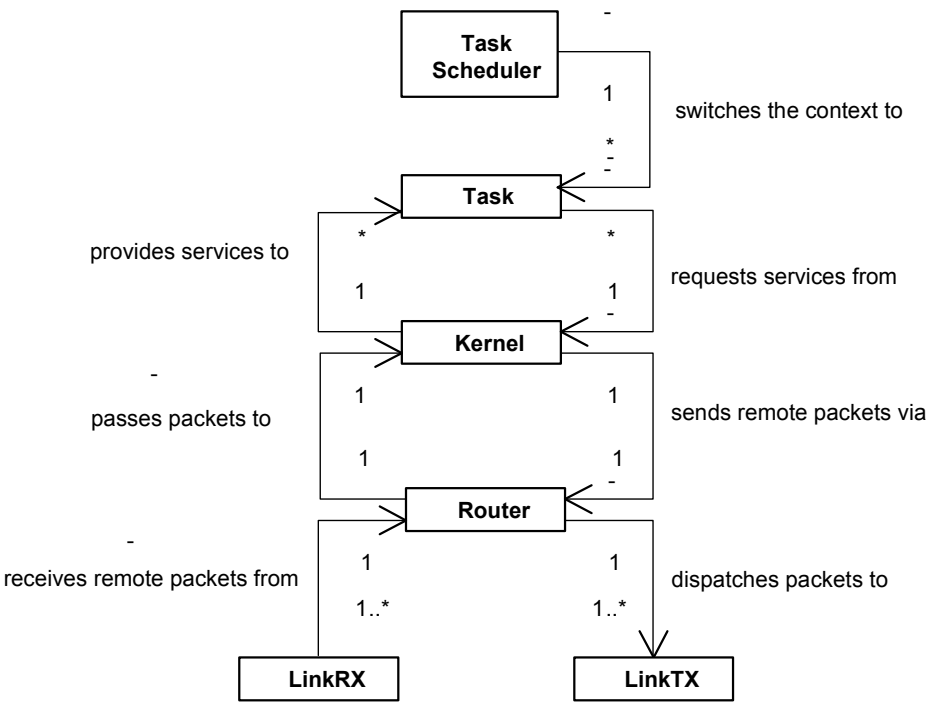


Figure 3. Functional relationship between entities of the distributed system

2.5. Inter-Task interaction

An inter-task interaction consists of two parts: sending and receiving a Packet via a Port. A typical scenario consists of logical actions, performed in one cycle of Packet interchanges between two communicating **Tasks**. When no data is interchanged (data size = zero), we call such an interchange of **Packets** "synchronisation". When data is exchanged as well, we call it communication. Note however, that at the level of **L0**, this is an issue for the application code running in the **Tasks**. From a point of view of the **Kernel** and the **Port**, just a **Packet** has been interchanged.

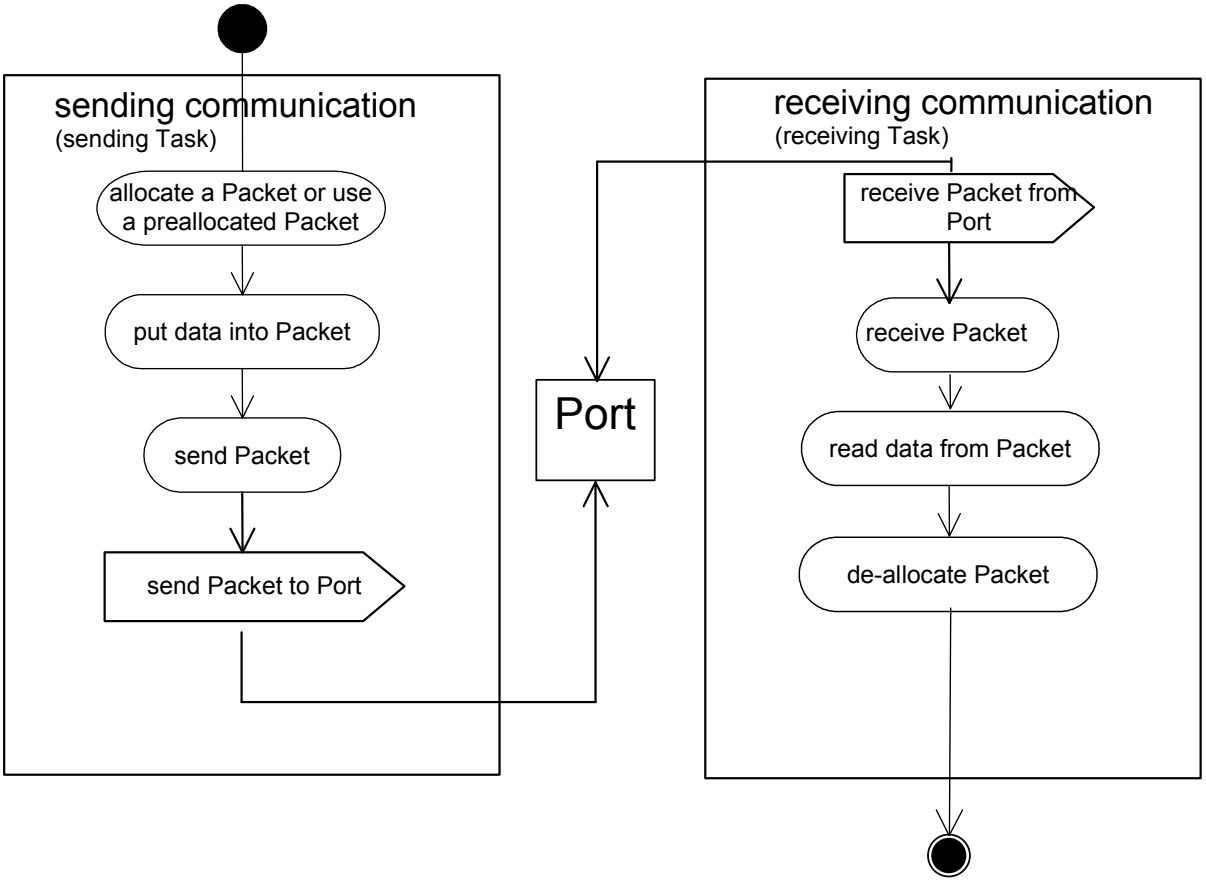


Figure 4. Scenario of a Packet interchange cycle (with data)

In most cases the send request is performed by one **Task** while the receive request is performed by another **Task**. However, as the interaction is through **Ports**, it can as well be that e.g. driver **Tasks** or hardware specific ISRs put a **Packet** in a **Port**. However, while an ISR can insert a **Packet** into a **Port** on which a driver task could wait to receive from, no ISR should attempt to receive a **Packet** from a **Port**. The reason is that ISRs are not allowed to wait and polling is just burning cycles. If an ISR needs to receive data it should get this data from an associated Driver Task that itself can wait to (asynchronously) receive from a Port.

As **OpenComRTOS** supports distributed systems, by default, the interacting **Tasks** and **Ports** can be located on different **Nodes**. For example, the sending **Task** can be located on **Node A**, the receiving **Task** can be located on **Node B** and the **Port** can be located on **Node C** (see the figure below for an example).

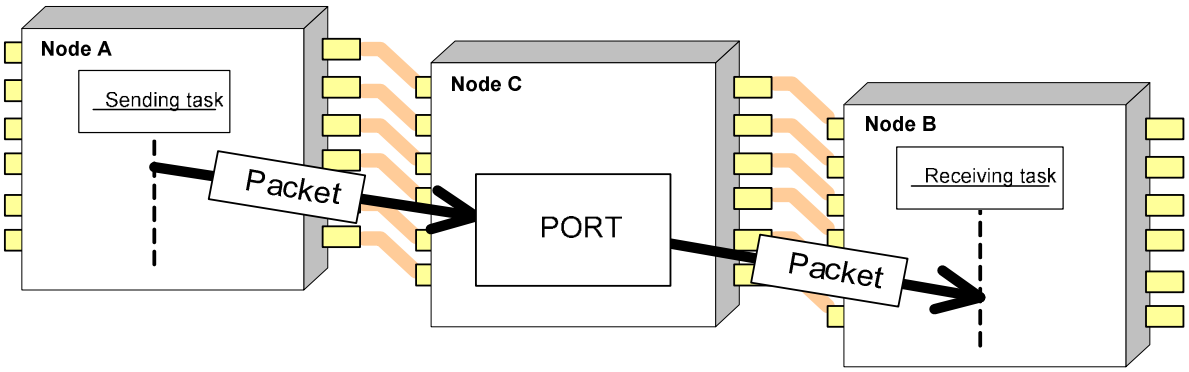


Figure 5. Possible distribution of Entities, involved into inter-task interaction.

2.6. Inter-Node interaction

OpenComRTOS provides topology independent interaction between tasks. All services, except when dictated otherwise by hardware dependencies, are from the application's task point of view independent of the location in the network of nodes. This applies e.g. to Task management services as well as to Port services. However, the layer of direct control of the **Link HW** may implement the communication very differently from one platform to another.

While in OpenComRTOS, tasks can interface directly with the hardware via Interrupt Service Routines, most often a (driver) Task will implement the higher level functionality of interfacing with the hardware. In particular, when multiple nodes are present in the system, these nodes will be able to exchange data through a dedicated software supported hardware mechanism. Independently of the hardware implementation, we call these dedicated communication mechanisms **LINKS**. **OpenComRTOS** defines dedicated **Tasks**, called **Link Driver Tasks**, that implements the OpenComRTOS system level communication protocol. Of course, in general, hardware will be accessed through a combination of an **ISR** and a **Driver Task**, but then a hardware and application specific protocol will be used.

> **A Link Driver Task is the only way to initiate transparent inter-node link communication**
>
> **Any Link Driver Task communicates only to other Tasks via a dedicated Port associated with it. This Port is called as a Task Input Port.**
>
> **Any Task communicates with a Link Driver Task only via a dedicated Port associated with it. This Port is called the Driver Input Port.**
>
> **The HW itself is controlled and accessed by the ISR layer. This layer may communicate with the Driver Tasks through shared memory and dedicated event signalling services.**

The interaction scheme is illustrated in the following figure.

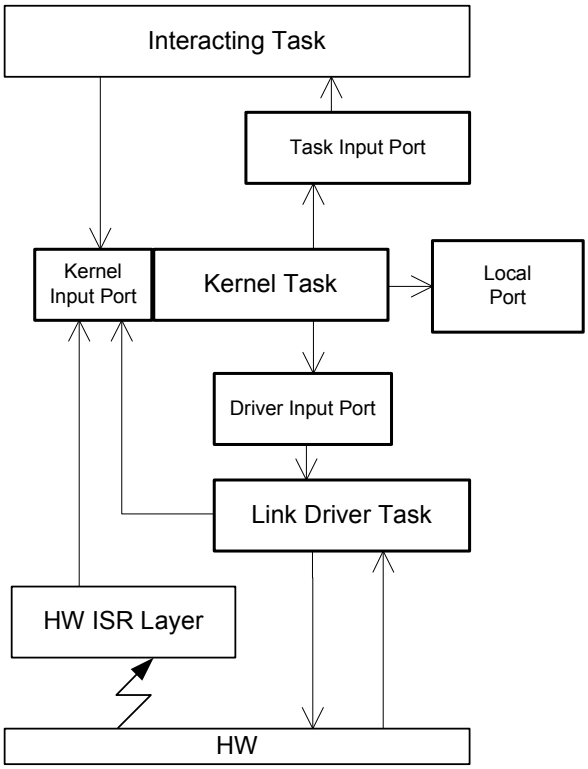


Figure 6. Interactions between HW, ISR Layer, Driver Task and application Tasks.

The Tasks, Link Driver Tasks and ISR layer interact with each other ONLY via the Kernel. See Section 2.4 / Figure 3 and Chapter 3.

3. Functional Design of Layer 0

The following figure represents the Functional model of **OpenComRTOS Layer L0**.

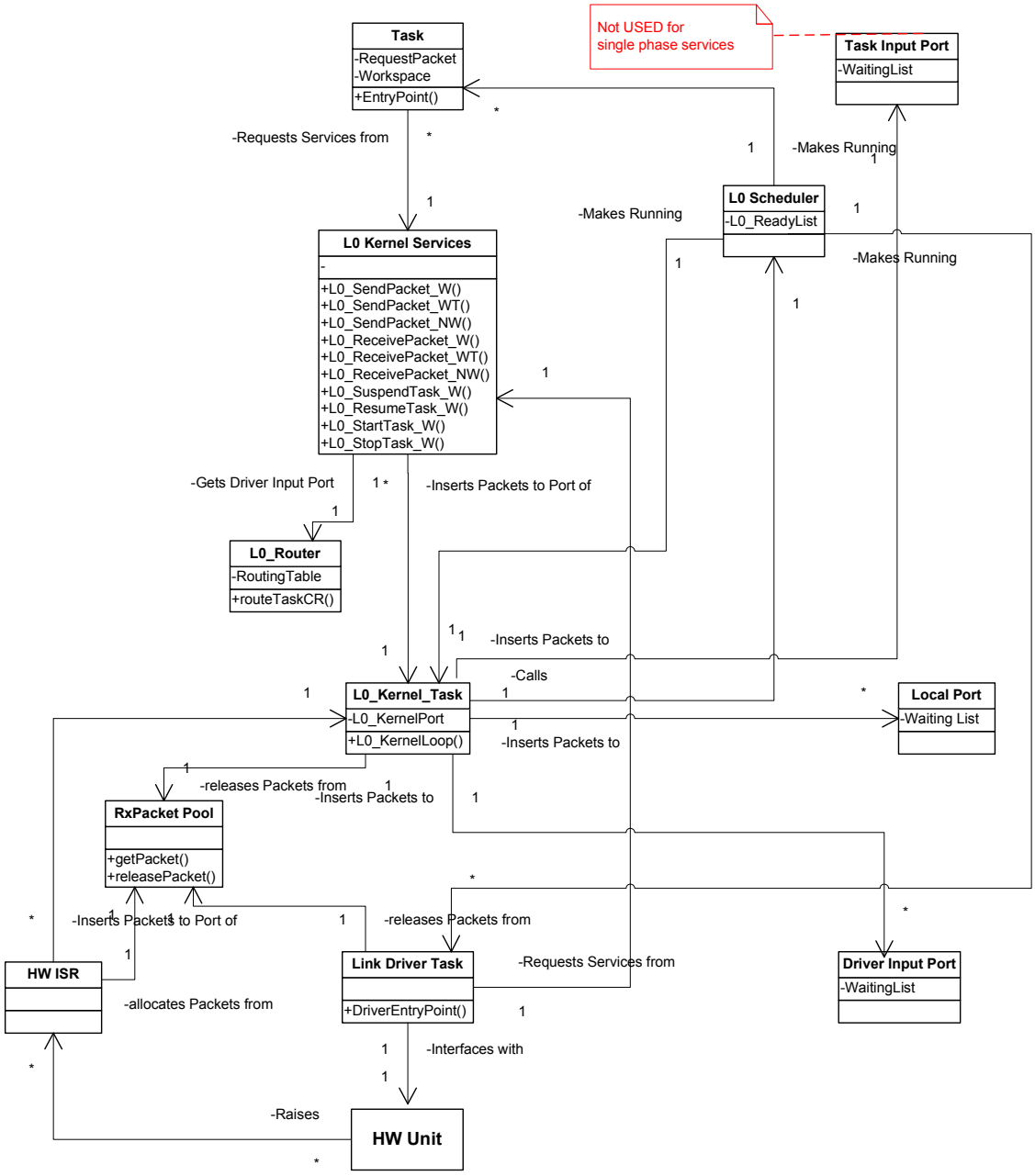


Figure 7. Functional model of OpenComRTOS

3.1. Task interactions

As defined in Section *2.4*, a synchronisation/communication between the Entities is a synchronisation/communication between the **Tasks** via intermediate **Ports**. To simplify the terminology, we call these **Task Interactions**.

