VirtuosoNext Application Programmer Interface

1.0.0.0

Contents

Part I

VirtuosoNext Fundamentals

Introduction

This document is intended as a manual that describes the use of VirtuosoNext, a network centric Real-time Operating System (RTOS) for developing embedded real-time applications. However, VirtuosoNext is more than that. VirtuosoNext was developed using formal modelling techniques from the ground up as a coherent runtime and programming system for "networked" embedded systems. It fits within a unified systems engineering methodology based on an "Interacting Entities" architectural paradigm. Almost any system can be developed following this paradigm, but often the tool support will be lacking. Many tools exist and many paradigms exist. The issue for the embedded (software) engineer is that each of these tools and methods have different semantics, making it very hard to combine them and to make sure that no remaining errors exist due to subtle side-effects as a result of the subtle differences in the semantics. This makes using VirtuosoNext particularly interesting for developing high-reliability or safety critical embedded systems. The RTOS kernel and its services were developed using a formal methodology, analyzed to the essential core and as a result VirtuosoNext has several unique properties that can make a big difference when developing embedded applications. We name some of the most important ones:

- Scalability: VirtuosoNext applications can be redeployed, mostly by recompilation of the application source code, from very small single micro controller systems to target systems with a large number of distributed heterogeneous processing Nodes.
- Extensible. VirtuosoNext can be extended with application specific services and entities without the need for the user to develop another middle-ware layer. Such services are integrated at the system level, itself based on a fine-grain microkernel architecture combined with packet switching. New services are integrated using a meta-modeling approach.
- Distributed operation. VirtuosoNext was designed from the start as a network centric runtime system. Whether the application Tasks and the kernel entities are placed on a single processing Nodes or are mapped onto several ones, the user does not need to care about where his Tasks and entities are mapped (except at configuration time). The system itself takes care of the routing and system level communication while the application source code is independent of the network and application topology. We call this a "Virtual Single Processor" runtime model.
- Efficiency. As a result of the formal modelling, the kernel entities and services are very orthogonal and generic. A major consequence is that the code size is very small (about 5 to 10 times smaller than equivalent classical implementations). Small code size also means that less time is spend in executing kernel services resulting in a lower overhead. Code size can be as small as 5 Kbytes while a fully featured distributed implementation only takes up about 20 Kbytes (processor dependent).
- Safety. All kernel services were modelled at the architectural design using a formal model checker. The final implementation was also verified using formal modelling to make sure that the implementation did not introduce potential errors. Thanks to its design based on packet switching, VirtuosoNext has no issue with memory fragmentation and buffer overflow. If it runs out of memory, the system will automatically start throttling allowing allocated resources to be freed up again.
- Hard real-time. In VirtuosoNext every operation inherits the priority of Tasks. Tas receive a priority at compile time and are preemptively scheduled with support for distributed priority inheritance

- Fine-grain partitioning. The Virtuoso kernel can make full use of the MMU and MPU support on the processor, hereby isolating each Task in memory, as well as preventing unauthorised access to memory regions, unles hey are explicitly shared. The overhead for this safety support is very small.
- Fault-tolerance: the fine grain partitioning can be used in conjunction with real-time fault recovery. Processor exceptions can be trapped allowing to recover the Task's state and to have it restarted in microseconds.
- Productivity. To relieve the programmer from tedious code writing (and also to reduce the error rate), VirtuosoNext adopted the principe of automatic code generation. Datastructures, initialisation code and build scripts are automatically generated from a higher level description and metamodels that contain all processor and board specific information.

As one can see, VirtuosoNext is much more than just another RTOS. It is a universal real-time programming system for embedded applications. It is also supported with easy-to-use tools like the Visual development Environment (VirtuosoNext-VE) supporting automatic code generation and visual tracing and debugging.

Scope

The scope of this document is limited to developing VirtuosoNext based applications. No detailed knowledge of its internal functioning is needed.

Chapter 1

General Concepts

1.1 Background of VirtuosoNext

The main purpose of VirtuosoNext is to provide a software runtime environment supporting a coherent and unified systems engineering methodology, based on Interacting Entities,

In VirtuosoNext the dominant active Interacting Entity is a software entity, called a “Task”. Other entities are specific instances of generic “Hubs” and they play an important role in the interactions between the Task entities. All Tasks interact only through Hubs, i.e. there are no direct Task-Task interactions, but specific types of Hubs will provide specific semantics for the kernel services used by the Tasks to interact. As such the basic functionality of a Hub is to synchronise between Tasks. The specific behaviour is determined by the logical predicates that govern the synchronization and by the action predicates that are invoked once a synchronization has taken place. This allowed us to redefine Hub services as the traditional services one finds in other RTOS, e.g. Events, Semaphores, Ports, FIFOs, Resources and Memory Pools. An additional one is a Packet Pool. Another difference is that this allows the user to integrate his own services in the RTOS system and that some services are available as asynchronous services.

A Task will be running on a computing device (CPU + RAM + Peripherals + etc.), called a “Node”.

There may be many Tasks running 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 Interacting Entities using only one Node, every Task virtualising a complete CPU instance.

Besides Tasks, VirtuosoNext provides services and Hub Entities allowing Tasks to synchronise and to exchange data using a specific behaviour for each type of Entity. This behaviour represent the system level interaction from which an application can build higher level Interactions, e.g. like communication protocols that consists of several Put/Get pairs.

VirtuosoNext 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 to communicate transparently across a network of processing Nodes. By design this means that one Node can be part of local network that is connected though internet with another Cluster at the other side of the world. This support for a transparent distributed operation however is an option that does not prevent using VirtuosoNext on a single CPU.

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

As such, VirtuosoNext comes with a PC hosted simulator. This “hostnode” can be integrated in an embedded system just like any fully embedded Node and allows embedded Nodes access to host services in a transparent way.

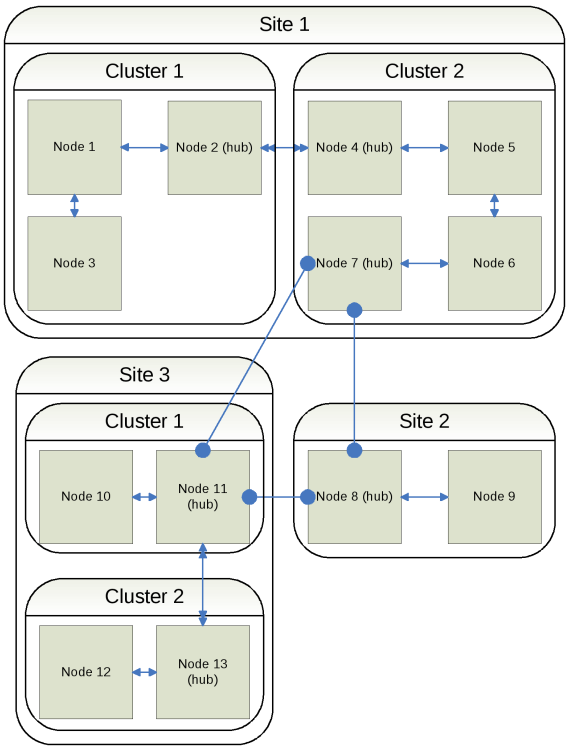

Figure 1.1: Generic structure of a distributed computing system

1.2 Physical structure of the target processing system

Figure 1.1 represents the physical structure of a generic and distributed computing system from the point of view of VirtuosoNext.

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

- Sites, consisting of
- Clusters, consisting of
- Nodes, hosting: Entities (e.g. Tasks, Hubs, . . .)

The Nodes communicate with each other via various physical communication channels (internal bus, IO buses, networks, IO Pipes, etc). There are also 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. From a logical point of view however there is no difference at the application level. Only the timing will differ.

1.3 Layered architecture of VirtuosoNext

VirtuosoNext is being developed using a scalable architecture. Each higher level layer builds on the lower layers and provides a specific functionalities. Given that each layer adds functional behaviour, one should view these layers as semantic layers instead of strictly functional ones. The layering however is still reflected in the use of different system Packets (L0, L1 and L2 with L0 and L1 merged in the implementation).

- L0 — The lowest semantic layer. It provides the basic primitive services, such as Task scheduling, routing of packets and a simple mechanism for intertask synchronization and communication. When there is more than one Node, it also provides an inter-Node communication mechanism.
- L1 — The next semantic layer. It provides flexible Task synchronization and coordination services. This layer can be used to emulate existing third party RTOS. L1 services include the layer L0 services.
- L2 — The highest semantic layer. This layer can support user-defined services, often supporting dynamic behaviour. Given that it may include widely distributed services, the communication delay can become important and the real-time behaviour can become "soft" real-time.

VirtuosoNext operates at the L0 layer by using just Ports and Packets. The Ports are used to exchange Packets between Tasks and synchronise by a Put_ and Get_ pair of service requests. The L1-Packets are atomic units containing a header and a payload zone for application specific data. The kernel mostly operates by shuffling the packets around while updating or using the header field information.

To implement the full L1 layer a generic Hub entity is used. It provides services with different functional behavior ranging from simple Event synchronization to a more complex behavior that includes buffering of data and copying it network wide.

1.4 The logical view of the L1 Layer

The distributed environment, described in the sections above is based on the existence of a fast and unified communication layer. The VirtuosoNext Layer L1 therefore is defined as providing the following functionalities:

1. a Packet-switching communication layer using Inter-node 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 an VirtuosoNext based system on a network of processing node is shown in Figure 1.2. For the application it will look like a Virtual Single Processor.

1.4.1 Principle of synchronization and communication

The distributed environment, described in the sections above is based on the existence of a unified communication layer, independent of the underlying communication protocol or the hardware. In terms of this communication layer, an abstraction of the physical inter-node communication medium is called an 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 links, 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. Note however, that such an Internode Link is not necessarily a physical point-to-point connection. It can be as well a shared memory that all Nodes have access to, or it can even be a virtual connection when e.g. some Nodes are hosted on top of legacy operating systems and VirtuosoNext communication uses "tunneling" (e.g. by calling the native socket communication) to connect the Nodes.

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.

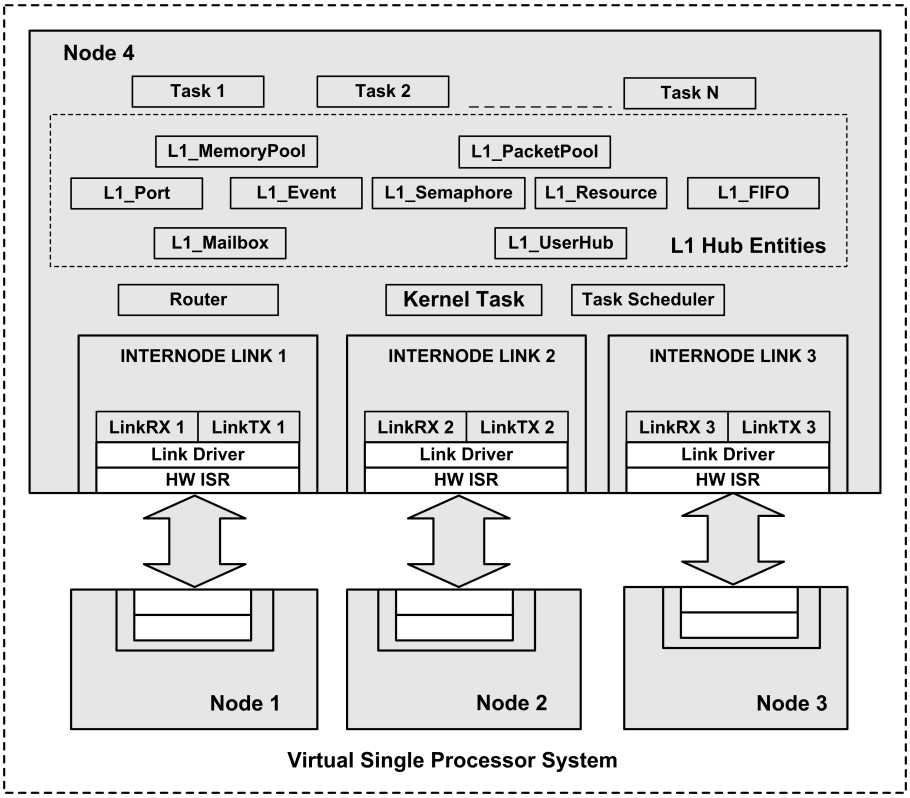

Figure 1.2: Logical structure of a distributed VirtuosoNext system

VirtuosoNext is based on a Packet-switching architecture. This means that Packets of a fixed size (that can be different for each application) are passed from one Entity to another Entity. As Tasks may be located on different Nodes, a 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 an intermediate Entity and are received from an intermediate Entity. This effectively isolates Tasks from each other and increases the scalability of the system. At the application view VirtuosoNext provides services with specific semantics and the underlying Entities and Packets can be "hidden" in the implementation and are 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 VirtuosoNext to reach the destination Node. The router itself is invisible to the application programmer. As all VirtuosoNext services are by default "distributed", the routing is void when routing between local Tasks.

1.4.2 Scheduling Tasks and Task interactions through the RTOS kernel

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

The Kernel is the logical entity that:

1. provides services to the Tasks and

2. also schedules the Tasks according to a real-time scheduling policy.

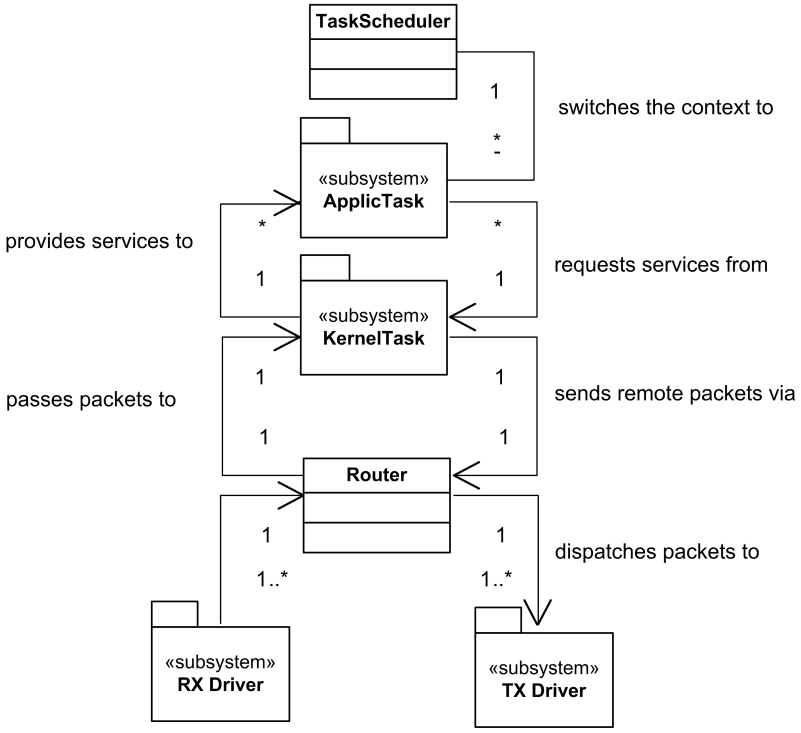

Figure 1.3: Functional relationship between entities of the distributed system

Although the functions are logically separate, in the practical implementation they are intertwined in VirtuosoNext.

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 the remote Node
- 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 Pipes
- LinkTX and LinkRX are provided by the Link hardware, managed by (hardware) specific link drivers and interrupt service routines (Link driver, HW Interrupt Service Routine (ISR))

The relations are represented in Figure 1.3.

1.5 Inter-Task interaction

An inter-task interaction consists of two parts: putting a Packet to a Hub and getting a Packet from the same Hub. These Packets are actually carriers for service requests and will be invisible to the programmer (except when using asynchronous services). 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 but more complex semantics are possible as well. Note however, that at the level of L1, this is an issue for the application code running in the Tasks. From a point of view of the Kernel and the Hub, just a Packet has been interchanged although in the implementation, just the relevant header fields and databytes are copied from one Packet to another.

A Port provides a minimum but complete functionality that includes synchronization and communication. It is really an instance of a more generic mechanism that was called a Hub. The Hub entity also provides services like Events, Semaphores, FIFO queues, Ports, Resources, Data Events, Memory Block Queues, Black Boards, and Memory Pools. The semantic differences are mainly determined by the actions associated upon synchronization. We call these actions the “Synchronising Predicate” and the action that results from it, i.e. the requested service, the “Action Predicate”.

The L1 Entities can be classified in groups as shown in Table 1.1.

Table 1.1: Interactions between Tasks and Hubs.

Hub type	**Request type**	**Guard**	**Action**
Port	Put	Waiting Get request	Both Task rescheduled, Packet exchanged
Port	Put	No waiting Get request	Task enters WAIT state
Port	Get	Waiting Put request	Both Tasks rescheduled, Packet exchanged
Port	Get	No waiting Put request	Task enters WAIT state
Event	Put	Event = FALSE	Event = TRUE, Task rescheduled
Event	Put	Event = TRUE	Task enters WAIT state
Event	Get	Event = TRUE	Event = FALSE, Task rescheduled,
Event	Get	Event = FALSE	Task enters WAIT state
Semaphore	Signal	Semaphore count < MAXINT	Semaphore incremented, Task rescheduled
Semaphore	Signal	Semaphore count = MAXINT	Task enters WAIT state
Semaphore	Get	Semaphore count > 0	Semaphore decremented, Task rescheduled
Semaphore	Get	Semaphore count = MAXINT	Task enters WAIT state
Resource	Lock	Resource has no owner Task	Task becomes owner,
			Task rescheduled
Resource	Lock	Resource has owner Task	Task enters WAIT state, priority inheritance applied
Resource	Unlock	Resource has no owner Task	Task rescheduled, return code RC_FAIL
Resource	Unlock	Resource has owner Task	Task rescheduled, return code RC_FAIL if owner Task different from self
FIFO	Enqueue	Count FIF0 entries between 1 and maximum	Task reschedules, data enqueued
FIFO	Enqueue	Count FIF0 entries = maximum	Task enters WAIT state
FIFO	Dequeue	Count FIF0 entries between 1 and maximum	Task reschedules, data dequeued
FIFO	Dequeue	Count FIF0 entries = zero	Task enter WAIT state
Packet Pool	Get	Packet available	Task reschedules, Packet removed from Pool
Packet Pool	Get	No Packet available	Task enters WAIT state

Continued on next page

Table 1.1 – continued from previous page

Hub type	Request type	Guard	Action
Packet Pool	Put		Task reschedules, Packet returned to Pool
Memory Pool	Get	Memory block available	Task reschedules, block removed from Pool
Memory Pool	Get	Memory block unavailable	Task enters WAIT state
Memory Pool	Put	Memory block available	Task reschedules, block returned to Pool
Data Event	Put	Event = FALSE	Event = TRUE, Data copied to Hub, Task rescheduled
Data Event	Put	Event = TRUE	Event = TRUE, Data copied to Hub, Task rescheduled
Data Event	Get	Event = TRUE	Event = FALSE, Data copied to Task, Task rescheduled
Data Event	Get	Event = FALSE	Task enters WAIT state
Memory Block Queue	Put	FreeBlocks > 0	FreeBlocks decremented, FullBlocks incremented, Task rescheduled
Memory Block Queue	Put	FreeBlocks = 0	Task enters WAIT state
Memory Block Queue	Get	FullBlock > 0	FreeBlocks incremented, FullBlocks decremented, Task rescheduled
Memory Block Queue	Get	FullBlocks = 0	Task enters WAIT state
Black Board	Put	MessageSize = 0	MessageSize = Task MessageSize, Data copied to Hub, Task rescheduled
Black Board	Put	MessageSize > 0	MessageSize = Task MessageSize, Data copied to Hub, Task rescheduled
Black Board	Get	MessageSize > 0	Data copied to Task, Task rescheduled
Black Board	Get	MessageSize = 0	Task enters WAIT state

In general, we can see that the generic mechanism is one of interaction between a Task that makes something available (the “put” operation) and a Task that wants to “get” it. Both are requesting the service through an intermediate Hub Entity via the kernel Task. In the context of common language used for such services, the “Put” operation can be called a “put”, “enqueue”, “insert”, “release”, “raise”, “free”, etc with the “Get” operation can be called “wait”, “get”, “lock”, “dequeue”, “read”, “allocate”, etc. In all cases one side of the interaction will make “something“available on which the other side can wait. For some services no explicit synchronization is needed while for some services two steps are needed. One in which both sides synchronise and the ‘something” is made available (e.g. with reservation in a waiting list) with a second step during which the “something” is actually obtained. The actual transfer from one side to another is governed by a Synchronising Predicate filter operation that is specific for the type of service and interaction Entity. If a data transfer and buffering is involved, it is to be seen as a side-effect of the synchronization performed by the matching filter. Figure 1.5 shows the available Hub types in VirtuosoNext 1.6.

In most cases the put request is performed by one Task while the get request is performed by another Task.

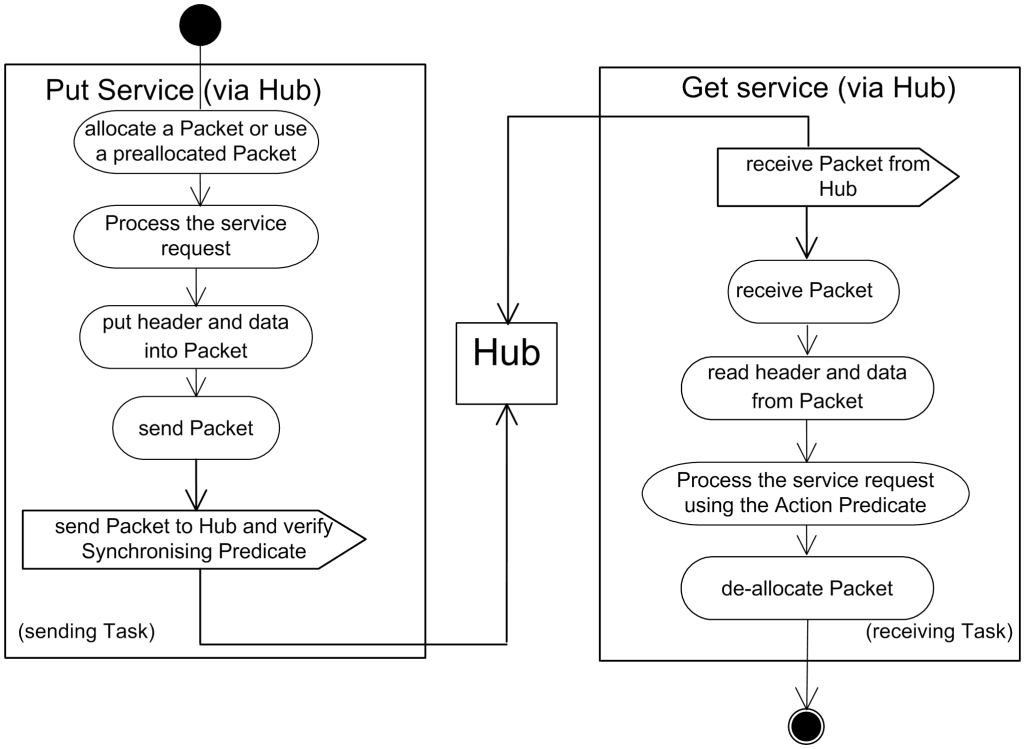

Figure 1.4: Generic scenario of a service request using the Hub entity

However, as the interaction is through Hubs, it can as well be that e.g. driver Tasks or hardware specific ISR put a Packet in a Hub. However, while an ISR can insert a Packet into a Hub on which a driver Task could wait to Get from, no ISR should attempt to Get a Packet from a Hub. The reason is that ISRs are not allowed to wait (polling is just burning cycles and monopolises the CPU when done inside an ISR) while in such a set-up no other Task can ever insert a packet as the ISR will monopolise the CPU. If an ISR needs to Get data it should get this data from an associated Driver Task that itself can wait it to Get from a Hub.

The general concept of a generic Hub is illustrated again in Figure 1.6

As VirtuosoNext supports distributed systems, by default, the interacting Tasks and Hubs can be located on different Nodes. For example, the Puting Task can be located on Node A, the receiving Task can be located on Node B and the Hub can be located on Node C. The data associated with such an interaction can even be located on still other Nodes as memory pools are also distributed. It is even possible to accept an interrupt on one node, passing it on via the network to another node and having the interrupt being processed on that other node.

1.6 Application specific services

Although not part of this manual, VirtuosoNext Hubs and their associated services can be customized in an application specific way without requiring a rebuild of the kernel. The developer needs to specify the synchronization predicate function and predicate function as well as the associated Hub states. The sytem generator needs to be adapted as well. This capability is decribed in the RTOS extension and porting kit. On the application level this approach has many advantages. First of all, it provides for more safety and scalability than with a traditionally designed RTOS. It also provides more performance as it avoids the need to write a middleware layer, often on top of the underlying OS and requiring often multiple service invocations to achieve the desired behaviour. Hence VirtuosoNext can be adapted to become another

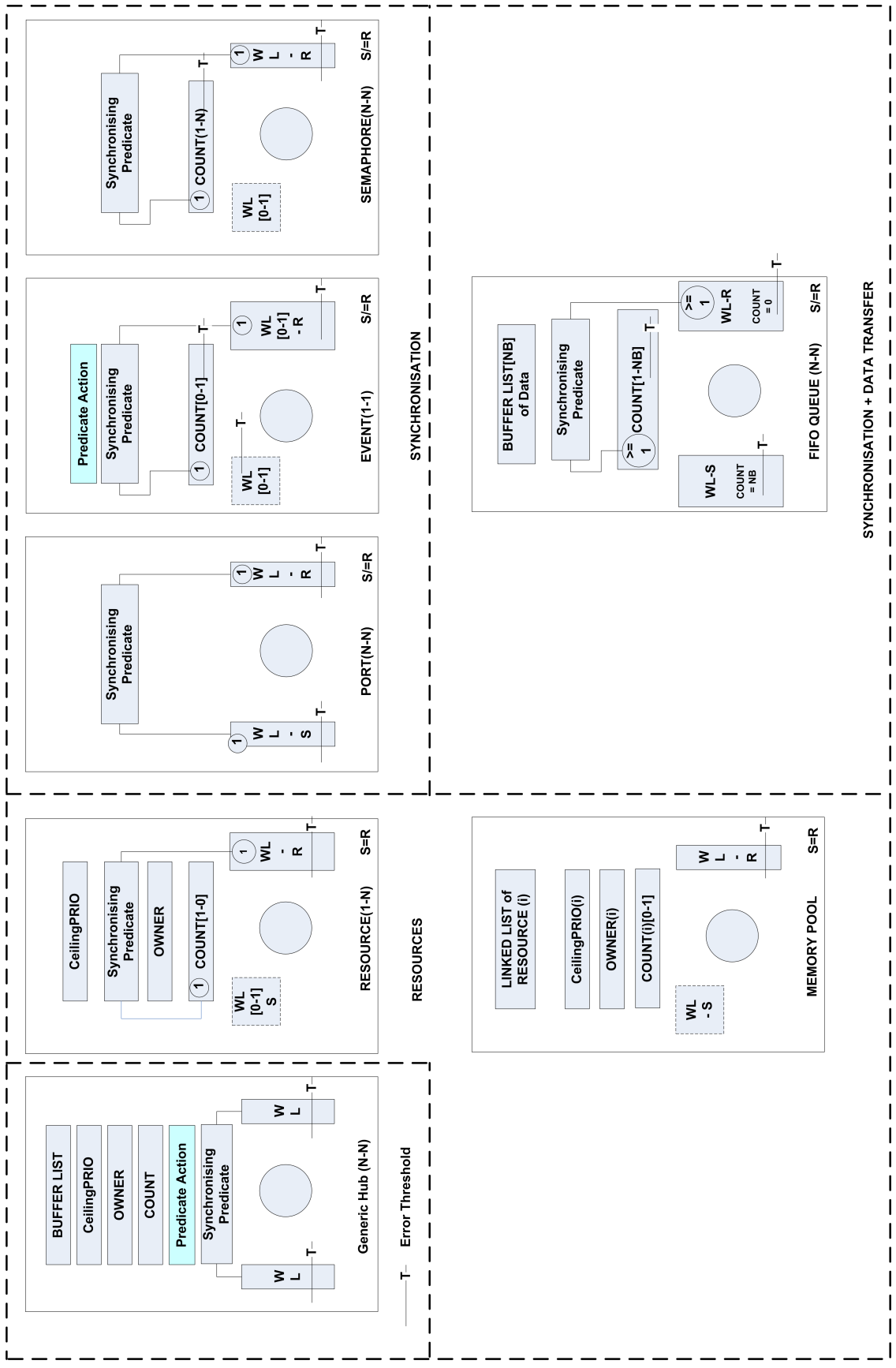

Figure 1.5: Graphical representation of the different Hub-types

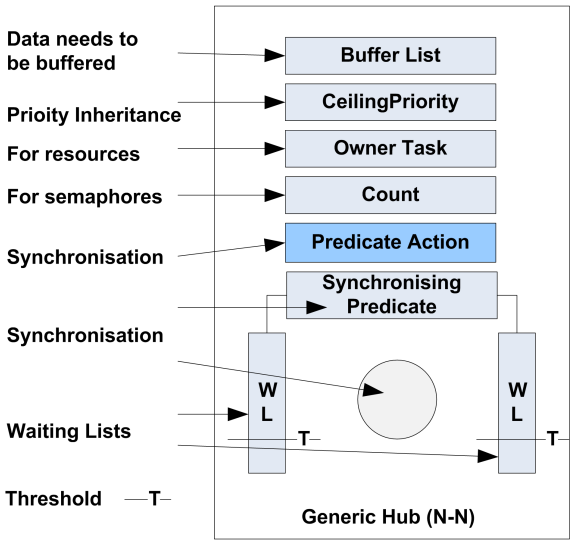

Figure 1.6: General conept of the generic Hub

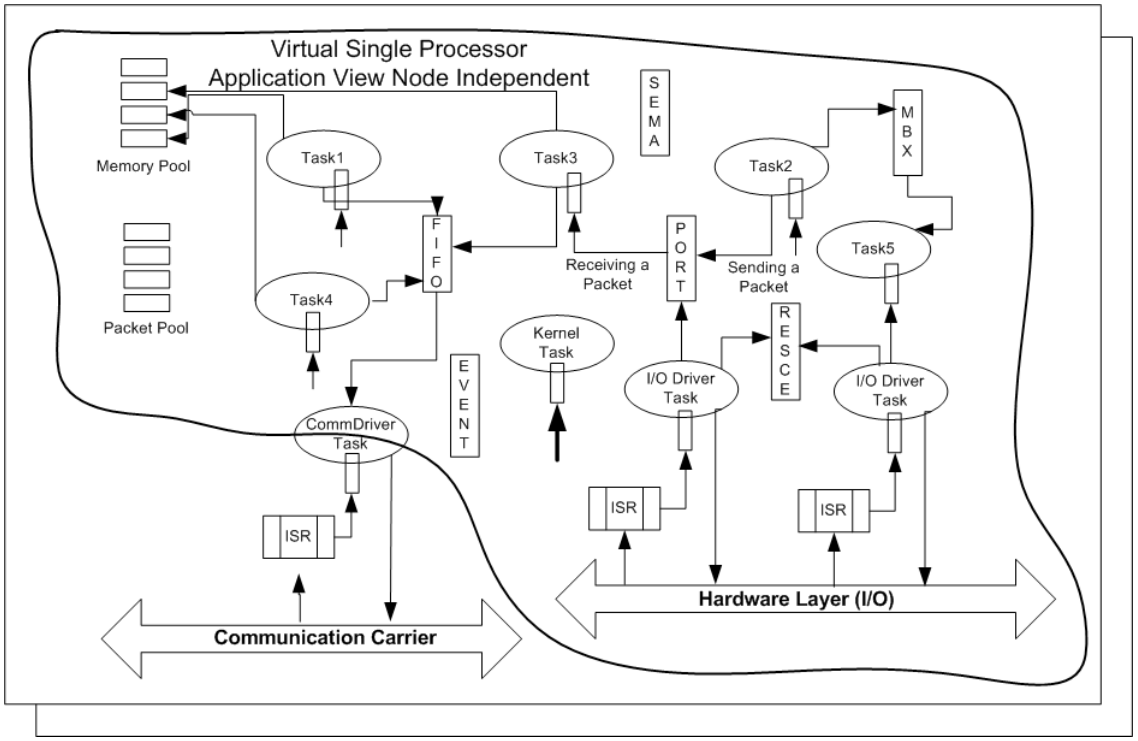

Figure 1.7: Possible distribution of Entities, involved in Task Interaction

RTOS as well, although the semantics might need some tweaking as most RTOS cannot support distributed environments (e.g. because they pass pointers to local memory in the service calls).

1.7 A new concurrent programming paradigm

The fact that one can create his own services, all based on a universal and generic "Hub" entity, makes that VirtuosoNext is much more than a network-centric RTOS. The concept of Tasks and Hubs embodies the fundamental concepts one needs to write concurrent programs, whether the target is a single processor system, a multicore systems, a parallel processing system or a widely distributed and loosely coupled system. This universal character provides for a natural way of programming such systems. Programming is in essence an activity whereby a model of a system is developed. Most systems (technical as well as non-technical ones) are naturally described as a set of Interacting Entities. In VirtuosoNext, the main Entities are Tasks and Hubs and the services they provide are the interactions. Interactions can be quite complex, but most interactions while consist of a synchronization point, guarded by a logical condition. When synchronisation has taken place, in a second step the real desired interaction will happen. E.g. it can allow a waiting entity to resume its operation, or information and/or data can be transferred or an action will be executed that acts on the external world (e.g. a motor is started). In VirtuosoNext this interaction behaviour is neatly separated and hence it is functionally scalable. Hubs are also separated from the Tasks, allowing scalability across networked Nodes.

One could argue that this type of concurrent programming is not really new. Indeed, a predecessor product (called Virtuoso at the time) allowed a similar programming style, but it was practically impossible to add new services. Virtuoso itself had found its inspiration in CSP (Communicating Sequential Processes), a process algebra thought out by C.A.R. Hoare. In CSP, Processes interact only synchronously through unidirectional channels. When they do, data can be passed from one process to the other and both processes can continue. In Virtuoso as well as in VirtuosoNext, this behaviour was externalized, as well as more complex semantics are supported. The Hubs are also independent of the Tasks, whereas in CSP the channels are tightly coupled between the processes. Hence, we could argue that VirtuosoNext are a pragmatic superset of CSP. Although the VirtuosoNext semantics were formally modelled and verified, we claim less rigour than with the strict semantics of CSP. The benefits obtained are a higher usability for real-world programming and more abstraction from the underlying implementation.