Ports are used as synchronisation Entities for the **Packets** sent and received by **Tasks**. Hence, **Ports** also decouple **Tasks** when interacting and they can be located physically on different **Nodes** than the sending and receiving **Tasks**. As a result, **Tasks** are isolated from each other while this mechanism is inherently scalable and topology independent.

3.1.1. Logical view of a Task

In **OpenComRTOS**, the software runtime environment can run many **Tasks** on a single **Node**. Each **Task** is a separate entity identified by its **TaskID**. The Task ID is a globally unique identifier in the distributed system. A **Task** is therefore defined as:

- **A Task is a uniquely identified operating resource**
- **A Task can issue service requests from the Kernel. These are implemented as a local function within a Task's workspace.**
- **The first instruction of a Task's function is called the entry point of the Task.**

The **Task Context** is defined by the following two parts:

- Its **Workspace** (often called **Stack Space**). This is an area of data memory that is involved in the logic **operation** of the **Task**. Normally, the logical data of a **Task** context is hardware independent. The logical data is an explicit part of the context that the **Task** manages itself and hence contains only data and variables that are only visible to the task itself.
- Its **CPU Context** that is the physical context of the Node**.** This is a set of data units that precisely defines the current state of the CPU. The **CPU Context** is an **implicit** part of the **Task Context**, not directly manipulated by the **Task**, but by the compiler, CPU and peripherals. Usually the **CPU Context** consists of the state of the essential CPU and other HW registers, like the Instruction Pointer (IP), Stack Pointer (SP), the Accumulating Registers, and I/O registers. The **CPU Context** is specific to the hardware (CPU + peripheral units, e.g. state information).

In **OpenComRTOS**, only one **Task** can execute at a given time on a given **Node**. This means that if there are many **Tasks** running on the same **Node**, the scheduler will divide the available processing time over the **Tasks** according to a **Task scheduling policy**. When using a priority based scheduler the priorities are to be assigned by the application developer who has to assure that all **Tasks** can meet all deadlines.

During their operation the **Tasks** may request the **Kernel** for services such as sending or receiving **Packets** via **Ports**. Typically, the **Tasks** will wait for events like the completion of such requests. Note that **Tasks** can run independently without issuing any service request, although this can lead to starvation for other Tasks. The "data" fields of a sent or received **Packet** may be "empty" (i.e. pure synchronization without data communication exchange).

A **Task** starts by being started from another Task or during kernel initialisation. It may have finished, which is called **STOPPED (sometimes called Terminated).**

Hence, a **Task** is further defined by:

A Task is an operating resource that is always in only one of the following states, managed by OpenComRTOS:

- **INACTIVE (**the initial state, similar to STOPPED)
- **RUNNING** (sometimes called "ACTIVE")
- **WAITING** (for a service request to complete, sometimes called "INACTIVE".)
- **READY** (to run)
- **SUSPENDED**

- **STOPPED**

Note that the normal states in operation are RUNNING, WAITING, READY and STOPPED. The SUSPENDED state is the result of an explicit suspend request and is orthogonal to the normal states. This means that a waiting status remains possible when the task is being suspended. It can only be changed by a resume request by e.g. a monitoring Task. Hence, a task should not suspend itself as the suspend state is introduced mainly to be able to handle exceptional application level conditions that require e.g. to prevent a Task from doing any potential harm. For example, it can be used to stop a robot arm from moving when an obstacle has been detected that should not be in its path.

When many **Tasks** run on the same **Node**, they compete for the CPU time in order of their priority. A higher priority means that when several **Tasks** are ready to run, the one with the highest priority will run first. Hence, a **Task** is further defined by:

> **A Task is an operating resource that has a PRIORITY. A priority has a value in the integer range from 0 to 255, with 0 being the highest priority.**

To provide many **Task** instances with the same (local) function, **OpenComRTOS** allows **Tasks** to start with a list of **Task** specific arguments.

Finally, at the system level but hidden from the application programmer, each Task including the Kernel Tasks and Driver Tasks, have a dedicated Input Port.

The logical model of a **Task** is represented by *Figure 8*:

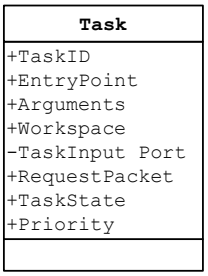


Figure 8. Logical model of a Task

3.1.2. Logical view of Packets

In **OpenComRTOS**, the interacting **Tasks** interchange **Packets** of a fixed size. The “fixed size” of a **Packet** means that the physical size of **Packet** is always the same and that the real size of the interchanged data in the **Packet** can not be greater than this size. A **Packet** contains so called header information that includes a number of the header specific fields, including the **size of the user data** (sometimes called payload). The Packet size is defined at compile time and can be application specific but it can never be smaller than the space needed for the header fields.

In each concrete case, the interchanged **Packet** is also supplied with the exact length of the attached interchanged data.

Hence, a **Packet** is defined as follows:

> **A Packet is an entity that consists of**
> - **A fixed size header including:**
> - **Service specific fields**
> - **the (user) Data Size field**
> - **The data limited in length to the Data Size field**
> - **Remaining unused space of the data portion of the packet (in any).**

The Data Size of a Packet can be zero or maximum equal to the Packet Size minus the size of Header.

The logical model of a data **Packet** is represented by *Figure 9*.

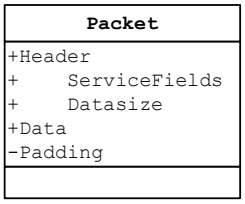


Figure 9. Logical model of a Packet

Note:
OpenComRTOS also has internal Packets that are not allocated dynamically and are not part the Packet Pool. These Packets (just like the Task Input Ports) are statically allocated at system configuration time.

NOTE:
In the text often the terms Send_ or Receive_Request_Packet will be used. Often, this is still same physical Packet but who's function has changed by an update of its header fields depending on the status of its processing.

3.1.3. Logical view of Ports

When sending a **Packet**, a **Task** sends it to the specified **Port**, where the **Packet** has to be delivered. If the service requires synchronization, a reference to the packet will be stored. In the implementation, copying of Packets is avoided and a pointer to the Packet will be passed. This implies that a Packet is owned by the Task that uses it to avoid that multiple tasks can modify a Packet's content or that the kernel tasks assures that only one task can write to the Packet at a given time. Similarly, when receiving a **Packet**, a **Task** receives it from the specified **Port**. The **Packet** is to be delivered to the Port by a sending Task. In both cases, a **Port** has to be specified, so it has to be identifiable. Hence, a **Port** is defined as:

> **A Port is an identifiable entity with a globally unique identifier in the distributed system.**

The purpose of a **Port** is defined as:

> **A Port is an entity used to synchronise the interchange of Packets between interacting Tasks. The synchronisation is handled by the Kernel Task.**

If a **Task** sends a **Packet** to a **Port**, and no other **Tasks** has yet requested to receive a **Packet** from that **Port**, then the sending **Task** will wait until such request arrives at the Port. Note that any number of **Tasks** (more than one) may send **Packets** to the same **Port** at any time.

Vice versa, if a **Task** requests to receive a **Packet** from a **Port**, and no sent **Packet**s are available, the **Task** becomes waiting until a sent **Packet** arrives at the **Port**. A number of **Tasks** (more than one) may request **Packets** from the same **Port** simultaneously. As such, a **Port** is further defined as:

> **A Port is an entity that buffers the requests to send or to receive Packets until synchronisation occurs.**

The sent and received **Packets** are "buffered" in a **Port** by means of a priority-ordered list of **Packets**. The priority of an element in the list is inherited from the requesting **Task**.

According to the above mentioned relationships between **Tasks**, **Ports** and data **Packets**, the logical model of a **Port** is represented by *Figure 10*. Note that while two waiting lists are indicated, at L0 level only one of them will be used. Both waiting lists are needed for L1 and L2 level services.

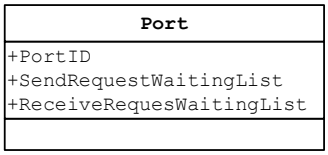


Figure 10. Logical model of a Port

3.1.4. Logical view of the Packet Pool

Every Task has a pre-allocated Packet that can be used for a single phase interaction between tasks. In order to allow two-phase interactions the **Task** has to allocate extra **Packets** from the **Packet Pool** that is located on its local **Node**. (see. 3.3.3).

After a **Task** has received and processed a **Packet**, the **Task** has to deallocate this **Packet** to return it to the **Packet Pool** that is located on its local **Node**.

> **The Packet pool of a Node is an operating resource that maintains a list of free Packets.**
>
> **If a Task requests a Packet from the Packet Pool, and the Packet Pool has no free Packets then the requesting task becomes waiting until another task has de-allocated a Packet so that this Packet can be allocated to satisfy the request.**

The requests to allocate **Packets** are "buffered" by means of a priority-ordered list. These are actually a list of pre-allocated packets used by OpenComRTOS to implement the service requests. The priority of an element in the list is inherited from the requesting **Task**.

The logical model of the **Packet Pool** is represented by *Figure 11*.

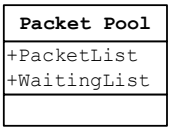


Figure 11. Logical model of the Packet Pool

3.2. Inter-node interactions

3.2.1. Logical view of Link Drivers and inter-node interactions

OpenComRTOS implements **INTER-NODE LINKs** (see Section *2.4*) using the relationship between a communicating **Task** and a **Link Driver Task**, explained in Section *3.2*.

- **The LinkTX of an INTER-NODE LINK is implemented through a dedicated Link Driver Task that transmits Packets to the directly connected remote Node via the appropriate hardware.**
- **The LinkRX of an INTER-NODE LINK is implemented through a dedicated SW entity in ISR LAYER that injects the received Packets in the Kernel Port. The Kernel will deliver the Packets to the appropriate local Ports and Task Input Ports, or route the Packets to the LinkTX of the appropriate INTER-NODE LINKs (i.e. to a Driver Input Port) as applicable.**

A Link Driver Task will implement the following behaviour:

- **The Link Driver Task is waiting for a Packet on the Driver Input Port.**

- **The Link Driver Task will process the Packet on the Driver Input Port.** (e.g. transmitting the packet over a LinkTX)

The interaction scheme of the involved entities is shown in the following figure.

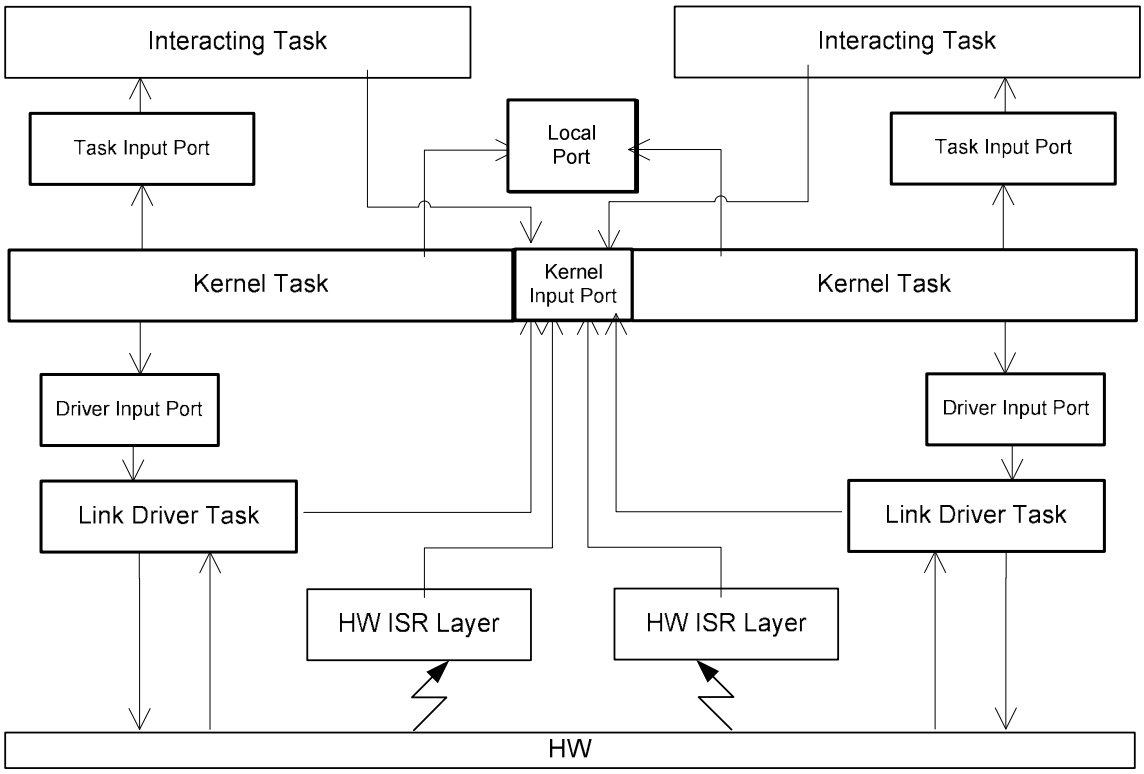


Figure 12. Communication between INTER-NODE LINKs and Tasks

Note: The Tasks, Link Driver Task and ISR layer interact with each other ONLY via the Kernel, as described below.

To provide the interacting **Tasks** with a simple and sufficient way for addressing the **INTER-NODE LINKs, OpenComRTOS** has adopted the following mechanism:

> **An INTER-NODE LINK is addressed by the Input Port of the Driver Task that is driving the link.**

When a **Task** calls a L0 service to send a **Packet** to a remote **Port**, the following sequence of actions is performed:

> **L0_SendPacket_W** (**Send_Request_Packet, Remote Port**)
>
> This function will in the context of the Task update the Header of the Packet to be sent to a Port and inserts it in the Kernel Input Port. The Kernel will call the Router function to connect the Remote Port with a local TX Driver Input Port. The Driver Task then forwards the Packet to the destination Node by the lower level protocols.
>
> When the Return Packet arrives, the Kernel will make the Task ready again and the task can retrieve the return value from its preallocated Packet.

When an interacting **Task** calls a **L0 service** to receive a **Packet** from a remote **Port**, the following sequence of actions is performed:

```
L0_ReceivePacket_W (Receive_Request_Packet, Remote_Port )

This function will in the context of the Task update the Header of the
Packet to be retrieved from a Port and inserts it in the Kernel Input
Port. The Kernel will call the Router function to connect the Remote Port
with a local TX Driver Input Port. The Driver Task then forwards the
Packet to the destination Node by the lower level protocols.

When the Return Packet arrives, the Kernel will make the Task ready again
and the task can retrieve the data and the return value from its
preallocated Packet.
```

When two **Inter-Node LINKs** of the same **Node** are used to pass a **Packet** from one remote **Port** to another (so-called through-routing), then only one operation is performed by the **Link Driver Task** that has received the **Packet** from the HW. After having passed on the **Packet** to the **Kernel**, the **Kernel** will insert the **Packet** in the **Driver Input Port** of the output **Inter-Node LINK**.