1.8 Inter-Node interaction

VirtuosoNext 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 the L1 interaction Entities. The link hardware layer may implement the communication very differently from one Platform to another.

While in VirtuosoNext Tasks can interface directly with the hardware via Interrupt Service Routines, most often driver Tasks will implement the higher level functionality the hardware interfaces . 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. VirtuosoNext defines dedicated Tasks, called Link Driver Tasks, that implements the VirtuosoNext 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

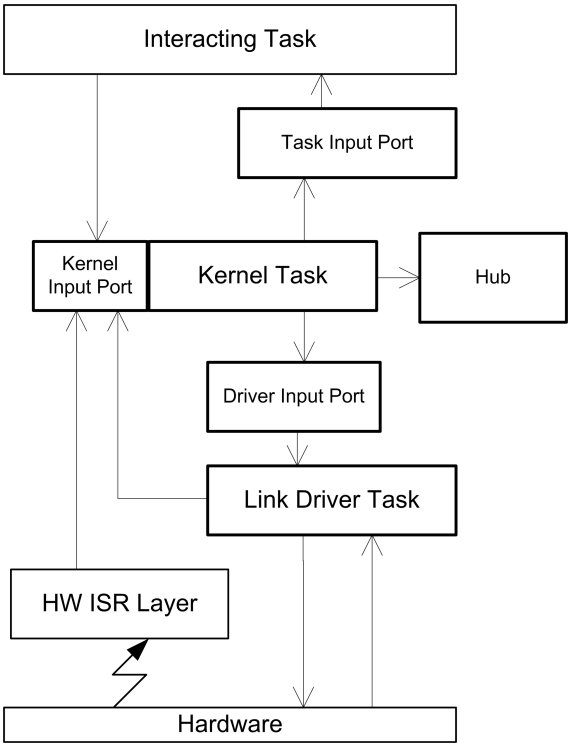

Figure 1.8: Interactions between HW, ISR Layer, Driver Task and application Tasks.

- 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 Tasks, Link Driver Tasks and ISR layer interact with each other ONLY via the Kernel Task.

The interaction scheme is illustrated in Figure 1.8.

Chapter 2

Functional Design of the L1 Layer

Figure 2.1 presents the functional model of the VirtuosoNext Layer L1, it is not a complete description of all the available services.

2.1 Task interactions

Entities that interact are a synchronizations and communications between the Tasks via intermediate Entities (e.g. Ports, Events, Semaphores, FIFOs, Hubs, etc.). To simplify the terminology, we call these Task Interactions. All these Entities can be derived from a common generic Entity (at least conceptually) that we called a “Hub”. Such a Hub provides first of all “synchronization” between “Puting”and “Receiving” Tasks. Synchronisation happens through the use of a “matching filter” we called the Synchronisation Predicate. It verifies that the conditions for synchronization are fulfilled resulting in e.g. a Task becoming ready again. From the application point of view, one can consider that during the synchronization a “resource” is made available from one Task to another, allowing the latter to continue when it gets the resource. The resource itself can be the notification that a certain event has happened, a piece of data or e.g. a logical entity that needs to be protected for atomic access.

Once synchronization has happened, the system will call the interaction specific Action Predicate. E.g. making a Task ready again, returning data or e.g. copying data from one memory area to another one. In general, one can imagine that a Hub can be used for very application specific interactions. An example would be that an alarm signal would be monitored but when a threshold level is reached a command is directly Put to an actuator to shut down a critical part of the application. This can be done without a middle-ware layer resulting in much faster reaction time? Because most of the code is system code, the risk for errors is also lower.

Hubs are used as synchronisation Entities between Tasks and operate by use of Packets sent and get by Tasks. These packets are most fo the time pre-allocated Task Packets and hence hidden in the API. Only for asynchronous services do we have to make the Packets explicit as multiple synchronization can be pending and hence a packet must be used that comes from a general Packet pool. Hence, Hubs also decouple Tasks when interacting and they can be located physically on different Nodes than the interacting Tasks. As a result, Tasks are isolated from each other while this mechanism is inherently scalable and topology independent.

2.1.1 Logical view of Task

In VirtuosoNext, 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 defined unique identifier in the distributed

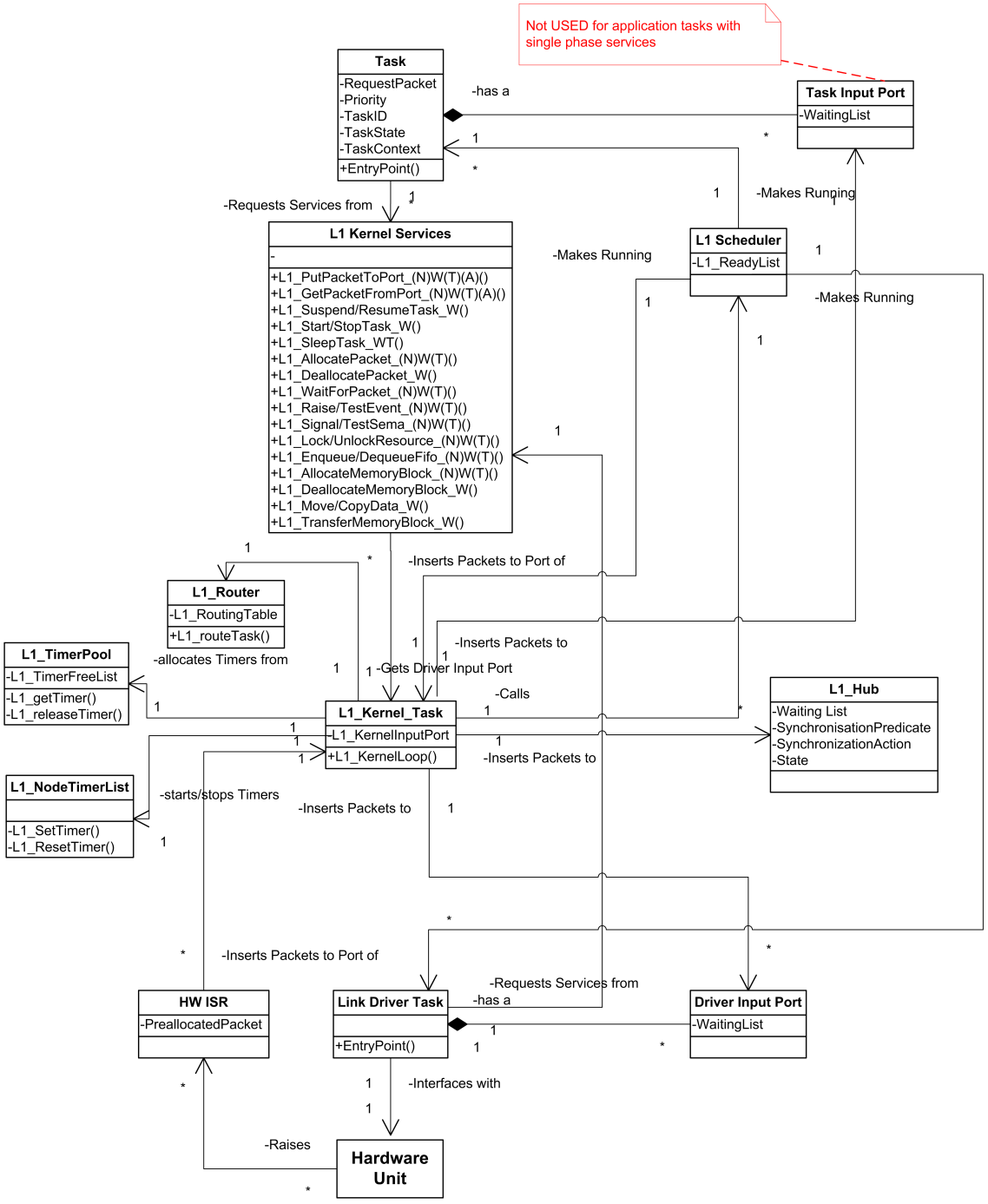

Figure 2.1: Functional model of VirtuosoNext

system. A Task is therefore defined as:

- A Task is a uniquely identified functional resource. It has its own context and can be considered as an independent unit of execution.
- A Task can issue service requests. 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 logical 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 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, the CPU and its 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 the I/O registers. The CPU Context is specific to the hardware (CPU + peripheral units, e.g. state information).

On any traditional CPU, only one Task can execute at a given time on a given Node. This is not a restriction of VirtuosoNext but the result of the von Neumann architecture of most CPUs. 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 Putting 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 get Packet may be "empty" (i.e. pure synchronization without data communication exchange).

A Task starts by being started from another Task or during kernel initialization. It may have finished, which is called STOPPED

Hence, a Task is further defined by its state. It is an operating resource that is always in only one of the following states, managed by VirtuosoNext:

- INACTIVE (the initial state)
- RUNNING (the Task is running on the CPU)
- WAITING (for a service request to complete)
- READY (to run and hence waiting to run in de ready list)
- SUSPENDED (orthogonal state to prevent the Task from running)
- STOPPED (used before a Task is reinitialized)

Note that the normal states in operation are RUNNING, WAITING, READY and STOPPED. The first three ones are sometimes collectively referred to as "ACTIVE". 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 issued by by another 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 preventing a Task from doing any potential harm.

Note also that stopping a Task is a much more drastic operation as this will also destroy the whole Task context and all information will be lost. Therefore precautions are needed to stop a Task in a correct way. This is typically achieved by calling an abort handling function before a new Task context is created.

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 its Priority

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, VirtuosoNext allows Tasks to start with a list of Task specific arguments. The functional code of the Task must be reentrant as well.

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, This Port is only accessed by through and by the Kernel.

2.1.2 Logical view of Packets

In VirtuosoNext, 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 for a given network and is defined at system generation time. The real size of the interchanged data in the Packet can not be greater than this size but the system can use multiple Packets to execute larger data transfers. A Packet contains so called header information that includes a number of 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 embedded interchanged data.

Hence, 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 at most be equal to the Packet Size minus the size of Header. The user is warned that the system will only copy the data in the payload section after synchronization in a Hub when this is part of the semantics of the service. E.g. with an Event no data will be copied, but with a Port data will be exchanged limited by the datasize parameter.

The basis of VirtuosoNext is the L1_Packet, with an application specific defined size. Such a Packet is sufficient to implement the L1_services like Task scheduling and Puting and receiving Packets to and from a Hub.

In the case of all "single-phase" services, these Packets are statically allocated at compile time. For some services, i.e. the "two-phase" services, the calling Task needs to use a dynamically allocated Packet. This Packet is allocated first from a Packet Pool that is managed by the local Kernel Task on the node. For more explanations on these single-phase and two-phase services see the service descriptions further in this document.

NOTE: In the text often the terms Put_ or Get_Request_Packet will be used. Often, this is still the same physical Packet but whose function is changed by an update of its header fields depending on the status of its processing.

2.1.3 Logical view of the generic L1 Hubs

When requesting a L1 kernel service, VirtuosoNext implements it by Puting a Packet to the specified entity called the L1 Hub. 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 Task assures that only one Task can write to the Packet at a given time. Similarly, when receiving a Packet, a Task Gets it from the specified Hub. The Packet having been delivered to the Hub by a Putting Task. Hence, a Hub is defined as follows: "A Hub is an identifiable entity with a globally unique identifier in the distributed system."

The purpose of a Hub is defined as follows:

- A Hub is an entity used to provide services between interacting Tasks, i.e. the Hub will implement the interaction. At the kernel level this behaviour is achieved by interchanging Packets between interacting Tasks through the Hub.
- The synchronization, eventually data exchange, is handled by the Kernel and depends on the specific behaviour defined by the packet header fields that are specific to the service request.

If a Task Puts a Service Request to a Hub, and no other Tasks have yet supplied a matching service request-Packet to that Hub, then the requesting Task will wait until such matching request Packet arrives at the Hub. This will be detected by the matching filter. Note that any number of Tasks (more than one) may Put service requests (i.e. Packets) to the same Hub at any time. Note, that this behavior is symmetric, although the behavior is often specified in terms of "Putting" Tasks en "receiving" Tasks. Hence, in the general case there will be waiting lists on both sides of a Hub.

"A Hub is an entity that buffers the service requests using Packets until synchronisation occurs."

The sent and get service requests are "buffered" in a Hub by means of a Priority-sorted list of Packets. The Priority of an element in the list is inherited from the requesting Task.

Above paragraphs explained the basic functionality of a Hub: synchronization between Tasks, making resources available and Tasks requesting resources all using Packets. Such a Hub has also some attributes, often filled in at runtime, that provide the service specific semantics. E.g. a counter can keep track of the number of Put or Get requests, the Hub can have an owner Task when used to provide atomic access and a Ceiling Priority can be associated with the Hub to provide support for Priority inheritance algorithms in the Task scheduler. It is also possible that some Hubs use buffers where requests or data are kept awaiting the synchronization to happen. Finally, after the synchronization often a callback function (the action predicate) will be called. This function can e.g. copy the data associated with the specific service after synchronization has happened. The Synchronising Predicate and the Action Predicate also enable to define new application specific services without the need to reimplement the basic Hub functionality. E.g. the user could for example define a Hub called an "AlarmWatcher". Driver Tasks could the Put sensor reading on a regular basis to this AlarmWatcher. The AlarmWacher then compares the sensor values with a pre-defined threshold value and when the threshold is surpassed, it activates an "Alarm Raising function" e.g. to disable the actuator driver Task.

According to the above mentioned relationships between Tasks, Hubs and data Packets. Note that while two waiting lists are indicated, for some classes of services (e.g. Events, semaphores, resources) only one of them will be used. The State attribute is dependent on the Hub Type and will contain information such as Owner Task, Ceiling Priority, Event flag, Semaphore count, and the Fifo buffer count. The Synchronization Predicate is a logical function that checks that synchronization can happen. The Synchronization Action

is a function to update the State when synchronization happens and to initiate the required action. The Synchronization Predicate and Synchronization Action are both dependent on the Hub Type, i.e. L1 service class.

2.1.4 On scheduling for real-time

One of the attributes of a Task is its Priority, defined to meet the application's timing requirements. The Priority will be defined by e.g. using a Rate Monotonic Analysis algorithm. In the "normal case" behaviour, this Priority attribute is used to sort in order of Priority all waiting lists, inclusive the lists of Tasks that are ready to run. Often this is the result of a service request that was fulfilled. However, it is not unlikely that while a Task is put in the ready list, another Task of a higher Priority is also requesting the same service (or resource). If this resource is unique (e.g. an Event was raised on which both Tasks are waiting) then the resource should be granted to the highest Priority one at the moment this Task became active (and not to the Task that was first inserted in the ready list). Hence, VirtuosoNext waiting lists are sorted in order of Priority.

Once the requesting Task become ready again, it is inserted on the ready list waiting to become active. If in the mean time a higher Priority Task also requests the same resource, it will be blocked by the lower Priority Task to which the resource was already granted. In VirtuosoNext the solution for this problem is achieved by decoupling the granting of the resource and the resource becoming available.. While the waiting Task is removed from the waiting list and inserted in the ready list, the resource is only 'reserved'. When this Task reaches the head of the ready list, a check is made to verify if it was still the Task with the highest Priority that was waiting for the resource. If not, the resource is granted to the other Task. These issues are applicable to all services, but in practice this is mostly an issue for resource related services. In VirtuosoNext, support for this functionality is provided using an application specific service.

Another issue is that once a Task owns a resource, it prevents other Tasks of higher Priority from receiving the resource. This is called "blocking" and the problem is called the 'Priority Inversion' problem. Given that a Task with an intermediate Priority can then start running, the lower Priority Tasks can block a highest Priority Task for an indeterminate period of time. This problem is created by the need for atomic access in the application, and while atomic access cannot be avoided, the blocking time can be minimized. This is achieved by raising the Priority of the lower Priority Task to the Priority of the waiting Task, reducing the blocking time. Often this Priority will be limited to a Ceiling Priority. The issue is also complicated by the fact that a Task can issue nested requests, i.e. requesting a new resource while already locking a granted resource. These issues are applicable to Resources, Memory Maps and Memory Pools but are in practice only implemented for resources as they define unique critical sections.

NOTE: When locking a Resource, the Task may block other Tasks requesting this Resource later. Hence, this time should be kept as short as possible. For this reason, it is assumed that while a Task locks a Resource it will not request any other service that can result in a waiting condition as this could result in long series of dependencies with no control over the real-time behavior. For the same reason a Task should not be stopped when owning resources. The Kernel cannot prevent such situations, so it is left to good programming practice.

2.1.5 On Timers

VirtuosoNext also maintains a Timer List. This is a List sorted on a Timer value holding events that need to happen in the future. When the event happens (its Timer value becomes a past event versus the actual Time), the Event is enabled and a typical action will be to insert a Packet into the Kernel Task Input Port. A typical event is a TimeOut related to a service request. Timer Events can be inserted into the Timer List as well as removed from the Timer List. Timers can also be used to implement Timer based scheduling.

2.1.6 On runtime errors

VirtuosoNext adopts a generic mechanism for handling runtime errors. No distinction is made between kernel errors and application errors. It is also possible that the error signal is to be seen as a warning, e.g. when a semaphore count reaches a threshold value to prevent forthcoming issues. When an error is raised, the kernel will insert an error package with all relevant into the input port of an error handling Task. This Task should run at the highest Priority one of all application Tasks on a given node. The application developer must define the actions to be taken when such an error is raised.

The occurence of a processor exception (memory violation, numerical error) causes the currently active Task to be aborted, and then restarted. It is possible to install an Abort Handler which gets invoked when a Task gets aborted, which then has the possiblity to perform cleanup / recovery operations.

2.1.7 Logical view of the Packet Pool

Every Task has a pre-allocated Packet that can be used for single phase interactions between Tasks. In order to allow two-phase interactions the Task has to allocate extra Packets from a Packet Pool that is located on its local Node (see 2.3.1 on page 25). In reality, this Packet Pool is also a Hub with a specific field that allows the kernel service to allocate or deallocate a packet from the Packet Pool. In this case, all Packets will be L1 Packets. Note that the same mechanism also supports different types of Packet Pools. E.g. the Packets can have a user defined size and are arranged in an array or they have a variable size. In these cases the ActionPredicate will be different and service specific names are just, e.g. MemoryArray or MemoryPool.

After a Task has get 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 available 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-sorted list. This is actually a list of pre-allocated packets used by VirtuosoNext to implement the service requests. The Priority of an element in the list is inherited from the requesting Task.

2.2 Inter-node interactions

2.2.1 Logical view of Link Drivers and inter-node interactions

VirtuosoNext implements Inter-node Links (see Section 1.4) using the relationship between a interacting 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 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:

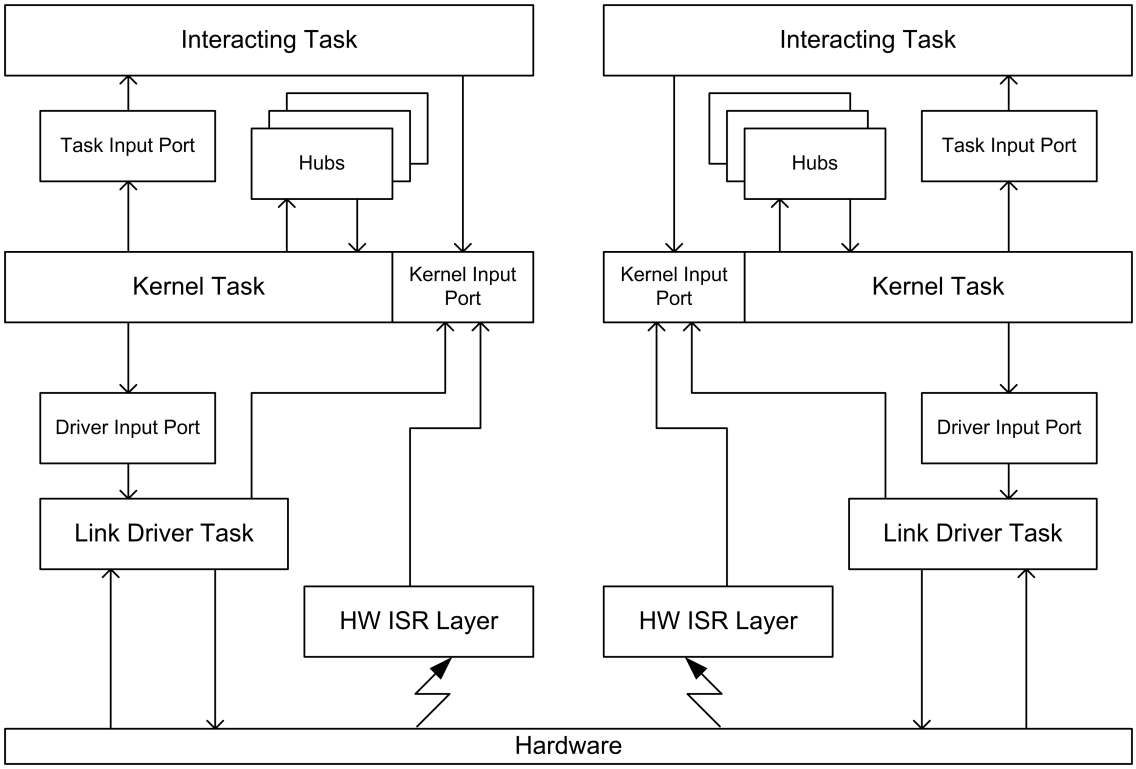

Figure 2.2: Communication between Inter-node Links and Tasks

- 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 Figure 2.2.

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, VirtuosoNext 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 service that uses a *remote* Hub as synchronising entity, the following sequence of actions is performed. Note that we illustrate this mechanism using the exchange of a Packet, but the same mechanism is used for all L1 services:

- `L1_PutPacket_W (Put_Request_Packet, Remote Hub)` or vice versa
- `L1_GetPacket_W (Get_Request_Packet, Remote_Hub)`

These functions will in the context of the Task update the Header of the Packet to be sent to a Hub and insert it in the Kernel Input Port. The Kernel will call the Router function to forward the request Packet to the Remote Hub using a local TX Driver Input Port. The Driver Task then forwards the Packet to the destination Node by the lower level LinkTX driver 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 two inter-node Links of the same Node are used to pass a Packet from one remote Hub to another (so-called through-routing), then only one operation is performed by the Link Driver Task that has Getd 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 LinkTX driver Task

2.2.2 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:

- Puting a Packet to a remote Hub
- Receiving a Packet from a remote Hub
- Forwarding a Packet from a neighbouring node to another neighbouring node

Note that there are no global routes calculated. Routing is based on a local routing table that tells the communication layer which communication port (TX-Link) to use to transmit a packet. The next step in the routing of the packet is done on the next receiving node.

2.3 Multi-tasking

As defined in Section 1.1, multiple Tasks may run on a single Node but only one Task can execute at a given time on a given Node.

2.3.1 Definition of multi-tasking

Multi-tasking as provided by VirtuosoNext, is defined as follows:

- 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 scheduling among equal Priority Tasks that are ready to run. Time-slicing, when enabled can only happen between Tasks of equal Priority.

2.3.2 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 with 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.

2.3.3 Logical view of the Kernel

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

“The Kernel of VirtuosoNext is a dedicated Task that serves the 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.”

VirtuosoNext 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 template algorithm describing how a Task requests a service from the Kernel is as follows:

1. Having passed a request to the Kernel, a Task goes into the waiting state, resulting in switching the context to the Task with the highest Priority among the Tasks that are READY to run.
2. The Kernel Task has a Priority higher than any other Task (incl. Link Driver Tasks).
3. 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, VirtuosoNext defines the following restriction:

- The ISR LAYER is only allowed to Put 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.
- It is possible to have another Task Get the return code from the ISR issued service (e.g. typically used by a Driver or monitoring Task).

Running as a Task, the Kernel performs the following sequence of operations in a loop. 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.

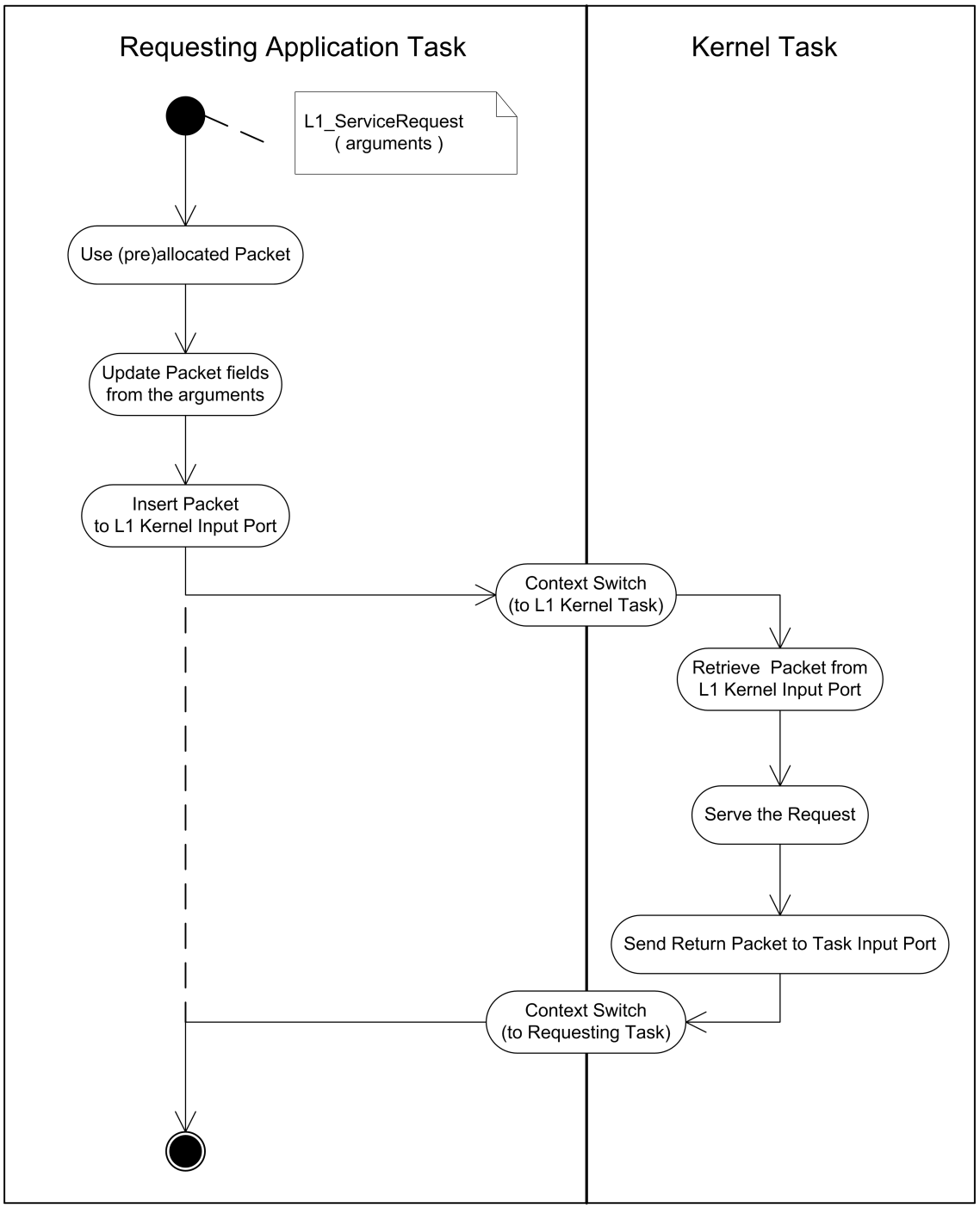

Figure 2.3: Template scenario of the serving of a request to the Kernel

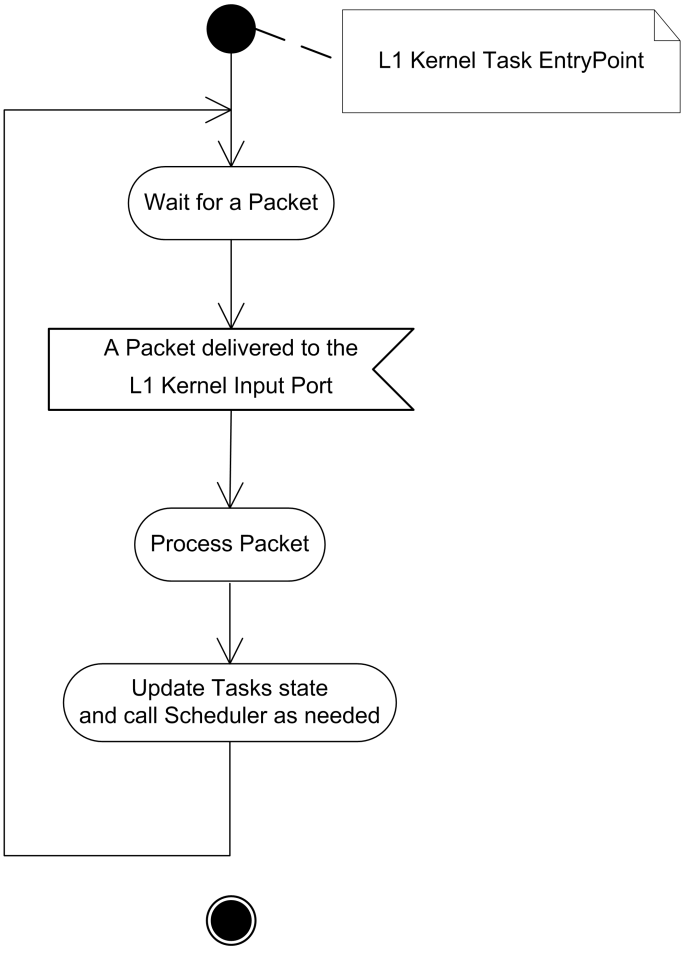

Figure 2.4: The Kernel Loop

2.3.4 Logical view of the Scheduler

For providing multi-tasking VirtuosoNext has a 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 and it has the highest Priority of all Tasks competing for the resource it reserved to use
  - 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 software module 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 (see Section 2.1.3) or by the logic of the service requested of the Kernel Task by a Task. (see Section 2.3.3).

Part II

Installation Instructions

Chapter 3

Installation Instructions

Introduction

This manual will guide you through the installation process of VisualDesigner-VirtuosoNext 1.1 which includes an VirtuosoNext Win64 port plus the corresponding examples. After guiding you through the installation process, the manual explains how to build the provided examples.

3.0.1 Folder Structure on Download Server

The download server contains two top level folders:

- `1.1.x.3\`, where 'x' is a number: This folder contains the binary and source code versions of the developed software, including these instructions. The $\star$ signs symbolise a wildcard. In this folder there are on the top level the following elements:
  - `VirtuosoNext_Installation_Instructions.pdf`: Document describing how to install VirtuosoNext. This is the document that you're currently reading.
  - `VirtuosoNext_OTL_Installation_Instructions.pdf`: Document describing how to install and build the VirtuosoNext-OTL
  - `VirtuosoNext_Support_Request.doc`: Use this form to issue a support request.
  - `README.TXT`: This information
  - `Binary_Distribution\`: Contains the binary redistributable of VirtuosoNext-1.1, its tools, and the Xilinx ZYNQ-BSP.
  - `OTL \`: Contains the source code obtained under the Open Technology License.
- `ExternalTools\`, which contains all the external tools needed to build and use the developed software. Whenever this document refers to the `ExternalTools\` folder, then this folder is meant.

3.1 VisualDesigner-VirtuosoNext Installation Instructions for MS-Windows

This section details how to setup the VisualDesigner-VirtuosoNext-1.0.0.0, which requires the MinGW toolchain, and the CMake build system. These instructions assume that you have downloaded the complete VisualDesigner-VirtuosoNext distribution.

Please ensure that already existing toolchains on the installation system do not conflict with the toolchains that we provide. This usually shows by the build process failing or the generated binary not working.

3.1.1 Install 7zip

Install the 7zip program on your Windows PC by executing the file:
`ExternalTools\7zip1900-x64.exe`

Follow the instructions given by the installer. This tool is required in order to install the MinGW Tool-chain in the next step.

3.1.2 MinGW Tool-chain for Windows

MinGW [1] is a GNU GCC version to compile programs for MS-Windows. It is available both under MS-Windows and Linux, which is one of the reasons why we use it. To install it please follow these steps:

1. Start 7zip;
2. Open the archive `ExternalTools\i686-8.1.0-release-win32-sjlj-rt-v6-rev0.7z`
3. Click on the button 'Extract'
4. In the dialoge that opens enter `c:\` as location for 'Copy to'.
5. Press on OK.
6. The toolchain has now been copied to: `c:\mingw32`

3.1.3 Adding MinGW to the System Binary Search Path

It is necessary to add the newly installed MinGW environment to the Windows System Path. To do this please follow the following steps:

1. Open the Start Search, type in "env", and choose "Edit the system environment variables":
2. Click the "Environment Variables..." button.
3. Under the "System Variables" section (the lower half), find the row with "Path" in the first column, and click edit.
4. The "Edit environment variable" UI will appear.
5. Click the "New" button.
6. Enter "`;c:\mingw32\bin`" in the newly created line.
7. Click the "OK" button.

Now the MinGW environment is part of your path. Warning, should you already have installed a mingw environment on your machine then please remove it from the path, as it could cause compatibility problems.

3.1.4 CMake Build System

VisualDesigner-VirtuosoNext uses the CMake build system [2] (version 3.0 or better) to build itself and applications using it. The following steps guide you through the installation process:

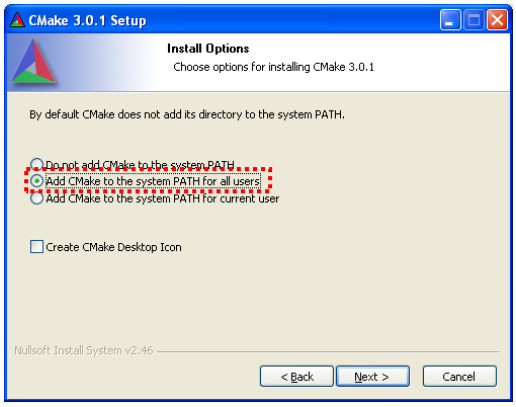

Figure 3.1: Adding CMake to the System Binary Search Path

1. Start the installation process by executing:
   `ExternalTools\cmake-3.0.1-win32-x86.exe`.
2. In the screen “Install Options” select “Add CMake to the system PATH for all users” (see Figure 3.1). This adds the CMake binary directory to the System Binary Search Path, which is necessary in order for the VirtuosoNext build system to be able to use CMake.

3.1.5 Installing VisualDesigner

The VisualDesigner installation image is available in the download folder. To install it, execute:
“`BinaryDistribution\VisualDesigner-VirtuosoNext-1.1.x.3.msi`”
where ‘`x`’ is a number representing the patch-level of the MSI. After this step the VisualDesigner including the VirtuosoNext Kernel Images for Win64 is installed.

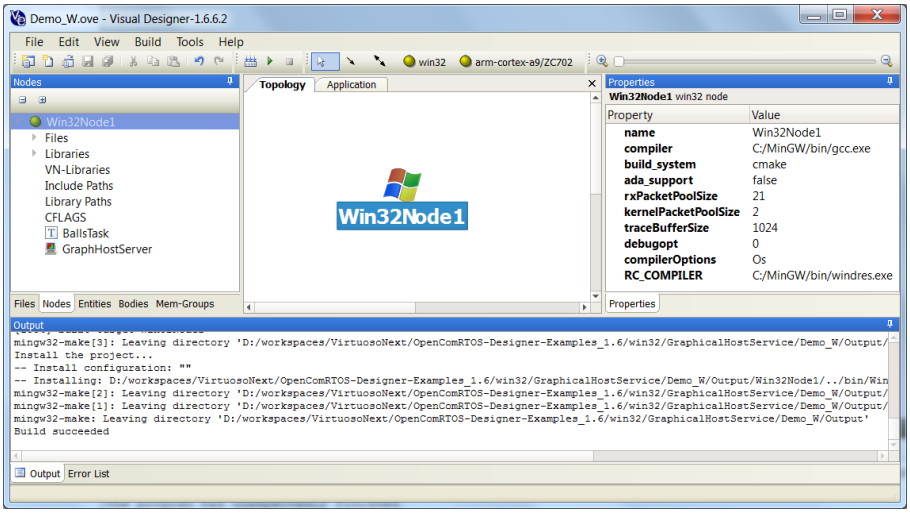

Figure 3.2: Topology of the Demo_W example

3.2 How to run an Example

This section first explains how to build one of the provided examples, before discussing each example in detail. All examples are located in the folder '`Examples\win64`' below the VisualDesigner-VirtuosoNext installation directory.

Please note that VirtuosoNext on MS-Windows is not timing accurate, it is a behaviour emulation of VirtuosoNext.

1. Start VisualDesigner:
   In the Start Menu of MS-Windows select open the group 'VisualDesigner-VirtuosoNext-1.0.0.0' (x representing the patch-level of VisualDesigner). Inside this group click on the entry labeled 'VisualDesigner' to start VisualDesigner.

2. Open the 'Demo_W' project in VisualDesigner:
   In the menu-bar click on 'File' and then on 'Open Project' to open the 'Open Project' dialogue of VisualDesigner. Now navigate to the folder '`examples\win64\GraphicalHostService\-Demo_W\Demo_W.ove`' below the VisualDesigner installation directory (usually `c:\VisualDesigner--VirtuosoNext-1.0.0.0`). There, select the file '`Demo_W.ove`' and click on the button labeled 'open' to open the project. You should now see a topology consisting of a Win64 Node, similar to the one shown in Figure 3.2. The topology diagram is a graphical representation of the project-topology.

3. Check the compiler settings for the Win32Node: Open the 'Properties' pane on the right hand side by clicking on 'Properties' and then pinning it down, using the little pin in the upper right corner of the Window. Left click on the Win64 node to display its properties. Now check whether or not the property 'Compiler' refers to the compiler to use for Win64 Nodes on your system (The MinGW compiler which you installed in Lecture 1)[1].

4. Take a look at the Application Diagram for the example (Figure 3.3). In the Application diagram the developer specifies Tasks and Hubs and their interactions. All necessary code to reflect the changes in the Application diagram gets automatically generated. The diagram updates itself whenever there are changes to the source code, this ensures that both source code and diagram are consistent at all times.

[1] If you followed the instructions given in Lecture 1, the compiler should be set to either "`c:\mingw32\-bin\gcc.exe`" or "`gcc.exe`".

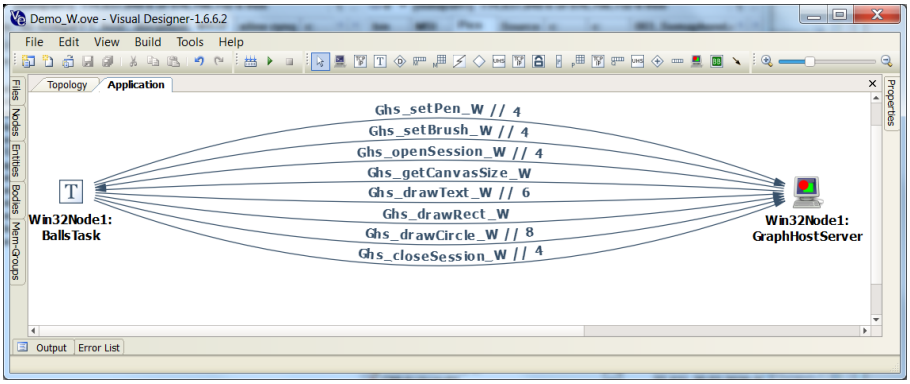

Figure 3.3: Application Diagram of the Demo_W example

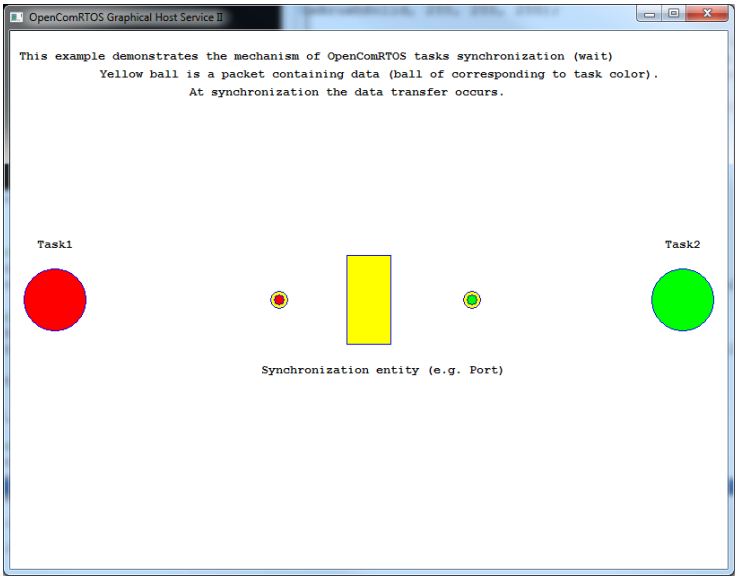

Figure 3.4: Screenshot of the running Demo_W example

5. Build the project:
   Compile the example application using the menu-item ‘Build’ form the ‘Build’ menu. The build run should end with: “Build successful”.

6. Execute the generated binary:
   `c:\VisualDesigner-VirtuosoNext-1.0.0.0\examples\win64\GraphicalHostService\-Demo_W\Output\bin\win32_node.exe`

3.3 Troubleshooting

This section is a collection of installation issues and their solution.

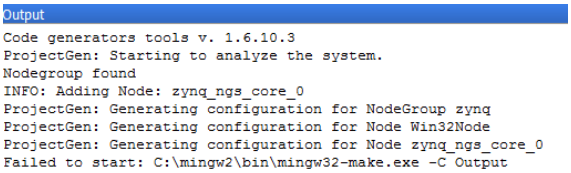

Figure 3.5: Example of Visual-Designer not finding `mingw32-make`.

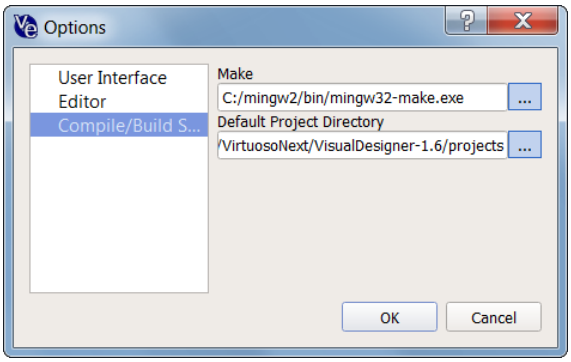

Figure 3.6: Visual-Designer Options Dialogue, 'Compile/Build ...'

3.3.1 mingw32-make not found

In case that Visual-Designer cannot execute the `mingw32-make` command it will show an error message in the Output pane similar to the one shown in fig. 3.5. To fix this issue open the Options Dialogue of Visual-Designer (Main Menu → Tools → Options...). In the Options Dialogue then select 'Compile/Build ...' (see fig. 3.6), and set the path to the make execuable.

3.4 Summary

This chapter gave the installation instructions for Visual-Designer and its dependencies, for MS-Windows 10. Showed how to build and run an example for Windows 64, and finally gave some troubleshooting hints.

Part III

VirtuosoNext

Chapter 4

Module Index

4.1 Modules

Here is a list of all modules:

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 6

Module Documentation

6.1 Task Management Operations

Macros

- #define L1_UNUSED_PARAMETER(x) (void)(x)

Functions

- L1_KernelTicks L1_getCurrentKernelTickCount (void)
- static L1_TaskID L1_getCurrentTaskId (void)

  This function returns the ID of the currently running Task.
- static L1_Priority L1_getCurrentTaskPriority (void)
- static L1_UINT32 L1_getCurrentTaskStackSize (void)
- static const char * L1_hubIdToHubName (L1_PortID hubId)

  This function retrieves the name of the Hub identified by the parameer hubId.
- L1_Time L1_KernelTicks2msec (L1_KernelTicks ticks)
- L1_KernelTicks L1_Msec2KernelTicks (L1_Time timeInMs)
- static __inline__ L1_ReturnCode L1_ResumeTask_W (L1_PortID task)
- static __inline__ L1_ReturnCode L1_StartTask_W (L1_PortID task)
- static __inline__ L1_ReturnCode L1_StopTask_W (L1_PortID task)
- static __inline__ L1_ReturnCode L1_SuspendTask_W (L1_PortID task)
- static const char * L1_taskIdToTaskName (L1_TaskID taskId)

  This function retrieves the name of the Task identified by the parameer taskId.
- static __inline__ L1_ReturnCode L1_WaitTask_WT (L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_WaitUntil_WT (L1_KernelTicks timePoint)
- static __inline__ L1_ReturnCode L1_Yield_W (void)

Variables

- const L1_HubNameToID * L1_HubNamesToIDs [ ]

  This array contains for each node-id how many Tasks are present on this node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`
- const L1_UINT32 L1_NBR_OF_NODES

Global variable containing the number of VirtuosoNext Nodes in the complete system. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

- const L1_UINT32 L1_NodeIdToNbrOfHubs [ ]

  This array contains for each node-id how many Hubs are present on this node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

- const L1_UINT32 L1_NodeIdToNbrOfTasks [ ]

  This array maps node-ids to arrays mapping local-ids to the name of the Hubs on each node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

- const L1_TaskNameToID ∗ L1_TaskNamesToIDs [ ]

  This array maps node-ids to arrays mapping local-ids to the name of the Tasks on each node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

6.1.1 Detailed Description

VirtuosoNext offers the following operations to manage Tasks.

6.1.2 Visual Designer

Figure 6.1: Application Diagram Icon

6.1.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- priority: The priority of the Task. A lower value means a higher priority. The user can select values in the range 3-254.
- stacksize: The size of the stack, in Bytes, for this Task.
- status: Status of the Task, started (L1_STARTED) or inactive (L1_INACTIVE).
- entrypoint: The name of the function that represents the Task-Entry-Point, it must be of type L1_-TaskFunction.
- arguments: A 32 Bit unsigned integer value that gets passed to the Task-Entry-Point when it gets started.

6.1.3 Macro Definition Documentation

6.1.3.1 #define L1_UNUSED_PARAMETER(*x*) (void)(x)

This is a macro that is used to indicate that a parameter is not being used in this function.

Parameters

x	The unused parameter.

6.1.4 Function Documentation

6.1.4.1 L1_KernelTicks L1_getCurrentKernelTickCount (void)

Returns the number of expired Kernel-Ticks.

Returns

The number of expired Kernel-Ticks since start of the Kernel-Task.

See Also

L1_WaitUntil_WT
L1_KernelTicks2msec
L1_KernelTicks2msec

6.1.4.2 static L1_TaskID L1_getCurrentTaskId (void) [inline], [static]

This function returns the ID of the currently running Task.

Returns

L1_TaskID The ID of the Task calling this function.

Warning

Do not use this function in ISR-Context, it will give incorrect results.

6.1.4.3 static L1_Priority L1_getCurrentTaskPriority (void) [inline], [static]

Returns

L1_Priority

Warning

Do not use this function in ISR-Context, it will give incorrect results.

6.1.4.4 static L1_UINT32 L1_getCurrentTaskStackSize (void) [inline], [static]

Returns

L1_UINT32

Warning

Do not use this function in ISR-Context, it will give incorrect results.

6.1.4.5 static const char* L1_hubIdToHubName (L1_PortID *hubId*) [inline], [static]

This function retrieves the name of the Hub identified by the parameer hubId.

Parameters

taskId	This is the ID of the Hub for which to determine the name.

Returns

!NULL Pointer to the C-String (NULL terminated) with the name of the Hub.
NULL A non existing hubId was given.

Warning

If an invalid ID (read an ID of a non-existing Hub) is given, then the function returns NULL.

6.1.4.6 L1_Time L1_KernelTicks2msec (L1_KernelTicks *ticks*)

Converts Kernel-Ticks to milliseconds

Parameters

ticks	The Kernel-Ticks value to convert.

Returns

The corresponding milliseconds

See Also

L1_WaitUntil_WT
L1_getCurrentKernelTickCount
L1_Msec2KernelTicks

6.1.4.7 L1_KernelTicks L1_Msec2KernelTicks (L1_Time *timeInMs*)

Converts milliseconds to Kernel-Ticks.

Parameters

timeInMs	The milliseconds value to convert.

Returns

The corresponding Kernel-Ticks.

See Also

L1_WaitUntil_WT
L1_getCurrentKernelTickCount
L1_KernelTicks2msec

6.1.4.8 static __inline__ L1_ReturnCode L1_ResumeTask_W (L1_PortID *task*) [static]

This service resumes the task at the point it was when suspended.

Parameters

task	ID of the Task to be resumed.

Returns

L1_ReturnCode:

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

Precondition

- Task was in suspend state

Postcondition

- Task resumed at the point it was when suspended.

Remarks

```
SPC Task states
SPC Resuming a Task
```

6.1.4.9 static __inline__ L1_ReturnCode L1_StartTask_W (L1_PortID *task*) [static]

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

Parameters

task	the ID of the Task to be started.

Returns

L1_ReturnCode:

- RC_OK, if the Task has started successfully.
- RC_FAIL, if the service failed.

Precondition

- Task is inactive
- Task is initialised and ready to start
- All elements of TaskControlRecord are filled in, including entry-point and stack pointer.
- The Task cannot start itself

Postcondition

- Task is on the READY list (case RC_OK)
- RC_Fail will be raised in following cases:
  - Task starts itself
  - Task is not yet initialised (i.e. not all TCR fields are filled in)

Remarks

SPC Task states

6.1.4.10 static __inline__ L1_ReturnCode L1_StopTask_W (L1_PortID *task*) [static]

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

Parameters

task	the ID of the Task to be stopped.

Returns

L1_ReturnCode:

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

Precondition

- Task is not stopped
- The Task is not the requesting task itself
- The Task should not lock any resource. (Task should release all resources first using a secondary entrypoint function

Postcondition

- Task is no longer on any waiting list
- Entry Point restored
- Any data may be lost
- Task is in stopped state

Note

Requests for the task can continue to arrive from other tasks No clean up yet for pending asynchronous packets The kernel task will discard any Packets with as destination a stopped Task.

Warning

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 be misbehaving (e.g. Stack overflow, numerical exception, etc.) Care should also be taken when stopping a driver task as this impacts the routing functionality. Additional kernel service (messages) are used for the clean-up of pending Packets in waiting list on other nodes Except for the case of two-phase services, it is sufficient to remove the (at most single waiting) Packet from the appropriate waiting list (either local or remote) (Waiting List of Port, Packet Pool or Kernel Input Port, or Driver Task Input Port). Only for (returning of) remote services, it is possible that a Packet is destined for a stopped Task.

Remarks

```
SPC Task states
SPC Stopping a Task
```

6.1.4.11 static __inline__ L1_ReturnCode L1_SuspendTask_W (L1_PortID *task*) [static]

This service suspends task and marks it as such in the Task Control Record.

Parameters

task	the ID of the Task to be suspended.

Returns

L1_ReturnCode:

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

Precondition

- The Task is not the requesting task itself

Postcondition

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

Note

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 needs 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. synchronisation. In particular, the Task may remain and inserted in the READY List. The task is however never made RUNNING. Hence, the suspend state of a Task is only changing the status of the task preventing it from being scheduled until the task is resumed.

Remarks

```
SPC Task states
SPC Suspending a Task
```

6.1.4.12 static const char* L1_taskIdToTaskName (L1_TaskID *taskId*) [inline], [static]

This function retrieves the name of the Task identified by the parameer taskId.

Parameters

taskId	This is the ID of the Task for which to determine the name.

Returns

!NULL Pointer to the C-String (NULL terminated) with the name of the Task.
NULL A non existing taskId was given.

Warning

If an invalid ID (read an ID of a non-existing Task) is given, then the function returns NULL.

6.1.4.13 static __inline__ L1_ReturnCode L1_WaitTask_WT (L1_Timeout *timeout*) [static]

This Kernel service is called by a Task to wait for a specified time interval.

Parameters

timeout	how many system ticks the task wants to wait.

Returns

L1_ReturnCode:

- RC_TO Service returned after Timeout.
- RC_FAIL service failed.

Precondition

- None

Postcondition

- Calling task ready.

Remarks

SPC Task states

6.1.4.14 static __inline__ L1_ReturnCode L1_WaitUntil_WT (L1_KernelTicks *timePoint*) [static]

This Kernel service allows a Task to wait until a specific time-point. Until this time-point has been reached the Task will be suspended.

Parameters

timePoint	The point in time the Tasks would like to be rescheduled. This value is in Kernel-Ticks.

Returns

RC_TO If the Task has been scheduled because the time-point has been reached.
RC_FAIL If there was a problem.

See Also

L1_getCurrentKernelTickCount
L1_Msec2KernelTicks

6.1.4.15 static __inline__ L1_ReturnCode L1_Yield_W (void) [static]

This function allows a Task to prematurely pass execution to another Task with at least the same priority than the currently active Task.

Returns

RC_OK
RC_FAIL

6.1.5 Variable Documentation

6.1.5.1 const L1_HubNameToID* L1_HubNamesToIDs[]

This array contains for each node-id how many Tasks are present on this node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

See Also

L1_taskIdToTaskName
L1_hubIdToHubName

6.1.5.2 const L1_UINT32 L1_NBR_OF_NODES

Global variable containing the number of VirtuosoNext Nodes in the complete system. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

See Also

L1_taskIdToTaskName
L1_hubIdToHubName

6.1.5.3 const L1_UINT32 L1_NodeIdToNbrOfHubs[]

This array contains for each node-id how many Hubs are present on this node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

See Also

L1_taskIdToTaskName
L1_hubIdToHubName

6.1.5.4 const L1_UINT32 L1_NodeIdToNbrOfTasks[]

This array maps node-ids to arrays mapping local-ids to the name of the Hubs on each node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

See Also

L1_taskIdToTaskName
L1_hubIdToHubName

6.1.5.5 const L1_TaskNameToID* L1_TaskNamesToIDs[]

This array maps node-ids to arrays mapping local-ids to the name of the Tasks on each node. This variable is defined at build time in the file `Output/src/L1_nodes_id_mapping.c`

See Also

L1_taskIdToTaskName
L1_hubIdToHubName

6.2 Asynchronous Services

Functions

- L1_ReturnCode L1_initialiseAsyncPacket (L1_Packet *packet, L1_PacketData *dataPart)
- L1_ReturnCode L1_WaitForPacket (L1_Packet **Packet, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_WaitForPacket_NW (L1_Packet **packet)
- static __inline__ L1_ReturnCode L1_WaitForPacket_W (L1_Packet **packet)
- static __inline__ L1_ReturnCode L1_WaitForPacket_WT (L1_Packet **packet, L1_Timeout timeout)

6.2.1 Detailed Description

6.2.2 Function Documentation

6.2.2.1 L1_ReturnCode L1_initialiseAsyncPacket (L1_Packet * *packet*, L1_PacketData * *dataPart*)

This function initalises an L1_Packet to be used for an Asynchronous interaction. The following fields get initialised:

- RequestingTaskID: Is set to the Task ID of the currently running Task.
- ListElement.Next: NULL;
- ListElement.Prev: NULL;
- ListElement.Priority: Is set to the IntrinsicPriority of the currently running Task.
- PendingRequestHandler: NULL;
- PendingRequestListElement.Prev = NULL;
- PendingRequestListElement.Next = NULL;
- PendingRequestListElement.Priority: Is set to the IntrinsicPriority of the currently running Task.
- OwnerPool: NULL, because this packet is not part of a Packet Pool.
- dataPart to the value of the parameter dataPart.

Parameters

packet	Pointer to the L1_Packet to Initialise.
dataPart	Pointer to the dataPart of the Packet if needed for the chosen interaction. Otherwise, it can be NULL.

Returns

RC_OK if the operation was successful
RC_FAIL if the operation was not successful.

Warning

Do not pass NULL to the parameter 'dataPart' unless you know what you're doing.

6.2.2.2 L1_ReturnCode L1_WaitForPacket (L1_Packet ** *Packet*, L1_Timeout *Timeout*)

Resynchronises a Task with the Packet it earlier requested by calling the L1_GetPacketFromPort_A or L1_PutPacketToPort_A service.

Returns when the Packet is available.

Parameters

Packet	Pointer to a pointer to an L1_Packet, the function will return the L1_Packet here.
Timeout	The number of ticks the function should wait for synchronisation.

Returns

RC_OK If the operation was successful.
RC_TO If the timeout expired.
RC_FAIL If the function was unable to perform the desired operation.

Remarks

```
SPC Input Port
```

6.2.2.3 static __inline__ L1_ReturnCode L1_WaitForPacket_NW (L1_Packet ** *packet*) [static]

Resynchronises a Task with the Packet it earlier requested by calling the L1_GetPacketFromPort_A or L1_PutPacketToPort_A service. Returns immediately.

Parameters

packet	will contain the pointer to the L1_Packet that was returned to the Task.

Returns

L1_ReturnCode

- RC_OK service completed successfully (there was a waiting Packet in the Port)/.
- RC_FAIL the service failed, Packet* is then set to NULL.

Precondition

-This service must have been preceded by a call to L1_GetPacketFromPort_A or L1_PutPacketToPort_A.

Postcondition

-The preallocated Packet must contain a pointer to a previously allocated Packet from the Packet Pool, containing the result of a preceding call to L1_PutPacketToPort_A or L1_GetPacketFromPort_A.

Remarks

SPC Input Port

6.2.2.4 static __inline__ L1_ReturnCode L1_WaitForPacket_W (L1_Packet ** *packet*) [static]

Resynchronises a Task with the Packet it earlier requested by calling the L1_GetPacketFromPort_A or L1_PutPacketToPort_A service.

Returns when the Packet is available.

Parameters

packet	will contain the pointer to the L1_Packet that was returned to the Task.

Returns

L1_ReturnCode

- RC_OK service completed successfully (there was a waiting Packet in the Port)/.
- RC_FAIL the service failed.

Precondition

-This service must have been preceded by a call to L1_GetPacketFromPort_A or L1_PutPacketToPort_A.

Postcondition

-The preallocated Packet must contain a pointer to a previously allocated Packet from the Packet Pool, containing the result of a preceding call to L1_PutPacketToPort_A or L1_GetPacketFromPort_A.

Remarks

SPC Input Port

6.2.2.5 static __inline__ L1_ReturnCode L1_WaitForPacket_WT (L1_Packet ** *packet*, L1_Timeout *timeout*) [static]

A Task calls this service to resynchronize on Packets earlier requested by calling the L1_GetPacketFromPort_A or L1_PutPacketToPort_A service. Returns when the Packet is available or when the timeout expires.

Parameters

packet	will contain the pointer to the L1_Packet that was returned to the Task.
timeout	the number of system ticks the call should wait for a packet to become available.

Returns

L1_ReturnCode

- RC_OK service completed successfully (there was a waiting Packet in the Port)/.
- RC_FAIL the service failed, Packet* is set to NULL.
- RC_TO the timeout expired, Packet* is set to NULL.

Precondition

-This service must have been preceded by a call to L1_GetPacketFromPort_A or L1_PutPacketToPort_A.

Postcondition

-The preallocated Packet must contain a pointer to a previously allocated Packet from the Packet Pool, containing the result of a preceding call to L1_PutPacketToPort_A or L1_GetPacketFromPort_A.

Remarks

SPC Input Port

6.3 Base Types

Modules

- L1_BYTE
- L1_UINT8
- L1_INT8

- L1_UINT16
- L1_INT16
- L1_UINT32
- L1_INT32
- L1_UINT64
- L1_INT64
- L1_Time
- L1_KernelTicks
- L1_BOOL
- L1_Priority
- L1_TaskArguments
- L1_ErrorCode
- L1_ReturnCode

6.3.1 Detailed Description

6.4 L1_BYTE

Variables

- const L1_BYTE L1_BYTE_MAX
- const L1_BYTE L1_BYTE_MIN

6.4.1 Detailed Description

L1_BYTE is a 8-bit unsigned integer type.

6.4.2 Variable Documentation

6.4.2.1 const L1_BYTE L1_BYTE_MAX

The maximal value of a L1_BYTE variable.

6.4.2.2 const L1_BYTE L1_BYTE_MIN

The minimal value of a L1_BYTE variable.

Remarks

```
SPC Data Types
```

6.5 L1_UINT8

Variables

- const L1_UINT8 L1_UINT8_MAX
- const L1_UINT8 L1_UINT8_MIN

6.5.1 Detailed Description

L1_UINT8 is a 8-bit unsigned integer type.

6.5.2 Variable Documentation

6.5.2.1 const L1_UINT8 L1_UINT8_MAX

The maximal value of a L1_UINT8 variable.

6.5.2.2 const L1_UINT8 L1_UINT8_MIN

The maximal value of a L1_UINT8 variable.

6.6 L1_INT8

Variables

- const L1_INT8 L1_INT8_MAX
- const L1_INT8 L1_INT8_MIN

6.6.1 Detailed Description

L1_INT8 is a 8-bit signed integer type.

6.6.2 Variable Documentation

6.6.2.1 const L1_INT8 L1_INT8_MAX

The maximal value of a L1_INT8 variable.

6.6.2.2 const L1_INT8 L1_INT8_MIN

The maximal value of a L1_INT8 variable.

6.7 L1_UINT16

Variables

- const L1_UINT16 L1_UINT16_MAX
- const L1_UINT16 L1_UINT16_MIN

6.7.1 Detailed Description

L1_UINT16 is a 16-bit unsigned integer type.

6.7.2 Variable Documentation

6.7.2.1 const L1_UINT16 L1_UINT16_MAX

The maximal value of a L1_UINT16 variable.

6.7.2.2 const L1_UINT16 L1_UINT16_MIN

The minimal value of a L1_UINT16 variable.

6.8 L1_INT16

Variables

- const L1_INT16 L1_INT16_MAX
- const L1_INT16 L1_INT16_MIN

6.8.1 Detailed Description

L1_INT16 is a 16-bit signed integer type.

6.8.2 Variable Documentation

6.8.2.1 const L1_INT16 L1_INT16_MAX

The maximal value of a L1_INT16 variable.

6.8.2.2 const L1_INT16 L1_INT16_MIN

The minimal value of a L1_INT16 variable.

6.9 L1_UINT32

Variables

- const L1_UINT32 L1_UINT32_MAX
- const L1_UINT32 L1_UINT32_MIN

6.9.1 Detailed Description

UINT32 is a 32-bit unsigned integer type.

6.9.2 Variable Documentation

6.9.2.1 const L1_UINT32 L1_UINT32_MAX

The maximal value of a L1_UINT32 variable.

6.9.2.2 const L1_UINT32 L1_UINT32_MIN

The minimal value of a L1_UINT32 variable.

6.10 L1_INT32

Variables

- const L1_INT32 L1_INT32_MAX
- const L1_INT32 L1_INT32_MIN

6.10.1 Detailed Description

L1_INT32 is a 32-bit signed integer type.

6.10.2 Variable Documentation

6.10.2.1 const L1_INT32 L1_INT32_MAX

The maximal value of a L1_INT32 variable.

6.10.2.2 const L1_INT32 L1_INT32_MIN

The minimal value of a L1_INT32 variable.

6.11 L1_UINT64

Variables

- const L1_UINT64 L1_UINT64_MAX
- const L1_UINT64 L1_UINT64_MIN

6.11.1 Detailed Description

L1_UINT64 is a 64-bit signed integer type.

6.11.2 Variable Documentation

6.11.2.1 const L1_UINT64 L1_UINT64_MAX

The maximal value of a L1_INT64 variable.

6.11.2.2 const L1_UINT64 L1_UINT64_MIN

The minimal value of a L1_INT64 variable.

6.12 L1_INT64

Variables

- const L1_INT64 L1_INT64_MAX
- const L1_INT64 L1_INT64_MIN

6.12.1 Detailed Description

L1_INT64 is a 64-bit signed integer type.

6.12.2 Variable Documentation

6.12.2.1 const L1_INT64 L1_INT64_MAX

The maximal value of a L1_INT64 variable.

6.12.2.2 const L1_INT64 L1_INT64_MIN

The minimal value of a L1_INT64 variable.

6.13 L1_Time

Variables

- const L1_UINT32 L1_Time_MAX
- const L1_UINT32 L1_Time_MIN

6.13.1 Detailed Description

6.13.2 Variable Documentation

6.13.2.1 const L1_UINT32 L1_Time_MAX

The maximal value of a L1_Time variable.

6.13.2.2 const L1_UINT32 L1_Time_MIN

The minimal value of a L1_Time variable.

6.14 L1_KernelTicks

Variables

- const L1_UINT32 L1_KernelTicks_MAX
- const L1_UINT32 L1_KernelTicks_MIN

6.14.1 Detailed Description

6.14.2 Variable Documentation

6.14.2.1 const L1_UINT32 L1_KernelTicks_MAX

The maximal value of a KernelTicks variable.

6.14.2.2 const L1_UINT32 L1_KernelTicks_MIN

The minimal value of a L1_KernelTicks variable.

6.15 L1_BOOL

Macros

- #define L1_FALSE 0U
- #define L1_TRUE 1U

6.15.1 Detailed Description

L1_BOOL is a basic integer type sufficient to represent the values: L1_TRUE and L1_FALSE. The size is target dependent.

6.15.2 Macro Definition Documentation

6.15.2.1 #define L1_FALSE 0U

Definition of the value for L1_FALSE.

6.15.2.2 #define L1_TRUE 1U

Definition of the value for L1_TRUE.

6.16 L1_Priority

6.16.1 Detailed Description

L1_Priority is a basic unsigned integer type sufficient to represent the values from 0 to 255, identifying the priority of a Task or a Packet.

6.17 L1_TaskArguments

6.17.1 Detailed Description

Argument to a Task Entry Point.

6.18 Types related to Timing

Typedefs

- typedef L1_UINT32 L1_KernelTicks
- typedef L1_UINT32 L1_Time
- typedef L1_UINT32 L1_Timeout

6.18.1 Detailed Description

6.18.2 Typedef Documentation

6.18.2.1 typedef L1_UINT32 L1_KernelTicks

This data type

See Also

L1_KernelTicks_MIN
L1_KernelTicks_MAX

6.18.2.2 typedef L1_UINT32 L1_Time

This data type is used to represent the number of expired ticks.

See Also

L1_Time_MIN
L1_Time_MAX

6.18.2.3 typedef L1_UINT32 L1_Timeout

L1_Timeout is a basic unsigned integer type that represents a timeout value in milliseconds. The maximum value, allowed by the appropriate L1_Timeout integer type, is interpreted as an infinite timeout. For example if L1_Timeout is provided by the means of a 16-bit or 32bit unsigned integer, then the infinite timeout is 0xFFFF(FFFF) Hex. The infinite timeout is (should be) referred as named constant L1_Infinite_TimeOut

6.19 L1_ErrorCode

Typedefs

- typedef L1_UINT32 L1_ErrorCode

Variables

- const L1_ErrorCode L1_ErrorCode_MAX

6.19.1 Detailed Description

L1-Service error code.

6.19.2 Typedef Documentation

6.19.2.1 typedef L1_UINT32 L1_ErrorCode

This data type is used to represent error codes in VirtuosoNext.

Warning

Only the upper 24bit of the L1_UINT32 are being used.

6.19.3 Variable Documentation

6.19.3.1 const L1_ErrorCode L1_ErrorCode_MAX

The maximal value of a L1_ErrorCode variable.

6.20 L1_ReturnCode

Macros

- #define RC_FAIL 0x1
- #define RC_FAIL_END 0x80
- #define RC_FAIL_NULL_POINTER 0x10
- #define RC_FAIL_OUT_OF_MEM 0x20
- #define RC_FAIL_UNSUPPORTED 0x3
- #define RC_OK 0x0
- #define RC_TO 0x2

Typedefs

- typedef L1_UINT32 L1_ReturnCode

6.20.1 Detailed Description

Return code representation.

6.20.2 Macro Definition Documentation

6.20.2.1 #define RC_FAIL 0x1

Return code for a failed request

6.20.2.2 #define RC_FAIL_END 0x80

VirtuosoNext has an area of 128 default return codes, any service that needs to define additional ones shall do starting at 129 (RC_FAIL_END + 1)

6.20.2.3 #define RC_FAIL_NULL_POINTER 0x10

A required pointer was NULL instead of the correct value.

6.20.2.4 #define RC_FAIL_OUT_OF_MEM 0x20

A memory allocation could not be served as the system is out of free memory.

6.20.2.5 #define RC_FAIL_UNSUPPORTED 0x3

Return code for if a request is not supported.

6.20.2.6 #define RC_OK 0x0

Return code for a successful request

6.20.2.7 #define RC_TO 0x2

Return code for a failed request after the timeout expired.

6.20.3 Typedef Documentation

6.20.3.1 typedef L1_UINT32 L1_ReturnCode

This data type represents a complete return code from a Function or a Service.