3.2.2. Logical view of the Router

The **Router** provides a way to map a target **Node** with a **Driver Input Port** that has to be used to route the **Packets**.

The **Router** is used in three cases:

- Sending a **Packet** to a remote **Port**
- Receiving a **Packet** from a remote **Port**
- Forwarding a Packet from a neighbouring node to another neighbouring node

3.3. Multi-tasking

As defined in Section 2.1, multiple **Tasks** may run on a single **Node** but only one **Task** can execute at a given time on a given **Node**.

3.3.1. Definition of multi-tasking

Multi-tasking as provided by **OpenComRTOS**, is defined as follows:

> **The multi-tasking is priority based, such that a higher priority Task that is ready to run gets the CPU in favour of a lower priority one (that is also ready to run)**
>
> **The multi-tasking is pre-emptive, such that when a higher priority Task becomes ready to run, it will pre-empt immediately a running Task of lower priority (hence the scheduler will switch contexts)**
>
> **The multi-tasking performs Round-Robin (or FIFO based) scheduling among equal priority Tasks that are ready to run. Time-slicing, when enabled can only happen between Tasks of equal priority.**

3.3.2. Logical view of the Context Switch

Logically, multi-tasking is supported by an atomic operation that switches the CPU context from one **Task** (to deactivate the running **Task**) to another one (to continue another ready **Task**). This operation is called the **Context Switch**.

> **The Context Switch is an atomic (non-interruptible) operation that saves the CPU context of the running Task, that is being deactivated, and restores the CPU context of another ready Task, that is being activated to run.**

In most practical implementations, the **context Switch** restores the essential CPU registers in such a way, that the resumed **Task** continues running right after the Context Switch from the point where its context was saved. The re-activated **Task** runs like if it was not ever deactivated. Note however that such states are orthogonal to the waiting and suspended states.

3.3.3. Logical view of the Kernel

The only way the **Tasks** can invoke the services of **OpenComRTOS Layer L0** is to request the services from the **Kernel**, which runs as a separate **Task**.

> **The Kernel of OpenComRTOS is a dedicated Task that serves the L0 service requests from the running Tasks and other software layers (e.g. from a HW ISR and Driver Tasks).**

All requests are passed to the **Kernel** using **Packets**, delivered to a dedicated input **Port** called the **Kernel Port**.

> **The Kernel Port is the only Port where the Packets are delivered directly in the context of a Task that inserts the Packet. Only the Kernel Task delivers the Packets to all other Ports.**

OpenComRTOS defines the following:

> **When a Packet is delivered to the Kernel Port, the requesting Task is set in the WAITING state.**
>
> **The Kernel sets the Requesting Task in the READY state only after the service request has been served (completed).**
>
> **The Kernel IS NOT ALLOWED TO access the Packet after having set the requesting Task back in the READY state.**

Each service of the **Kernel** is provided as a dedicated function call, exported to other SW layers as a part of the **Kernel API**.

> **The Kernel provides the following functional API calls to allocate, send, receive and release the Packets:**

Single Phase Services	These services will always return before the task can issue a new service.
Task management services	
L0_SuspendTask_W	Suspends a Task at its current instruction.
L0_ResumeTask_W	Resumes a Task from its current instruction.
L0_StartTask_W	Starts a Task from its entry point
L0_StopTask_W	Stops a Task and resets the instruction pointer to the Task's entry point
Packet Pool services	
L0_AllocatePacket_W	Waits until a Packet has been allocated.
L0_AllocatePacket_WT	Waits until either a Packet has been allocated or the specified timeout has expired. If the timeout has expired the return value indicates a failed allocation (there was no available Packet in the Packet pool)
L0_AllocatePacket_NW	As above but returns immediately either with the allocated Packet or with a return value indicating failure (if there was no available Packet in the

	Packet pool).
L0_DeallocatePacket_W	De-allocates a Packet.
Port based services	
L0_SendPacket_W	Waits until the sent request has synchronised with a corresponding request to receive a Packet from the specified Port.
L0_SendPacket_WT	Waits until either the sent request has synchronised with a corresponding request to receive a Packet from the specified Port, or else the specified timeout has expired. If the timeout has expired the return value indicates a failed request (there was no corresponding request to receive a Packet from the specified Port) and the sent Packet is removed from the specified Port. Upon failure the requesting Task has to take appropriate action, e.g. de-allocate the Packet.
L0_SendPacket_NW	As above but returns immediately after the Packet was delivered to the specified Port. Indicates either success (there was a corresponding request to receive a Packet from the destination Port) or failure (there was no corresponding request to receive a Packet from the specified Port; in that case the sent Packet is NOT buffered in the specified Port). Upon failure the requesting Task has to take appropriate action, e.g. de-allocate the Packet. *Note: if the specified Port is remote than the return time includes a communication delay.*
L0_ReceivePacket_W	Waits until the request has encountered a corresponding sent packet delivered to the specified Port. Upon success the calling Task has to de-allocate the Packet after processing it.
L0_ReceivedPacket_WT	Waits until either the request has encountered a corresponding send request delivered to the specified Port, or either the specified timeout has expired. If the timeout has expired the return value indicates a failed request (there was no corresponding request to receive a Packet from the specified Port) and the receive Packet is removed from the Specified Port. Upon success, the calling Task has to de-allocate the Packet after processing it.
L0_ReceivePacket_NW	Returns immediately after the request was delivered to the specified Port, indicating either success (there was a corresponding send request at the specified Port) or a failure (there was no send request at the specified Port; in that case the receive Packet is NOT buffered in the specified Port). Upon success the calling Task has to de-allocate the Packet having it processed. *Note: if the specified Port is remote than the return time includes a communication delay.*

Two Phase services	These services decouple the service request from the return of the service by requiring the task to call two paired services. In between, the task can call other services but the programmer is responsible for a correct operation.
L0_SendPacket_Async	Send a Packet (that must be allocated from the

	Packet Pool) to a Port and return immediately without being put in the waiting state. Completion is deferred till a corresponding L0_WaitForPacket service request which will return any of the packets previously send asynchronously.
L0_ReceivePacket_Async	Request to Receive a Packet from a Port without being put in the waiting state. This service requires that the receive requests use a Packet has been allocated from the Packet Pool). The completion is deferred till a corresponding L0_WaitForPacket service request which will return any of the packets previously allocated but filled in with the data of a corresponding send_request to one of the Ports.
L0_WaitForPacket_(N)W(T)	This service waits for any Packet. The programmer is responsible to take care of correct bookkeeping in terms of the allocated Packets.

The template algorithm describing how a **Task** requests a service from the **Kernel** is represented by *Figure 14.*

Having passed a request to the Kernel, a Task becomes in the waiting state, resulting in switching the context to the Task with the highest priority among the Tasks that are READY to run.

The Kernel Task has a priority higher than any other Task (incl. Link Driver Tasks).

The Kernel Task will process all requests on its Input Port until the waiting list is empty before calling the scheduler to execute the next highest priority Task on the ready list.

Tasks from the **Application Layer** are not the only ones that may request a service from the **Kernel**. In particular, a **HW ISR** can request a service. As the **HW ISR** environment (further **ISR LAYER**) cannot be set in a waiting state, **OpenComRTOS** defines the following restriction:

The ISR LAYER is only allowed to send a Packet to the local Kernel Task Input Port.

The Packets, being sent, are delivered to the Port in the context of the ISR LAYER (i.e. without switching to the Kernel Task).

These Packets will contain a Service ID that will be used by the Kernel task to invoke a specific function as needed by the application.

Running as a **Task**, the **Kernel** performs the following sequence of operations in a loop, shown in *Figure 15*. When the **Kernel** has processed all requests retrieved from its input Port, it comes in the state of waiting for other requests, and as such passes the CPU back to other **Tasks**.

The logical model of the **Kernel** is represented by *Figure 13*:

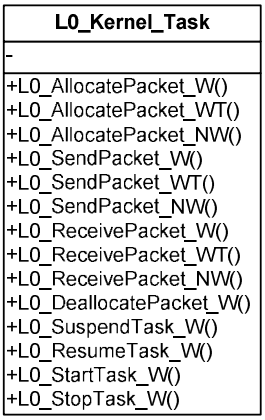


Figure 13. Logical view of the Kernel

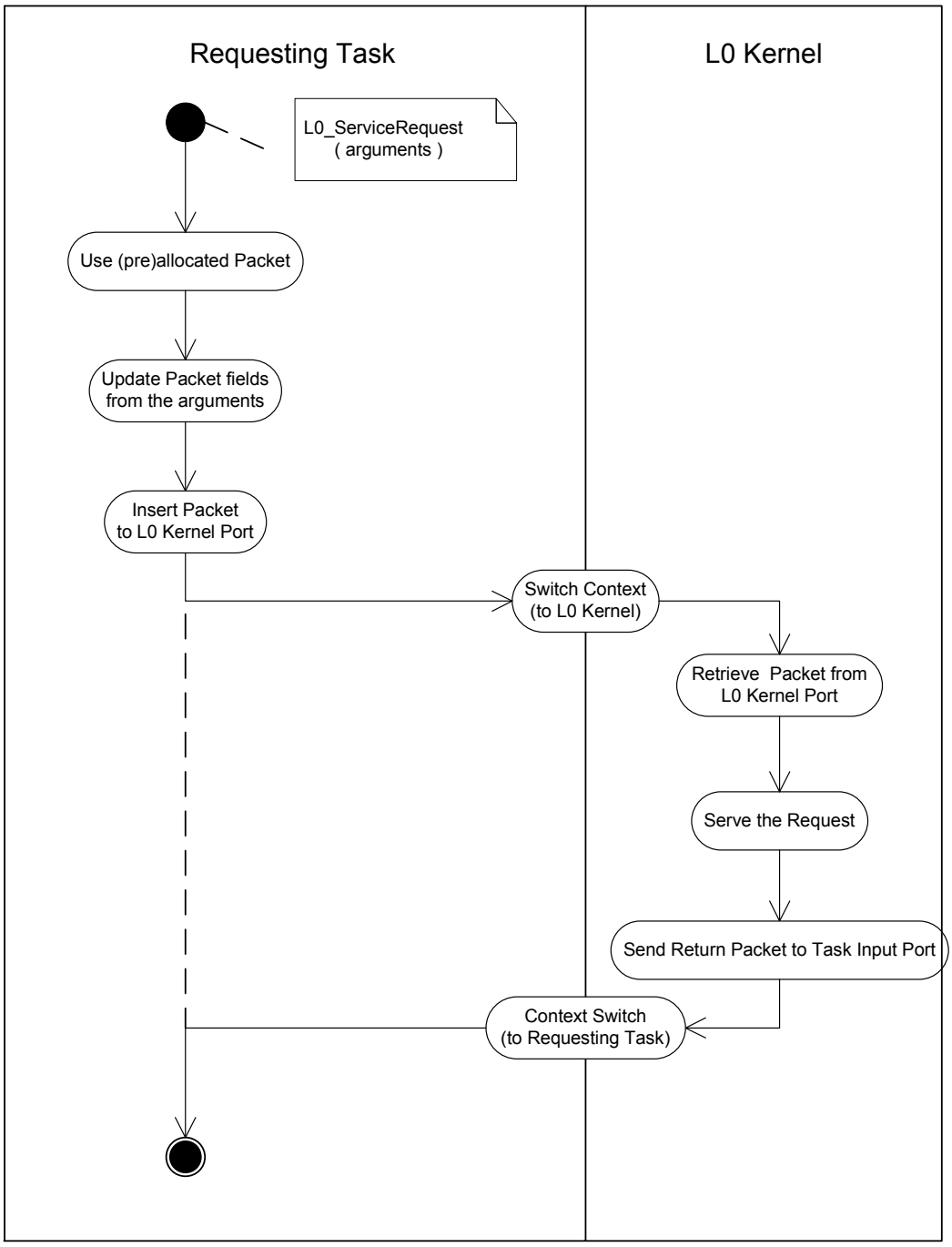


Figure 14. Template scenario of the serving of a request to the Kernel

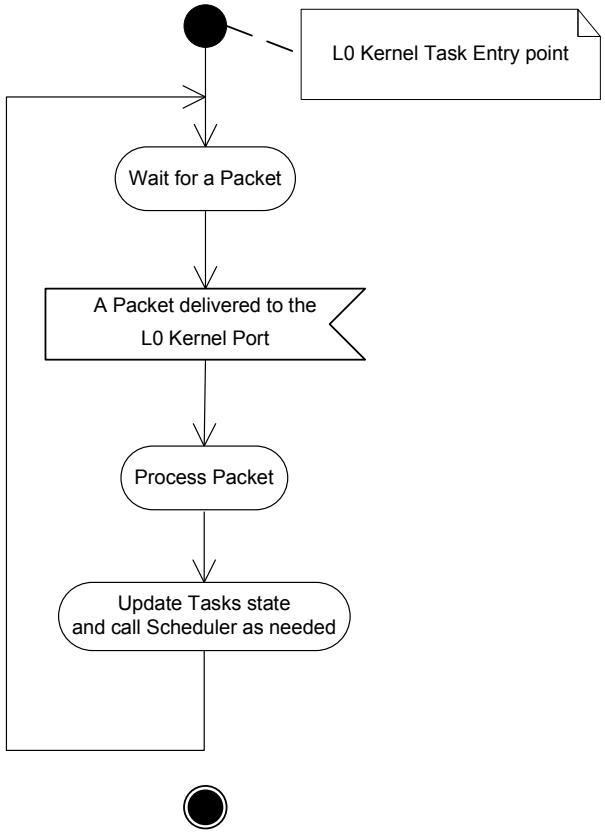


Figure 15. The Kernel Loop

3.3.4. Logical view of the L0 Scheduler

For providing multi-tasking **OpenComRTOS** has a **L0 Scheduler,** that is defined in the following way:

> **The Scheduler is a functional entity that decides which Task has to execute next, among all Tasks ready to run.**
>
> **To know what Tasks are READY to run, the Scheduler manages a dedicated (and only one) list of Tasks, called the READY list.**
>
> **The Scheduler is invoked to decide what Task to run next only in case of the following state changes in the OS environment:**
>
> - **a Task becomes ready to run and has been put into the READY list.**
> - **If a Task is no longer READY to run, it will be removed from the READY list.**

The **READY** list is a priority-ordered list of **Tasks**.