6.21 VirtuosoNext Hub

Modules

- Black Board Hub
- Data Event Hub
- Data-Queue Hub
- Event Hub
- FIFO Hub
- Memory Pool Hub
- Packet Pool Hub
- Port Hub
- Resource Hub
- Semaphore Hub
- Memory Block Queue Hub

6.21.1 Detailed Description

6.22 Developer Information

Data Structures

- struct _struct_L1_Hub_

Macros

- #define L1_HubNodeID(h) (((h) & L1_GLOBALID_MASK) >> 8)
- #define L1_id2localhub(h) (&L1_LocalHubs[((h) & ~L1_GLOBALID_MASK)])
- #define L1_isControlPacket(p) ((((p)->ServiceID) % 256) == L1_SID_IOCTL_HUB)
- #define L1_isLocalHubID(h) (((h) & L1_GLOBALID_MASK) == ((L1_KernelInputPortID) & L1-_GLOBALID_MASK))
- #define L1_isPutPacket(p) ((((p)->ServiceID) % 256) == L1_SID_PUT_TO_HUB)

Typedefs

- typedef void(* L1_HubControlFunction)(L1_Hub *hub, L1_Packet *packet, L1_BYTE ioctl_type)
- typedef void(* L1_HubStateUpdateFunction)(L1_Hub *hub, L1_Packet *packet)
- typedef L1_BOOL(* L1_HubSyncConditionFunction)(L1_Hub *hub, L1_Packet *packet)
- typedef void(* L1_HubSynchronizeFunction)(L1_Hub *hub, L1_Packet *packet, L1_Packet *waiting-Packet)

Functions

- void L1_Hub_exchangePacketData (L1_Packet *packet, L1_Packet *waitingPacket)

6.22.1 Detailed Description

6.22.2 Macro Definition Documentation

6.22.2.1 #define L1_HubNodeID(*h*) (((h) & L1_GLOBALID_MASK) >> 8)

Extracts the Node ID part from the L1_Hub ID given in parameter h.

Parameters

h	L1_HubID from which to extract the Node IF part.

Returns

The Node ID part of the L1_HubID h.

6.22.2.2 #define L1_id2localhub(*h*) (&L1_LocalHubs[((h) & ~L1_GLOBALID_MASK)])

Converts the L1_HubID h to a pointer to the Hub on the local Node.

Parameters

h	The L1_HubID to be converted.

Returns

The pointer to the Hub identified by the L1_HubID h.

Warning

This function does not check whether this L1_HubID identifies a local Hub. For this the developer must use L1_isLocalHubID().

See Also

L1_isLocalHubID

6.22.2.3 #define L1_isControlPacket(*p*) ((((p)->ServiceID) % 256) == L1_SID_IOCTL_HUB)

Determines whether the L1_Packet parameter p points to contains an IOCTL-Request

Parameters

p	Pointer the L1_Packet to be checked.

Returns

L1_TRUE The L1_Packet pointed to by p is a Control-Packet
L1_FALSE Otherwise.

6.22.2.4 #define L1_isLocalHubID(h) (((h) & L1_GLOBALID_MASK) == ((L1_KernelInputPortID) & L1_GLOBALID_MASK))

Checks whether or not the Hub identified by the L1_HubID h is on this Node, i.e. is locally available.

Parameters

h	The L1_HubID to be checked.

Returns

L1_TRUE The HubID identifies a Hub on this Node.
L1_FALSE Otherwise.

6.22.2.5 #define L1_isPutPacket(p) ((((p)->ServiceID) % 256) == L1_SID_PUT_TO_HUB)

Remarks

```
SPC Number of Tasks and Hubs
SPC Number of Nodes
SPC System-Wide IDs
```

Determines whether the L1_Packet parameter p points to contains a Put-Request

Parameters

p	Pointer the L1_Packet to be checked.

Returns

L1_TRUE The L1_Packet pointed to by p is a Put-Packet
L1_FALSE Otherwise.

6.22.3 Typedef Documentation

6.22.3.1 typedef void(* L1_HubControlFunction)(L1_Hub *hub, L1_Packet *packet, L1_BYTE ioctl_type)

Ioctl like function, e.g. to initialize, set and get state parameters. This function can also be used to implement hubs which abstract hardware devices.

Parameters

hub	Pointer to the generic Hub state structure associated with this Hub.
packet	Pointer to the L1_Packet which caused the function to be called. It will contain additional information.
ioctl_type	Control operation to be executed (L1_IOCTL_HUB_OPEN only for this hub).

6.22.3.2 typedef void(∗ L1_HubStateUpdateFunction)(L1_Hub ∗hub, L1_Packet ∗packet)

Remarks

```
SPC Hubs are derived from the Generic Hub
```

This function updates the state of a Hub.

Parameters

hub	Pointer to the generic Hub state structure associated with this Hub.
packet	Pointer to the L1_Packet which should be used to update the Hub state.

Precondition

Empty waiting list, as complimentary packets already accounted for.

6.22.3.3 typedef L1_BOOL(∗ L1_HubSyncConditionFunction)(L1_Hub ∗hub, L1_Packet ∗packet)

This function checks whether or not the Packet results in a synchronisation with the Hub taking place.

Parameters

hub	Pointer to the generic Hub state structure associated with this Hub.
packet	Pointer to the L1_Packet for which to determine whether a synchronisation happened or not.

Returns

L1_TRUE Synchronisation happened.
L1_FALSE Otherwise.

Precondition

Empty waiting list and exactly one packet non-null

6.22.3.4 typedef void(∗ L1_HubSynchronizeFunction)(L1_Hub ∗hub, L1_Packet ∗packet, L1_Packet ∗waitingPacket)

Function to be called when the Generic-Hub detects that synchronisation was achieved, i.e. a Packet waiting in the Hub-WaitingList, and it's counter part just having arrived. Thus both Packet and WaitingPacket will never be NULL.

The function shall return the Packets to their Tasks!

Parameters

hub	Pointer to the generic Hub state structure associated with this Hub.
packet	Pointer to the L1_Packet which caused the function to be called.
waitingPacket	Pointer to the L1_Packet which was waiting on the Hub-WaitingList.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL

6.22.4 Function Documentation

6.22.4.1 void L1_Hub_exchangePacketData (L1_Packet * *packet*, L1_Packet * *waitingPacket*)

This function copies the data from the Put-Packet to the Get-Packet. It automatically determines which one is which.

Parameters

packet	Pointer to the L1_Packet that just arrived from a Task.
waitingPacket	Pointer to the Packet that was waiting in the Hub.

Precondition

packet NOT NULL
waitingPacket NOT NULL

6.23 Black Board Hub

Data Structures

- struct L1_BlackBoard_Board
- struct L1_BlackBoard_HubState

Functions

- L1_BOOL BlackBoardHub_SyncCondition (L1_Hub *Hub, L1_Packet *Packet)
- void BlackBoardHub_Synchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)
- void BlackBoardHub_Update (L1_Hub *Hub, L1_Packet *Packet)
- static __inline__ L1_ReturnCode L1_Drv_Isr_UpdateBlackBoard_NW (L1_HubID hubID, L1_Packet *packet, L1_BYTE *message, L1_UINT32 messageSize)
- static __inline__ L1_ReturnCode L1_Drv_Isr_UpdateDataEvent_NW (L1_HubID hubID, L1_Packet *packet, L1_BYTE *data, L1_UINT32 length)
- static __inline__ L1_BOOL L1_isBlackBoardHub (L1_Hub *pHub)
- L1_ReturnCode L1_ReadBlackBoard (L1_HubID hubID, L1_Packet *packet, L1_BYTE *messageBuffer, L1_UINT32 bufferSize, L1_UINT32 *receivedMessageSize, L1_UINT32 *messageNumber, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_ReadBlackBoard_NW (L1_HubID hubID, L1_BYTE *messageBuffer, L1_UINT32 bufferSize, L1_UINT32 *receivedMessageSize, L1_UINT32 *messageNumber)
- static __inline__ L1_ReturnCode L1_ReadBlackBoard_W (L1_HubID hubID, L1_BYTE *messageBuffer, L1_UINT32 bufferSize, L1_UINT32 *receivedMessageSize, L1_UINT32 *messageNumber)
- static __inline__ L1_ReturnCode L1_ReadBlackBoard_WT (L1_HubID hubID, L1_BYTE *messageBuffer, L1_UINT32 bufferSize, L1_UINT32 *receivedMessageSize, L1_UINT32 *messageNumber, L1_Timeout timeout)
- L1_ReturnCode L1_UpdateBlackBoard (L1_HubID hubID, L1_Packet *packet, L1_BYTE *message, L1_UINT32 messageSize, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_UpdateBlackBoard_NW (L1_HubID hubID, L1_BYTE *message, L1_UINT32 messageSize)
- L1_ReturnCode L1_WipeBoard (L1_HubID hubID, L1_Packet *packet, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_WipeBoard_NW (L1_HubID hubID)

6.23.1 Detailed Description

6.23.2 Hub Description

The Blackboard Hub is meant as a 'safe global data structure', where data can be published to. It is safer than a normal global data structure for the following reasons:

- There is no read operation taking place while a write operation takes place.
- The readers of the black board get told how many times the message on the board has been updated already, thus they can check whether they missed an update and thus take corrective action if needed.
- If the blackboard is empty, i.e. no data is present, the readers will be put onto the Waiting List of the Hub (_WT and _NW semantics are obeyed) and will be released upon the writer posting a message onto the board. There are two reasons why no data might be present on the board, the first one being because no one has written any data yet, i.e. after system start up. The second reason is that the board has been wiped.
- The readers have their own private copy of the contents of the blackboard and they decide when it will be updated with new contents. This prevents changes of the global variable contents half way during processing.

Remarks

```
REQ Blackboard
SPC Blackboard
```

6.23.3 Visual Designer

Figure 6.2: Application Diagram Icon

6.23.3.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.

6.23.4 Function Documentation

6.23.4.1 L1_BOOL BlackBoardHub_SyncCondition (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function returns L1_TRUE when synchronization must happen in a blackboard hub when receiving a packet.

Parameters

Hub	ID of the blackboard hub.
Packet	Pointer to the L1_Packet received at the blackboard Hub.

Returns

L1_BOOL:

- L1_FALSE Synchronization is not needed.
- L1_TRUE Synchronization must take place.

Precondition

empty waiting list.

Postcondition

None

6.23.4.2 void BlackBoardHub_Synchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall update the state of the Hub using the message contained in the L1_Packet packet and then return it to its Task. If packet->DataSize = 0 the function shall wipe the board, and insert the L1-_Packet waitingPacket onto the WaitingList of the Hub again. Otherwise, it shall update the content of the L1_Packet waitingPacket with the content of the Black Board, and return it to its Task. If the Hub WaitingList is not empty, each waiting Packet is removed, their content updated with the content of the Black Board and then returned to their Tasks.

Parameters

hub	Pointer to a hub of type Black Board.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL
packet->ServiceID = L1_SID_PUT_TO_HUB
waitingPacket->ServiceID = L1_SID_GET_FROM_HUB

Postcondition

packet->Status = RC_OK

6.23.4.3 void BlackBoardHub_Update (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function performs a blackboard update operation. The packet and blackboard have to be updated when a message is written to it (put packet), and when data is read (get Packet). When receiving a Put packet, the data of the packet is copied to the blackboard as a message, the size of the data in the blackboard is updated and the number of messages increases. When receiving a Get packet, the data from the blackboard is copied to the packet. The message is kept in the blackboard until a wipe operation occurs.

Parameters

Hub	ID which identifies the blackboard hub to be updated.
Packet	Put or Get packet that updates the blackboard..

Precondition

None

Postcondition

None

6.23.4.4 static __inline__ L1_ReturnCode L1_Drv_Isr_UpdateBlackBoard_NW (L1_HubID *hubID,* L1_Packet * *packet,* L1_BYTE * *message,* L1_UINT32 *messageSize*) [static]

Writes a message onto the Black Board. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.
packet	Pointer to an initialised L1_Packet to be used for this interaction.
message	Pointer to the message to write onto the Black Board.
messageSize	Size of the message that should be written onto the Black Board. The size of the message must be less or equal 'L1_PACKET_SIZE - sizeof(L1_UINT32)', otherwise the interaction will fail.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

Warning

Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

SPC Packets from an ISR

Remarks

SPC Data size range

6.23.4.5 static __inline__ L1_ReturnCode L1_Drv_Isr_UpdateDataEvent_NW (L1_HubID *hubID*, L1_Packet * *packet*, L1_BYTE * *data*, L1_UINT32 *length*) [static]

This interaction posts data to the DataEvent-Hub id. If the DataEvent-Hub previously contained data already then this will be overwritten, as it has been succeeded by the new data. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters

hubID	ID of the DataEvent-Hub to which to post the data.
packet	Pointer to an initialised L1_Packet to be used for this interaction.
data	Pointer to the buffer which contains the data to be posted.
length	Number of bytes that are in the buffer pointed to by data. The maximum number of bytes that can be posted is L1_PACKET_DATA_SIZE.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

Warning

Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

SPC Packets from an ISR

Remarks

SPC Data size range

6.23.4.6 static __inline__ L1_BOOL L1_isBlackBoardHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a BlackBoard-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a BlackBoard-Hub.

Returns

L1_TRUE if the data structure represents a BlackBoard-Hub.
L1_FALSE otherwise.

6.23.4.7 L1_ReturnCode L1_ReadBlackBoard (L1_HubID *hubID*, L1_Packet * *packet*, L1_BYTE * *messageBuffer*, L1_UINT32 *bufferSize*, L1_UINT32 * *receivedMessageSize*, L1_UINT32 * *messageNumber*, L1_Timeout *Timeout*)

Reads a message from a Black Board Hub and copies it into a buffer.

Parameters

hubID	ID of the Black Board Hub.
packet	Packet to be used for the interaction. The data and data size fields of this packet are copied into the message buffer.
messageBuffer	Pointer to the buffer where to store the message retrieved from the Black Board Hub. If NULL this function returns RC_FAIL;
bufferSize	size of the buffer, must be greater or equal to the message retrieved from the Black Board Hub, otherwise RC_FAIL will be returned.
received-MessageSize	Pointer to a variable of type L1_UINT32 where the size in byte of the retrieved message will be stored. This parameter may be set to NULL if this information is not desired.
message-Number	Pointer to a variable of type L1_UINT32 where the number of the message will be stored. The message number gets incremented by the Black Board Hub every time a Task writes a message onto the board.
Timeout	Timeout value for the operation. When set to max, timeout is disabled.

Returns

L1_ReturnCode:

- RC_OK operation successful
- RC_FAIL operation failed
- RC_TO operation timed out

Precondition

- None

Postcondition

- None

6.23.4.8 static __inline__ L1_ReturnCode L1_ReadBlackBoard_NW (L1_HubID *hubID,* L1_BYTE * *messageBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *receivedMessageSize,* L1_UINT32 * *messageNumber*) [static]

Reads a message from a Black Board Hub and copies it into a buffer.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.
messageBuffer	pointer to the buffer where to store the message retrieved from the Black Board Hub. If NULL this interaction will return RC_FAIL;
bufferSize	size of the buffer, must be greater or equal to the message retrieved from the Black Board Hub, otherwise RC_FAIL will be returned.
received-MessageSize	pointer to a variable of type L1_UINT32 where the size in byte of the retrieved message will be stored. This parameter may be set to NULL if this information is not desired.
message-Number	pointer to a variable of type L1_UINT32 where the number of the message will be stored. The message number gets incremented by the Black Board Hub every time a Task writes a message onto the board.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

6.23.4.9 static __inline__ L1_ReturnCode L1_ReadBlackBoard_W (L1_HubID *hubID*, L1_BYTE * *messageBuffer*, L1_UINT32 *bufferSize*, L1_UINT32 * *receivedMessageSize*, L1_UINT32 * *messageNumber*) [static]

Reads a message from a Black Board Hub and copies it into a buffer.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.
messageBuffer	pointer to the buffer where to store the message retrieved from the Black Board Hub. If NULL this interaction will return RC_FAIL;
bufferSize	size of the buffer, must be greater or equal to the message retrieved from the Black Board Hub, otherwise RC_FAIL will be returned.
receivedMessageSize	pointer to a variable of type L1_UINT32 where the size in byte of the retrieved message will be stored. This parameter may be set to NULL if this information is not desired.
messageNumber	pointer to a variable of type L1_UINT32 where the number of the message will be stored. The message number gets incremented by the Black Board Hub every time a Task writes a message onto the board.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

6.23.4.10 static __inline__ L1_ReturnCode L1_ReadBlackBoard_WT (L1_HubID *hubID*, L1_BYTE * *messageBuffer*, L1_UINT32 *bufferSize*, L1_UINT32 * *receivedMessageSize*, L1_UINT32 * *messageNumber*, L1_Timeout *timeout*) [static]

Reads a message from a Black Board Hub and copies it into a buffer.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.
messageBuffer	pointer to the buffer where to store the message retrieved from the Black Board Hub. If NULL this interaction will return RC_FAIL;
bufferSize	size of the buffer, must be greater or equal to the message retrieved from the Black Board Hub, otherwise RC_FAIL will be returned.
receivedMessageSize	pointer to a variable of type L1_UINT32 where the size in byte of the retrieved message will be stored. This parameter may be set to NULL if this information is not desired.
messageNumber	pointer to a variable of type L1_UINT32 where the number of the message will be stored. The message number gets incremented by the Black Board Hub every time a Task writes a message onto the board.

timeout	of type L1_Timeout, the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed
- RC_TO service timed out

Precondition

- None

Postcondition

- None

6.23.4.11 L1_ReturnCode L1_UpdateBlackBoard (L1_HubID *hubID,* L1_Packet * *packet,* L1_BYTE * *message,* L1_UINT32 *messageSize,* L1_Timeout *Timeout*)

Writes a message onto the Black Board. This function copies the message into the packet and inserts it into the hub with the specified timeout.

Parameters

hubID	ID of the identifies the blackboard hub.
packet	Packet to be used to write message to blackboard hub.
message	Pointer to the message to write onto the Black Board.
messageSize	Size of the message that should be written onto the Black Board. The size of the message must be less or equal 'L1_PACKET_SIZE - sizeof(L1_UINT32)', otherwise the interaction will fail.
Timeout	Timeout value the operation.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed
- RC_TO service timed out

Precondition

- None

Postcondition

- None

6.23.4.12 static __inline__ L1_ReturnCode L1_UpdateBlackBoard_NW (L1_HubID *hubID,* L1_BYTE * *message,* L1_UINT32 *messageSize*) [static]

Writes a message onto the Black Board.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.
message	Pointer to the message to write onto the Black Board.
messageSize	Size of the message that should be written onto the Black Board. The size of the message must be less or equal 'L1_PACKET_SIZE - sizeof(L1_UINT32)', otherwise the interaction will fail.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

6.23.4.13 L1_ReturnCode L1_WipeBoard (L1_HubID *hubID,* L1_Packet * *packet,* L1_Timeout *Timeout*)

This function erases the message from the black board. Any Task trying to copy the message from the black board will be put onto the waiting list. Interactions with the blackboard can have an infinite timeout, or be set to expire after a specific time.

Parameters

hubID	ID which identifies the blackboard hub.
packet	Message to be written to the blackboard.
Timeout	Amount of milliseconds to wait for the interaction to expire. Can be set to the max value to wait forever.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed
- RC_TO service timed out

Precondition

- None

Postcondition

- None

6.23.4.14 static __inline__ L1_ReturnCode L1_WipeBoard_NW (L1_HubID *hubID*) [static]

This interaction erases the message from the black board. Any Task trying to copy the message from the black board will be put onto the waiting list.

Parameters

hubID	of type L1_HubID, which identifies the Black Board Hub.

Returns

L1_ReturnCode:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- None

Postcondition

- None

6.24 Data Event Hub

Data Structures

- struct L1_DataEvent_HubState

Functions

- void DataEventHub_Ioctl (L1_Hub ∗Hub, L1_Packet ∗Packet, L1_BYTE ioctl_type)
- L1_BOOL DataEventHub_SyncCondition (L1_Hub ∗Hub, L1_Packet ∗Packet)
- void DataEventHub_Synchronize (L1_Hub ∗hub, L1_Packet ∗packet, L1_Packet ∗waitingPacket)
- void DataEventHub_Update (L1_Hub ∗Hub, L1_Packet ∗Packet)
- static __inline__ L1_ReturnCode L1_ClearDataEvent_NW (L1_HubID id)
- static __inline__ L1_ReturnCode L1_ReadDataEvent_NW (L1_HubID id, L1_BYTE ∗data, L1_UINT32 length, L1_UINT32 ∗pNbrOfReadBytes)
- static __inline__ L1_ReturnCode L1_ReadDataEvent_W (L1_HubID id, L1_BYTE ∗data, L1_UINT32 length, L1_UINT32 ∗pNbrOfReadBytes)
- static __inline__ L1_ReturnCode L1_ReadDataEvent_WT (L1_HubID id, L1_BYTE ∗data, L1_UINT32 length, L1_UINT32 ∗pNbrOfReadBytes, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_UpdateDataEvent_NW (L1_HubID id, L1_BYTE ∗data, L1_UINT32 length)

6.24.1 Detailed Description

The DataEvent-Hub is a crossover between am Event-Hub and a Port-Hub. It's usage scenario is to efficiently transfer information about state changes from a Sender-Task to a Receiver-Task. For this purpose the Hub allows the Sender-Task to post data together with raising and Event. Upon a data post, the Hub will wake up the first Task on it's waiting list and pass the data to it as well. If there is currently no Task on the waiting list, the Hub will store the data in an internal buffer and remember that data was posted to it. Upon a Task reading the data the Hub will forget the previously pending data. Thus the Hub performs an auto-reset.

Remarks

```
REQ DataEvent
SPC Data Event
```

6.24.2 Visual Designer

Figure 6.3: Application Diagram Icon

6.24.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.

6.24.3 Function Documentation

6.24.3.1 void DataEventHub_Ioctl (L1_Hub * *Hub*, L1_Packet * *Packet*, L1_BYTE *ioctl_type*)

This function shall handle the initialisation and the clearing of the DataEvent Hub.

Parameters

Hub	Pointer to a hub of type L1_DATAEVENT.
Packet	Pointer to the L1_Packet with the request.
ioctl_type	The type of IOCTL Operation requested. It may have only the following values: • L1_IOCTL_HUB_OPEN Brings the Hub to it's default state.

6.24.3.2 L1_BOOL DataEventHub_SyncCondition (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function shall check whether or not the incoming request results in a sychronisation or not.

Parameters

Hub	Pointer to a hub of type L1_DATAEVENT.
Packet	Pointer to the L1_Packet with the request.

Returns

L1_TRUE Synchronisation was achieved.
L1_FALSE Synchronisation was not achieved.

6.24.3.3 void DataEventHub_Synchronize (L1_Hub * *hub*, L1_Packet * *packet*, L1_Packet * *waitingPacket*)

This function shall transfer the data of the L1_Packet packet to the L1_Packet waitingPacket, and then return both L1_Packets to their Tasks.

Parameters

hub	Pointer to a hub of type L1_DATAEVENT.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL
packet is a Put-Packet
waitingPacket is a Get-Packet

Postcondition

waitingPacket->Data = packet->Data
waitingPacket->DataSize = packet->DataSize
packet->Status = RC_OK
waitingPacket->Status = RC_OK

6.24.3.4 void DataEventHub_Update (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function shall update the state of the DataEvent Hub based on the content of the L1_Packet Packet. If Packet is a Put-Packet then the data gets copied into an internal buffer, together with the data-size, and the DataEvent is marked as raised. If Packet is a Get-Packet then the internal buffer contents get copied to Packet together with the data-size, and the DataEvent is marked as not raised.

Parameters

Hub	Pointer to a hub of type L1_DATAEVENT.
Packet	Packet Pointer to the L1_Packet with the request.

6.24.3.5 static __inline__ L1_ReturnCode L1_ClearDataEvent_NW (L1_HubID *id*) [static]

This interaction clears the content of a DataEvent-Hub. This means the Hub forgets that data has been posted to it previously.

Parameters

id	ID of the DataEvent-Hub to clear.

Returns

RC_OK The DataEvent-Hub could be cleared.
RC_FAIL Otherwise.

6.24.3.6 static __inline__ L1_ReturnCode L1_ReadDataEvent_NW (L1_HubID *id*, L1_BYTE * *data*, L1_UINT32 *length*, L1_UINT32 * *pNbrOfReadBytes*) [static]

This interaction tries to read data from a DataEvent-Hub. This interaction always returns directly.

Parameters

id	ID of the DataEvent-Hub from which to read data.
data	Pointer to the buffer which should contain the data that was read from the DataEvent--Hub.
length	The size of the buffer pointed to by data in Bytes. It must be equal or larger than the data posted to the DataEvent-Hub, otherwise the interaction will fail with RC_FAIL.
pNbrOfRead-Bytes	Pointer to a variable of type L1_UINT32 which will contain the number of Bytes that were actually read from the DataEvent-Hub.

Returns

RC_OK The data could be read.
RC_FAIL The data could not be read. One cause of this is a too small buffer.

6.24.3.7 static __inline__ L1_ReturnCode L1_ReadDataEvent_W (L1_HubID *id*, L1_BYTE * *data*, L1_UINT32 *length*, L1_UINT32 * *pNbrOfReadBytes*) [static]

This interaction tries to read data from a DataEvent-Hub. If the DataEvent-Hub is currently holding data the interaction will return directly, otherwise it will wait until another Task posts data to the Hub.

Parameters

id	ID of the DataEvent-Hub from which to read data.
data	Pointer to the buffer which should contain the data that was read from the DataEvent--Hub.
length	The size of the buffer pointed to by data in Bytes. It must be equal or larger than the data posted to the DataEvent-Hub, otherwise the interaction will fail with RC_FAIL.
pNbrOfRead-Bytes	Pointer to a variable of type L1_UINT32 which will contain the number of Bytes that were actually read from the DataEvent-Hub.

Returns

RC_OK The data could be read.
RC_FAIL The data could not be read. One cause of this is a too small buffer.

6.24.3.8 static __inline__ L1_ReturnCode L1_ReadDataEvent_WT (L1_HubID *id*, L1_BYTE * *data*, L1_UINT32 *length*, L1_UINT32 * *pNbrOfReadBytes*, L1_Timeout *timeout*) [static]

This interaction tries to read data from a DataEvent-Hub. If the DataEvent-Hub is currently holding data the interaction will return directly, otherwise it will wait until either another Task posts data to the Hub, or the timeout expires.

Parameters

id	ID of the DataEvent-Hub from which to read data.
data	Pointer to the buffer which should contain the data that was read from the DataEvent--Hub.
length	The size of the buffer pointed to by data in Bytes. It must be equal or larger than the data posted to the DataEvent-Hub, otherwise the interaction will fail with RC_FAIL.
pNbrOfRead-Bytes	Pointer to a variable of type L1_UINT32 which will contain the number of Bytes that were actually read from the DataEvent-Hub.
timeout	How long the interaction shall wait, in system ticks, to perform the interaction.

Returns

RC_OK The data could be read.
RC_TO The timeout expired.
RC_FAIL The data could not be read. One cause of this is a too small buffer.

6.24.3.9 static __inline__ L1_ReturnCode L1_UpdateDataEvent_NW (L1_HubID *id*, L1_BYTE * *data*, L1_UINT32 *length*) [static]

This interaction posts data to the DataEvent-Hub id. If the DataEvent-Hub previously contained data already then this will be overwritten, as it has been succeeded by the new data. This interaction always returns directly.

Parameters

id	ID of the DataEvent-Hub to which to post the data.
data	Pointer to the buffer which contains the data to be posted.
length	Number of bytes that are in the buffer pointed to by data. The maximum number of bytes that can be posted is L1_PACKET_DATA_SIZE.

Returns

RC_OK The data could be posted.
RC_FAIL Otherwise.

6.25 Data-Queue Hub

Data Structures

- struct _struct_L1_DataQueueElement_
- struct _struct_L1_DataQueueState_

Typedefs

- typedef struct _struct_L1_DataQueueState_ L1_DataQueue_HubState
- typedef struct _struct_L1_DataQueueElement_ L1_DataQueueElement

Functions

- L1_BOOL DataQueueHub_SyncCondition (L1_Hub *Hub, L1_Packet *Packet)
- void DataQueueHub_Synchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)
- void DataQueueHub_Update (L1_Hub *Hub, L1_Packet *Packet)
- L1_ReturnCode L1_DataQueue_get (L1_HubID hubID, L1_Packet *packet, L1_BYTE *buffer, L1_UINT32 bufferSize, L1_UINT32 *pNbrOfBytesRetrieved, L1_Timeout timeout)
- L1_ReturnCode L1_DataQueue_put (L1_HubID hubID, L1_Packet *packet, L1_BYTE *buffer, L1_UINT32 length, L1_BOOL urgent, L1_Timeout timeout)
- static __inline__ L1_BOOL L1_isDataQueueHub (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isDataQueueHubEmpty (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isDataQueueHubFull (L1_Hub *pHub)

6.25.1 Detailed Description

The DataQueue-Hub offers buffered data exchange between Tasks.

6.25.2 Visual Designer

Figure 6.4: Application Diagram Icon

6.25.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- nbrOfElements: How many elements can be stored in the data-queue.
- elementSize: The size of each element stored in the data-queue.

6.25.3 Typedef Documentation

6.25.3.1 typedef struct _struct_L1_DataQueueState_ L1_DataQueue_HubState

State of a DataQueue-Hub.

6.25.3.2 typedef struct _struct_L1_DataQueueElement_ L1_DataQueueElement

This structure represents an Element in the FIFO-Hub. It buffers the data given to it.

6.25.4 Function Documentation

6.25.4.1 L1_BOOL DataQueueHub_SyncCondition (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function signals if synchronization occurs when receiving put/get packets in a fifo. When receiving a put packet, returns true when the fifo is not full (more packets can be received). When receiving a get packet, returns true when the fifo is not empty (there is data to be read from the fifo).

Parameters

Hub	ID of the Fifo Hub.
Packet	Packet used for the synchronization operation.

Returns

L1_TRUE If synchronisation took place.
L1_FALSE If no synchronisation took place.

6.25.4.2 void DataQueueHub_Synchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall ensure that the data from the Put-Packet gets inserted into the FIFO and that the Get--Packet gets the oldest set of data that is in the FIFO.

Parameters

hub	Pointer to a hub of type L1_FIFO.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL

6.25.4.3 void DataQueueHub_Update (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function updates the state of a Fifo Hub, depending on the type of packet received. Put packets increase the count of the Fifo, data from the packet is copied into the fifo buffer and the data size is updated. Get packets decrease the count of the Fifo and copies the data from the hub's buffer to the data field of the packet.

Parameters

Hub	ID of the Fifo Hub.
Packet	Packet used for the update operation.

6.25.4.4 L1_ReturnCode L1_DataQueue_get (L1_HubID *hubID,* L1_Packet * *packet,* L1_BYTE * *buffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *pNbrOfBytesRetrieved,* L1_Timeout *timeout*)

Parameters

hubID	The ID of the Data-Queue Hub where to enqueue the the data at.
packet	Pointer to the L1_Packet to use for the interaction.
buffer	Pointer to the memory location where the data retrieved from the Data-Queue Hub shall be placed.
bufferSize	The number of bytes that can be stored in the buffer, pointed to by the parameter buffer.
pNbrOfBytes-Retrieved	Pointer to a variable of type L1_UINT32 where the number of retrieved bytes shall be stored after the interaction completed successfully. The value shall be less or equal to the value stored in the parameter bufferSize.
timeout	How long to wait for this interaction to synchronise if the Data-Queue Hub has no occupied element available. The parameter has the following meanings: • 0: Non waiting (_NW) semantics. • L1_INFINITE_TIMEOUT: Waiting semantics (_W). The interaction waits until it synchronised, potentially forever or until the system terminates. • Any other value: The number of milliseconds to wait for the interaction to synchronise (_WT).

Returns

RC_OK: The interaction was performed successfully, the data has been inserted in the Data-Queue.
RC_TO: The interaction timed out, the data has not been inserted in the Data-Queue
RC_FAIL: The interaction failed, there are multiple reasons for this possible:

- When the timeout parameter is set to 0 (_NW) and there was no free element available in the Data-Queue Hub.
- The amount of data to be retrieved from the Data-Queue Hub was smaller than the Data-Queue Hub element size.
- The amount of data to be retrieved from the Data-Queue Hub was larger than the payload size of the L1_Packet minus one byte (L1_PACKET_DATA_SIZE - 1).
- The parameter buffer was a NULL-Pointer.
- The parameter packet was a NULL-Pointer.
- The parameter pNbrOfBytesRetrieved was a NULL-Pointer.

6.25.4.5 L1_ReturnCode L1_DataQueue_put (L1_HubID *hubID,* L1_Packet * *packet,* L1_BYTE * *buffer,* L1_UINT32 *length,* L1_BOOL *urgent,* L1_Timeout *timeout*)

Parameters

hubID	The ID of the Data-Queue Hub where to enqueue the the data at.
packet	Pointer to the L1_Packet to use for the interaction.
buffer	Pointer to the memory location that contains the data to be enqueued.
length	The number of bytes to enqueue at the Data-Queue Hub.
urgent	This indicates whether or not the data is urgent. If set to L1_TRUE the element gets inserted at the front of the queue, otherwise at the end. Be aware that this flag does not affect the priority of the L1_Packet in the system.
timeout	How long to wait for this interaction to synchronise if the Data-Queue Hub has no free element available. The parameter has the following meanings: • 0: Non waiting (_NW) semantics. • L1_INFINITE_TIMEOUT: Waiting semantics (_W). The interaction waits until it synchronised, potentially forever or until the system terminates. • Any other value: The number of milliseconds to wait for the interaction to synchronise (_WT).

Returns

RC_OK: The interaction was performed successfully, the data has been inserted in the Data-Queue.
RC_TO: The interaction timed out, the data has not been inserted in the Data-Queue
RC_FAIL: The interaction failed, there are multiple reasons for this possible:

- When the timeout parameter is set to 0 (_NW) and there was no free element available in the Data-Queue Hub.
- The amount of data to be inserted in the Data-Queue Hub was larger than the Data-Queue Hub element size.
- The amount of data to be inserted in the Data-Queue Hub was larger than the payload size of the L1_Packet minus one byte (L1_PACKET_DATA_SIZE - 1).
- The parameter buffer was a NULL-Pointer.
- The parameter packet was a NULL-Pointer.

6.25.4.6 static __inline__ L1_BOOL L1_isDataQueueHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a FIFO-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a FIFO-Hub.

Returns

L1_TRUE If the data structure represents a FIFO-Hub.
L1_FALSE Otherwise.

6.25.4.7 static __inline__ L1_BOOL L1_isDataQueueHubEmpty (L1_Hub * *pHub*) [static]

Determines whether the FIFO-Hub identified by pHub is empty.

Parameters

pHub	Pointer to the Hub data structure of a FIFO-Hub.

Returns

L1_TRUE If the FIFO is empty.
L1_FALSE If the FIFO is not empty.

6.25.4.8 static __inline__ L1_BOOL L1_isDataQueueHubFull (L1_Hub * *pHub*) [static]

Determines whether the FIFO-Hub identified by pHub is full.

Parameters

pHub	Pointer to the Hub data structure of a FIFO-Hub.

Returns

L1_TRUE If the FIFO is full.
L1_FALSE If the FIFO is not full.

Remarks

SPC FIFO-nbrOfElement upper bound

6.26 Event Hub

Data Structures

- struct _struct_L1_EventState_

Macros

- #define L1_Event_State(h) ((L1_Event_HubState*)(h)->HubState)

Typedefs

- typedef struct _struct_L1_EventState_ L1_Event_HubState

Functions

- L1_BOOL EventSyncCondition (L1_Hub ∗Hub, L1_Packet ∗Packet)
- void EventUpdate (L1_Hub ∗Hub, L1_Packet ∗Packet)
- static __inline__ L1_ReturnCode L1_Drv_Isr_RaiseEvent_NW (L1_HubID event, L1_Packet ∗packet)
- static __inline__ L1_BOOL L1_isEventHub (L1_Hub ∗pHub)
- static __inline__ L1_BOOL L1_isHubEventSet (L1_Hub ∗pHub)
- static __inline__ L1_ReturnCode L1_RaiseEvent_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_RaiseEvent_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_RaiseEvent_WT (L1_HubID HubID, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_TestEvent_A (L1_HubID HubID, L1_Packet ∗packet)
- static __inline__ L1_ReturnCode L1_TestEvent_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_TestEvent_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_TestEvent_WT (L1_HubID HubID, L1_Timeout timeout)

6.26.1 Detailed Description

An Event-Hub synchronises two Tasks based on a boolean condition.

6.26.2 Visual Designer

Figure 6.5: Application Diagram Icon

6.26.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.

6.26.3 Example

This example illustrates the use of the Event Hub. Task1 periodically raises the Event Event1 on which the Task2 is waiting. When the Event is raised the waiting Task2 will receive a RC_OK return value.

The program uses the L1_TestEvent_W and L1_RaiseEvent_W waiting kernel services.

6.26.3.1 Entities

- Task1: Task1EntryPoint, shown in section Source Code of Task1EntryPoint
- Task2: Task2EntryPoint, shown in section Source Code of Task2EntryPoint
- Event1: The Event Hub used to synchronise between Task1 and Task2.
- StdioHostServer1: Stdio Host Server used to print messages onto the screen.

- StdioHostServer1Res: Resource Lock used to prevent disruptions while printing messages onto the console using StdioHostServer1.

6.26.4 Source Code of Task1EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>

void Task1EntryPoint(L1_TaskArguments Arguments)
{
    L1_INT32 EventCounter = 0;
    while(1)
    {
        // Here Event1 gets raised.
        if(RC_OK == L1_RaiseEvent_W(Event1))
        {
            L1_LockResource_W(StdioHostServer1Res);
            Shs_putString_W(StdioHostServer1, "Task1 raised the Event1 N \n");
            Shs_putInt_W(StdioHostServer1, EventCounter++, 'd');
            Shs_putChar_W(StdioHostServer1, '\n');
            L1_UnlockResource_NW(StdioHostServer1Res);
        }
    }
}
```

6.26.5 Source Code of Task2EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>

void Task2EntryPoint(L1_TaskArguments Arguments)
{
    L1_INT32 EventCounter = 0;

    while(1)
    {
    // Here Event1 gets tested.
        if(RC_OK == L1_TestEvent_W(Event1))
        {
            L1_LockResource_W(StdioHostServer1Res);
            Shs_putString_W(StdioHostServer1, "Task2 tested Event1 N ");
            Shs_putInt_W(StdioHostServer1, EventCounter++, 'd');
            Shs_putString_W(StdioHostServer1, " - synchronization is done\n");
            L1_UnlockResource_NW(StdioHostServer1Res);
        }
    }
}
```

Remarks

```
REQ Event
SPC Event
```

6.26.6 Macro Definition Documentation

6.26.6.1 #define L1_Event_State(*h*) ((L1_Event_HubState*)(h)->HubState)

This macro casts the HubState void pointer to a pointer to L1_Event_HubState.

Parameters

h	Pointer to a type of type L1_EVENT.

6.26.7 Typedef Documentation

6.26.7.1 typedef struct _struct_L1_EventState_ L1_Event_HubState

The state of an Event-Hub.

Remarks

```
SPC Event state variable
```

6.26.8 Function Documentation

6.26.8.1 L1_BOOL EventSyncCondition (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function evaluates if the update function should be executed, depending on the type of packet received. For send packets, the event sync condition is true if the event state is not set. If the event is set, the update condition is false. For receive packets, the event sync condition is true if the event state is set. If the event is not set, the update condition is false.

Parameters

Hub	Event hub that is tested for synchronization.
Packet	Send or receive packet received by the hub.

Returns

L1_BOOL

- L1_TRUE Update condition is true (update function should be called).
- L1_FALSE Update condition is false (update function should not be called).

Precondition

- Hub is of type L1_EVENT.
- Empty waiting list, as complimentary packets are already accounted for.

6.26.8.2 void EventUpdate (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function updates the state of an Event Hub, depending on the type of packet received. Send packets raise the event. Receive packets test the event.

Parameters

Hub	Event hub that is updated.
Packet	Send or receive packet received by the hub.

Precondition

- Hub is of type L1_EVENT.

6.26.8.3 static __inline__ L1_ReturnCode L1_Drv_Isr_RaiseEvent_NW (L1_HubID *event,* L1_Packet * *packet*) [static]

This interaction tries to raise an event from the ISR context. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters:

Parameters

event	ID of the Event-Hub that should be raised.
packet	Pointer to the L1_Packet that will be used to represent the interaction. This L1_Packet must have been once initialised using the function L1_Drv_Isr_initialisePacket().

Returns

RC_OK The packet that raises the Event could be inserted into the Kernel Input Port.
RC_FAIL The packet that raises the Event could not be inserted into the Kernel Input Port.

Warning

Must not be used with Event-Hubs located at another Node.
Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

SPC Packets from an ISR

6.26.8.4 static __inline__ L1_BOOL L1_isEventHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a Event-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a Event-Hub.

Returns

L1_TRUE if the data structure represents a Event-Hub.
L1_FALSE otherwise.

6.26.8.5 static __inline__ L1_BOOL L1_isHubEventSet (L1_Hub * *pHub*) [static]

This function determines whether or not the Event-Hub identified by the argument pHub is set.

Parameters

pHub	Pointer to a variable of type L1_Hub of an Event-Hub.

Warning

pHub must not be a NULL-Pointer, this function does not check this.

Returns

L1_TRUE if the Event-Hub is set.
L1_FALSE if the Event-Hub is not set.

6.26.8.6 static __inline__ L1_ReturnCode L1_RaiseEvent_NW (L1_HubID *HubID*) [static]

This service raises an Event from False to True. This service returns immediately independent of whether or not it could raise the event.

Parameters:

Parameters

HubID	is of type L1_HubID, identifies the Event, i.e. Hub, that the calling Task wants to raise.

Returns

L1_ReturnCode:

- RC_OK service successful (the Event has been raised)
- RC_FAIL service failed (the Event has not been raised)

Precondition

- Packet is the preallocated Packet
- Hub is of Event type

Postcondition

- Header fields of preallocated Packet filled in

6.26.8.7 static __inline__ L1_ReturnCode L1_RaiseEvent_W (L1_HubID *HubID*) [static]

This service raises an Event from False to True. If the Event is already set, wait.

Parameters:

Parameters

HubID	is of type L1_HubID, identifies the Event, i.e. Hub, that the calling Task wants to raise.

Returns

L1_ReturnCode:

- RC_OK service successful (the Event has been raised)
- RC_FAIL service failed (the Event has not been raised)

Precondition

- Packet is the preallocated Packet
- Hub is of Event type

Postcondition

- Header fields of preallocated Packet filled in

6.26.8.8 static __inline__ L1_ReturnCode L1_RaiseEvent_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

This service raises an Event from False to True. This call waits until either the event could be raised or the timeout expired.

Parameters:

Parameters

HubID	of type L1_HubID, identifies the Event, i.e. Hub, that the calling Task wants to raise.
timeout	of type L1_Timeout, the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode:

- RC_OK service successful (the Event has been raised)
- RC_FAIL service failed (the Event has not been raised)
- RC_TO service timed out.

Precondition

- Packet is the preallocated Packet
- Hub is of Event type

Postcondition

- Header fields of preallocated Packet filled in

6.26.8.9 static __inline__ L1_ReturnCode L1_TestEvent_A (L1_HubID *HubID*, L1_Packet * *packet*) [static]

Request to test an Event-Hub without being put in the waiting state. The completion is deferred until a corresponding L1_WaitForPacket call happens.

Parameters

HubID	is of type L1_HubID and identifies the Event, that the calling Task wants to test.
packet	Pointer to the L1_Packet that will be used for this Asynchronous-Interaction. There are two ways of acquiring such an L1_Packet: • Allocate an L1_Packet in your Task, and then initialise it using the function L1_initialiseAsyncPacket(). • Allocate an L1_Packet from a local (i.e. same Node) Packet Pool-Hub, using one of the following interactions: L1_AllocatePacket_W(), L1_AllocatePacket_NW(), and L1_AllocatePacket_WT().

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- Packet is a preallocated L1_Packet, initialised using L1_initialiseAsyncPacket().

Postcondition

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

Warning

This Interaction only works locally (Task and Hub on the same Node), i.e. not distributed (Task and Hub on different Nodes).

See Also

L1_initialiseAsyncPacket
L1_AllocatePacket_W
L1_AllocatePacket_NW
L1_AllocatePacket_WT

Remarks

SPC Runtime parameter check

6.26.8.10 static __inline__ L1_ReturnCode L1_TestEvent_NW (L1_HubID *HubID*) [static]

This service tests an Event. Returns immediately.

Parameters

HubID	is of type L1_HubID and identifies the Event, that the calling Task wants to test.

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful (there was a set Event)
- RC_FAIL service failed (there was no set Event)

Precondition

- Packet is the preallocated Packet

Postcondition

- Header fields of preallocated Packet filled in

6.26.8.11 static __inline__ L1_ReturnCode L1_TestEvent_W (L1_HubID *HubID*) [static]

This service tests an Event. This call waits until the Event has been signalled.

Parameters

HubID	is of type L1_HubID and identifies the Event, that the calling Task wants to test.

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful (there was a set Event)
- RC_FAIL service failed (there was no set Event)

Precondition

- Packet is the preallocated Packet

Postcondition

- Header fields of preallocated Packet filled in

6.26.8.12 static __inline__ L1_ReturnCode L1_TestEvent_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

This service tests an Event. This call waits until either the Event has been signalled, or the timeout expired.

Parameters

HubID	is of type L1_HubID and identifies the Event, that the calling Task wants to test.
timeout	of type L1_Timeout, the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful (there was a set Event).
- RC_FAIL service failed (there was no set Event).
- RC_TO timeout expired.

Precondition

- Packet is the preallocated Packet

Postcondition

- Header fields of preallocated Packet filled in

6.27 FIFO Hub

Data Structures

- struct _struct_L1_FifoState_

Typedefs

- typedef struct _struct_L1_FifoState_ L1_Fifo_HubState

Functions

- void Fifo_Ioctl (L1_Hub ∗Hub, L1_Packet ∗Packet, L1_BYTE ioctl_type)
- L1_BOOL FifoSyncCondition (L1_Hub ∗Hub, L1_Packet ∗Packet)
- void FifoSynchronize (L1_Hub ∗hub, L1_Packet ∗packet, L1_Packet ∗waitingPacket)
- void FifoUpdate (L1_Hub ∗Hub, L1_Packet ∗Packet)
- static __inline__ L1_ReturnCode L1_DequeueFifo_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_DequeueFifo_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_DequeueFifo_WT (L1_HubID HubID, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_Drv_Isr_EnqueueFifo_NW (L1_HubID fifo, L1_Packet ∗packet)
- static __inline__ L1_ReturnCode L1_EnqueueFifo_NW (L1_HubID HubID)

- static __inline__ L1_ReturnCode L1_EnqueueFifo_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_EnqueueFifo_WT (L1_HubID HubID, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_GetDataFromFifo_NW (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived)
- static __inline__ L1_ReturnCode L1_GetDataFromFifo_W (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived)
- static __inline__ L1_ReturnCode L1_GetDataFromFifo_WT (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived, L1_Timeout timeout)
- static __inline__ L1_BOOL L1_isFifoHub (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isHubFifoEmpty (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isHubFifoFull (L1_Hub *pHub)
- static __inline__ L1_ReturnCode L1_PutDataToFifo_NW (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size)
- static __inline__ L1_ReturnCode L1_PutDataToFifo_W (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size)
- static __inline__ L1_ReturnCode L1_PutDataToFifo_WT (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size, L1_Timeout timeout)

6.27.1 Detailed Description

The FIFO-Hub offers buffered data exchange between Tasks.

6.27.2 Visual Designer

Figure 6.6: Application Diagram Icon

6.27.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- size: How many buffers the FIFO provides.

6.27.3 Example

This example illustrates the use of the FIFO Hub. Task1 puts a character into a packet and sends this to FIFO1. Task2 initially waits for 2 seconds for the FIFO to fill up and then retrieves the packets from FIFO1 and displays their content.

6.27.3.1 Entities

- FIFO1: The FIFO bugger between Task1 and Task2, it can store 5 elements.
- Task1: Task1EntryPoint, shown in section Source Code of Task1EntryPoint
- Task1: Task2EntryPoint, shown in section Source Code of Task2EntryPoint
- StdioHostServer1: A Stdio Host Server component which provides access to the console.
- StdioHostServer1Res: A Resource Hub to ensure that a second task does not interfere with console access.

6.27.4 Source Code of Task1EntryPoint

```
void Task1EntryPoint(L1_TaskArguments Arguments)
{
    L1_BYTE ch;
    L1_Packet *Packet = L1_CurrentTaskCR->RequestPacket;

    for(L1_UINT32 i=0; i<5; i++)
    {
        for(ch = 'a'; ch < 'j'; ch++)
        {
            Packet->DataSize = sizeof(L1_BYTE);
            Packet->Data[0] = ch;

            if(RC_OK == L1_EnqueueFifo_W(FIFO1))
            {
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1,
                                "The Task1 put into the FIFO1 the symbol ");
                Shs_putChar_W(StdioHostServer1, ch);
                Shs_putChar_W(StdioHostServer1, '\n');
                L1_UnlockResource_NW(StdioHostServer1Res);
            }else
            {
                Shs_putString_W(StdioHostServer1,
                                "A symbol is not put by Task1 into FIFO1\n");
            }
        }
    }
}
```

6.27.5 Source Code of Task2EntryPoint

```
#include <L1_api.h>
#include "L1_node_config.h"
#include <StdioHostService/StdioHostClient.h>

void Task2EntryPoint(L1_TaskArguments Arguments)
{
    L1_Packet *Packet = L1_CurrentTaskCR->RequestPacket;
```

```
    L1_BYTE i, ch;

    while(1)
    {
        L1_LockResource_W(StdioHostServer1Res);
        Shs_putString_W(StdioHostServer1, "Task2 sleeps for 2 s waiting for the FIFO
        L1_UnlockResource_NW(StdioHostServer1Res);
        L1_WaitTask_WT(2000);

        for(i = 'a'; i < 'j'; i++)
        {
            if(RC_OK == L1_DequeueFifo_W(FIFO1))
            {
                ch = Packet->Data[0];
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1,
                                "The Task2 read from the FIFO1 the symbol ");
                Shs_putChar_W(StdioHostServer1, ch);
                Shs_putChar_W(StdioHostServer1, '\n');
                L1_UnlockResource_NW(StdioHostServer1Res);
            }else
            {
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1,
                                "A symbol is not read by Task2 from FIFO1\n");
                L1_UnlockResource_NW(StdioHostServer1Res);
            }
        }
    }
}
```

Remarks

REQ FIFO
SPC FIFO

6.27.6 Typedef Documentation

6.27.6.1 typedef struct _struct_L1_FifoState_ L1_Fifo_HubState

State of a FIFO-Hub.

Remarks

SPC FIFO state variables

6.27.7 Function Documentation

6.27.7.1 void Fifo_Ioctl (L1_Hub * *Hub*, L1_Packet * *Packet*, L1_BYTE *ioctl_type*)

This function shall handle the initialisation and the clearing of the FIFO Hub.

Parameters

Hub	Pointer to a hub of type L1_FIFO.
Packet	Pointer to the L1_Packet with the request.
ioctl_type	The type of IOCTL Operation requested. It may have only the following values: • L1_IOCTL_HUB_OPEN Brings the Hub to it's default state.

6.27.7.2 L1_BOOL FifoSyncCondition (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function signals if synchronization occurs when receiving put/get packets in a fifo. When receiving a put packet, returns true when the fifo is not full (more packets can be received). When receiving a get packet, returns true when the fifo is not empty (there is data to be read from the fifo).

Parameters

Hub	ID of the Fifo Hub.
Packet	Packet used for the synchronization operation.

Returns

L1_TRUE If synchronisation took place.
L1_FALSE If no synchronisation took place.

6.27.7.3 void FifoSynchronize (L1_Hub * *hub*, L1_Packet * *packet*, L1_Packet * *waitingPacket*)

This function shall ensure that the data from the Put-Packet gets inserted into the FIFO and that the Get--Packet gets the oldest set of data that is in the FIFO.

Parameters

hub	Pointer to a hub of type L1_FIFO.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL

6.27.7.4 void FifoUpdate (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function updates the state of a Fifo Hub, depending on the type of packet received. Put packets increase the count of the Fifo, data from the packet is copied into the fifo buffer and the data size is updated. Get packets decrease the count of the Fifo and copies the data from the hub's buffer to the data field of the packet.

Parameters

Hub	ID of the Fifo Hub.
Packet	Packet used for the update operation.

6.27.7.5 static __inline__ L1_ReturnCode L1_DequeueFifo_NW (L1_HubID *HubID*) [static]

Retrieves data from a FIFO, the data is stored in the payload of the task's Request-Packet.

This call returns immediately, even if there is no packet available in the FIFO.

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)

Precondition

- None

Postcondition

- Calling task ready

6.27.7.6 static __inline__ L1_ReturnCode L1_DequeueFifo_W (L1_HubID *HubID*) [static]

Retrieves data from a FIFO, the data is stored in the payload of the task's Request-Packet. This call waits until there is data in the FIFO to be retrieved.

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)

Precondition

- None

Postcondition

- Calling task ready

6.27.7.7 static __inline__ L1_ReturnCode L1_DequeueFifo_WT (L1_HubID *HubID*, L1_Timeout *timeout*) [static]

Retrieves data from a FIFO, the data is stored in the payload of the task's Request-Packet. Waits until either data becomes available or the timeout expired, depending on what happens first.

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	the L1_HubID which identifies the FIFO-Hub to use.
timeout	the number of system ticks the call should wait for a packet to become available.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)
- RC_TO the timeout expired without a package being available.

Precondition

- None

Postcondition

- Calling task ready

6.27.7.8 static __inline__ L1_ReturnCode L1_Drv_Isr_EnqueueFifo_NW (L1_HubID *fifo*, L1_Packet * *packet*) [static]

Enqueues a fifo. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters:

Parameters

fifo	is the L1_HubID which identifies the Fifo, that the calling ISR wants to enqueue.
packet	Pointer to the L1_Packet that will be used to represent the interaction. This L1_Packet must have been once initialized using the function L1_Drv_Isr_initialisePacket().

Returns

RC_OK The packet that enqueues the fifo could be inserted into the Kernel Input Port.
RC_FAIL The packet that enqueues the fifo could not be inserted into the Kernel Input Port.

Warning

Must not be used with Fifo-Hubs located at another Node.
Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

SPC Packets from an ISR

6.27.7.9 static __inline__ L1_ReturnCode L1_EnqueueFifo_NW (L1_HubID *HubID*) [static]

Inserts the payload-data of a task's Request-Packet into a FIFO. This call returns immediately, even if the packet could not be enqueued in the FIFO.

Parameters

HubID	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)

Precondition

- None

Postcondition

- Calling task ready

6.27.7.10 static __inline__ L1_ReturnCode L1_EnqueueFifo_W (L1_HubID *HubID*) [static]

Inserts the payload-data of a task's Request-Packet into a FIFO. This service waits until it could insert the data into the specified FIFO.

Parameters

HubID	identifies the FIFO-Hub to use.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)

Precondition

- None

Postcondition

- Calling task ready

6.27.7.11 static __inline__ L1_ReturnCode L1_EnqueueFifo_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

Inserts the payload-data of a task's Request-Packet into a FIFO. This service tries to enqueue a packet into the FIFO until the the timeout expires.

Parameters

HubID	identifies the FIFO-Hub to use.
timeout	the number of system ticks the call should wait while trying to enqueue the packet.

Returns

L1_ReturnCode

- RC_OK service successful (the data was inserted in the FIFO)
- RC_FAIL service failed (the data was not inserted in the FIFO)
- RC_TO the timeout expired.

Precondition

- None

Postcondition

- Calling task ready

6.27.7.12 static __inline__ L1_ReturnCode L1_GetDataFromFifo_NW (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.
bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail. – The FIFO-Hub was empty, hence there is no data that could be retrieved.

6.27.7.13 static __inline__ L1_ReturnCode L1_GetDataFromFifo_W (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.
bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.

6.27.7.14 static __inline__ L1_ReturnCode L1_GetDataFromFifo_WT (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived,* L1_Timeout *timeout*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.
bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.
timeout	The timeout to use for the interaction with the Hub.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.
- RC_TO: The timeout of the interaction expired.

6.27.7.15 static __inline__ L1_BOOL L1_isFifoHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a FIFO-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a FIFO-Hub.

Returns

L1_TRUE If the data structure represents a FIFO-Hub.
L1_FALSE Otherwise.

6.27.7.16 static __inline__ L1_BOOL L1_isHubFifoEmpty (L1_Hub * *pHub*) [static]

Determines whether the FIFO-Hub identified by pHub is empty.

Parameters

pHub	Pointer to the Hub data structure of a FIFO-Hub.

Returns

L1_TRUE If the FIFO is empty.
L1_FALSE If the FIFO is not empty.

6.27.7.17 static __inline__ L1_BOOL L1_isHubFifoFull (L1_Hub * *pHub*) [static]

Determines whether the FIFO-Hub identified by pHub is full.

Parameters

pHub	Pointer to the Hub data structure of a FIFO-Hub.

Returns

L1_TRUE If the FIFO is full.
L1_FALSE If the FIFO is not full.

Remarks

SPC FIFO-Size upper bound

6.27.7.18 static __inline__ L1_ReturnCode L1_PutDataToFifo_NW (L1_HubID *hubID,* L1_BYTE * *data,* L1_UINT32 *size*) [static]

This function performs a data transfer to a FIFO-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail. – There was no space available in the FIFO-Hub, hence the data could not be enqueued.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.27.7.19 static __inline__ L1_ReturnCode L1_PutDataToFifo_W (L1_HubID *hubID*, L1_BYTE * *data*, L1_UINT32 *size*) [static]

This function performs a data transfer to a FIFO-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.27.7.20 static __inline__ L1_ReturnCode L1_PutDataToFifo_WT (L1_HubID *hubID*, L1_BYTE * *data*, L1_UINT32 *size*, L1_Timeout *timeout*) [static]

This function performs a data transfer to a FIFO-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.
timeout	The timeout in ticks for the interaction.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.
- RC_TO: The timeout of the interaction expired.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.28 Memory Pool Hub

Data Structures

- struct L1_MemoryPool_HubState

Macros

- #define L1_isMemoryPoolHub(h) ((h)->HubType == L1_MEMORYPOOL)
- #define L1_MemoryPool_State(h) ((L1_MemoryPool_HubState∗)(h)->HubState)

Functions

- L1_ReturnCode L1_AllocateMemoryBlock (L1_HubID HubID, L1_BYTE ∗∗Memory, L1_UINT16 Size, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_NW (L1_HubID memoryPool, L1_BYTE ∗∗memoryBlock, L1_UINT16 size)
- static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_W (L1_HubID memoryPool, L1_BYTE ∗∗memoryBlock, L1_UINT16 size)
- static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_WT (L1_HubID memoryPool, L1_BYTE ∗∗memoryBlock, L1_UINT16 size, L1_Timeout timeout)
- L1_ReturnCode L1_DeallocateMemoryBlock_NW (L1_HubID memoryPool, void ∗memoryBlock)
- void MemoryPoolIoctl (L1_Hub ∗Hub, L1_Packet ∗Packet, L1_BYTE ioctl_type)
- L1_BOOL MemoryPoolSyncCondition (L1_Hub ∗Hub, L1_Packet ∗Packet)
- void MemoryPoolSynchronize (L1_Hub ∗hub, L1_Packet ∗packet, L1_Packet ∗waitingPacket)
- void MemoryPoolUpdate (L1_Hub ∗Hub, L1_Packet ∗Packet)

6.28.1 Detailed Description

The Memory-Pool-Hub offers Tasks to dynamically allocate and deallocate blocks of memory.

6.28.2 Visual Designer

Figure 6.7: Application Diagram Icon

6.28.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- sizeOfBlocks: The size of the memory blocks the Hub provides, in Bytes.
- numOfBlocks: How many memory blocks the Hub provides.

6.28.3 Example

The the code shown in section MemoryPoolExampleTEP shows a Task that utilises a Memory Pool Hub to allocate one block of 1024 bytes of memory. It then prints the address of the memory block onto the console before deallocating the memory block, before releasing it again.

6.28.3.1 Entities

- MPool1: Memory Pool Hub:
  - BlockSize = 1024
  - NumberOfBlocks = 1
- Shs1: A Stdio Host Server
- Task1: The Task that performs the operations, uses the function MemoryPoolExampleTEP() as Task Entry Point.

6.28.3.2 MemoryPoolExampleTEP

```
void MemoryPoolExampleTEP(L1_TaskArguments Arguments)
{
    // Pointer of the memory block, to be allocated
    L1_BYTE * memoryBlock = NULL;

    // Allocating the memory block.
    if( RC_FAIL == L1_AllocateMemoryBlock_W(MPool1, &memoryBlock, 1024) )
    {
        Shs_putString_W(Shs1, "Error could not allocate the memory block.\n");
        exit(-1);
    }
    Shs_putString_W(Shs1, "Could successfully allocate the memory block at: ");
    Shs_putInt_W(Shs1, memoryBlock, 'd');
    Shs_putString_W(Shs1, "\n");

    // Deallocating the previously allocated memory block
    if( RC_FAIL == L1_DeallocateMemoryBlock_NW(MPool1, memoryBlock) )
    {
        Shs_putString_W(Shs1, "Error in deallocation of the memory block\n");
        exit(-2);
    }
    Shs_putString_W(Shs1, "\nPress enter to terminate the program\n");
}
```

Remarks

```
SPC Memory Pool
SPC Memory Management
```

6.28.4 Macro Definition Documentation

6.28.4.1 #define L1_isMemoryPoolHub(*h*) ((h)->HubType == L1_MEMORYPOOL)

This expression checks whether the pointer h, points to a Memory-Pool hub.

Parameters

h	Pointer to an L1_Hub.

Warning

It is the responsibility of the developer to ensure that h points to a structure of type L1_Hub.
This function does not check whether or not h is NULL.

6.28.4.2 #define L1_MemoryPool_State(*h*) ((L1_MemoryPool_HubState*)(h)->HubState)

This macro casts the HubState void pointer to a pointer of type L1_MemoryPool_HubState.

Parameters

h	Pointer to an L1_Hub.

Warning

It is the responsibility of the developer to ensure that h points to a structure of type L1_Hub.
This function does not check whether or not the Hub is of type L1_MEMORYPOOL. It is the responsibility of the developer to check this beforehand.
This function does not check whether or not h is NULL.

See Also

L1_isMemoryPoolHub

6.28.5 Function Documentation

6.28.5.1 L1_ReturnCode L1_AllocateMemoryBlock (L1_HubID *HubID,* L1_BYTE ** *Memory,* L1_UINT16 *Size,* L1_Timeout *Timeout*)

Acquires a memory-block from a local memory pool.

Parameters

HubID	the ID of the MemoryPool from which to acquire a region of memory with the size specified by the parameter Size.
Memory	if the service completed successfully, this will point to a pointer where the allocated memory block is located. This memory can then be used by the Task. Otherwise, this variable will point to a NULL pointer.
Size	the desired size of the MemoryBlock.
Timeout	Indicates the timeout value/mode for the request.

Returns

RC_OK The service completed successfully, Memory points to a pointer which points to the allocated MemoryBlock.
RC_FAIL The service failed, Memory will point to a NULL pointer.

Warning

The memory pool must be mapped to the same node as the task calling this function.

Precondition

- memoryPool must be local

Postcondition

- Calling task ready.

6.28.5.2 static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_NW (L1_HubID *memoryPool,* L1_BYTE ** *memoryBlock,* L1_UINT16 *size*) [static]

Acquires a memory-block from a memory pool. This call returns immediately independent of whether or not a MemoryBlock was available or not.

Parameters

memoryPool	the ID of the MemoryPool from which to acquire a region of memory with the size specified by the parameter Size.
memoryBlock	if the service completed successfully, this will point to a pointer where the allocated memory block is located.This memory can then be used by the Task. Otherwise, this variable will point to a NULL pointer.
size	the desired size of the MemoryBlock. However, it is currently not used correctly by the function.

Returns

RC_OK The service completed successfully, memoryBlock points to a pointer which points to the allocated MemoryBlock.
RC_FAIL The service failed, memoryBlock will point to a NULL pointer.

Warning

The memory pool must be mapped to the same node as the task calling this function.

Precondition

- memoryPool must be local

Postcondition

- Calling task ready.

6.28.5.3 static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_W (L1_HubID *memoryPool,* L1_BYTE ** *memoryBlock,* L1_UINT16 *size*) [static]

Acquires a memory-block from a local memory pool. This service waits till a memory block is available.

Parameters

memoryPool	the ID of the MemoryPool from which to acquire a region of memory with the size specified by the parameter Size.
memoryBlock	if the service completed successfully, this will point to a pointer where the allocated memory block is located. This memory can then be used by the Task. Otherwise, this variable will point to a NULL pointer.
size	the desired size of the MemoryBlock.

Returns

RC_OK The service completed successfully, Memory points to a pointer which points to the allocated MemoryBlock.
RC_FAIL The service failed, Memory will point to a NULL pointer.

Warning

The memory pool must be mapped to the same node as the task calling this function.

Precondition

- memoryPool must be local

Postcondition

- Calling task ready.

6.28.5.4 static __inline__ L1_ReturnCode L1_AllocateMemoryBlock_WT (L1_HubID *memoryPool,* L1_BYTE ** *memoryBlock,* L1_UINT16 *size,* L1_Timeout *timeout*) [static]

Acquires a memory-block from a memory pool. Waits until either a memory-block becomes available or the timeout expired, depending on what happens earlier.

Parameters

memoryPool	the ID of the MemoryPool from which to acquire a region of memory with the size specified by the parameter Size.
memoryBlock	if the service completed successfully, this will point to a pointer where the allocated memory block is located. This memory can then be used by the Task. Otherwise, this variable will point to a NULL pointer.
size	the desired size of the MemoryBlock. However, it is currently not used correctly by the function.
timeout	of type L1_Timeout, the number of system ticks the call should wait for a MemoryBlock to become available.

Returns

RC_OK The service completed successfully, memoryBlock points to a pointer which points to the allocated MemoryBlock.
RC_FAIL The service failed, memoryBlock will point to a NULL pointer.
RC_TO The timeout expired without a MemoryBlock becoming available, memoryBlock will point to a NULL pointer.

Warning

The memory pool must be mapped to the same node as the task calling this function.

Precondition

- memoryPool must be local

Postcondition

- Calling task ready.

6.28.5.5 L1_ReturnCode L1_DeallocateMemoryBlock_NW (L1_HubID *memoryPool,* void * *memoryBlock*)

This Kernel service is called by a Task to release a memory-block back to its memory pool.

Parameters

memoryPool	identifies the MemoryPool.
memoryBlock	pointer to the memory-block to release

Returns

RC_OK service successful (a memory block was released to the memory pool)
RC_FAIL service failed (the memory block was not released to the memory pool)

Precondition

- None

Postcondition

- Calling task ready

6.28.5.6 void MemoryPoolIoctl (L1_Hub * *Hub,* L1_Packet * *Packet,* L1_BYTE *ioctl_type*)

This function provides a standard handler to the memory pool initialization function through an ioctl operation to open memory pool hub.

Parameters

Hub	Pointer to a Hub of type L1_MEMORYPOOL.
Packet	Pointer to the L1_Packet which caused the function to be called. It will contain additional information.
ioctl_type	Control operation to be executed (L1_IOCTL_HUB_OPEN only for this hub).

6.28.5.7 L1_BOOL MemoryPoolSyncCondition (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function evaluates the synchronization condition. If a deallocation is requested (put packet), the function returns true if the packet has been previously allocated. For allocation requests (get packet), the function evaluates to true if the free memory block list is not empty.

Parameters

Hub	Pointer to a Hub of type L1_MEMORYPOOL.
Packet	Pointer to the L1_Packet used to test for the synchronization condition.

Returns

L1_TRUE If synchronisation was achieved.
L1_FALSE If synchronisation was not achieved.

6.28.5.8 void MemoryPoolSynchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall first check whether the deallocation request in the L1_Packet packet is valid. If it is valid it shall then return the Memory Block to the Hub, before handling the allocation request in the L1_Packet waitingPacket. Afterwards both Packets get returned to their Tasks with the field Status set to RC_OK.

In the case that packet contains an invalid deallocation request the function shall reinsert the L1_Packet waitingPacket into the Hub WaitingList before returning packet with packet->Status set to RC_FAIL.