> **The Scheduler is the only SW Unit that does the Context Switch between Tasks**
>
> **The Scheduler DOES NOT decide which Task becomes READY to run and which Task becomes WAITING, it just schedules the task that has the highest priority on the READY List. The decisions are always made by the logic of interaction with the Ports (see Section *3.1.3*) or by the logic of the service requested of the Kernel Task by a Task. (see Section *3.3.3*).**

The logical model of the **Scheduler** is shown in *Figure 16*:

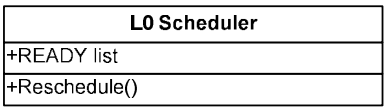


Figure 16. Logical view of the Scheduler

4. Design view of Layer 0

4.1. Predefined constants

L0_PacketSize	Size of the L0_Packets
L1_PacketSize	Size of the L1_Packets
L2_PacketSize	Size of the L2_Packets
L0_DataSize	Size of the Data in a L0 Packet (in Bytes)
L0_InfiniteTimeOut	Infinite timeout (0xFFFF Hex)

4.2. Data types

4.2.1. BYTE

BYTE is a 8-bit unsigned integer type.

4.2.2. INT16

INT16 is a 16-bit signed integer type.

4.2.3. INT32

INT32 is a 32-bit signed integer type.

4.2.4. UINT16

UINT16 is a 16-bit unsigned integer type.

4.2.5. UINT32

UINT32 is a 32-bit unsigned integer type.

4.2.6. BOOL

BOOL is a basic integer type sufficient to represent the values: TRUE and FALSE.

4.2.7. EntityAddress

EntityAddress is an abstract type that represents an identifier of an Entity.
EntityAddress is a system wide address represented by a 32 bit data structure with the following 8bit fields: LocalEntityID, NodeID, SiteID, ClusterID.
In practice at L0 we will only find EntityAdresses for Tasks en Ports and the context will allow to distinguish between them. In this context we call them TaskID and PortID.

4.2.7.1 TaskID

TaskID is a type that represents an identifier of a **Task** .

`TaskID` is a system wide identifier represented by a 32 bit data structure divided in the following 8bit fields: LocalTaskID, NodeID, SiteID, ClusterID.

4.2.7.2 PortID

`PortID` is an type that represents an identifier of a **Port** on a **Node**.
`PortID` is a system wide identifier represented by a 32 bit datastructure divided in the following 8bit fields: LocalPortID, NodeID, SiteID, ClusterID.

4.2.8. L0_Prio

`L0_Prio` is a basic unsigned integer type sufficient to represent the values from 0 to 255, identifying the priority of a Task or a Packet.

4.2.9. L0_Timeout

`L0_Timeout` is a basic unsigned integer type that represents a timeout value in **milliseconds**. The maximum value, allowed by the appropriate **`L0_Timeout`** integer type, is interpreted as an **infinite** timeout. For example if **`L0_Timeout`** is provided by the means of a 16-bit unsigned integer, then the **infinite** timeout is **0xFFFF Hex.** The **infinite** timeout is (should be) referred as named constant **`L0_Infinite_TimeOut`**

4.2.10. L0_ListElement

`L0_ListElement` is a structure that is an element of a bidirectional linked list.

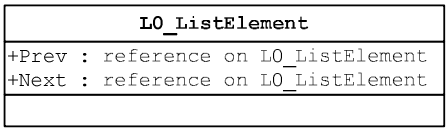


Figure 17 L0_ListElement

4.2.11. L0_PrioListElement

`L0_PrioListElement` is a structure that is an element of an ordered bidirectional linked list. The list is ordered is descending order of the priorities of the list elements.

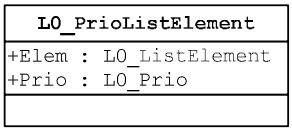


Figure 18 L0_PrioListElement

Remarks:

`ListElement` defines a prioritized list element as an ordinary (not prioritized) list element. Actually, *`ListElement`* is the “head” part of **`L0_PrioListElement`**. So, the pointer to an object of type **`L0_PrioListElement`** coincides with the pointer to its element *`ListElement`*.

4.2.12. L0_List

`L0_List` is a structure that maintains **L0_ListElement** elements in a linear arrangement and allows efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing an element of the same type.

Figure 19 L0_List

4.2.13. L0_PrioList

L0_PrioList is a structure that maintains **L0_PrioListElement** elements in a linear arrangement and allows efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing an element of the same type. The list is ordered is descending order of the priorities of the list elements.

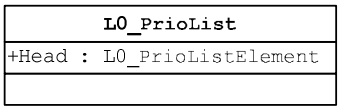


Figure 20 L0_PrioList

4.2.14. L0_TaskArguments

L0_TaskArguments is a data structure representing the startup arguments of a Task. The semantics of each concrete use of **L0_TaskArguments** is specific for each **Task**. For example, **L0_TaskArguments** can be assigned to a pointer to an array of parameters, or being cast to an integer for a single parameter.

4.2.15. L0_TaskFunction

L0_TaskFunction is a pointer to a function with one input parameter of type **L0_TaskArgs**. The function, pointed to by **L0_TaskFunction** is used as an entry point to start a **Task**.

4.2.16. L0_Status

L0_Status is an enumeration type used to specify the result of a service request (success, failure, etc.).

RC_OK	Return code for a successful request
RC_Fail	Return code for a failed request
RC_TimeOut	Return code for a failed request after the timeout expired.

4.2.17. L0_ServiceID

L0_ServiceID is an enumeration type used to identify the services, provided by the **Kernel**.

L0_SID_AllocatePacket	Service identifier for allocation of a packet
L0_SID_DeallocatePacket	Service identifier for deallocation of a packet
L0_SID_SuspendTask	Service identifier for suspension of a task
L0_SID_ResumeTask	Service identifier for resumption of a task
L0_SID_StartTask	Service identifier for starting a task
L0_SID_StopTask	Service identifier for stopping a task
L0_SID_SendPacket	Service identifier for sending of a packet
L0_SID_ReceivePacket	Service identifier for receiving of a packet
L0_SID_ReceiveAnyPacket	Service identifier for receiving any packet (only used by TX Driver Tasks)

Figure 21 L0_ServiceID

4.3. The design view of a Task

A **Task** is instantiated by a data structure **`L0_TaskControlRecord`**. The whole set of L0_TaskControlRecords is called the **L0_TaskControlBlock**.

A Error! Reference source not found.**`Record`** does not contain an explicit indication of the **Task** state unless **SUSPENDED**. The states are implicitly indicated by the following conditions:

- A **Task** is **RUNNING**, if it is an element of the **READY List** of the **L0 Scheduler**.
- A **Task** is **WAITING**, if it has a **Packet** in a waiting list (of a **Port** or of the **Packet Pool**).
- A **Task** is **SUSPENDED** if it is marked as **SUSPENDED**.

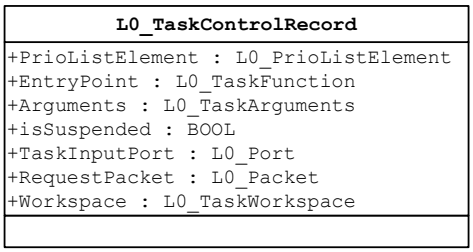


Figure 22 L0_TaskControlRecord

Remarks:

- *`PrioListElement`* defines a **Task** as a prioritized list element. Actually, *`PrioListElement`* is the first field of **`L0_TaskControlRecord`**. The pointer to an object of type **`L0_TaskControlRecord`** coincides with the pointer to its element *`PrioritizedListElement`*.
- *`Prio`* of *`PrioListElement`* is the priority of the **Task**.
- *`EntryPoint`* specifies the function of a **Task**.
- *`Arguments`* specifies arguments of a **Task**.
- *`IsSuspended`* indicates if a **Task** is suspended.
- *`InputPort`* identifies the **Input Port** of the **Task**. (**Driver Input Port** or **Task Input Port**.
- *`RequestPacket`* specified in Section **Error! Reference source not found.**, which is a Task's pre-allocated (static) **Packet**.
- *`TaskContext`* is a platform dependent part used to store essential descriptors of the current state of a **Task** when switching the CPU context to another task. (i.e. the CPU Context defined in Section 3.1.1)

4.4. The design view of a Packet

A **Packet** is instantiated by a data structure Error! Reference source not found.. When in use**,** a **Packet** will often be an element of waiting lists. The model of a Packet is derived from the model of the prioritized list element (see **`L0 PrioListElement`**).

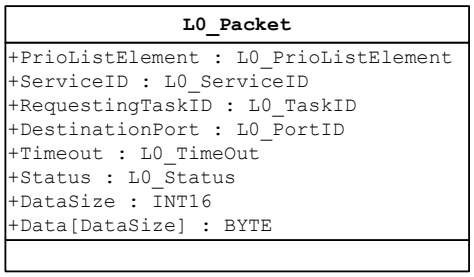


Figure 23 L0_Packet

Remarks:

PrioListElement defines a **Packet** as a prioritized list element. Actually, *PrioListElement* is the first field of a **L0_Packet**. So, the pointer to an object of type **L0_Packet** coincides with the pointer to its element *PrioListElement*. The *Priority* of *PrioListElement* is the priority of the **Packet**.

ServiceID specifies the Service that is requested by a **Packet** from the **Kernel**.

RequestingTaskID specifies the Task that owns the **Packet**. This includes the LocalTaskID, NodeID, ClusterID and SiteID.

DestinationPortID specifies the **Port** to/from which the **Packet** has to be sent/received. This includes the LocalPortID, NodeID, ClusterID and SiteID.

TimeOut specifies the timeout (if any) associated with the requested service.

Status indicates the status of completion of the service, set by the **Kernel** when it finishes serving the request.

DataSize specifies the size of the user data, supplied to the **Packet**. It can be zero.

Data specifies the data, supplied to the **Packet**. Depending on the value of *ServiceID*, the content of *Data* may be one of the following:

- If *ServiceID* is SID_ALLOCATE_Packet, then *Data* is never used.
- If *ServiceID* is SID_DEALLOCATE_Packet, then *Data* is never used.
- If *ServiceID* is SID_SendPacket or SID_ReceivePacket, then *Data* contains the data that has to be delivered to a receiving **Task**.

4.5. The design view of a Port

L0_Port is a data structure representing a **Port**.

The architecture defines the logical view of a **Port** as one that has two waiting lists: the **Receive Request Waiting List** and the **Send Request Waiting List**. From design point of view there is no need to operate with two waiting lists as the requests cancel each other out. At any given point in time, there can either be only receive request(s), or only send request(s) or the waiting lists are empty.

Inserting or removing an element in the waiting list must be an atomic operation.

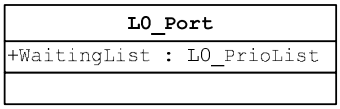


Figure 24 L0_Port

5. Procedures and algorithms

5.1. Kernel API calls

5.1.1. L0_Status L0_StartTask_W (TaskID)

This will start the task with TaskID and add it to the READY list of the node on which the Task resides.

Parameters:

`TaskID` - the Task to start

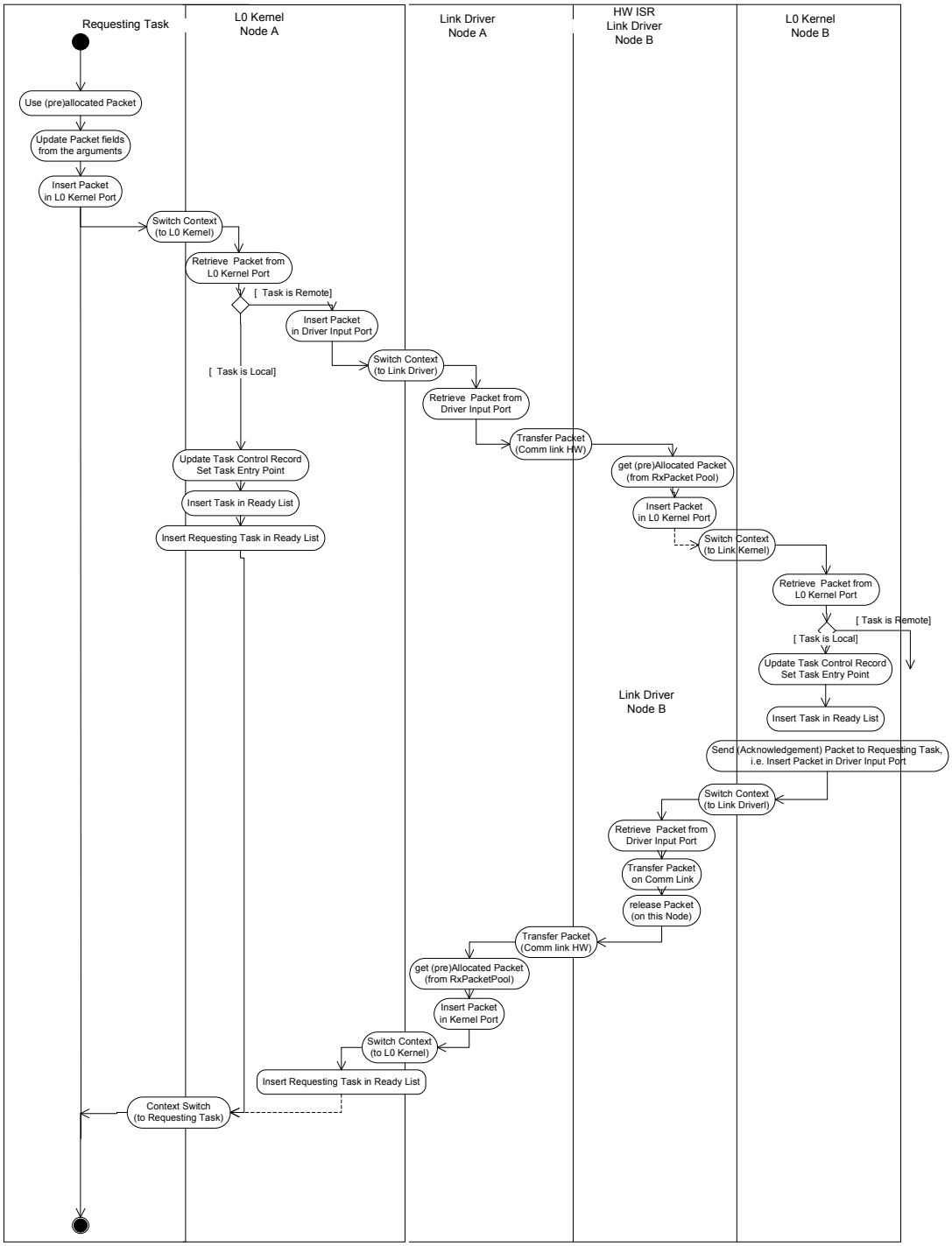
Return value:
`RC_OK` - the Task has started successfully.
`RC_FAIL` - the service failed.