Parameters

hub	Pointer to a Hub of type L1_MEMORYPOOL.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL
packet->ServiceID = L1_SID_PUT_TO_HUB
waitingPacket->ServiceID = L1_SID_GET_FROM_HUB

6.28.5.9 void MemoryPoolUpdate (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function updates the memory pool hub when receiving packets. A get packet allocates a memory block. To allocate a memory block, a free block is taken from the memory pool and it is inserted into the occupied memory block list. A void pointer is returned in the data field of the packed used to allocate the block. A put packet deallocates a memory block. The inverse process is to remove the block from the occupied list, and insert it into the free memory block list. The data size of the packet is set to zero to indicate that not data is returned.

Parameters

Hub	Pointer to a hub of type L1_MEMORYPOOL.
Packet	Pointer to the L1_Packet used for the update operation.

6.29 Packet Pool Hub

Data Structures

- struct _struct_L1_PacketPoolState_

Typedefs

- typedef struct _struct_L1_PacketPoolState_ L1_PacketPool_HubState

Functions

- L1_ReturnCode L1_AllocatePacket (L1_HubID HubID, L1_Packet **Packet, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_AllocatePacket_NW (L1_HubID packetPool, L1_Packet **packet)
- static __inline__ L1_ReturnCode L1_AllocatePacket_W (L1_HubID packetPool, L1_Packet **packet)
- static __inline__ L1_ReturnCode L1_AllocatePacket_WT (L1_HubID packetPool, L1_Packet **packet, L1_Timeout timeout)
- L1_ReturnCode L1_DeallocatePacket_NW (L1_HubID packetPool, L1_Packet *packet)
- static __inline__ L1_BOOL L1_isHubPacketPoolPacketAvailable (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isPacketPoolHub (L1_Hub *pHub)
- static __inline__ L1_PacketPool_HubState * L1_PacketPool_State (L1_Hub *pHub)
- void PacketPoolIoctl (L1_Hub *Hub, L1_Packet *Packet, L1_BYTE ioctl_type)
- L1_BOOL PacketPoolSyncCondition (L1_Hub *Hub, L1_Packet *Packet)
- void PacketPoolSynchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)
- void PacketPoolUpdate (L1_Hub *Hub, L1_Packet *Packet)

6.29.1 Detailed Description

Remarks

```
SPC Packet Pool
```

A Packet Pool allows Tasks to allocate and deallocated additional L1_Packets during runtime.

6.29.2 Visual Designer

Figure 6.8: Application Diagram Icon

6.29.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- size: How many L1_Packets the Hub provides.

6.29.3 Typedef Documentation

6.29.3.1 typedef struct _struct_L1_PacketPoolState_ L1_PacketPool_HubState

The state of a Packet-Pool-Hub.

Remarks

```
SPC Packet Pool state variables
```

6.29.4 Function Documentation

6.29.4.1 L1_ReturnCode L1_AllocatePacket (L1_HubID *HubID*, L1_Packet ** *Packet*, L1_Timeout *Timeout*)

This function allocates a packet from a packet pool and initializes the task ID and priority fields.

Parameters

HubID	The ID of the pool where the packet will be allocated.
Packet	The allocated packet will be pointed by this variable when the function succeeds.
Timeout	Indicates the timeout value/mode for the request.

Returns

- RC_OK service completed successfully (there was an available Packet in the Packet Pool).
- RC_FAIL service failed (no available Packet in the Packet Pool), Packet will the point to NULL.

Precondition

- There is a Kernel Packet Pool on the Node
- This service cannot be called from the ISR LAYER

Postcondition

- 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.

6.29.4.2 static __inline__ L1_ReturnCode L1_AllocatePacket_NW (L1_HubID *packetPool*, L1_Packet ** *packet*) [static]

This service allocates a Packet from a Packet-Pool on the local Node. The service returns immediately either with the allocated Packet or with a return value indicating failure (if there was no available Packet in the Packet pool).

Parameters

packetPool	the ID of the PacketPool on the local Node.
packet	will contain a pointer to Packet upon successful return.

Returns

- RC_OK the service completed successfully (there was an available Packet in the Packet Pool).
- RC_FAIL service failed (no available Packet in the Packet Pool), Packet will the point to NULL.

Precondition

- There is a local Packet Pool on the Node
- This service cannot be called from the ISR LAYER

Postcondition

- 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.

6.29.4.3 static __inline__ L1_ReturnCode L1_AllocatePacket_W (L1_HubID *packetPool*, L1_Packet ** *packet*) [static]

This service allocates a Packet from a Packet-Pool on the local Node. It waits until a Packet has been allocated.

Parameters

packetPool	the ID of the PacketPool on the local Node.
packet	of type L1_Packet**, will contain the Packet upon successful return

Returns

- RC_OK service completed successfully (there was an available Packet in the Packet Pool).
- RC_FAIL service failed (no available Packet in the Packet Pool), Packet will the point to NULL.

Precondition

- There is a Kernel Packet Pool on the Node
- This service cannot be called from the ISR LAYER

Postcondition

- 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.

6.29.4.4 static __inline__ L1_ReturnCode L1_AllocatePacket_WT (L1_HubID *packetPool,* L1_Packet ** *packet,* L1_Timeout *timeout*) [static]

This service allocates a Packet from a Packet-Pool on the local Node. This service 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).

Parameters

packetPool	of type L1_HubID, the ID of the PacketPool on the local Node.
packet	will contain the Packet upon successful return.
timeout	the number of system ticks the call should wait for a packet to become available.

Returns

- RC_OK service completed successfully (there was an available Packet in the Packet Pool).
- RC_FAIL service failed (no available Packet in the Packet Pool), Packet* is set to NULL.
- RC_TO the timeout has expired, Packet will point to NULL.

Precondition

- There is a Packet Pool on the Node.
- This service cannot be called from the ISR LAYER.

Postcondition

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

6.29.4.5 L1_ReturnCode L1_DeallocatePacket_NW (L1_HubID *packetPool,* L1_Packet * *packet*)

This service deallocates a Packet and returns it to the Packet Pool.

Parameters

packetPool	the ID of the Packet Pool to which to return packet.
packet	the Packet that needs to be de-allocated.

Returns

- RC_OK service completed successfully.
- RC_FAIL service failed.

Precondition

- This service cannot be called by the ISR LAYER.
- Packet must have been allocated by L1_AllocatePacket.
- Packet must be a Packet on a local PacketPool.

Postcondition

- Packet is no longer available for use by the Task.
- Packet is available for use by other Tasks.

Note

The L1_DeallocatePacket kernel service is served by the Kernel Task of the Node at which the requesting Task resides. Hence, the destination Port is implicitly set to the KernelPort.

6.29.4.6 static __inline__ L1_BOOL L1_isHubPacketPoolPacketAvailable (L1_Hub * *pHub*) [static]

This function determines whether or not the Packet Pool pHub still contains L1_Packets which can be allocated from it.

Parameters

pHub	Pointer to an L1_Hub structure of the Packet Pool for which to check the availability of Packets.

Returns

L1_TRUE If there are L1_Packets available.
L1_FLASE Otherwise.

6.29.4.7 static __inline__ L1_BOOL L1_isPacketPoolHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a PacketPool-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a PacketPool-Hub.

Returns

L1_TRUE if the data structure represents a PacketPool-Hub.
L1_FALSE otherwise.

Warning

This function does not check whether or not pHub is NULL.

6.29.4.8 static __inline__ L1_PacketPool_HubState* L1_PacketPool_State (L1_Hub * *pHub*) [static]

This function returns the Hub-State of the Hub pHub as a pointer of type L1_PacketPool_HubState.

Parameters

pHub	Pointer to a Hub of type of type L1_PACKET_POOl.

Warning

This function does not check whether or not the Hub is of type L1_PACKET_POOL. It is the responsibility of the developer to check this beforehand.
This function does not check whether or not pHub is NULL.

See Also

L1_isPacketPoolHub

6.29.4.9 void PacketPoolIoctl (L1_Hub * *Hub,* L1_Packet * *Packet,* L1_BYTE *ioctl_type*)

This function provides a standard handler to the packet pool initialization function through an ioctl operation to open packet pool hub.

Parameters

Hub	Pointer to the packet pool hub.
Packet	Pointer to the L1_Packet which caused the function to be called. It will contain additional information.
ioctl_type	Control operation to be executed (L1_IOCTL_HUB_OPEN only for this hub).

6.29.4.10 L1_BOOL PacketPoolSyncCondition (L1_Hub * *Hub,* L1_Packet * *Packet*)

When deallocating a packet, synchronization should always succeed. For the case of packet allocation, it should succeed only if the packet pool has available packets to allocate.

Parameters

Hub	Pointer to the packet pool hub.
Packet	Packet used to test for the synchronization condition.

Returns

L1_TRUE If synchronisation was achieved.
L1_FALSE If synchronisation was not achieved. *

6.29.4.11 void PacketPoolSynchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall deallocate the L1_Packet provided in packet, before handling the allocation request in waitingPacket. Afterwards both Packets get returned to their Tasks with the field Status set to RC_OK.

In the case that packet contains an invalid deallocation request the function shall reinsert the L1_Packet waitingPacket into the Hub WaitingList before returning packet with packet->Status set to RC_FAIL.

Parameters

hub	Pointer to a hub of type L1_PACKETPOOL.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL
packet->ServiceID = L1_SID_PUT_TO_HUB
waitingPacket->ServiceID = L1_SID_GET_FROM_HUB

6.29.4.12 void PacketPoolUpdate (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function updates the packet pool when receiving allocate and deallocate requests. To allocate a packet, a packet is taken from the hub's packet list and returned as part of the data field of the get packet that requested the allocation. A deallocation returns the packet to the packet list.

Parameters

Hub	Pointer to the packet pool hub.
Packet	Packet used for the update operation.

6.30 Port Hub

Functions

- static __inline__ L1_ReturnCode L1_Drv_Isr_PutPacketToPort_NW (L1_HubID HubID, L1_Packet *packet)
- static __inline__ L1_ReturnCode L1_GetDataFromPort_NW (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived)
- static __inline__ L1_ReturnCode L1_GetDataFromPort_W (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived)
- static __inline__ L1_ReturnCode L1_GetDataFromPort_WT (L1_HubID hubID, L1_BYTE *dataBuffer, L1_UINT32 bufferSize, L1_UINT32 *bytesReceived, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_GetPacketFromPort_A (L1_HubID HubID, L1_Packet *packet)
- static __inline__ L1_ReturnCode L1_GetPacketFromPort_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_GetPacketFromPort_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_GetPacketFromPort_WT (L1_HubID HubID, L1_Timeout timeout)

- static __inline__ L1_BOOL L1_isLocalPortHub (L1_Hub *pHub)
- static __inline__ L1_ReturnCode L1_PutDataToPort_NW (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size)
- static __inline__ L1_ReturnCode L1_PutDataToPort_W (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size)
- static __inline__ L1_ReturnCode L1_PutDataToPort_WT (L1_HubID hubID, L1_BYTE *data, L1_UINT32 size, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_PutPacketToPort_A (L1_HubID HubID, L1_Packet *packet)
- static __inline__ L1_ReturnCode L1_PutPacketToPort_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_PutPacketToPort_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_PutPacketToPort_WT (L1_HubID HubID, L1_Timeout timeout)
- L1_BOOL LocalPortSyncCondition (L1_Hub *Hub, L1_Packet *Packet)
- void LocalPortSynchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)

6.30.1 Detailed Description

The Port Hub is uses to exchange data between two Tasks in a reliable way. It's behavior is similar to a CSP-Channel.

6.30.2 Visual Designer

Figure 6.9: Application Diagram Icon

6.30.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.

6.30.3 Example

The this example shows how to transfer data from one Task to another Task using a Port.

6.30.3.1 Entities

- Port1: Port which is used to exchange data between Task1 and Task2
- Task1: Task1EntryPoint, shown in section Source Code for Task1EntryPoint
- Task2: Task2EntryPoint, shown in section Source Code for Task2EntryPoint
- StdioHostServer1: Access to the console.
- StdioHostServer1Res: Ensuring that a second task does not interfere with console access.

6.30.4 Source Code for Task1EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>

void Task1EntryPoint(L1_TaskArguments Arguments)
{
    L1_Packet *Packet = L1_CurrentTaskCR->RequestPacket;
    L1_BYTE ch;
    for (ch = 'a'; ch <= 'z'; ch++)
    {
        Packet->DataSize = sizeof(L1_BYTE);
        Packet->Data[0]  = ch;

        if (RC_FAIL == L1_PutPacketToPort_W(Port1))
        {
           exit(-1);
        }
    }
}
```

6.30.5 Source Code for Task2EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>

void Task2EntryPoint(L1_TaskArguments Arguments)
{
    L1_Packet *Packet = L1_CurrentTaskCR->RequestPacket;

    L1_BYTE ch, i;

    for(i = 0; i < 26; i++)
    {
        if(RC_OK == L1_GetPacketFromPort_W(Port1))
        {
            Packet->DataSize = sizeof(L1_BYTE);
            ch = Packet->Data[0];
            L1_LockResource_W(StdioHostServer1Res);
            Shs_putString_W(StdioHostServer1, "The following symbol was get from Port
            Shs_putChar_W(StdioHostServer1, ch);
            Shs_putChar_W(StdioHostServer1, '\n');
            L1_UnlockResource_NW(StdioHostServer1Res);
        }else
        {
            Shs_putString_W(StdioHostServer1, "Error: Could not acquire a symbol from
      \n");
        }
    }
}
```

Remarks

```
REQ Port
SPC Port
```

6.30.6 Function Documentation

6.30.6.1 static __inline__ L1_ReturnCode L1_Drv_Isr_PutPacketToPort_NW (L1_HubID *HubID,* L1_Packet * *packet*) [static]

This service puts the L1_Packet given in parameter packet of the task calling it into a Port. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters:

Parameters

HubID	is the L1_HubID which identifies the Port-Hub, that the calling ISR wants to send the Packet to.
packet	Pointer to the L1_Packet that will be used to represent the interaction.

This L1_Packet must have been once initialized using the function L1_Drv_Isr_initialisePacket().

Returns

RC_OK The packet could be inserted into the Kernel Input Port.
RC_FAIL The packet could not be inserted into the Kernel Input Port.

Warning

Must not be used with Port-Hubs located at another Node.
Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

```
SPC Packets from an ISR
```

6.30.6.2 static __inline__ L1_ReturnCode L1_GetDataFromPort_NW (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.

bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.

6.30.6.3 static __inline__ L1_ReturnCode L1_GetDataFromPort_W (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.
bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.

6.30.6.4 static __inline__ L1_ReturnCode L1_GetDataFromPort_WT (L1_HubID *hubID,* L1_BYTE * *dataBuffer,* L1_UINT32 *bufferSize,* L1_UINT32 * *bytesReceived,* L1_Timeout *timeout*) [static]

Receives data from a Port-Hub.

Parameters

hubID	ID of the Hub to send data to.
dataBuffer	This is the pointer to the buffer where to store the data received from the Hub.
bufferSize	This is the size of the dataBuffer in L1_BYTE. This does not indicate that the interaction will receive so much data, it is just an indication of the maximum number of bytes the function can store safely in the dataBuffer. The real number of received bytes is stored, upon return in the parameter bytesReceived.
bytesReceived	Pointer to an L1_UINT32 which will contain the number of bytes that were received from the Hub after the interaction was performed successfully. This parameter may be set to NULL to indicate that no interest in this value exists. However, this is generally not advised.
timeout	The timeout to use for the interaction with the Hub.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail: – Not enough space in the dataBuffer. In this case the data is not lost but is still present in the field Data of the used L1_Packet. – dataBuffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.
- RC_TO: The timeout of the interaction expired.

6.30.6.5 static __inline__ L1_ReturnCode L1_GetPacketFromPort_A (L1_HubID *HubID,* L1_Packet * *packet*) [static]

Request to get a Packet from a Port without being put in the waiting state. This service requires that the Get requests use a Packet has been allocated from the Packet Pool). The completion is deferred till a corresponding L1_WaitForPacket service request which will return any of the packets previously allocated but filled in with the data of a corresponding Put_request to one of the Ports.

Parameters

HubID	of type L1_HubID which identifies the Port.
packet	Pointer to the L1_Packet that will be used for this Asynchronous-Interaction. There are two ways of acquiring such an L1_Packet: • Allocate an L1_Packet in your Task, and then initialise it using the function L1_initialiseAsyncPacket(). • Allocate an L1_Packet from a local (i.e. same Node) Packet Pool-Hub, using one of the following interactions: L1_AllocatePacket_W(), L1_AllocatePacket_NW(), and L1_AllocatePacket_WT().

Returns

L1_ReturnCode

- RC_OK service completed successfully.
- RC_FAIL service failed

Precondition

- Packet must have been allocated by the function L1_AllocatePacket().

Postcondition

- The calling task will remain on the READY List.

Warning

This Interaction only works locally (Task and Hub on the same Node), i.e. not distributed (Task and Hub on different Nodes).

See Also

L1_initialiseAsyncPacket
L1_AllocatePacket_W
L1_AllocatePacket_NW
L1_AllocatePacket_WT

6.30.6.6 static __inline__ L1_ReturnCode L1_GetPacketFromPort_NW (L1_HubID *HubID*) [static]

Retrieves a packet from a port using the task's Request-Packet. Returns immediately after the get request was delivered to the specified Port, indicating either success (there was a corresponding put request at the specified Port) or a failure (there was no put request at the specified Port; in that case the Get Packet is NOT buffered in the specified Port).

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	L1_HubID which identifies the Port.

Note

If the specified Port is remote than the return time includes a communication delay.

Returns

- RC_OK service successful (there was a waiting Put request in the Port)
- RC_FAIL service failed (no corresponding put request in the Port)

Precondition

- Packet is the preallocated SystemPacket

Postcondition

- Header fields of Put Packet filled in the Task's System Packet.
- Data of Put Packet will have been filled in.

6.30.6.7 static __inline__ L1_ReturnCode L1_GetPacketFromPort_W (L1_HubID *HubID*) [static]

Retrieves a packet from a port using the task's Request-Packet. This service waits until the get request has synchronised with a corresponding put packet delivered to the specified Port.

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	L1_HubID which identifies the Port.

Returns

- RC_OK service successful (there was a waiting Put request in the Port)
- RC_FAIL service failed (no corresponding put request in the Port)

Precondition

- Packet is the preallocated SystemPacket

Postcondition

- Header fields of Put Packet filled in the Task's System Packet.
- Data of Put Packet will have been filled in.

6.30.6.8 static __inline__ L1_ReturnCode L1_GetPacketFromPort_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

Retrieves a packet from a port using the task's Request-Packet. Waits until either the get request has synchronised with a corresponding put 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 get a Packet from the specified Port) and the get Packet is removed from the Specified Port.

Warning

The payload part of the task specific request-packet gets overwritten as soon as another request gets sent. Therefore, always copy the payload-data to a local buffer using L1_memcpy(...).

Parameters

HubID	L1_HubID which identifies the Port.
timeout	of type L1_Timeout, the number of system ticks the call should wait for synchronisation.

Returns

- RC_OK service successful (there was a waiting Put request in the Port)
- RC_FAIL service failed (no corresponding put request in the Port)
- RC_TO service timed out.

Precondition

- Packet is the preallocated SystemPacket

Postcondition

- Header fields of Put Packet filled in the Task's System Packet.
- Data of Put Packet will have been filled in.

6.30.6.9 static __inline__ L1_BOOL L1_isLocalPortHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a Port-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a Port-Hub.

Returns

L1_TRUE if the data structure represents a BlackBoard-Hub.
L1_FALSE otherwise.

6.30.6.10 static __inline__ L1_ReturnCode L1_PutDataToPort_NW (L1_HubID *hubID,* L1_BYTE * *data,* L1_UINT32 *size*) [static]

This function performs a data transfer to a Port-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.
- RC_TO: The timeout of the interaction expired.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.30.6.11 static __inline__ L1_ReturnCode L1_PutDataToPort_W (L1_HubID *hubID,* L1_BYTE * *data,* L1_UINT32 *size*) [static]

This function performs a data transfer to a Port-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.30.6.12 static __inline__ L1_ReturnCode L1_PutDataToPort_WT (L1_HubID *hubID,* L1_BYTE * *data,* L1_UINT32 *size,* L1_Timeout *timeout*) [static]

This function performs a data transfer to a Port-Hub. It copies the data stored in the buffer indicated by data into the data part of an L1_Packet and then sends this packet to the Hub.

Parameters

hubID	ID of the Hub to send data to.
data	pointer to a buffer of type L1_BYTE which contains the data that should be sent to the Hub.
size	size of the buffer pointed to by data. This must not be larger than L1_PACKET_SIZE, otherwise the interaction will return RC_FAIL.
timeout	The timeout in ticks for the interaction.

Returns

L1_ReturnCode:

- RC_OK: The interaction was successful;
- RC_FAIL: The interaction failed. There are multiple reasons for the interaction to fail; – The buffer data does not fit in the data part of the L1_Packet (L1_PACKET_DATA_SIZE). – data-Buffer is NULL – packet is NULL – General failure of the system causing the interaction with the Hub to fail.
- RC_TO: The timeout of the interaction expired.

Warning

Only use this functions to interact with Port- or Fifo-Hubs. Interacting with other types of Hubs may have undesired effects.

6.30.6.13 static __inline__ L1_ReturnCode L1_PutPacketToPort_A (L1_HubID *HubID,* L1_Packet * *packet*) [static]

Puts a Packet (that must be allocated from the Packet Pool) to a Port and returns immediately without being put in the waiting state. Completion is deferred till a corresponding Get service request which will return any of the packets previously Put asynchronously.

Note

The algorithm for L1_PutPacketToPort_A is the same as of L1_PutPacket_W, except that L1_insert-PacketInKernel will not remove the calling Task from the ReadyList.

Parameters

HubID	of type L1_HubID, which identifies the Port.
packet	Pointer to the L1_Packet that will be used for this Asynchronous-Interaction. There are two ways of acquiring such an L1_Packet: • Allocate an L1_Packet in your Task, and then initialise it using the function L1_-initialiseAsyncPacket(). • Allocate an L1_Packet from a local (i.e. same Node) Packet Pool-Hub, using one of the following interactions: L1_AllocatePacket_W(), L1_AllocatePacket_-NW(), and L1_AllocatePacket_WT().

Returns

L1_ReturnCode:

- RC_OK service successful (there was a waiting Get request in the Port)
- RC_FAIL service failed (no corresponding Get request in the Port)

Precondition

- Packet must have been allocated by the function L1_AllocatePacket.

Postcondition

- The calling task will remain on the READY List.

Warning

This Interaction only works locally (Task and Hub on the same Node), i.e. not distributed (Task and Hub on different Nodes).

See Also

L1_initialiseAsyncPacket
L1_AllocatePacket_W
L1_AllocatePacket_NW
L1_AllocatePacket_WT

6.30.6.14 static __inline__ L1_ReturnCode L1_PutPacketToPort_NW (L1_HubID *HubID*) [static]

This service puts the Request-Packet of the task calling it into a Port. The service returns immediately after the Packet was delivered to the specified Port. Indicates either success (there was a corresponding request to get a Packet from the destination Port) or failure (there was no corresponding request to get a Packet from the specified Port; in that case the put Packet is NOT buffered in the specified Port).

Note

If the specified Port is remote than the return time includes a communication delay.

Parameters

HubID	of type L1_HubID, which identifies the Port.

Returns

L1_ReturnCode:

- RC_OK service successful (there was a waiting Get request in the Port)
- RC_FAIL service failed (no corresponding Get request in the Port)

Precondition

- None
- Packet is the preallocated Packet

Postcondition

- The Header field of the RequestPacket are filled in.
- Header fields of preallocated Packet filled in

6.30.6.15 static __inline__ L1_ReturnCode L1_PutPacketToPort_W (L1_HubID *HubID*) [static]

This service puts the Request-Packet of the task calling it into a Port. This service waits until the put request has synchronised with a corresponding request to get a Packet from the specified Port.

Parameters

HubID	of type L1_HubID, which identifies the Port.

Returns

L1_ReturnCode:

- RC_OK service successful (there was a waiting Get request in the Port)
- RC_FAIL service failed (no corresponding Get request in the Port)

Precondition

- None
- Packet is the preallocated Packet

Postcondition

- The Header field of the RequestPacket are filled in.
- Header fields of preallocated Packet filled in

6.30.6.16 static __inline__ L1_ReturnCode L1_PutPacketToPort_WT (L1_HubID *HubID*, L1_Timeout *timeout*) [static]

This service puts the Request-Packet of the task calling it into a Port. Waits until either the put request has synchronised with a corresponding request to get 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 get a Packet from the specified Port) and the put Packet is removed from the specified Port.

Parameters

HubID	of type L1_HubID, which identifies the Port.
timeout	of type L1_Timeout, the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode:

- RC_OK service successful (there was a waiting Get request in the Port)
- RC_FAIL service failed (no corresponding Get request in the Port)
- RC_TO service timed out.

Precondition

- None
- Packet is the preallocated Packet

Postcondition

- The Header field of the RequestPacket are filled in.
- Header fields of preallocated Packet filled in

6.30.6.17 L1_BOOL LocalPortSyncCondition (L1_Hub * *Hub,* L1_Packet * *Packet*)

Remarks

```
SPC Port
```

This function returns L1_TRUE when synchronisation happens in a Port Hub.

Parameters

Hub	Pointer to the L1_Hub structure of the Port Hub to check..
Packet	Pointer to the L1_Packet received at the Port Hub.

Returns

L1_BOOL:

- L1_FALSE Synchronisation occurred.
- L1_TRUE Synchronisation did not occur.

Precondition

- empty waiting list.

Postcondition

- None

6.30.6.18 void LocalPortSynchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall copy the data and data size from the Put-Packet to the Get-Packet and then return both Packets to their Tasks.

Parameters

hub	Pointer to a hub of type L1_PORT.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL

Postcondition

packet->Status = RC_OK
waitingPacket->Status = RC_OK

6.31 Resource Hub

Data Structures

- struct _struct_L1_ResourceState_

Typedefs

- typedef struct
  _struct_L1_ResourceState_ L1_Resource_HubState

Functions

- static __inline__ L1_BOOL L1_isHubResourceLocked (L1_Hub *pHub)
- static __inline__ L1_BOOL L1_isResourceHub (L1_Hub *pHub)
- static __inline__ L1_ReturnCode L1_LockResource_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_LockResource_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_LockResource_WT (L1_HubID HubID, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_UnlockResource_NW (L1_HubID HubID)
- L1_BOOL ResourceSyncCondition (L1_Hub *Hub, L1_Packet *Packet)
- void ResourceSynchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)
- void ResourceUpdate (L1_Hub *Hub, L1_Packet *Packet)

6.31.1 Detailed Description

The Resource-Hub is a synchronisation mechanism to prevent two or more Tasks accessing the same global resource at the same time.

6.31.2 Visual Designer

Figure 6.10: Application Diagram Icon

6.31.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- ceilingPriority: The ceiling priority of the Entity.

6.31.3 Example

This examples illustrates how a Resource Hub can be used to guard access to a shared resource, in this case a Stdio Host Server. It consists of two tasks: Task1 and Task2, which both count from 0 to 19 and print out the counting messages onto the console using the Stdio Host Server StdioHostServer1.

6.31.3.1 Entities

- Task1: Task1EntryPoint, shown in section Source Code of Task1EntryPoint
- Task1: Task2EntryPoint, shown in section Source Code of Task2EntryPoint
- StdioHostServer1: A Stdio Host Server component which provides access to the console.
- StdioHostServer1Res: A Resource Hub to ensure that a second task does not interfere with console access.

6.31.4 Source Code of Task1EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>

void Task1EntryPoint(L1_TaskArguments Arguments)
{
    L1_UINT32 i = 0;
        for(i = 0; i < 20; i++)
        {
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1, "Task 1 outputs: 0x");
                Shs_putInt_W(StdioHostServer1, i, 'x');
                Shs_putChar_W(StdioHostServer1, '\n');
                L1_UnlockResource_NW(StdioHostServer1Res);
       }
}
```

6.31.5 Source Code of Task2EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>
```

```
void Task2EntryPoint(L1_TaskArguments Arguments)
{
    L1_UINT32 i = 0;
        for(i = 0; i < 20; i++)
        {
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1, "Task 2 outputs: 0x");
                Shs_putInt_W(StdioHostServer1, i, 'x');
                Shs_putChar_W(StdioHostServer1, '\n');
                L1_UnlockResource_NW(StdioHostServer1Res);
       }
}
```

Remarks

```
REQ Resource
SPC Resource
```

6.31.6 Typedef Documentation

6.31.6.1 typedef struct _struct_L1_ResourceState_ L1_Resource_HubState

State of a Resource-Hub.

Remarks

```
SPC Resource state variables
```

6.31.7 Function Documentation

6.31.7.1 static __inline__ L1_BOOL L1_isHubResourceLocked (L1_Hub * *pHub*) [static]

Determines whether the Resource-Hub identified by pHub is locked.

Parameters

pHub	Pointer to the Hub data structure of a Resource-Hub.

Returns

L1_TRUE If the FIFO is locked.
L1_FALSE If the FIFO is unlocked.

6.31.7.2 static __inline__ L1_BOOL L1_isResourceHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a Resource-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a Resource-Hub.

Returns

L1_TRUE If the data structure represents a Resource-Hub.
L1_FALSE Otherwise.

6.31.7.3 static __inline__ L1_ReturnCode L1_LockResource_NW (L1_HubID *HubID*) [static]

Locks a logical Resource. This service does return immediately, even if it could not lock the resource.

Parameters

HubID	identifies the Resource-Hub, that the calling Task wants to lock.

Returns

RC_OK service successful (the resource was acquired and locked)
RC_FAIL service failed (the resource was not acquired)

Precondition

- None

Postcondition

- Calling task ready

6.31.7.4 static __inline__ L1_ReturnCode L1_LockResource_W (L1_HubID *HubID*) [static]

Locks a logical Resource. This service waits until it could lock the logical Resource.

Parameters

HubID	identifies the Resource-Hub, that the calling Task wants to lock.

Returns

RC_OK service successful (the resource was acquired and locked)
RC_FAIL service failed (the resource was not acquired)

Precondition

- None

Postcondition

- Calling task ready

6.31.7.5 static __inline__ L1_ReturnCode L1_LockResource_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

Locks a logical Resource. This service waits until it either could lock the resource or the timeout expired.

Parameters

HubID	identifies the Resource-Hub, that the calling Task wants to lock
timeout	the number of system ticks the call should wait for synchronisation.

Returns

RC_OK service successful (the resource was acquired and locked)
RC_FAIL service failed (the resource was not acquired)
RC_TO service timed out.

Precondition

- None

Postcondition

- Calling task ready

6.31.7.6 static __inline__ L1_ReturnCode L1_UnlockResource_NW (L1_HubID *HubID*) [static]

Unlocks a logical Resource. This service returns immediately, independent whether or not it could unlock the resource.

Parameters

HubID	identifies the Resource-Hub, that the calling Task wants to unlock.

Returns

RC_OK service successful (the resource was released)
RC_FAIL service failed (the resource could not be unlocked)

Precondition

- None

Postcondition

- Calling task ready

6.31.7.7 L1_BOOL ResourceSyncCondition (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function returns true when synchronization is achieved. For a lock resource request, it returns true if the resource is unlocked. If the resource was previously locked, the function returns false and RC_FAIL in the status of the packet received. For an unlock resource request, it returns L1_TRUE if the resource was previously locked and the task unlocking it is the same than the one that set the lock. Otherwise it returns L1_FALSE and RC_FAIL in the status of the packet received.

Parameters

Hub	Pointer to a hub of type L1_RESOURCE.
Packet	Packet used for the synchronization operation.

Returns

L1_TRUE If synchronisation was achieved.
L1_FALSE If synchronisation was not achieved.

6.31.7.8 void ResourceSynchronize (L1_Hub * *hub,* L1_Packet * *packet,* L1_Packet * *waitingPacket*)

This function shall utilise the function ResourceSyncCondition() to check whether the L1_Packet packet contains a valid request. In which case it shall call ResourceUpdate with the L1_Packet packet as parameter, followed by a call to ResourceUpdate with the L1_Packet waitingPacket as parameter.

Otherwise, the function shall return the L1_Packet packet with packet->Status set to RC_FAIL and insert the L1_Packet waitingPacket into the Hub-WaitingList.

Parameters

hub	Pointer to a hub of type L1_RESOURCE.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL
packet->ServiceID = L1_SID_PUT_TO_HUB
waitingPacket->ServiceID = L1_SID_GET_FROM_HUB

See Also

ResourceSyncCondition
ResourceUpdate

6.31.7.9 void ResourceUpdate (L1_Hub * *Hub,* L1_Packet * *Packet*)

This function updates the state of the resource when required. When a request to lock a resource is received, it sets the resource to locked, and it sets the owner to the task that locked the resource. When a request to unlock the resource is received, the resource is unlocked.

Parameters

Hub	Pointer to a hub of type L1_RESOURCE.
Packet	Packet used for the update operation.

6.32 Semaphore Hub

Data Structures

- struct _struct_L1_SemaphoreState_

Typedefs

- typedef struct _struct_L1_SemaphoreState_ L1_Semaphore_HubState

Functions

- static __inline__ L1_ReturnCode L1_Drv_Isr_SignalSemaphore_NW (L1_HubID semaphore, L1_Packet ∗packet)
- static __inline__ L1_BOOL L1_isHubSemaphoreSet (L1_Hub ∗pHub)
- static __inline__ L1_BOOL L1_isSemaphoreHub (L1_Hub ∗pHub)
- static __inline__ L1_ReturnCode L1_SignalSemaphore_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_SignalSemaphore_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_SignalSemaphore_WT (L1_HubID HubID, L1_Timeout timeout)
- static __inline__ L1_ReturnCode L1_TestSemaphore_A (L1_HubID HubID, L1_Packet ∗packet)
- static __inline__ L1_ReturnCode L1_TestSemaphore_NW (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_TestSemaphore_W (L1_HubID HubID)
- static __inline__ L1_ReturnCode L1_TestSemaphore_WT (L1_HubID HubID, L1_Timeout timeout)
- L1_BOOL SemaphoreSyncCondition (L1_Hub ∗Hub, L1_Packet ∗Packet)
- void SemaphoreUpdate (L1_Hub ∗Hub, L1_Packet ∗Packet)

6.32.1 Detailed Description

Represents a counting semaphore, this means that the Semaphore-Hub counts how often it has been signaled and then permits to being tested the same amount of times.

6.32.2 Visual Designer

Figure 6.11: Application Diagram Icon

6.32.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- ceilingPriority: The ceiling priority of the Entity.

6.32.3 Example

This example demonstrates the Tasks synchronization mechanism via the Semaphore Hub, by implementing a so called Semaphore-loop. In the Semaphore-loop Task1 signals Seamaphore Sema1, while Task2 waits for Sema1 to be signalled. Upon being signalled Task2 signals Sema2 for which Task1 waits to become signalled. Then the whole thing repeats.

6.32.3.1 Entities

- Task1: Task1EntryPoint, shown in section Source Code of Task1EntryPoint
- Task1: Task2EntryPoint, shown in section Source Code of Task2EntryPoint
- Sema1: Semaphore Hub
- Sema2: Semaphore Hub
- StdioHostServer1: Stdio Host Server used to print messages onto the screen.

6.32.4 Source Code of Task1EntryPoint

```
#include <L1_api.h>
#include "L1_node_config.h"
#include <StdioHostService/StdioHostClient.h>

void Task1EntryPoint(L1_TaskArguments Arguments)
{
    while(1)
    {
        Shs_putString_W(StdioHostServer1, "Task 1 signals Sema 1\n");
        if(RC_OK != L1_SignalSemaphore_W(Sema1))
        {
            Shs_putString_W(StdioHostServer1, "Not Ok\n");
        }

        Shs_putString_W(StdioHostServer1, "Task 1 tests Sema 2\n");
        if(RC_OK != L1_TestSemaphore_W(Sema2))
        {
            Shs_putString_W(StdioHostServer1, "Not Ok\n");
        }
    }
}
```

6.32.5 Source Code of Task2EntryPoint

```
#include <L1_api.h>
#include <L1_node_config.h>
#include <StdioHostService/StdioHostClient.h>

void Task2EntryPoint(L1_TaskArguments Arguments)
{
    while(1)
    {
        Shs_putString_W(StdioHostServer1, "Task 2 tests Sema 1\n");
        if(RC_OK != L1_TestSemaphore_W(Sema1))
        {
            Shs_putString_W(StdioHostServer1, "Not Ok\n");
        }
        Shs_putString_W(StdioHostServer1, "Task 2 signals Sema 2\n");
        if(RC_OK != L1_SignalSemaphore_W(Sema2))
        {
            Shs_putString_W(StdioHostServer1, "Not Ok\n");
        }
    }
}
```

Remarks

```
REQ Semaphore
SPC Semaphore
```

6.32.6 Typedef Documentation

6.32.6.1 typedef struct _struct_L1_SemaphoreState_ L1_Semaphore_HubState

State of a Semaphore-Hub.

Remarks

```
SPC Semaphore state variable
```

6.32.7 Function Documentation

6.32.7.1 static __inline__ L1_ReturnCode L1_Drv_Isr_SignalSemaphore_NW (L1_HubID *semaphore*, L1_Packet * *packet*) [static]

Signals a semaphore, i.e. increases the semaphore count. This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters:

Parameters

semaphore	is the L1_HubID which identifies the Semaphore, that the calling ISR wants to signal/
packet	Pointer to the L1_Packet that will be used to represent the interaction. This L1_Packet must have been once initialized using the function L1_Drv_Isr_initialisePacket().

Returns

RC_OK The packet that raises the Event could be inserted into the Kernel Input Port.
RC_FAIL The packet that raises the Event could not be inserted into the Kernel Input Port.

Warning

Must not be used with Semaphore-Hubs located at another Node.
Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

SPC Packets from an ISR

6.32.7.2 static __inline__ L1_BOOL L1_isHubSemaphoreSet (L1_Hub * *pHub*) [static]

Determines if the count of an L1_Semaphore_HubState is higher than zero (semaphore set).

Parameters

pHub	Pointer to a Hub data structure of a Semaphore-Hub.

Returns

L1_TRUE If the Semaphore-Hub is set.
L1_FALSE Otherwise.

6.32.7.3 static __inline__ L1_BOOL L1_isSemaphoreHub (L1_Hub * *pHub*) [static]

Checks whether or not the given data structure represents a Semaphore-Hub.

Parameters

pHub	Pointer to the data structure, of type L1_Hub, which should be checked whether or not it represents a Semaphore-Hub.

Returns

L1_TRUE If the data structure represents a Semaphore-Hub.
L1_FALSE Otherwise.

6.32.7.4 static __inline__ L1_ReturnCode L1_SignalSemaphore_NW (L1_HubID *HubID*) [static]

Signals a semaphore, i.e. increases the semaphore count. This call returns immediately.

Parameters:

Parameters

HubID	is the L1_HubID which identifies the Semaphore, that the calling Task wants to signal

Returns

L1_ReturnCode:

- RC_OK service successful (the semaphore count was incremented)
- RC_FAIL service failed (the semaphore count was not incremented)

Precondition

- None

Postcondition

- Semaphore count incremented
- Calling tasks ready

6.32.7.5 static __inline__ L1_ReturnCode L1_SignalSemaphore_W (L1_HubID *HubID*) [static]

Signals a semaphore, i.e. increases the semaphore count. This call waits until it could increment the Semaphore count.

Parameters:

Parameters

HubID	the L1_HubID which identifies the Semaphore, that the calling Task wants to signal

Returns

L1_ReturnCode:

- RC_OK service successful (the semaphore count was incremented)
- RC_FAIL service failed (the semaphore count was not incremented)

Precondition

- None

Postcondition

- Semaphore count incremented
- Calling tasks ready

6.32.7.6 static __inline__ L1_ReturnCode L1_SignalSemaphore_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

Signals a semaphore, i.e. increases the semaphore count. This service waits until it either could increment the semaphore count or the timeout expired.

Parameters:

Parameters

HubID	is the L1_HubID which identifies the Semaphore, that the calling Task wants to signal
timeout	the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode:

- RC_OK service successful (the semaphore count was incremented)
- RC_FAIL service failed (the semaphore count was not incremented)
- RC_TO service timed out.

Precondition

- None

Postcondition

- Semaphore count incremented
- Calling tasks ready

6.32.7.7 static __inline__ L1_ReturnCode L1_TestSemaphore_A (L1_HubID *HubID,* L1_Packet * *packet*) [static]

Tests whether or not a Semaphore is ready, i.e. the semaphore count is larger than zero. The completion is deferred until a corresponding L1_WaitForPacket call happens.

Parameters

HubID	is of type L1_HubID and identifies the Event, that the calling Task wants to test.
packet	Pointer to the L1_Packet that will be used for this Asynchronous-Interaction. There are two ways of acquiring such an L1_Packet: • Allocate an L1_Packet in your Task, and then initialise it using the function L1_-initialiseAsyncPacket(). • Allocate an L1_Packet from a local (i.e. same Node) Packet Pool-Hub, using one of the following interactions: L1_AllocatePacket_W(), L1_AllocatePacket_-NW(), and L1_AllocatePacket_WT().

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful
- RC_FAIL service failed

Precondition

- Packet is a preallocated L1_Packet, initialised using L1_initialiseAsyncPacket().

Postcondition

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

Warning

This Interaction only works locally (Task and Hub on the same Node), i.e. not distributed (Task and Hub on different Nodes).

See Also

L1_initialiseAsyncPacket
L1_AllocatePacket_W
L1_AllocatePacket_NW
L1_AllocatePacket_WT

6.32.7.8 static __inline__ L1_ReturnCode L1_TestSemaphore_NW (L1_HubID *HubID*) [static]

Tests whether or not a Semaphore is ready, i.e. the semaphore count is larger than zero. This service returns immediately, even if it could not decrement the semaphore counter.

Parameters

HubID	identifies the Semaphore, that the calling Task wants to test.

Returns

L1_ReturnCode

- RC_OK The service call was successful (the semaphore count was >1 and decremented)
- RC_FAIL The service call failed.

Precondition

- None

Postcondition

- Semaphore count is 0 or decremented by one.
- Calling tasks ready

6.32.7.9 static __inline__ L1_ReturnCode L1_TestSemaphore_W (L1_HubID *HubID*) [static]

Tests whether or not a Semaphore is ready, i.e. the semaphore count is larger than zero. This service waits until it could decrement the semaphore count.

Parameters

HubID	identifies the Semaphore-Hub, that the calling Task wants to test.

Returns

L1_ReturnCode

- RC_OK The service call was successful (the semaphore count was >1 and decremented)
- RC_FAIL The service call failed.

Precondition

- None

Postcondition

- Semaphore count is 0 or decremented by one.
- Calling tasks ready

6.32.7.10 static __inline__ L1_ReturnCode L1_TestSemaphore_WT (L1_HubID *HubID,* L1_Timeout *timeout*) [static]

Tests whether or not a Semaphore is ready, i.e. the semaphore count is larger than zero. This service waits until it either could decrement the semaphore or the timeout expired.

Parameters

HubID	is of type L1_HubID and identifies the Semaphore, that the calling Task wants to test.
timeout	the number of system ticks the call should wait for synchronisation.

Returns

L1_ReturnCode, the following return values are possible:

- RC_OK service successful (there was a set Event)
- RC_FAIL service failed (there was no set Event)
- RC_TO service timed out.

Precondition

- None

Postcondition

- Semaphore count is 0 or decremented by one.
- Calling tasks ready

6.32.7.11 L1_BOOL SemaphoreSyncCondition (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function evaluates if the update function should be executed, depending on the type of packet received. For put packets, the semaphore sync condition is always true. For get packets, the semaphore sync condition is true if the count is larger than zero.

Parameters

Hub	Semahpore hub that is tested for synchronization.
Packet	Put or get packet received by the hub.

Returns

L1_BOOL

- L1_TRUE Update condition is true (update function should be called).
- L1_FALSE Update condition is false (update function should not be called).

Precondition

- Hub is of L1_Hub type.
- empty waiting list, as complimentary packets are already accounted for.

6.32.7.12 void SemaphoreUpdate (L1_Hub * *Hub*, L1_Packet * *Packet*)

This function updates the state of a Semaphore Hub, depending on the type of packet received. Put packets signal the semaphore. Get packets test the semaphore.

Parameters

Hub	Event hub that is updated.
Packet	Put or get packet received by the hub.

Precondition

- Hub is of L1_Hub type.

6.33 Memory Block Queue Hub

Data Structures

- struct L1_MemoryBlockQueue_HubState

Macros

- #define L1_isMemoryBlockQueueHub(h) ((h)->HubType == L1_MEMORY_BLOCK_QUEUE)

Enumerations

- enum MemoryBlockQueueHub_IOCTL_CODES { , L1_IOCTL_MBQ_ISR_SEND_BLOCK = (L1_IOCTL_HUB_END + 3) }

Functions

- L1_ReturnCode L1_AcquireMemoryBlock_NW (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock)
- L1_ReturnCode L1_DequeueMemoryBlock (L1_HubID hubID, L1_Packet *packet, L1_MemoryBlock **ppMemoryBlock, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_NW (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock)
- static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_W (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock)
- static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_WT (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock, L1_Timeout Timeout)
- static L1_ReturnCode L1_Drv_Isr_EnqueueMemoryBlock_NW (L1_HubID mbq, L1_Packet *packet, L1_BYTE *buffer, L1_UINT32 dataSize, L1_BOOL urgent)
- L1_ReturnCode L1_EnqueueMemoryBlock (L1_HubID hubID, L1_Packet *packet, L1_MemoryBlock **ppMemoryBlock, L1_BOOL urgent, L1_Timeout Timeout)
- static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_NW (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock, L1_BOOL urgent)
- static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_W (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock, L1_BOOL urgent)
- static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_WT (L1_HubID hubID, L1_MemoryBlock **ppMemoryBlock, L1_BOOL urgent, L1_Timeout Timeout)
- static __inline__ L1_BYTE * L1_MB_getMemory (L1_MemoryBlock *pMemoryBlock)
- static __inline__ L1_UINT32 L1_MB_getNbrOfUsedBytes (L1_MemoryBlock *pMemoryBlock)
- static __inline__ L1_UINT32 L1_MB_getSize (L1_MemoryBlock *pMemoryBlock)
- static __inline__ void L1_MB_setNbrOfUsedBytes (L1_MemoryBlock *pMemoryBlock, L1_UINT32 nbrOfUsedBytes)
- L1_ReturnCode L1_ReturnMemoryBlock_NW (L1_HubID hubID, L1_MemoryBlock *pMemoryBlock)
- void MemoryBlockQueueHub_Ioctl (L1_Hub *hub, L1_Packet *packet, L1_BYTE ioctl_type)
- L1_BOOL MemoryBlockQueueHub_SyncCondition (L1_Hub *hub, L1_Packet *packet)
- void MemoryBlockQueueHub_Synchronize (L1_Hub *hub, L1_Packet *packet, L1_Packet *waitingPacket)
- void MemoryBlockQueueHub_Update (L1_Hub *hub, L1_Packet *packet)

6.33.1 Detailed Description

The Memory Block Queue (MBQ) Hub offers a zero-copy mechanism to exchange data between two Tasks.

6.33.2 Visual Designer

Figure 6.12: Application Diagram Icon

6.33.2.1 Properties

The Entity has the following Properties:

- node: The name of the Node to which the Entity is mapped.
- name: Name of the Entity instance.
- sizeOfBlocks: The size of the memory blocks the Hub provides, in Bytes.
- nbrOfBlocks: How many memory blocks the Hub provides.

Remarks

```
SPC Memory Block Queue
SPC Memory Management
```

6.33.3 Macro Definition Documentation

6.33.3.1 #define L1_isMemoryBlockQueueHub(*h*) ((h)->HubType == L1_MEMORY_BLOCK_QUEUE)

This macro checks whether or not a Hub is of type MemoryBlockQueue

Returns

- 0: If the Hub is not a MemoryBlockQueue Hub.
- 1: If the Hub is a MemoryBlockQueue Hub.

6.33.4 Enumeration Type Documentation

6.33.4.1 enum MemoryBlockQueueHub_IOCTL_CODES

IOCTL-codes for the Memory Block Queue Hub.

Enumerator

L1_IOCTL_MBQ_ISR_SEND_BLOCK The data-part of packets with this IOCTL-Code gets copied into a free Memory-Block and then added to the Queue.

6.33.5 Function Documentation

6.33.5.1 L1_ReturnCode L1_AcquireMemoryBlock_NW (L1_HubID *hubID,* L1_MemoryBlock ** *ppMemoryBlock*)

Tries to acquire a free memory-block from a memory-buffer-queue-hub.

Parameters

hubID	[in] of the Memory-Block-Queue-Hub.
ppMemory-Block	[inout] Pointer to a pointer to an L1_MemoryBlock, where the pointer to the L1_-MemoryBlock will be returned to.

Returns

RC_OK Could acquire the Memory Block.
RC_FAIL Could not acquire the Memory Block.

6.33.5.2 L1_ReturnCode L1_DequeueMemoryBlock (L1_HubID *hubID*, L1_Packet * *packet*, L1_MemoryBlock ** *ppMemoryBlock*, L1_Timeout *Timeout*)

Sends a Get-Packet containing the pointer to the Memory-Block indicated by ppMemoryBlock to the MB-Q-Hub. There the Hub checks whether the Memory-Block is of the correct size. If that is not the case the interaction will return RC_FAIL. Furthermore, the Hub checks whether there is currently a used Memory-Block available, i.e. if the Hub is currently empty or not. If the Hub is empty the Get-Packet will be inserted into the waiting list depending on the chosen interaction semantics. Otherwise the Hub will insert the Memory-Block into the list of free Memory-Blocks, and retrieve a Memory-Block from the full Memory--Block list. Upon returning ppMemoryBlock will return the pointer to the Memory-Block retrieved from the free Memory-Block list.

Parameters

hubID	[in] ID of the Memory-Block-Queue-Hub.
packet	[inout] Pointer to the Packet to use to perform the interaction
ppMemory-Block	[inout] Pointer to a pointer to an L1_MemoryBlock.
Timeout	[in] How long to wait for the execution of the operation.

Returns

RC_OK The Interaction was done successfully.
RC_TO The Interaction timed out.
RC_FAIL The interaction failed.

6.33.5.3 static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_NW (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*) [static]

Acquires a full Memory-Block from the Memory Block Queue Hub with the ID hubID. The interaction expects that ppMemoryBlock points to a now free Memory-Block which shall be returned to the hub.

If currently no full Memory-Block is available, the interaction shall return directly and return RC_FAIL.

Parameters

hubID	[in] ID of the Memory-Block-Queue-Hub.
ppMemory-Block	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the full Memory-Block that was dequeued.
RC_FAIL The interaction was unsuccessful.

6.33.5.4 static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_W (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*) [static]

Acquires a full Memory-Block from the Memory Block Queue Hub with the ID hubID. The interaction expects that ppMemoryBlock points to a now free Memory-Block which shall be returned to the hub.

If currently no full Memory-Block is available, the interaction shall wait until a full Memory-Block becomes available.

Parameters

hubID	[in] ID of the Memory-Block-Queue-Hub.
ppMemory-Block	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the full Memory-Block that was dequeued.
RC_FAIL The interaction was unsuccessful.

6.33.5.5 static __inline__ L1_ReturnCode L1_DequeueMemoryBlock_WT (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*, L1_Timeout *Timeout*) [static]

Acquires a full Memory-Block from the Memory Block Queue Hub with the ID hubID. The interaction expects that ppMemoryBlock points to a now free Memory-Block which shall be returned to the hub.

If currently no full Memory-Block is available, the interaction shall wait until a full Memory-Block becomes available, or the timeout expires, before returning.

Parameters

hubID	[in] ID of the Memory-Block-Queue-Hub.
ppMemory-Block	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.
Timeout	How long the interaction shall wait, in system ticks, to perfrom the interaction.

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the full Memory-Block that was dequeued.
RC_FAIL The interaction was unsuccessful.
RC_TO The timeout expired.

6.33.5.6 static L1_ReturnCode L1_Drv_Isr_EnqueueMemoryBlock_NW (L1_HubID *mbq*, L1_Packet * *packet*, L1_BYTE * *buffer*, L1_UINT32 *dataSize*, L1_BOOL *urgent*) [inline], [static]

This function inserts the content of the parameter buffer into a Memory Buffer Queue Hub, without the need to previously allocating a Buffer.

This function returns directly, like an Asynchronous Interaction, but the packet will never be returned to the ISR.

Parameters

mbq	[in] Is the L1_HubID which identifies the Memory Block Queue Hub, that the calling ISR wants to enqueue the buffer at.
packet	[inout] Pointer to the L1_Packet that will be used to represent the interaction.
buffer	[in] Pointer to the buffer that will be enqueued in the Memory Buffer Queue.
dataSize	[in] Size of the buffer to enqueue, in bytes. Must not be larger than the size of the Memory Blocks handled by the Message Buffer Queue.

urgent	[in] Indicates whether a message in the Memory-Block is urgent (L1_TRUE) or not (L1_FALSE). In case of the message being urgent the Memory-Block gets inserted in the beginning of the list of the list, otherwise it will be enqueued at the end of the list.

Returns

RC_OK The packet could be inserted into the Kernel Input Port.
RC_FAIL The packet could not be inserted into the Kernel Input Port.

Warning

Must not be used with Memory Buffer Queue Hubs located at another Node.
Only to be used within an ISR, not within a Task!

See Also

L1_Drv_Isr_initialisePacket

Remarks

```
SPC Packets from an ISR
```

Remarks

```
SPC Memory Block Queue initial state of MBQ-FullBlocks
```

6.33.5.7 L1_ReturnCode L1_EnqueueMemoryBlock (L1_HubID *hubID*, L1_Packet * *packet*, L1_MemoryBlock ** *ppMemoryBlock*, L1_BOOL *urgent*, L1_Timeout *Timeout*)

Sends a Put-Packet containing the pointer to the indicated by ppBuffer to the MBQ-Hub. There the Hub checks whether the Memory-Block has the correct size. If that is not the case the interaction will return RC_FAIL. Furthermore, the Hub checks whether there is currently a free Memory-Block available, i.e. if the Hub is currently full or not. If the Hub is full the Put-Packet will be inserted into the waiting list depending on the chosen interaction semantics. Otherwise, the Hub will insert the Memory-Block into the list of used Memory-Blocks, and retrieve a Memory-Block from the free Memory-Block list. Upon returning ppMemoryBlock will return the pointer to the Memory-Block retrieved from the free Memory-Block list.

Parameters

hubID	[in] ID of the Memory-Block-Queue-Hub.
packet	[inout] Pointer to the Packet to use to perform the interaction
ppMemory-Block	[inout] Pointer to a pointer to an L1_MemoryBlock.
urgent	[in] Indicates whether a message in the Memory-Block is urgent (L1_TRUE) or not (L1_FALSE). In case of the message being urgent the Memory-Block gets inserted in the beginning of the list of the list, otherwise it will be enqueued at the end of the list.
Timeout	How long to wait for the execution of the operation.

Returns

RC_OK The Interaction was done successfully.
RC_TO The Interaction timed out.
RC_FAIL The interaction failed.

6.33.5.8 static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_NW (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*, L1_BOOL *urgent*) [static]

Enqueues the Memory-Block ppMemoryBlock points to in the MBQ-Hub identified by the parameter hubID. It allows to specify whether the Memory-Block should be inserted at the beginning of the queue (urgent = L1_TRUE) or at the end (urgent = L1_FALSE). Upon successful return of the function ppMemoryBlock points to a free Memory-Block which can be used by the Task directly.

If currently no free Memory-Block is available in the Hub, then this function shall return directly with the return value RC_FAIL.

Parameters

hubID	[in] ID of the Memory Block Queue Hub to interact with.
ppMemoryBlock	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.
urgent	[in] Indicates whether the Memory-Block is urgent (L1_TRUE) or not (L1_FALSE).

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the free Memory-Block to be used by the Task.
RC_FAIL The interaction was unsuccessful.

6.33.5.9 static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_W (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*, L1_BOOL *urgent*) [static]

Enqueues the Memory-Block ppMemoryBlock points to in the MBQ-Hub identified by the parameter hubID. It allows to specify whether the Memory-Block should be inserted at the beginning of the queue (urgent = L1_TRUE) or at the end (urgent = L1_FALSE). Upon successful return of the function ppMemoryBlock points to a free Memory-Block which can be used by the Task directly.

If currently no free Memory-Block is available in the Hub, then this function shall wait until a free Memory-Block becomes available before returning.

Parameters

hubID	[in] ID of the Memory Block Queue Hub to interact with.
ppMemoryBlock	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.
urgent	[in] Indicates whether the Memory-Block is urgent (L1_TRUE) or not (L1_FALSE).

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the free Memory-Block to be used by the Task.
RC_FAIL The interaction was unsuccessful.

6.33.5.10 static __inline__ L1_ReturnCode L1_EnqueueMemoryBlock_WT (L1_HubID *hubID*, L1_MemoryBlock ** *ppMemoryBlock*, L1_BOOL *urgent*, L1_Timeout *Timeout*) [static]

Enqueues the Memory-Block ppMemoryBlock points to in the MBQ-Hub identified by the parameter hubID. It allows to specify whether the Memory-Block should be inserted at the beginning of the queue (urgent

= L1_TRUE) or at the end (urgent = L1_FALSE). Upon successful return of the function ppMemoryBlock points to a free Memory-Block which can be used by the Task directly.

If currently no free Memory-Block is available in the Hub, then this interaction shall wait until a free Memory-Block becomes available, or the timeout expires, before returning.

Parameters

hubID	[in] ID of the Memory Block Queue Hub to interact with.
ppMemory-Block	[inout] Pointer to the full Memory-Block to be enqueued. Upon successful return of the function this will point to a free Memory-Block.
urgent	[in] Indicates whether the Memory-Block is urgent (L1_TRUE) or not (L1_FALSE).
Timeout	How long the interaction shall wait, in system ticks, to perform the interaction.

Returns

RC_OK The interaction was successful, ppMemoryBlock points to the free Memory-Block to be used by the Task.
RC_FAIL The interaction was unsuccessful.
RC_TO The timeout expired.

6.33.5.11 static __inline__ L1_BYTE* L1_MB_getMemory (L1_MemoryBlock * *pMemoryBlock*) [static]

Returns the pointer to the first byte of the data-part of the Memory-Block indicated by pMemoryBlock.

Parameters

pMemoryBlock	[in] Pointer to the Memory-Block from which to acquire the data-part.

Returns

Pointer to the first byte of the data-part of the Memory-Block.

6.33.5.12 static __inline__ L1_UINT32 L1_MB_getNbrOfUsedBytes (L1_MemoryBlock * *pMemoryBlock*) [static]

Returns the number of bytes currently used in the Memory-Block pointed to by pMemoryBlock.

Parameters

pMemoryBlock	[in] Pointer to the Memory-Block from which to get the number of used bytes.

Returns

The number of bytes used in the Memory-Block.

See Also

L1_MB_setNbrOfUsedBytes

6.33.5.13 static __inline__ L1_UINT32 L1_MB_getSize (L1_MemoryBlock * *pMemoryBlock*) [static]

Returns the maximum number of bytes the Memory-Block pointed to by pMemoryBlock can store.

Parameters

pMemoryBlock	[in] Pointer to the Memory-Block from which to get the size.

Returns

The maximum number of bytes the Memory-Block can store.

6.33.5.14 static __inline__ void L1_MB_setNbrOfUsedBytes (L1_MemoryBlock * *pMemoryBlock,* L1_UINT32 *nbrOfUsedBytes*) [static]

Sets the number of bytes currently used in the Memory-Block pointed to by pMemoryBlock.

Parameters

pMemoryBlock	[in] Pointer to the Memory-Block where to set the number of used bytes.
nbrOfUsed-Bytes	[in] The the value of used bytes to be set in the Memory-Block.

See Also

L1_MB_getNbrOfUsedBytes

6.33.5.15 L1_ReturnCode L1_ReturnMemoryBlock_NW (L1_HubID *hubID,* L1_MemoryBlock * *pMemoryBlock*)

Returns the MemoryBlock pointed to by pMemoryBlock to the Memory Block Queue Hub with the ID hubID.

Parameters

hubID	[in] of the Memory-Block-Queue-Hub.
pMemoryBlock	[in] Pointer to the Memory-Block to return to the Hub.

Returns

RC_OK Could return the Memory Block.
RC_FAIL Could not return the Memory Block.

6.33.5.16 void MemoryBlockQueueHub_Ioctl (L1_Hub * *hub,* L1_Packet * *packet,* L1_BYTE *ioctl_type*)

Parameters

hub	Pointer to a Hub of type L1_MEMORY_BLOCK_QUEUE.
packet	Pointer to the L1_Packet which caused the function to be called. It will contain additional information.
ioctl_type	The command code, the following command exist: • L1_IOCTL_HUB_OPEN: Initialised the Hub, only used by the Kernel-Task. • L1_IOCTL_MBQ_ACQUIRE_BLOCK: Tries to acquire a Memory Block from the list of free Memory Blocks. • L1_IOCTL_MBQ_RETURN_BLOCK: Returns a Memory Block to the list of free Memory Blocks. • L1_IOCTL_MBQ_ISR_SEND_BLOCK: Copies the data in the data-part of the L1_Packet to a free Memory Block and inserts it into the list of used Memory Blocks.

See Also

L1_AcquireMemoryBlock_NW
L1_ReturnMemoryBlock_NW
L1_Drv_Isr_EnqueueMemoryBlock_NW

6.33.5.17 L1_BOOL MemoryBlockQueueHub_SyncCondition (L1_Hub * *hub*, L1_Packet * *packet*)

Parameters

hub	Pointer to a Hub of type L1_MEMORY_BLOCK_QUEUE.
packet	Pointer to the L1_Packet which caused the function to be called. It will contain additional information.

Returns

L1_TRUE The request in the L1_Packet packet results in synchronisation and the Hub-State shall be updated.
L1_FALSE The request in the L1_Packet packet, either does not result in synchronisation (packet->Status := RC_OK) or is incorrect (packet->Status := RC_FAIL).

6.33.5.18 void MemoryBlockQueueHub_Synchronize (L1_Hub * *hub*, L1_Packet * *packet*, L1_Packet * *waitingPacket*)

This function shall synchronise the two complementary requests in the L1_Packets packet and waitingPacket. It shall first check whether or not the request in packet is valid.

If the request in the L1_Packet packet is valid, it shall update the Hub-State using it, before updating the Hub-State using the L1_Packet waitingPacket. Afterwards it shall return both Packets to their Tasks.

If the request in the L1_Packet packet is invalid, the function shall raise an error (packet->Status := RC_FAIL) and return packet to its Task. The function shall then reinsert the L1_Packet waitingPacket into the Hub WaitingList.

Parameters

hub	Pointer to a Hub of type L1_MEMORY_BLOCK_QUEUE.
packet	Pointer to the newly arrived L1_Packet.
waitingPacket	Pointer to the L1_Packet which is complementary to the one in the parameter packet.

Precondition

hub NOT NULL
packet NOT NULL
waitingPacket NOT NULL

6.33.5.19 void MemoryBlockQueueHub_Update (L1_Hub * *hub*, L1_Packet * *packet*)

This function shall handle Put- and Get-Packets and update the state of the Hub accordingly.

Parameters

hub	Pointer to a Hub of type L1_MEMORY_BLOCK_QUEUE.
packet	Pointer to the L1_Packet which contains the request that results in updating the state of the Memory Block Queue Hub.

6.34 Hardware Abstraction Layer

Functions

- void L1_deinitializeContextOfTask (L1_TaskControlRecord *TaskCR)
- void L1_enterCriticalSection (void)
- void L1_enterISR (void)
- L1_UINT32 L1_hal_SMP_getCoreNumber (void)
- void L1_initializeContextOfTask (L1_TaskControlRecord *TaskCR)
- void L1_initializePlatform (L1_UINT32 NodeNumberOfTasks)
- void L1_leaveCriticalSection (void)
- void L1_leaveISR (void)
- void L1_restoreStatusRegister (L1_INTERRUPT_STATUS msr)
- L1_INTERRUPT_STATUS L1_saveStatusRegister (void)
- void L1_startTasks (void)
- void L1_switchContext (L1_TaskControlRecord *Task2Preempt, L1_TaskControlRecord *TaskCR2Restore)

6.34.1 Detailed Description

This is the interface between the Kernel and the hardware specific platform.

6.34.2 Function Documentation

6.34.2.1 void L1_deinitializeContextOfTask (L1_TaskControlRecord * *TaskCR*)

Platform dependent clearance and removal of CPU context of a task.

Parameters

TaskCR	Reference to the Task Control Record for which the CPU context should be cleared or removed.

Example:

```
void L1_deinitializeContextOfTask (
      L1_TaskControlRecord *TaskCR)
{
    L1_TaskContext *TaskContext = TaskCR->Context;

    // Free all storage and reset any context fields
    // here or in L1_initializeContextOfTask
    ...
    TaskContext->State = TASK_CONTEXT_STATE_TERMINATED;
}
```

Remarks

Function called in response to a L1_StopTask_W service. Function can be empty if the context is never changed or if it is followed by a L1_initializeContextOfTask

See Also

L1_StopTask_W

6.34.2.2 void L1_enterCriticalSection (void)

Synchronization to ensure a single source to access a piece of code or object, i.e. for single entrant code. Platform dependent support to implement an atomic operation or mutex. Paired with L1_leaveCriticalSection().

Used to protect:

- Manipulations of the Kernel Input Port waiting list
- Manipulations of Rx driver packet pool

Warning

Should not be used in new code, instead use L1_saveStatusRegister()

6.34.2.3 void L1_enterISR (void)

Platform dependent prologue of an interrupt service routine.

In most platforms this function does not exist but is part of the Interrupt Controller Driver.

Example:

```
void L1_enterISR(void)
{
    // push registers onto the stack
    // save the task context if not yet done so
    ... enter critical section for nested interrupts ...
    L1_ISRNesting++;
    if (L0_ISRNesting == 1) {
        ... // save context
        ... // switch to ISR context
    }
    .. leave critical section for nested interrupts ...
}
```

Remarks

It is recommended to use a separate context and stack for ISRs.

6.34.2.4 L1_UINT32 L1_hal_SMP_getCoreNumber (void)

Returns the Core-ID of the Core this function is executed on. It is used in SMP-System to emulate Virtuoso-Next standard MP behaviour, i.e. one Kernel per core.

Returns

The number of the core, starting from 0.

6.34.2.5 void L1_initializeContextOfTask (L1_TaskControlRecord * *TaskCR*)

Platform dependent creation and initialization of CPU context of a task.

Parameters

TaskCR	Reference to the Task Control Record for which the CPU context should be initialized.

Example:

```
void L1_initializeContextOfTask (L1_TaskControlRecord  *
      TaskCR)
{
  L1_TaskContext * TaskContext;

  TaskContext = TaskCR->Context;

  // Configure the Stack to call so that loading the context
  // causes the function L1_runTask() to be called.

  ...

  // initialize all context fields
  ...
}
```

Remarks

Function called by L1_initializePlatform and in response to a L1_StartTask_W service.

See Also

L1_StartTask_W
L1_initializePlatform

6.34.2.6 void L1_initializePlatform (L1_UINT32 *NodeNumberOfTasks*)

Remarks

```
SPC HAL API Specification
```

Global platform initialization and creation of all tasks and CPU contexts. The function is called when starting VirtuosoNext to provide any platform dependent initialization and creation of tasks.

Parameters

NodeNumberOfTasks	The number of Tasks on this node, which need to be initialised using the function L1_initializeContextOfTask.

See Also

L1_initializeContextOfTask

Example:

```
//Initialize ISR context, if any
L1_IsrStackPtr = ...;

// Initialize application tasks
for (i = 0; i < NodeNumberOfTasks; i++) {
    if (L1_TaskControlBlock[i].TaskState == L1_STARTED) {
        L1_initializeContextOfTask(&(L1_TaskControlBlock[i]));
     }
}
```

6.34.2.7 void L1_leaveCriticalSection (void)

Synchronization to re-allow multiple sources to access a piece of code and reentrant code. Platform dependent support to implement an atomic operation or mutex. Paired with L1_enterCriticalSection.

Used to protect:

- Manipulations of the Kernel Input Port waiting list
- Manipulations of Rx driver packet pool

Warning

Should not be used in new code, instead use L1_restoreStatusRegister()

6.34.2.8 void L1_leaveISR (void)

Platform dependent epilogue of an interrupt service routine.

In most platforms this function does not exist but is part of the Interrupt Controller Driver.