Pre-conditions:

- Task is inactive
- Task is initialised and ready to start
- All elements of TaskControlRecord are filled in, incl. entrypoint and stack address
- The Task cannot start itself

Post-conditions:

- Task is on the READY list (case RC_OK)


Requesting Task
L0 Kernel Node A
Link Driver Node A
HW ISR Link Driver Node B
L0 Kernel Node B
Use (pre)allocated Packet
Update Packet fields from the arguments
Insert Packet in L0 Kernel Port
Switch Context (to L0 Kernel)
Retrieve Packet from L0 Kernel Port
[Task is Remote]
Insert Packet in Driver Input Port
Switch Context (to Link Driver)
[Task is Local]
Retrieve Packet from Driver Input Port
Transfer Packet (Comm link HW)
Update Task Control Record Set Task Entry Point
get (pre)Allocated Packet (from RxPacket Pool)
Insert Task in Ready List
Insert Packet in L0 Kernel Port
Insert Requesting Task in Ready List
Switch Context (to Link Kernel)
Retrieve Packet from L0 Kernel Port
[Task is Remote]
[Task is Local]
Update Task Control Record Set Task Entry Point
Link Driver Node B
Insert Task in Ready List
Send (Acknowledgement) Packet to Requesting Task, i.e. Insert Packet in Driver Input Port
Switch Context (to Link Driverl)
Retrieve Packet from Driver Input Port
Transfer Packet on Comm Link
release Packet (on this Node)
Transfer Packet (Comm link HW)
get (pre)Allocated Packet (from RxPacketPool)
Insert Packet in Kernel Port
Switch Context (to L0 Kernel)
Insert Requesting Task in Ready List
Context Switch (to Requesting Task)

5.1.2. L0_Status L0_StopTask_W (TaskID)

This will stop the task with TaskID, remove it from the READY list, remove any pending Packets on all waiting lists and restore the entry point.

Parameters:

`TaskID`	- the Task to stop

Return value:

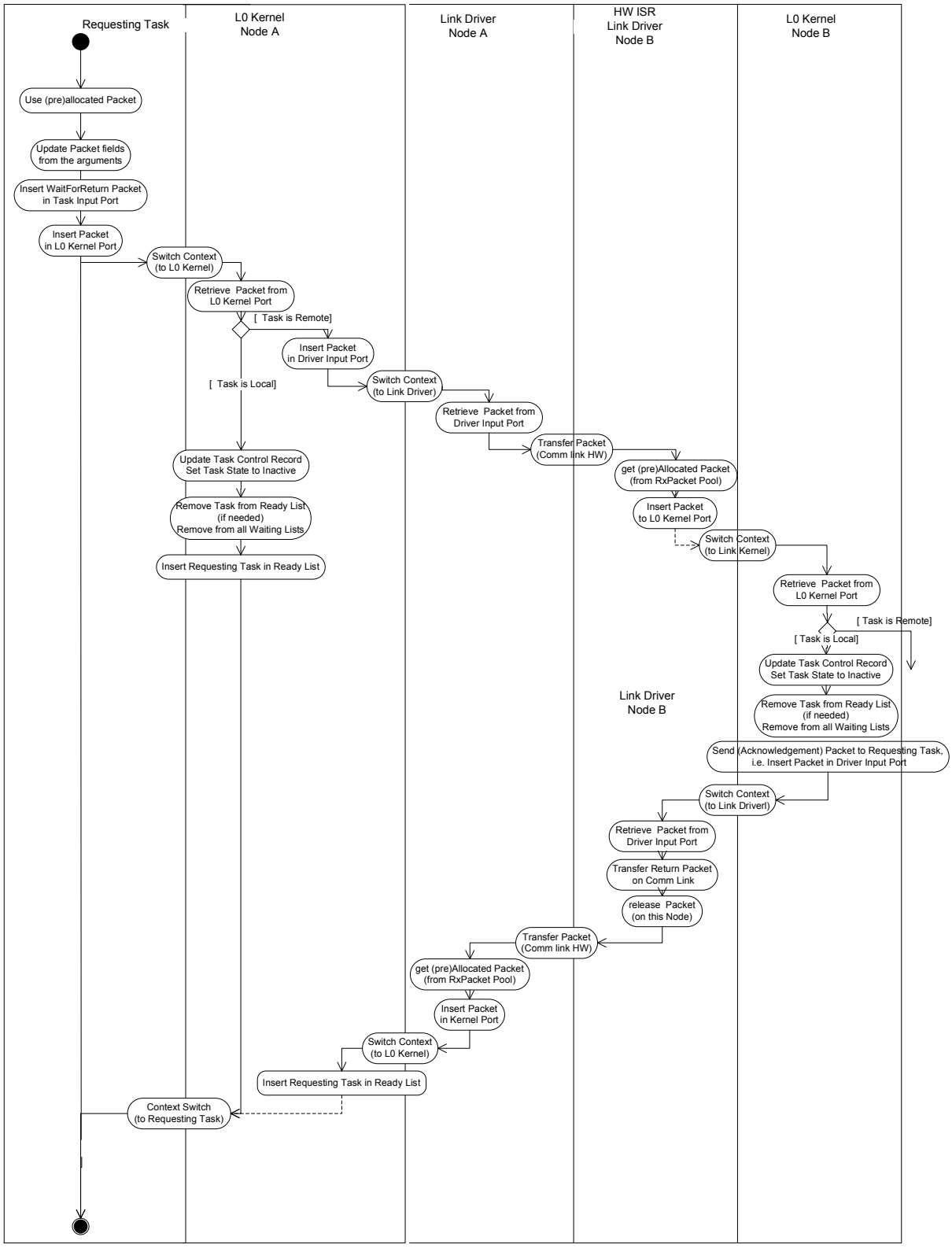
`RC_OK`	- the Task has started successfully.
`RC_FAIL`	**-** the service failed.

Pre-conditions:

- Task is not inactive and not stopped
- The Task is not the requesting task itself

Post-conditions:

- Task is no longer on any waiting list **(see release notes)**
- Entry Point restored
- Any data may be lost


Requesting Task
L0 Kernel
Node A
Link Driver
Node A
HW ISR
Link Driver
Node B
L0 Kernel
Node B
Use (pre)allocated Packet
Update Packet fields
from the arguments
Insert WaitForReturn Packet
in Task Input Port
Insert Packet
in L0 Kernel Port
Switch Context
(to L0 Kernel)
Retrieve Packet from
L0 Kernel Port
[Task is Remote]
[Task is Local]
Insert Packet
in Driver Input Port
Switch Context
(to Link Driver)
Retrieve Packet from
Driver Input Port
Transfer Packet
(Comm link HW)
get (pre)Allocated Packet
(from RxPacket Pool)
Insert Packet
to L0 Kernel Port
Switch Context
(to Link Kernel)
Retrieve Packet from
L0 Kernel Port
[Task is Remote]
[Task is Local]
Update Task Control Record
Set Task State to Inactive
Remove Task from Ready List
(if needed)
Remove from all Waiting Lists
Send (Acknowledgement) Packet to Requesting Task,
i.e. Insert Packet in Driver Input Port
Link Driver
Node B
Update Task Control Record
Set Task State to Inactive
Remove Task from Ready List
(if needed)
Remove from all Waiting Lists
Insert Requesting Task in Ready List
Switch Context
(to Link Driverl)
Retrieve Packet from
Driver Input Port
Transfer Return Packet
on Comm Link
release Packet
(on this Node)
Transfer Packet
(Comm link HW)
get (pre)Allocated Packet
(from RxPacket Pool)
Insert Packet
in Kernel Port
Switch Context
(to L0 Kernel)
Insert Requesting Task in Ready List
Context Switch
(to Requesting Task)

Note:

This service must be used with caution. It assumes perfect knowledge about the system by the invoking Task. Normally only to be used when the Task is found to misbehaving (e.g. Stack overflow, numerical exception, etc.)

Additional kernel service (messages) may be used for the clean-up of pending Packets in waiting list on other nodes. This “clean-up” behaviour is NOT shown in the Figure above.

5.1.3. L0_Status L0_SuspendTask_W (TaskID)

This will suspend the task with TaskID and mark it as such in the Task Control Record.

Parameters:

`TaskID`	- the Task to suspend

Return value:

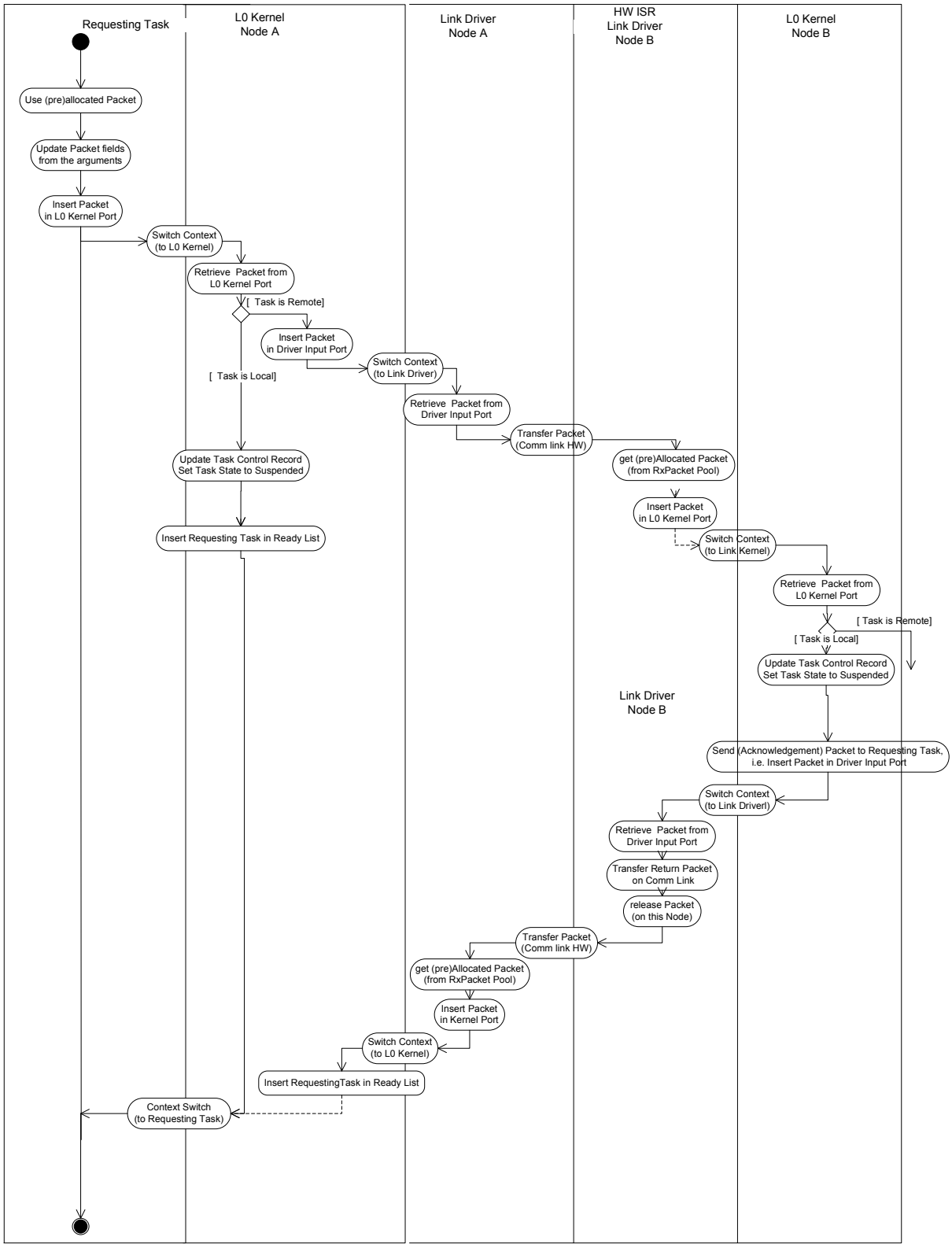
`RC_OK`	- the Task has been suspended successfully.
`RC_FAIL`	**-** the service failed.

Pre-conditions:

- The Task is not the requesting task itself

Post-conditions:

- Task is marked as suspended
- Requests for the task can continue to arrive from other tasks


Requesting Task
L0 Kernel Node A
Link Driver Node A
HW ISR Link Driver Node B
L0 Kernel Node B
Use (pre)allocated Packet
Update Packet fields from the arguments
Insert Packet in L0 Kernel Port
Switch Context (to L0 Kernel)
Retrieve Packet from L0 Kernel Port
[Task is Remote]
[Task is Local]
Insert Packet in Driver Input Port
Switch Context (to Link Driver)
Retrieve Packet from Driver Input Port
Transfer Packet (Comm link HW)
get (pre)Allocated Packet (from RxPacket Pool)
Insert Packet in L0 Kernel Port
Switch Context (to Link Kernel)
Retrieve Packet from L0 Kernel Port
[Task is Remote]
[Task is Local]
Update Task Control Record Set Task State to Suspended
Update Task Control Record Set Task State to Suspended
Insert Requesting Task in Ready List
Link Driver Node B
Send (Acknowledgement) Packet to Requesting Task, i.e. Insert Packet in Driver Input Port
Switch Context (to Link Driverl)
Retrieve Packet from Driver Input Port
Transfer Return Packet on Comm Link
release Packet (on this Node)
Transfer Packet (Comm link HW)
get (pre)Allocated Packet (from RxPacket Pool)
Insert Packet in Kernel Port
Switch Context (to L0 Kernel)
Insert RequestingTask in Ready List
Context Switch (to Requesting Task)

The suspend service is the fastest way to prevent a Task from executing any further code. It should only be used when the application has a good reason and need to be followed by an analysis, eventually resulting in a corrective action (e.g. by an operator or stopping and restarting a Task).

Pending Packets in any waiting list remain pending, and are continued to be processed e.g. synchronization. In particular, the Task may remain and inserted in the READY List. The task is however never made RUNNING.

5.1.4. L0_Status L0_ResumeTask_W (TaskID)

This call will resume the task at the point it was when suspended

Parameters:

`TaskID`	- the Task to resume

Return value:

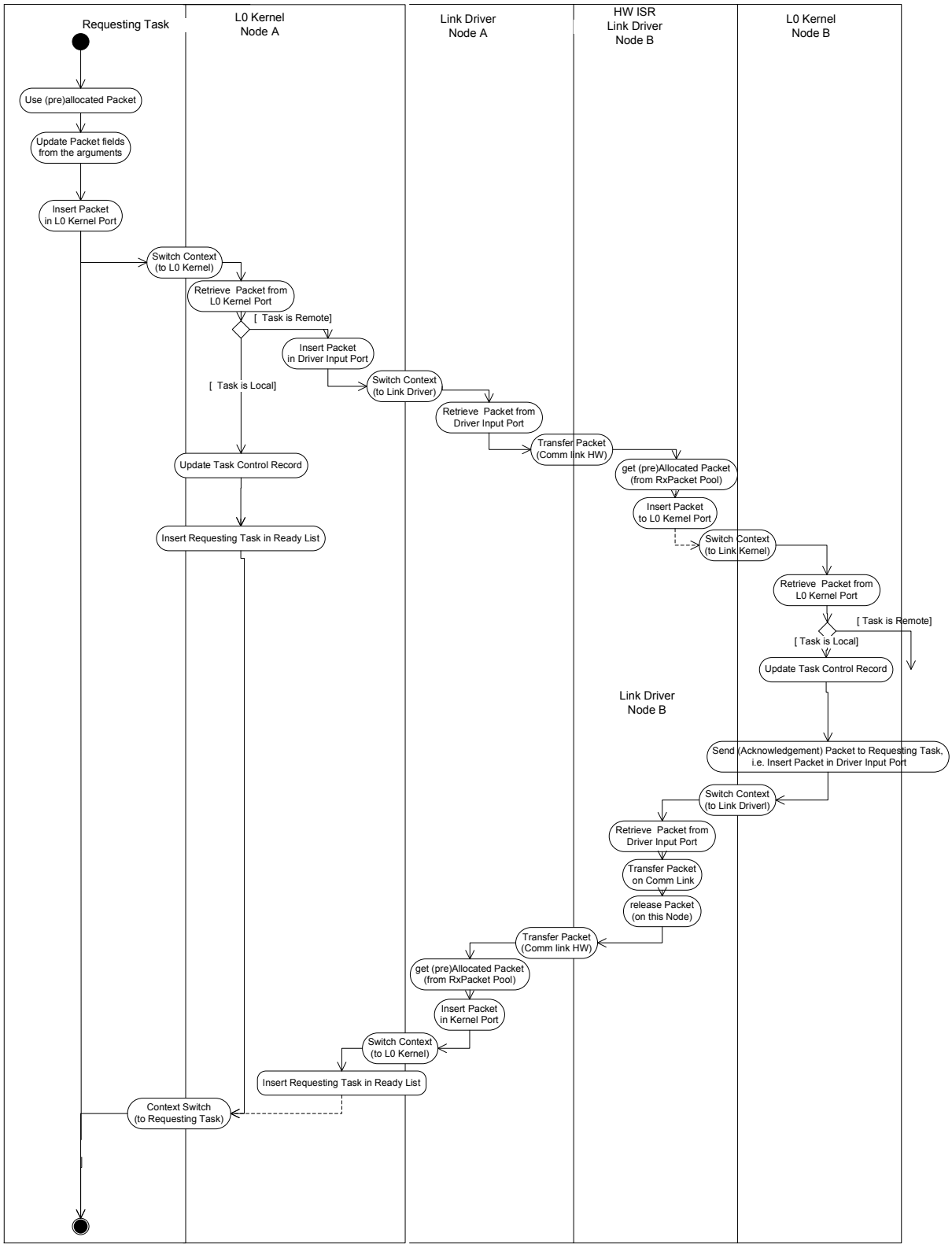
`RC_OK`	- the Task has been resumed successfully.
`RC_FAIL`	- the service failed.

Pre-conditions:

- Task was in suspend state

Post-conditions:

- Task resumed at the point it was when suspended.


Requesting Task
L0 Kernel
Node A
Link Driver
Node A
HW ISR
Link Driver
Node B
L0 Kernel
Node B
Use (pre)allocated Packet
Update Packet fields
from the arguments
Insert Packet
in L0 Kernel Port
Switch Context
(to L0 Kernel)
Retrieve Packet from
L0 Kernel Port
[Task is Remote]
Insert Packet
in Driver Input Port
[Task is Local]
Switch Context
(to Link Driver)
Retrieve Packet from
Driver Input Port
Transfer Packet
(Comm link HW)
get (pre)Allocated Packet
(from RxPacket Pool)
Insert Packet
to L0 Kernel Port
Switch Context
(to Link Kernel)
Retrieve Packet from
L0 Kernel Port
[Task is Remote]
[Task is Local]
Update Task Control Record
Update Task Control Record
Insert Requesting Task in Ready List
Link Driver
Node B
Send (Acknowledgement) Packet to Requesting Task,
i.e. Insert Packet in Driver Input Port
Switch Context
(to Link Driverl)
Retrieve Packet from
Driver Input Port
Transfer Packet
on Comm Link
release Packet
(on this Node)
Transfer Packet
(Comm link HW)
get (pre)Allocated Packet
(from RxPacket Pool)
Insert Packet
in Kernel Port
Switch Context
(to L0 Kernel)
Insert Requesting Task in Ready List
Context Switch
(to Requesting Task)

Note:

5.1.5. L0_Status L0_AllocatePacket _{NW|W|WT } (L0_Packet *Packet, [L0_Timeout Timeout])

This **Kernel** service is called by a **Task** to allocate a **Packet** from the **Packet Pool**.

Parameters:

L0_Packet *Packet*	– will contain the **Packet** upon successful return.
L0_Timeout *Timeout*	– Timeout value. **Timeout > 0 and < InfiniteTimeout** **Timeout = InfiniteTimeout(_W variant)** **Timeout = 0 stands for _NW variant**

Return value:

RC_OK	- service terminated successfully (there was an available **Packet** in the **Packet Pool**).
RC_FAIL	- service failed (no available **Packet** in the **Packet Pool**).
RC_FAIL_TO	- service failed and returned after Timeout.

Pre-conditions:

- This service cannot be called from the **ISR LAYER**

Post-conditions:

- *ServiceID* of the pre-allocated **Packet** of the calling **Task** will be set to **SID_Allocate_Packet**.
- Task is on READY list upon return
- ***Packet*** can be used for two-phase services

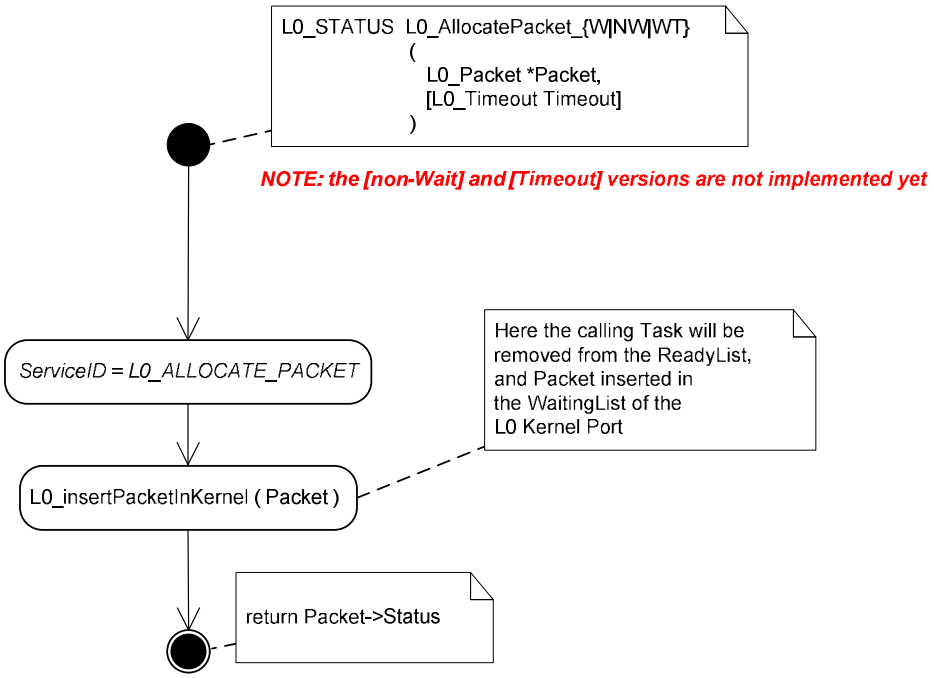


Figure 25. Algorithm of the procedure L0_AllocatePacket

5.1.6. void L0_DeallocatePacket_W(L0_Packet Packet)

This **Kernel** service is called by a **Task** to **DEALLOCATE** a **Packet** and return it to the **Packet Pool**.

Parameters:

L0_Packet *Packet* – the **Packet** that has to be de-allocated.

Return value:

RC_OK or RC_Fail

Pre-conditions:

- This service cannot be called by **ISR LAYER**
- *Packet* must have been allocated by **L0_AllocatePacket**

Post-conditions:

- Packet is no longer available for use by Task

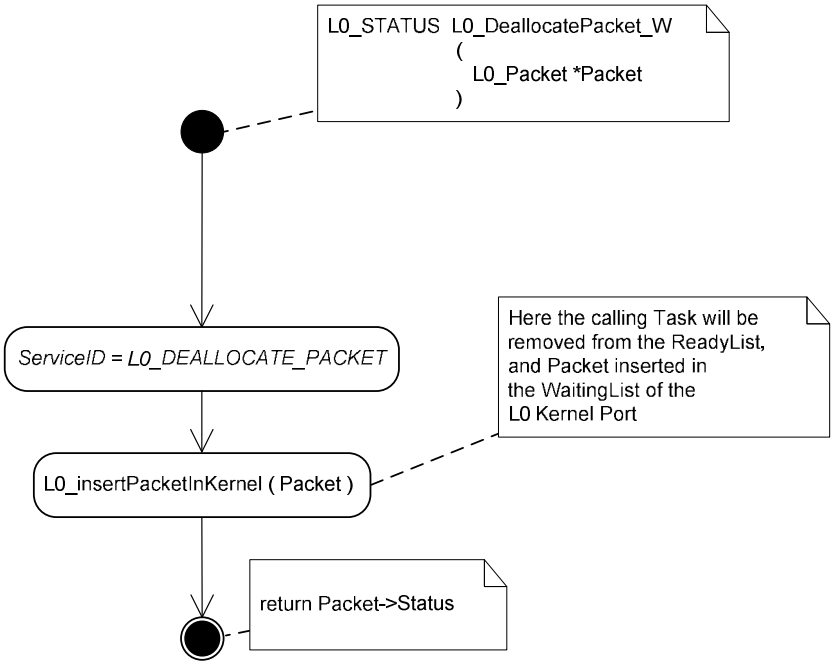


Figure 26. Algorithm of the procedure L0_DeallocatePacket

5.1.7. L0_Status L0_SendPacket_{NW|W|WT } (L0_PortID Port, L0_Packet Packet, [L0_Timeout Timeout])

This **Kernel** service is called by a **Task** to send a **Packet** to a **Port**.

Parameters:

L0_PortID *Port* – identifies the **Port**, to which the calling **Task** wants to send a **Packet**.

L0_Packet *Packet* – the **Packet** that has to be sent.

L0_Timeout *Timeout* – Timeout value.

Return value:

RC_OK - service successful (there was a waiting receive request in the **Port**)

RC_FAIL - service failed (no corresponding receive request in the **Port**)

RC_FAIL_TO - service failed and returned after Timeout

Pre-conditions:

- *Packet* is the preallocated Packet

Post-conditions:

- Header fields of preallocated Packet filled in

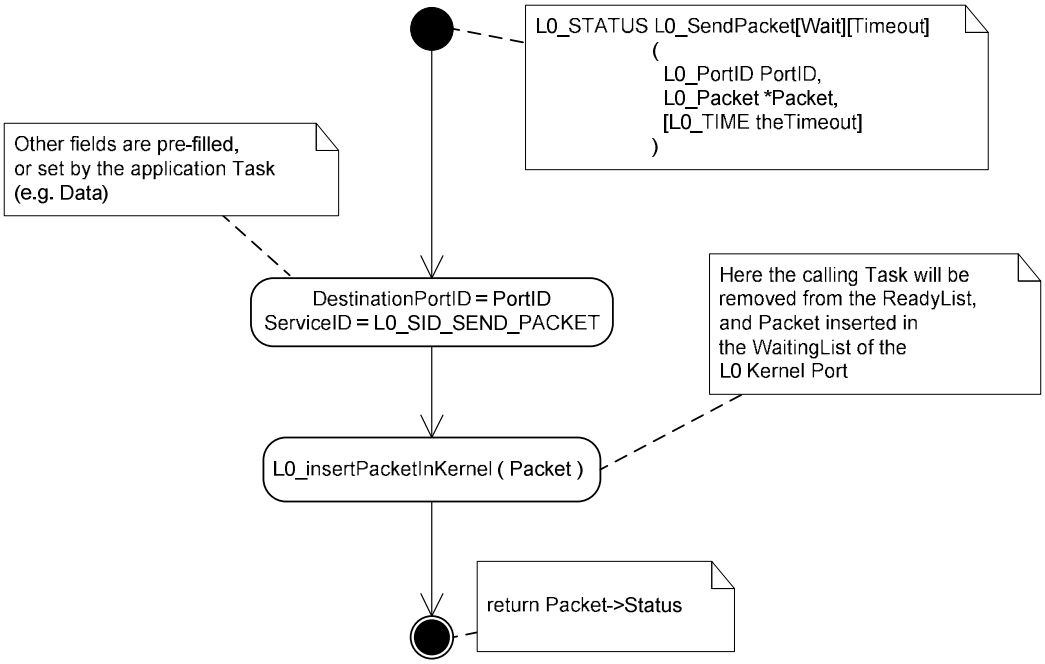


Figure 27 Algorithm of the procedure L0_SendPacket

5.1.8. L0_Status L0_ReceivePacket_{NW|W|WT } (L0_PortID Port, L0_Packet Packet, [L0_Timeout Timeout])

This **Kernel** service is called by a **Task** to receive a **Packet** from a **Port**.

Parameters:

`L0_PortID` *`Port`*	– identifies the **Port**, to which the calling **Task** wants to send a **Packet**.
`L0_Packet` *`Packet`*	– the pre-allocated Packet
`L0_Timeout` *`Timeout`*	– Timeout value.

Return value:

`RC_OK`	- service successful (there was a waiting send request in the **Port**)
`RC_FAIL`	**-** service failed (no corresponding send request in the **Port**)
`RC_FAIL_TO`	- service failed and returned after Timeout

Pre-conditions:

- *`Packet`* is the preallocated Packet

Post-conditions:

- Header fields of preallocated Packet filled in
- Data of send Packet will have been filled in

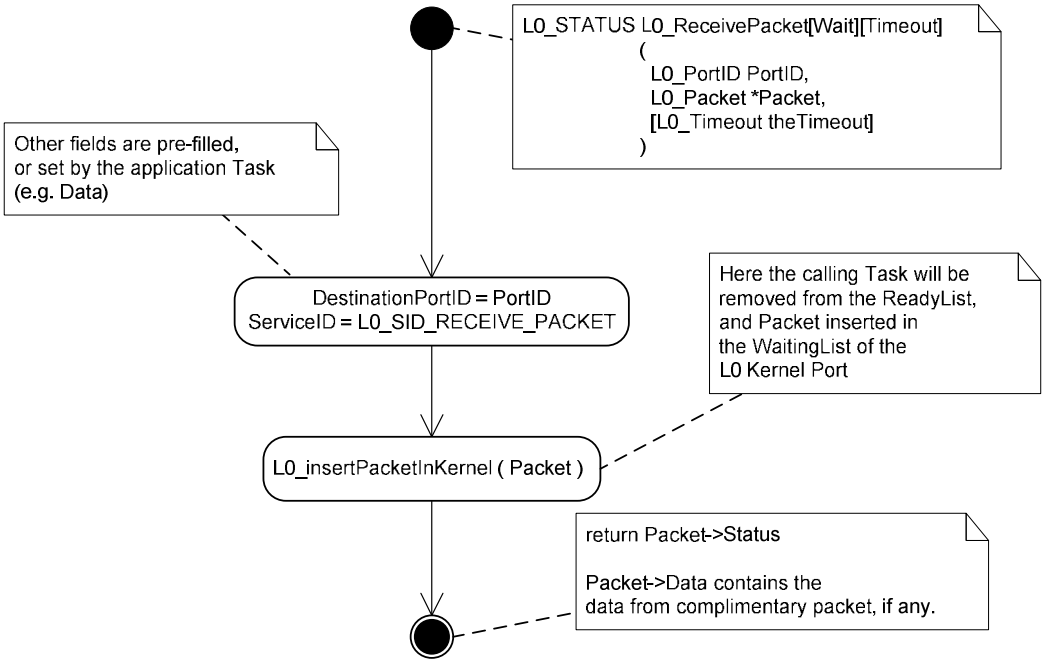


Figure 28 Algorithm of the procedure L0_ReceivePacket

5.1.9. void L0_SendPacket_Async (L0_Port Port, L0_Packet Packet)

This **Kernel** service is called by a **Task** to send a **Packet** asynchronously to a **Port**.

Parameters:

`L0_PortID` *`Port`* – identifies the **Port**, to which the calling **Task** wants to send a **Packet**.

`L0_Packet` *`Packet`* – the **Packet** that has to be sent, allocated from the packet Pool

Return value:

None

Pre-conditions:

- *`Packet`* must have been allocated by the function `L0_AllocatePacket`.

Post-conditions:

- The calling task will remain on the **READY List**

FIGURE: TBD

5.1.10. void L0_ReceivePacket_Async (L0_Port Port, L0_Packet Packet)

This **Kernel** service is called by a **Task** to receive a **Packet** asynchronously from a **Port**.

Parameters:

`L0_PortID` *`Port`* – identifies the **Port**, to which the calling **Task** wants to send a **Packet**.

`L0_Packet` *`Packet`* – a packet allocated from the Packet Pool

Return value:

None

Pre-conditions:

- *Packet* must have been allocated by the function **L0_AllocatePacket**.

Post-conditions:

- The calling task will remain on the **READY List**

FIGURE: TBD

5.1.11. L0_Status L0_WaitForPacket__{NW|W|WT } (L0_PortID Port, L0_Packet, L0_Timeout Timeout)

This **Kernel** service is called by a **Task** to resynchronize on Packets send earlier using the L0_SendPacketAsync service

Parameters:

L0_PortID *Port*	– identifies the **Port**, to which the calling **Task** wants to send a **Packet**.
L0_Packet *Packet*	– the preallocated Packet
L0_Timeout *Timeout*	– Timeout value.

Return value:

RC_OK	- service terminated successfully (there was a waiting Packet in the **Port**)
RC_FAIL	- service failed (no corresponding Packet in the **Port**)
RC_FAIL_TO	- service failed and returned after Timeout

Pre-conditions:

- This service should have been preceded by a L0_SendPacket_Async or L0_ReceivePacket_Async

Post-conditions:

- The preallocated Packet should contain a pointer to a previously allocated Packet from the Packet Pool

FIGURE: TBD

5.2. Function called internally by the API services in the context of the calling Task

5.2.1. void L0_InsertPacketInKernel (L0_Packet)

This function is called internally by the API services in the context of the calling Task to insert a service request packet into the Kernel and switches context to the kernel. Upon return the context will have been restored.

Parameters:

`L0_Packet` *`Packet`* – the (pre)allocated Service Request Packet

Return value:

None

Pre-conditions:

- The Packet header fields must have been correctly filled in according to the requested service

Post-conditions:

- The return value and all other header fields and data are updated

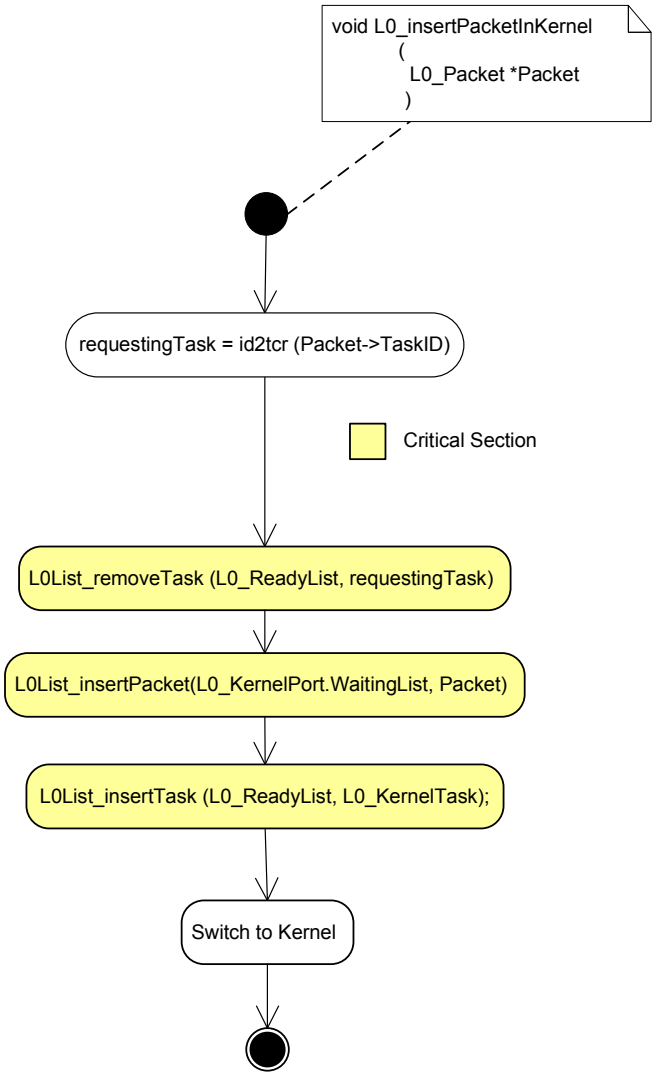


Figure 29. Algorithm of the procedure L0_insertPacketInKernel

5.3. Kernel Internal API calls

5.3.1. void L0_KernelLoop (void)

The **Kernel Task** executes its function to perform the main loop of the **Kernel Task**.

Parameters:

No parameters

Pre-conditions:
to be started before all other tasks and when all nodes have been booted

Post-conditions:
void (as infinite loop)

Remarks:

The **Kernel Task** executes `L0_Kernel_Loop` to process all requests from the **Tasks** and from the hardware layer. These requests have been put as **Packets** on the **Waiting List** of the **Kernel Port**. Whenever the **Waiting List** is empty, the **Kernel Task** is set in the **WAITING** state (i.e. removed from the **READY List)**.
The **Kernel Task** is re-inserted in the **READY List** when a new **Packet** is put on the Waiting List of the **Kernel Port**. If the **Kernel Task** is the highest priority task in the **READY List**, the **CPU Context** is switched to it.

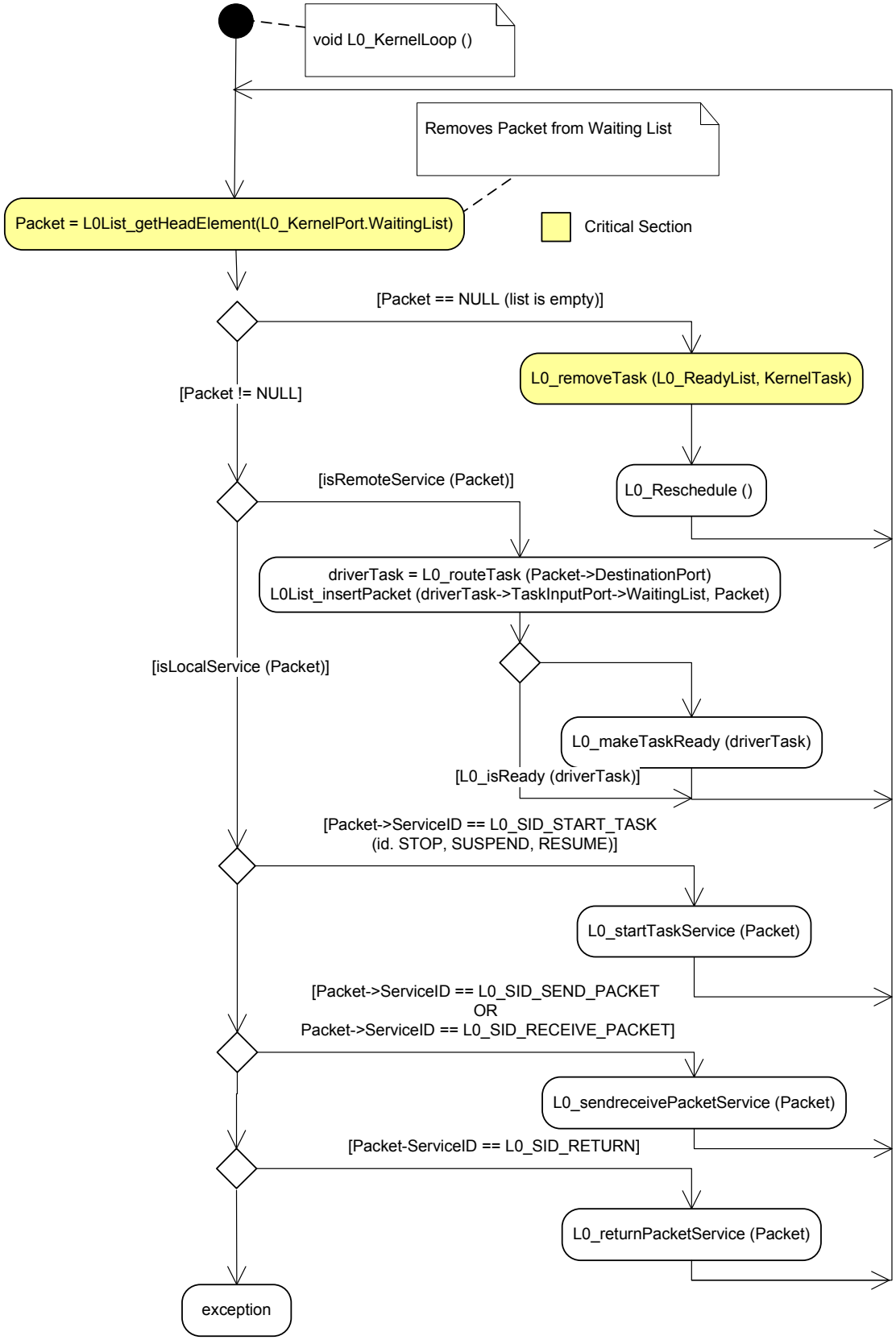


Figure 30. Algorithm of the procedure L0_KernelLoop

5.3.2. void L0_AllocatePacketService(L0_Packet Packet)

The **Kernel Task** calls this procedure to process the specified **Packet** that requests to allocate a **Packet** from the **Packet Pool**.

FIGURE TBD

Parameters:

`L0_Packet` *`Packet`* **-** the **Packet** that has to be served.

Return value:

`void`

Pre-conditions:

- The **Task**, owning *`Packet`*, is not on the **READY List**
- The calling **Task** is the **Kernel Task**

Post-conditions:

- If the **Packet Pool** contains a **Packet**, then that **Packet** is passed to the highest priority **Task** waiting for the requested allocation
- Next the **Task** is added to the **READY List**. At that, *`Status`* of *`Packet`* *is* set to **`L0_RC_OK`**.
- In case of the **[_W][_T]** version: If the **Packet Pool** does not contain a **Packet**, then *`Packet`* is set on the **Waiting List** of the **Packet Pool**.
- In case of the **[_NW]** version: If the **Packet Pool** does not contain a **Packet**, then the **Task** is added to the **READY List**. At that, *`Status`* of *`Packet`* *is* set to **`RC_FAIL`**

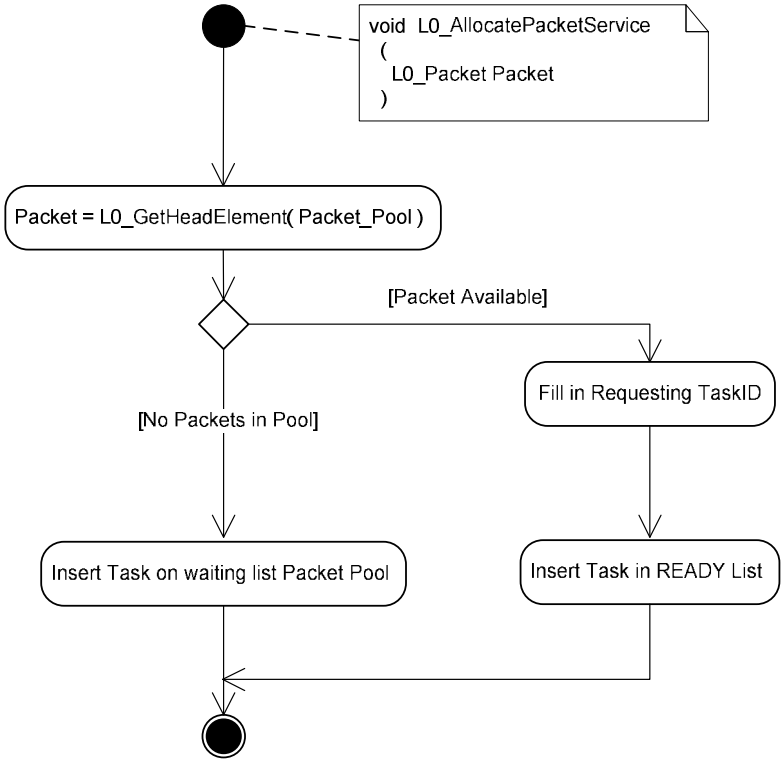


Figure 31. Algorithm of the procedure L0_AllocatePacketservice TBD

5.3.3. void L0_DeallocatePacketService(L0_Packet Packet)

The **Kernel Task** calls this procedure to de-allocate the specified **Packet**.

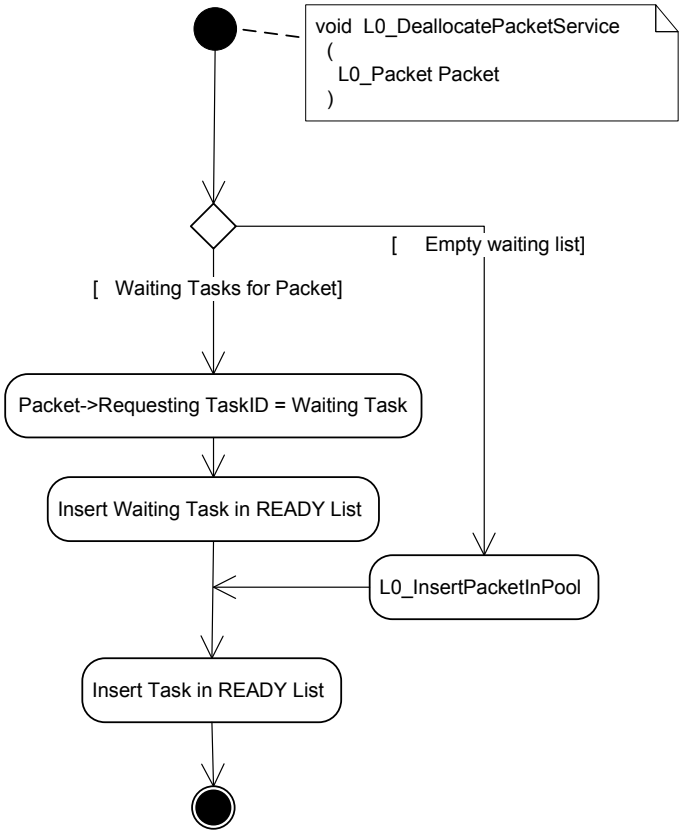


Figure 32. Algorithm of the procedure L0_DeallocatePacketservice

FIGURE TBD

Parameters:

`L0_Packet Packet -` the **Packet** that has to be served.

Return value:
`void`

Pre-conditions:

- The **Task**, owning `Packet`, is not in the **READY List**
- The calling **Task** is the **Kernel Task**

Post-conditions:

- If there is (are) waiting allocation request(s) in the **Packet Pool**, then `Packet` is passed to the highest priority **Task** one
- Next this waiting **Task** is put on the **READY List**.
- The deallocating **Task** is added to the **READY List**. At that `Status` of `Packet` is set to **`L0_RC_OK`**.

5.3.4. void L0_SendPacketService(L0_Packet Packet)

The **Kernel Task** calls this procedure to process a send request

Parameters:

`L0_Packet` *`Packet`* - the **Packet** that has to be send

Return value:

`void`

Pre-conditions:

- The Packet is a send request packet
- The destination Port is local

Post-conditions:

- If the *`DestinationPort`* of *`Packet`* is remote than the **Packet** is inserted into the **Input Port** of the appropriate **Link Driver Task**.
- If there is a complementary **Packet** on the **Waiting List** of the **Port**[1] then *Packet has synchronized with that complementary Packet*.
- In case of the [_W][_T] version:If there is no complementary **Packet** waiting in the **Waiting List** of the **Port** then *`Packet`* *set on* the **Waiting List** of the **Port**.

- In case of the [**Wait**-less/**Timeout**-less] version: TBD.

[1] Local Port or Task Input Port or Driver Input Port

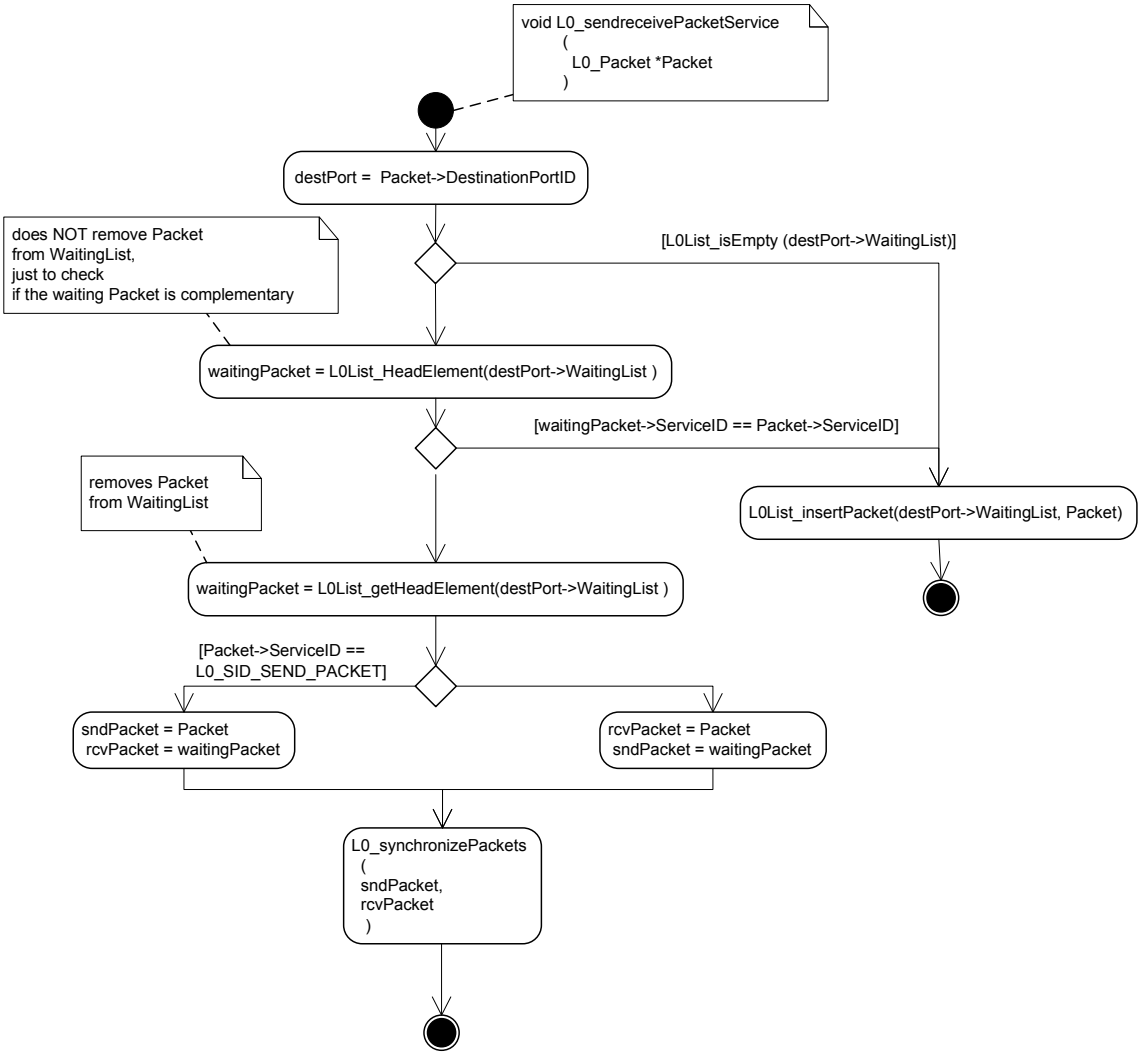


Figure 33. Algorithm of the procedure L0_SendPacketservice

5.3.5. void L0_ReceivePacketService(L0_Packet Packet)

The **Kernel Task** calls this procedure to process the receive request of the specified **Packet**.

See Section 5.3.4.

5.3.6. void L0_ReturnPacketService(L0_Packet Packet)

The **Kernel Task** calls this function to process the return or acknowledgement of a kernel request service and makes the requesting Task READY again..

Parameters:

`L0_Packet` *`Packet`* - the return **Packet** of the requested service

Pre-conditions:

- The **Task** originally requesting the kernel service is not in the **READY List**
- The **Packet** originates from a remote **Kernel Task** and was copied into a Packet allocated by a **Rx Driver.**

Post-conditions:

- The **Task** originally requesting the kernel service is put on the **READY List**
- The **Task** originally requesting the kernel service will return with the status in the **Packet**.header fields
- Header fields and Data are copied into the Task's preallocated Packet
- The **Packet** allocated by the RxDriver is released

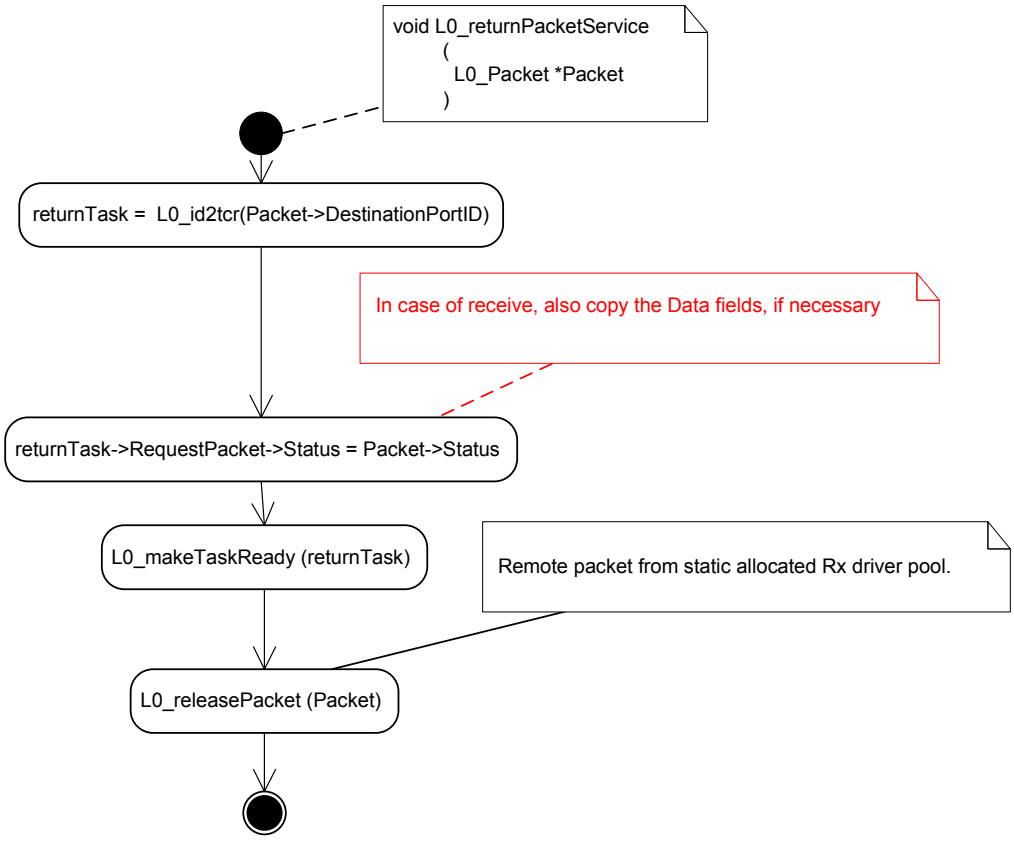


Figure 34. Algorithm of the procedure L0_ReturnPacketService

5.3.7. void L0_SynchronizePackets (L0_Packet SendRequestPacket, L0_Packet ReceiveRequestPacket)

The **Kernel Task** calls this function to synchronise a send and receive request, meanwhile swapping the complementary fields

Parameters:

L0_Packet *SendRequestPacket* - the **Packet** that was sent
L0_Packet *ReceiveRequestPacket* - the **Packet** used to receive a Packet

Pre-conditions:

- The Packets are complementary

Post-conditions:

- Fields are swapped
- The Tasks are made READY when local
- Return Packet(s) sent to remote nodes when Task is remote

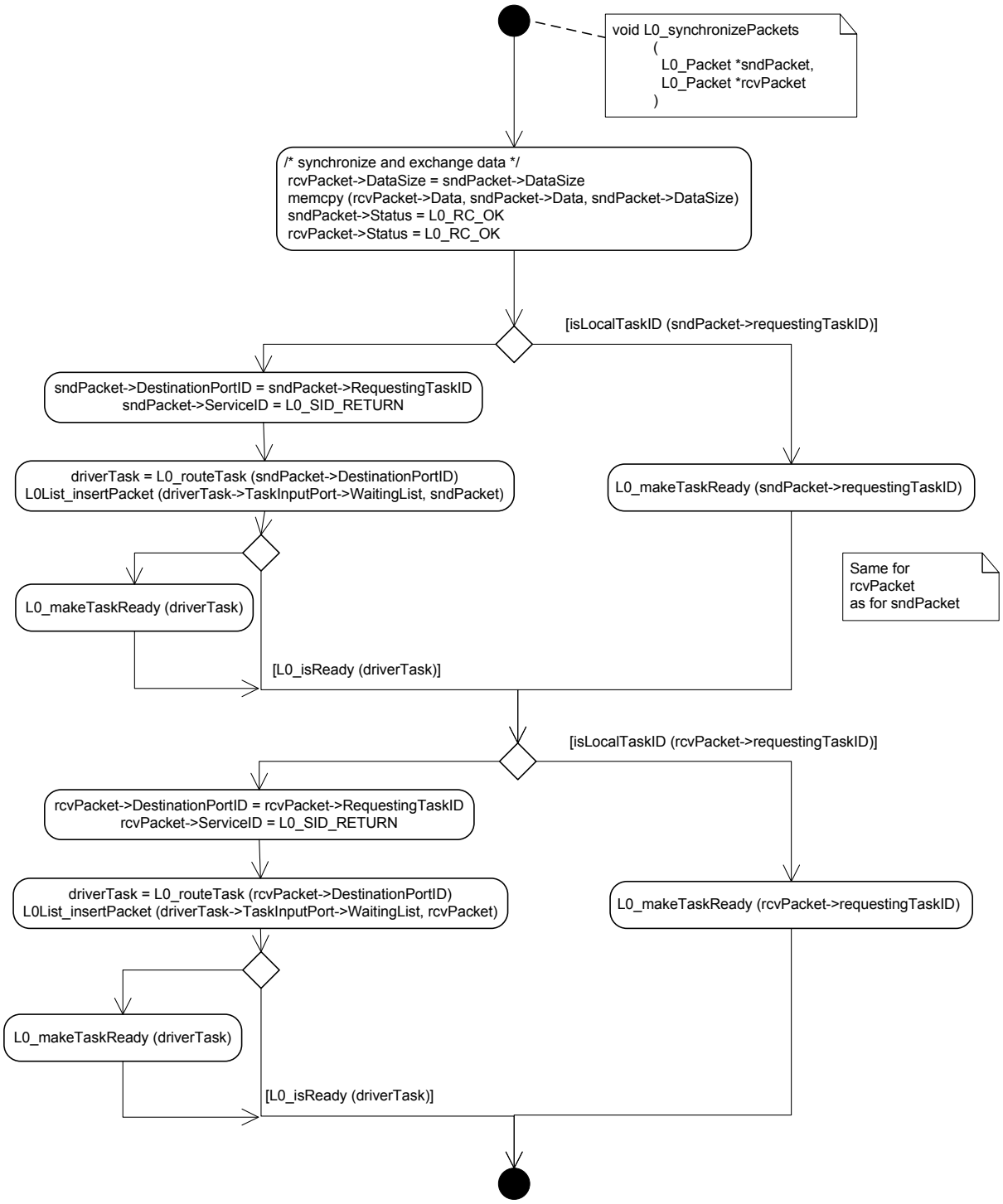


Figure 35. Algorithm of the procedure L0_SynchronizePackets

5.3.8. void L0_MakeTaskReady(L0_TaskControlRecord Task)

The **Kernel Task** calls this function to put the specified **Task** on the **READY List**.

Parameters:
`L0_TaskControlRecord` *`Task`* (the **Task** to be made READY)

Return value:
`void`

Pre-conditions:

- *`Task`* is not in the **READY List**
- The calling **Task** is the **Kernel Task**
- Task is not the **Kernel Task**

Post-conditions:

- *`Task`* is put on the **READY List**.

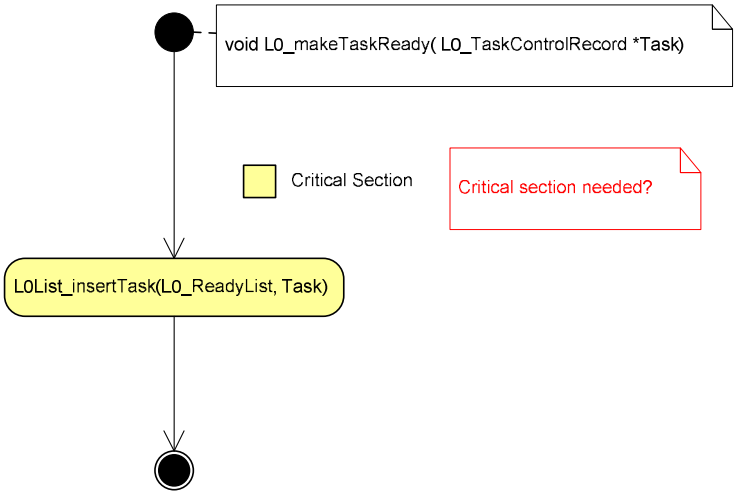


Figure 36. Algorithm of the procedure L0Kernel_MakeTaskReady

5.3.9. void L0_Reschedule (L0_TaskControlRecord Task)

TBD

5.4. Implementation notes

- Remote service handling
- Ready list manipulation
- ISRs
- Rx/Tx driver tasks

TBD

6. Issues

ID of ISSUE	Matter of Issue