Example:

```
L1_leaveISR(void){
    ... enter critical section for nested interrupts ...

    L1_ISRNesting--;
    if (L1_ISRNesting == 0) {
        if (L1_CurrentTaskCR == L1_KernelTaskCR) {
            // only if in process of deschedule
            L1_ScheduleRequest = L1_TRUE;
            TaskCR2Restore = L1_CurrentTaskCR;
        } else if (L1_ScheduleRequest == L1_TRUE) {
            // requested by a user task
            // leave it up to normal flow, since task is about to switch to KernelTas
        } else {
            // we are in a user task, not in a kernel request service, switch to kerr
            L1_ScheduleRequest = L1_FALSE;
            // restore and switch context
            ... L1_switchContext (L1_CurrentTaskCR, L1_KernelTaskCR) ...
        }
        // return
        // resumed the suspended thread again, or other and finish processing this IS
    } else {
        // nested interrupts, switch context when returning from first/original ISR
        // normal flow
    }
    ... restore context ...
    ... leave critical section for nested interrupts ...
    // pop registers from the stack
    ...
}
```

Remarks

It is recommended to use a separate context and stack for ISRs

6.34.2.9 void L1_restoreStatusRegister (L1_INTERRUPT_STATUS *msr*)

Restoring of current machine status info, e.g. interrupt mask, and leaving of critical section for context switch as a side-effect. Paired with L1_saveStatusRegister.

Parameters

msr	Platform dependent status information, e.g. interrupt mask.

Warning

The parameter msr must be a return value from L1_saveStatusRegister.

Remarks

Called with value returned from L1_saveStatusRegister
Should be used instead of L1_leaveCriticalSection in new code.

6.34.2.10 L1_INTERRUPT_STATUS L1_saveStatusRegister (void)

Retrieving of current machine status info, e.g. interrupt mask, and entering of critical section for context switch as a side-effect. Paired with L1_restoreStatusRegister.

Example

```
L1_INTERRUPT_STATUS L1_saveStatusRegister(void)
{
    L1_INTERRUPT_STATUS status;
    status = SREG;    // get Status Register
    ... disable interrupts ...
    return status;
}
```

Returns

The value of the MSR before interrupts got disabled.

Remarks

Return value used as argument to L1_restoreStatusRegister.
Should be used instead of L1_enterCriticalSection in new code.

6.34.2.11 void L1_startTasks (void)

Platform dependent function to start execution of the VirtuosoNext tasks on the CPU. The context of the first task to start should be loaded, and the CPU resources given to that task (typically the Kernel Task).

Example:

```
void L1_startTasks (void)
{
    L1_CurrentTaskCR = L1_KernelTaskCR;
    Stack2SwitchON  = L1_CurrentTaskCR->Context->StackPtr;

    // load context
    ... load the new stack pointer Stack2SwitchON ...
    ... pop registers from the stack ...
}
```

6.34.2.12 void L1_switchContext (L1_TaskControlRecord * *Task2Preempt*, L1_TaskControlRecord * *TaskCR2Restore*)

Platform dependent function to yield execution to another task to run on the CPU.

The context of the task to pre-empt should be saved, and the context of the task to resume should be restored. The CPU resources will then be given to the restored task.

Parameters

Task2Preempt	Reference to the Task that will yield its execution (typically L1_CurrentTaskCR)
TaskCR2-Restore	Reference to the Task that will resume execution.

Example:

```
void L1_switchContext (L1_TaskControlRecord * Task2Preempt,
      L1_TaskControlRecord * Task2Restore)
{
    Stack2SwitchON  = Task2Restore->Context->StackPtr;
    Stack2SwitchOFF = &(Task2Preempt->Context->StackPtr);
    L1_CurrentTaskCR = Task2Restore;

    // save context
    ... push registers on the stack ...
    ... store the current stack pointer Stack2SwitchOFF ...
    // load context
    ... load the new stack pointer Stack2SwitchON ...
    ... pop registers from the stack ...
}
```

Remarks

VirtuosoNext selects the task to restore. Function is called from within a L1_enterCriticalSection / L1_leaveCriticalSection pair.

See Also

L1_saveStatusRegister
L1_restoreStatusRegister

6.35 Internal Kernel API

Data Structures

- struct _struct_L1_Port_
- struct L1_TaskControlRecord

Macros

- #define L1_id2localport(p) (&L1_LocalPorts[((p) & ~L1_GLOBALID_MASK)])

- #define L1_isLocalPortID(p) (((p) & L1_GLOBALID_MASK) == ((L1_KernelInputPortID) & L1-_GLOBALID_MASK))
- #define L1_isLocalTaskID(tid) (((tid) & L1_GLOBALID_MASK) == ((L1_KernelTaskID) & L1_-GLOBALID_MASK))
- #define L1_PortNodeID(p) (((p) & L1_GLOBALID_MASK) >> 8)
- #define L1_thisNodeID ((L1_NodeID)((L1_KernelTaskID) & L1_GLOBALID_MASK))

Typedefs

- typedef struct _struct_L1_Port_ L1_InputPort

Enumerations

- enum L1_TaskStatus { L1_INACTIVE, L1_STARTED, L1_ABORTED }

Functions

- void inputPortService (L1_InputPort ∗Port, L1_Packet ∗Packet)
- void L1_abortTaskService (L1_Packet ∗packet)
- void L1_anyPacketService (L1_Packet ∗Packet)
- L1_ReturnCode L1_buildAndInsertPacket (L1_PortID PortID, L1_Packet ∗Packet, L1_UINT16 Service-ID, L1_Timeout Timeout)
- void L1_changeTaskPriority (L1_TaskControlRecord ∗TaskCR, L1_Priority newPriority)
- void L1_idleTask (L1_TaskArguments Arguments)
- void L1_initLinkDriver (void)
- void L1_KernelEntryPoint (L1_TaskArguments Arguments)
- void L1_KernelLoop (void)
- L1_Packet ∗ L1_KernelPacketPool_getPacket (void)
- static void L1_List_insertTask (L1_List ∗list, L1_TaskControlRecord ∗taskCR)
- static void L1_List_removeTask (L1_TaskControlRecord ∗taskCR)
- void L1_makeTaskReady (L1_TaskControlRecord ∗TaskCR)
- void L1_remoteService (L1_Packet ∗Packet)
- void L1_resetTimer (L1_Packet ∗waitingPacket)
- void L1_resumeTaskService (L1_Packet ∗Packet)
- void L1_returnPacketService (L1_Packet ∗Packet)
- void L1_returnToTask (L1_Packet ∗Packet)
- int L1_runRTOS (void)
- void L1_runTask (L1_TaskControlRecord ∗taskCR)
- static int L1_runVirtuosoNext (L1_UINT32 nbrOfTasks, L1_UINT32 nbrOfHubs)
- L1_ReturnCode L1_setTimer (L1_Packet ∗Packet)
- void L1_startTaskService (L1_Packet ∗Packet)
- void L1_stopTaskService (L1_Packet ∗Packet)
- void L1_suspendTaskService (L1_Packet ∗Packet)
- void L1_timerPacketService (L1_Packet ∗Packet)
- void L1_timerPacketService_tick (L1_Packet ∗packet)

Variables

- L1_TimerList L1_NodeTimerTimeoutList

6.35.1 Detailed Description

Remarks

SPC Kernel Internal API

6.35.2 Macro Definition Documentation

6.35.2.1 #define L1_id2localport(*p*) (&L1_LocalPorts[((p) & ~L1_GLOBALID_MASK)])

Converts the Port ID p to a pointer to the Hub on the local Node.

Parameters

p	The Port ID to be converted.

Returns

The pointer to the L1_IntputPort identified by the Port ID p.

Warning

This function does not check whether this Port ID identifies a local Port. For this the developer must use L1_isLocalPortID().

See Also

L1_isLocalPortID

6.35.2.2 #define L1_isLocalPortID(*p*) (((p) & L1_GLOBALID_MASK) == ((L1_KernelInputPortID) & L1_GLOBALID_MASK))

Checks whether or not the L1_InputPort identified by parameter p is on this Node, i.e. is locally available.

Parameters

p	The Port ID to be checked.

Returns

L1_TRUE The p identifies an Input Port on this Node.
L1_FALSE Otherwise.

6.35.2.3 #define L1_isLocalTaskID(*tid*) (((tid) & L1_GLOBALID_MASK) == ((L1_KernelTaskID) & L1_GLOBALID_MASK))

Checks whether or not the Task identified by parameter tid is on this Node, i.e. is locally available.

Parameters

tid	The Task ID to be checked.

Returns

L1_TRUE The Task ID tid identifies a Task on this Node.
L1_FALSE Otherwise.

6.35.2.4 #define L1_PortNodeID(*p*) (((p) & L1_GLOBALID_MASK) >> 8)

Extracts the Node ID part from the Port ID p.

Parameters

p	The Port ID from which to extract the Node ID part.

Returns

Node ID part of the Port ID p.

6.35.2.5 #define L1_thisNodeID ((L1_NodeID)((L1_KernelTaskID) & L1_GLOBALID_MASK))

The Node-ID of this Node.

6.35.3 Typedef Documentation

6.35.3.1 typedef struct _struct_L1_Port_ L1_InputPort

Remarks

```
SPC Input Port
```

This structure represents a Task Input Port (TIP), which is used to pass L1_Packets to the Kernel Task or Link Driver Tasks for processing. In case of Asynchronous Interactions the Task Input Port contains the Asynchronous Packets that have been returned to the Task.

6.35.4 Enumeration Type Documentation

6.35.4.1 enum L1_TaskStatus

L1_TaskStatus is an enumeration type used to specify the state of a Task.

Remarks

```
SPC Task states
```

Enumerator

L1_INACTIVE State of a task which is not started when system boots, or which has been stopped.

L1_STARTED State of a task which has been started when system boots, or which has been started by start task service.

L1_ABORTED State of a task that caused an exception to be triggered. Only used while the Abort--Handler is or will be active.

6.35.5 Function Documentation

6.35.5.1 void inputPortService (L1_InputPort * *Port,* L1_Packet * *Packet*)

Performs the synchronisation of the L1_Packets in a Task-Input-Port.

Parameters

Port	Pointer to the Input-Port to synchronise.
Packet	Pointer to the L1_Packet to process.

6.35.5.2 void L1_abortTaskService (L1_Packet * *packet*)

Given a Request-Packet of a Task this function will abort the Task that owns the Request-Packet.

Parameters

packet	Pointer to the Request-Packet of the Task that should be aborted. The field errorCode shall be set to the reason for the abort.

See Also

L1_Drv_Isr_abortTask

6.35.5.3 void L1_anyPacketService (L1_Packet * *Packet*)

This service gets invoked by the Kernel if a Task is waiting for an L1_Packet to be placed in it's Input--Port. This function only returns once there is an L1_Packet in the input Port of the Task identified by Packet->DestinationPortID, or the Timeout expired.

Parameters

Packet	Pointer to the L1_Packet to be processed.

Postcondition

Packet->Data[0-4] contain the pointer to the L1_Packet that was inserted into the Input-Port of the Task.

6.35.5.4 L1_ReturnCode L1_buildAndInsertPacket (L1_PortID *PortID,* L1_Packet * *Packet,* L1_UINT16 *ServiceID,* L1_Timeout *Timeout*)

Configures the L1_Packet identified by the parameter Packet, with values of the PortID, ServiceID, and Timeout. The packet is then inserted into the Kernel-Input-Port and a context switch gets performed to the Kernel-Task to process the Packet.

Parameters

PortID	The ID of the Port the Packet should be sent to.
Packet	Pointer to the Packet to send.
ServiceID	The Service ID of the Packet, must be a value defined in L1_ServiceID.
Timeout	The amount of ticks the Packet should wait for the Interaction to take place.

Returns

The function returns the return value of the interaction:

- RC_OK The interaction was successful.
- RC_FAIL The interaction failed.
- RC_TO The timeout expired.

6.35.5.5 void L1_changeTaskPriority (L1_TaskControlRecord * *TaskCR,* L1_Priority *newPriority*)

This function changes the priority of the Task identified by TaskCR to the value of newPriority. If the Task is currently on the Ready-List it will be reinserted into it to ensure that the new priority is reflected.

Parameters

TaskCR	Pointer to the Task Control Record of the Task to adjust the priority of.
newPriority	The new priority the Task should have.

6.35.5.6 void L1_idleTask (L1_TaskArguments *Arguments*)

This is a forward declaration of the idle task given in the user's project. It is the lowest priority task in the system. Usually implemented as an infinite loop that ignores the arguments received where performance metrics may be performed.

Parameters

Arguments	function implementation dependent.

6.35.5.7 void L1_initLinkDriver (void)

This function shall be defined by every Node of a Multi Node (MP) System. Its purpose is to establish the links between this Node and its neighbor Nodes.

6.35.5.8 void L1_KernelEntryPoint (L1_TaskArguments *Arguments*)

This function initializes the system timer, initializes the link drivers in MP mode, and starts the Kernel loop function.

Parameters

Arguments	The arguments passed to this function are ignored.

6.35.5.9 void L1_KernelLoop (void)

This function represents the Kernel-Task. It performs packet-switching on incoming L1_Packets routing them either to a local Hub, a local Task, or another Node for processing. If there are no L1_Packets to process, it deschedules.

6.35.5.10 L1_Packet* L1_KernelPacketPool_getPacket (void)

Allocates a Packet from the KernelPacketPool and returns a pointer it.

Returns

NULL indicates that no packet could be allocated
!NULL is the pointer to the L1_Packet that was allocated from the KernelPacketPool.

6.35.5.11 static void L1_List_insertTask (L1_List * *list*, L1_TaskControlRecord * *taskCR*) [inline], [static]

Inserts a the L1_TaskControlRecord identified by taskCR into the L1_List identified by list

Parameters

list	Pointer to the L1_List where task shall be inserted.
taskCR	Pointer to the L1_TaskControlRecord that shall be inserted into the L1_List list.

Precondition

list NOT NULL
taskCR NOT NULL
taskCR is not element of any L1_List.

Postcondition

taskCR is element of the L1_List list.

6.35.5.12 static void L1_List_removeTask (L1_TaskControlRecord * *taskCR*) [inline], [static]

Removes the L1_TasckControlRecord identified by taskCR from the list it is element of, if any.

Parameters

taskCR	Pointer to the L1_TaskControlRecord that shall be removed from the list it is element of.

Precondition

taskCR NOT NULL

Postcondition

taskCR is not any longer element of any L1_List.

6.35.5.13 void L1_makeTaskReady (L1_TaskControlRecord * *TaskCR*)

Makes the Task identified by the parameter TaskCR ready. This means that it adds the TaskCR to the Ready-List.

Parameters

TaskCR	Pointer to the Task Control Record of the Task which should be made ready.

Precondition

- TaskCR must belong to a local Task.
- TaskCR must not be on the Ready-List

Postcondition

- TaskCR is on the Ready-List.

6.35.5.14 void L1_remoteService (L1_Packet * *Packet*)

Routes the L1_Packet, identified by the parameter Packet, to the corresponding Driver-Task.

Parameters

Packet	Pointer to the L1_Packet to be routed to the corresponding Driver-Task.

Warning

If the Driver-Task is inactive the Packet gets discarded.

6.35.5.15 void L1_resetTimer (L1_Packet * *waitingPacket*)

Cancels the timeout for the L1_Packet, identified by the parameter waitingPacket.

Parameters

waitingPacket	Pointer to the L1_Packet for which to cancel the timeout.

6.35.5.16 void L1_resumeTaskService (L1_Packet * *Packet*)

This function resumes the task that has the destination ID of the Packet. A task that is in the suspended state will be resumed.

Parameters

Packet	The task that has the id of the destination ID field of this packet will be resumed.

6.35.5.17 void L1_returnPacketService (L1_Packet * *Packet*)

This service gets triggered when a Remote Node returns an L1_Packet to the local node. This service then takes care to return the Packet to the correct Task and make it ready if necessary.

Parameters

Packet	Pointer to the L1_Packet to return to its Task.

6.35.5.18 void L1_returnToTask (L1_Packet * *Packet*)

This function returns an L1_Packet to the Task that owns it. If the Task is currently not ready to be run, it will be added to the Ready-List. If the Packet is from a remote Node this function will route it to the correct Link-Driver.

Parameters

Packet	Pointer to the L1_Packet to return to its Task.

6.35.5.19 int L1_runRTOS (void)

This function starts the RTOS. The function initializes platform specific hardware, initializes the key list for priority inheritance, initializes the hubs in the node by setting the HubControlFunction, and initializes the ready list. After that it starts the tasks in the system. It inserts all the tasks whose state field is L1_STARTED into the ready list and starts the kernel task.

Returns

int -1 when there is a problem. 0 when it all went fine.

6.35.5.20 void L1_runTask (L1_TaskControlRecord * *taskCR*)

Given a task control record, the function initializes the sequence number of the request packet and starts the entry point function with the arguments pointed by the control block structure.

Parameters

taskCR	is a pointer to the task control record of the Task to be run.

Remarks

```
SPC Task arguments
```

6.35.5.21 static int L1_runVirtuosoNext (L1_UINT32 *nbrOfTasks,* L1_UINT32 *nbrOfHubs*) [inline], [static]

This function starts the RTOS. It takes the number of tasks and hubs as parameter.

Parameters

nbrOfTasks	How many Tasks are on the Node.
nbrOfHubs	How many Hubs are on the Node.

Returns

int -1 when there is a problem. 0 when it all went fine.

6.35.5.22 L1_ReturnCode L1_setTimer (L1_Packet * *Packet*)

This function inserts the L1_Packet, identified by the parameter Packet, into the Timer-List. j

Parameters

Packet	Pointer to the L1_Packet which should be inserted into the Timer-List.

Returns

L1_ReturnCode

- RC_OK If the Packet could be inserted into the Timer-List.
- RC_FAIL If the Packet could not be inserted into the Timer-List.

6.35.5.23 void L1_startTaskService (L1_Packet * *Packet*)

This function starts the task that has the destination ID of the Packet. If the task is inactive, it initializes the task context and sets its state to L1_STARTED. The task is made ready and the kernel is instructed to return to task.

Parameters

Packet	The task that has the id of the destination ID field of this packet will be started.

6.35.5.24 void L1_stopTaskService (L1_Packet * *Packet*)

This function stops the task that has the destination ID of the Packet. If the task is ready, it removes the task from the ready list. Otherwise it removes the task's packet from any waiting list and the suspended field is set to false to be able to start the task again. A task that is stopped is set to the L1_INACTIVE state.

Parameters

Packet	The task that has the id of the destination ID field of this packet will be stopped

6.35.5.25 void L1_suspendTaskService (L1_Packet * *Packet*)

This function suspends the task that has the destination ID of the Packet.

Parameters

Packet	The task that has the id of the destination ID field of this packet will be suspended.

6.35.5.26 void L1_timerPacketService (L1_Packet * *Packet*)

This service handles any incoming timer related tasks. It either suspends or wakes up a Task, depending on the requested Service in field Packet->ServiceID.

Parameters

Packet	Pointer to the L1_Packet which contains the request.

6.35.5.27 void L1_timerPacketService_tick (L1_Packet * *packet*)

This function is meant to be called by the periodic ticker timers, whenever a tick occurred.

The function will inject an L1_Packet into the Kernel Input Port to inform the Kernel-Task about the tick.

Parameters

packet	Pointer to the L1_Packet to inject into the Kernel.

Warning

The ISR shall afterwards schedule the Kernel-Task.

Returns

6.35.6 Variable Documentation

6.35.6.1 L1_TimerList L1_NodeTimerTimeoutList

This list contains all timeouts registered at this node.

Chapter 7

Data Structure Documentation

7.1 _struct_L1_DataQueueElement_ Struct Reference

`#include <L1_hub_data_queue.h>`

Data Fields

- L1_BYTE * data
- L1_UINT16 dataSize

7.1.1 Detailed Description

This structure represents an Element in the FIFO-Hub. It buffers the data given to it.

7.1.2 Field Documentation

7.1.2.1 L1_BYTE* data

The data stored in this

7.1.2.2 L1_UINT16 dataSize

Number of bytes that are used in the field Data

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_data_queue.h

7.2 _struct_L1_DataQueueState_ Struct Reference

`#include <L1_hub_data_queue.h>`

Data Fields

- L1_UINT16 count
- L1_DataQueueElement * elements
- const L1_UINT16 elementSize
- L1_UINT16 head
- const L1_UINT16 nbrOfElements
- L1_UINT16 tail

7.2.1 Detailed Description

State of a DataQueue-Hub.

7.2.2 Field Documentation

7.2.2.1 L1_UINT16 count

Current number of used elements of the array Buffer.

7.2.2.2 L1_DataQueueElement* elements

Array of L1_DataQueueElements which are used to store the data passed to the DataQueue-Hub.

7.2.2.3 const L1_UINT16 elementSize

The number of bytes each L1_DataQueueElement can store.

7.2.2.4 L1_UINT16 head

Index of the Head element in the array Buffer.

7.2.2.5 const L1_UINT16 nbrOfElements

Number of L1_DataQueueEntry elements the field elements points to.

7.2.2.6 L1_UINT16 tail

Index of the Tail element in the array Buffer.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_data_queue.h

7.3 _struct_L1_EventState_ Struct Reference

`#include <L1_hub_event.h>`

Data Fields

- L1_BOOL isSet

7.3.1 Detailed Description

The state of an Event-Hub.

Remarks

```
SPC Event state variable
```

7.3.2 Field Documentation

7.3.2.1 L1_BOOL isSet

Indicates whether or not the Event-Hub is signaled.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_event.h

7.4 _struct_L1_FifoState_ Struct Reference

```
#include <L1_hub_fifo.h>
```

Data Fields

- L1_PacketData **const Buffer
- L1_UINT16 Count
- L1_PacketData *const DataParts
- L1_UINT16 Head
- const L1_UINT16 Size
- L1_UINT16 Tail

7.4.1 Detailed Description

State of a FIFO-Hub.

Remarks

```
SPC FIFO state variables
```

7.4.2 Field Documentation

7.4.2.1 L1_PacketData** const Buffer

Array of Pointers to L1_PacketData elements used to exchange data between Tasks and the FIFO.

7.4.2.2 L1_UINT16 Count

Current number of used elements of the array Buffer.

7.4.2.3 L1_PacketData* const DataParts

Array of L1_PacketData elements which are used to store the data passed to the FIFO-Hub.

7.4.2.4 L1_UINT16 Head

Index of the Head element in the array Buffer.

7.4.2.5 const L1_UINT16 Size

Number of L1_FifoData elements the array Buffer contains.

Remarks

```
SPC Configurable FIFO-Size
```

7.4.2.6 L1_UINT16 Tail

Index of the Tail element in the array Buffer.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_fifo.h

7.5 _struct_L1_Hub_ Struct Reference

```
#include <L1_hub.h>
```

Data Fields

- const L1_HubControlFunction HubControlFunction
- const void * HubState
- const L1_HubSyncConditionFunction HubSyncConditionFunction
- const L1_HubSynchronizeFunction HubSynchronizeFunction
- const L1_ServiceType HubType
- const L1_HubStateUpdateFunction HubUpdateFunction
- L1_List WaitingList

7.5.1 Detailed Description

Generic Hub State Structure.

7.5.2 Field Documentation

7.5.2.1 const L1_HubControlFunction HubControlFunction

Pointer to the Hub-Control function for this Hub. If not used this may be NULL.

7.5.2.2 const void* HubState

Pointer to the Hub specific data. If not used this may be NULL.

7.5.2.3 const L1_HubSyncConditionFunction HubSyncConditionFunction

Pointer to the Hub-SyncCondition function for this Hub. If not used this may be NULL.

7.5.2.4 const L1_HubSynchronizeFunction HubSynchronizeFunction

Pointer to the Hub-Synchronize function for this Hub, this function gets called, the Generic Hub, when it has a pair of Get- and Put-Packets, i.e. achieved synchronisation. If not used this may be NULL.

7.5.2.5 const L1_ServiceType HubType

The type of hub this structure belongs to. This is relevant to ensure that the field HubState can be casted to the correct type.

7.5.2.6 const L1_HubStateUpdateFunction HubUpdateFunction

Pointer to the Hub-Update function for this Hub. If not used this may be NULL.

7.5.2.7 L1_List WaitingList

On this list any L1_Packet that currently does not result in a synchronisation is kept.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub.h

7.6 _struct_L1_MemoryBlock_ Struct Reference

`#include <L1_memory_api.h>`

Data Fields

- L1_BYTE * Data
- L1_MemoryBlockHeader Header

7.6.1 Detailed Description

This structure represents a memory block that can be allocated form a MemoryPool Hub. It consists of a header and a data part.

Remarks

SPC Memory Block Size and Data fields

7.6.2 Field Documentation

7.6.2.1 L1_BYTE* Data

Payload, the user pointer, i.e. user memory space the size is determined by the memory pool parameters and size request at allocation time.

7.6.2.2 L1_MemoryBlockHeader Header

The header of the MemoryBlock

The documentation for this struct was generated from the following file:

- include/kernel/L1_memory_api.h

7.7 _struct_L1_MemoryBlockHeader_ Struct Reference

`#include <L1_memory_api.h>`

Data Fields

- L1_UINT32 BlockSize
- L1_ListElement ListElement
- L1_TaskID ownerTaskID
- L1_UINT32 UsedBytes

7.7.1 Detailed Description

Remarks

SPC Memory Management This structure represents the header part of a memory block.

7.7.2 Field Documentation

7.7.2.1 L1_UINT32 BlockSize

The requested size of this block, can be different from the MemPool Blocksize.

Remarks

SPC Memory Block Size

7.7.2.2 L1_ListElement ListElement

linkage in free list, may overlap with other fields.

Remarks

```
SPC Memory Block insertion into a List
```

7.7.2.3 L1_TaskID ownerTaskID

ID of the Task that allocated this Memory Block. Only used for Memory-Pool-Hubs.

7.7.2.4 L1_UINT32 UsedBytes

The number of bytes currently being in use. This is important for the MemoryBlockQueue users.

Remarks

```
SPC Memory Block used bytes
```

The documentation for this struct was generated from the following file:

- include/kernel/L1_memory_api.h

7.8 _struct_L1_Packet_ Struct Reference

```
#include <L1_packet_api.h>
```

Data Fields

- L1_PacketData * dataPart
- L1_PortID DestinationPortID
- L1_ErrorCode errorCode
- L1_BOOL inUse
- L1_ListElement ListElement
- L1_List * OwnerPool
- L1_PendingRequestHandler PendingRequestHandler
- L1_ListElement PendingRequestListElement
- L1_TaskID RequestingTaskID
- L1_UINT32 SequenceNumber
- L1_UINT16 ServiceID
- L1_ReturnCode Status
- L1_Timeout Timeout
- L1_TimerTimeout TimeoutTimer

7.8.1 Detailed Description

Entities interact using L1_Packets in VirtuosoNext.

Remarks

```
SPC Identical structure
SPC Header and data
```

7.8.2 Field Documentation

7.8.2.1 L1_PacketData* dataPart

Pointer to the data-part of an L1_Packet. This field shall never be NULL.

7.8.2.2 L1_PortID DestinationPortID

Specifies the Hub to/from which the Packet has to be put/get. Notes:

- in case of a task management service, DestinationHubID is a TaskID
- in case of an L1 service, DestinationHubID is a HubID

7.8.2.3 L1_ErrorCode errorCode

Indicates the last error the Packet / Task experienced.

Remarks

```
WPT Advanced Error Reporting
```

7.8.2.4 L1_BOOL inUse

This flag indicates whether or not this L1_Packet is currently being used. This is used by Interrupt Service Routines to check whether or not their Packet has been already returned to them.

Warning

This is only meant to be used in ISRs where the Packet gets returned implicitly! Do not modify this flag!

7.8.2.5 L1_ListElement ListElement

Remarks

```
SPC Packet priority
```

7.8.2.6 L1_List* OwnerPool

is a reference to the PacketPool from which this Packet was allocated, or NULL if the Packet was not allocated from a PacketPool.

7.8.2.7 L1_PendingRequestHandler PendingRequestHandler

Reference to the function that will be called by the Kernel-Task to handle the pending request.

7.8.2.8 L1_ListElement PendingRequestListElement

This is the ListElement is used to add an L1_Packet to the Pending Requests Queue (PRQ).

7.8.2.9 L1_TaskID RequestingTaskID

Specifies the ID of the Task that owns the Packet.

7.8.2.10 L1_UINT32 SequenceNumber

Internally used by the tracing subsystem to identify packets.

7.8.2.11 L1_UINT16 ServiceID

Specifies the Service that is requested by a Packet from the Kernel.

7.8.2.12 L1_ReturnCode Status

Indicates the status of completion of the service, set by the Kernel when it finishes serving the request.

Remarks

```
SPC Return Packets
```

7.8.2.13 L1_Timeout Timeout

Specifies the timeout (if any) associated with the requested service.

7.8.2.14 L1_TimerTimeout TimeoutTimer

This field is used to enable the L1_Packet to be inserted into an L1_TimerList.

The documentation for this struct was generated from the following file:

- include/kernel/L1_packet_api.h

7.9 _struct_L1_PacketPoolState_ Struct Reference

```
#include <L1_hub_packet_pool.h>
```

Data Fields

- L1_PacketData ∗const PacketDataPool
- L1_List PacketList
- L1_Packet ∗const PacketPool
- L1_UINT16 Size

7.9.1 Detailed Description

The state of a Packet-Pool-Hub.

Remarks

```
SPC Packet Pool state variables
```

7.9.2 Field Documentation

7.9.2.1 L1_PacketData∗ const PacketDataPool

Pointer to an array of L1_PacketData elements which has at least Size number of elements.

7.9.2.2 L1_List PacketList

List which will contain the currently free packets of the pool.

7.9.2.3 L1_Packet∗ const PacketPool

Pointer to an array of L1_Packets which has at least Size number of elements.

7.9.2.4 L1_UINT16 Size

The total number of packets in the pool, only used for initialization.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_packet_pool.h

7.10 _struct_L1_Port_ Struct Reference

```
#include <L1_port_api.h>
```

Data Fields

- L1_List WaitingList

7.10.1 Detailed Description

Remarks

```
SPC Input Port
```

This structure represents a Task Input Port (TIP), which is used to pass L1_Packets to the Kernel Task or Link Driver Tasks for processing. In case of Asynchronous Interactions the Task Input Port contains the Asynchronous Packets that have been returned to the Task.

7.10.2 Field Documentation

7.10.2.1 L1_List WaitingList

The Priority ordered List of L1_Packets in the Input Port.

The documentation for this struct was generated from the following file:

- include/kernel/L1_port_api.h

7.11 _struct_L1_ResourceState_ Struct Reference

```
#include <L1_hub_resource.h>
```

Data Fields

- L1_Priority CeilingPriority
- L1_BOOL Locked
- L1_Priority OwnerBoostedToPriority
- L1_TaskID OwningTaskID

7.11.1 Detailed Description

State of a Resource-Hub.

Remarks

```
SPC Resource state variables
```

7.11.2 Field Documentation

7.11.2.1 L1_Priority CeilingPriority

Defines the maximum Priority to which the Task owning this Resource may be boosted.

Remarks

```
SPC Configurable Ceiling Priority
```

7.11.2.2 L1_BOOL Locked

Identifies whether or not the Resource is currently locked or not.

7.11.2.3 L1_Priority OwnerBoostedToPriority

The Priority to which the Task owning this Resource has been boosted to already. It is used to avoid repeated boosting of the Priority in cases where a Task has already been boosted to a higher Priority. The maximum priority this field may contain is the value given in the field CeilingPriority.

7.11.2.4 L1_TaskID OwningTaskID

If the Resource is locked, i.e. Locked==L1_TRUE, then this field contains the ID of the Task that owns this Resource.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_resource.h

7.12 _struct_L1_SemaphoreState_ Struct Reference

`#include <L1_hub_semaphore.h>`

Data Fields

- L1_UINT16 Count

7.12.1 Detailed Description

State of a Semaphore-Hub.

Remarks

```
SPC Semaphore state variable
```

7.12.2 Field Documentation

7.12.2.1 L1_UINT16 Count

How often the Semaphore-Hub has been signaled.

Remarks

```
SPC Semaphore counter lower bound
```

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_semaphore.h

7.13 _struct_tracebuffer_ Struct Reference

`#include <L1_types.h>`

Data Fields

- L1_UINT32 param0
- L1_UINT32 param1
- L1_UINT32 param2
- L1_UINT32 param3

7.13.1 Detailed Description

This structure represents one event recorded in the Trace-Buffer.

7.13.2 Field Documentation

7.13.2.1 L1_UINT32 param0

This is the first parameter in the Trace-Buffer, it should be used to encode the 8 bit type in the lowest 8bit of the 32Bit word, the highest 24Bit are used to store the elapsed system ticks (usually ms).

7.13.2.2 L1_UINT32 param1

The second parameter of an event contains the value of the high counter, usually the clock value.

7.13.2.3 L1_UINT32 param2

The content of this parameter depends upon the type of event that was traced.

7.13.2.4 L1_UINT32 param3

The content of this parameter depends upon the type of event that was traced.

The documentation for this struct was generated from the following file:

- include/L1_types.h

7.14 L1_BlackBoard_Board Struct Reference

`#include <L1_hub_black_board.h>`

Data Fields

- L1_BYTE message [L1_PACKET_DATA_SIZE-sizeof(L1_UINT32)]
- L1_UINT32 messageNumber

7.14.1 Detailed Description

Represents the content of a Blackboard.

Remarks

SPC BlackBoard State Variables

7.14.2 Field Documentation

7.14.2.1 L1_BYTE message[L1_PACKET_DATA_SIZE-sizeof(L1_UINT32)]

This is the message that got published.

7.14.2.2 L1_UINT32 messageNumber

This is a copy of the variable messageNumber in the Hub, but in network endianness.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_black_board.h

7.15 L1_BlackBoard_HubState Struct Reference

`#include <L1_hub_black_board.h>`

Data Fields

- L1_BlackBoard_Board board
- L1_UINT32 dataSize
- L1_UINT32 messageNumber

7.15.1 Detailed Description

The state of a BlackBoard Hub.

Remarks

SPC BlackBoard State Variables

7.15.2 Field Documentation

7.15.2.1 L1_BlackBoard_Board board

This is the board where the Message gets published. This gets copied verbosely to the reader.

7.15.2.2 L1_UINT32 dataSize

The size of the message on the board. Must be less or equal L1_PACKET_DATA_SIZE - sizeof(L1_UINT32).

7.15.2.3 L1_UINT32 messageNumber

This counts the number of messages that have been written to the board. This is in local CPU endianness.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_black_board.h

7.16 L1_DataEvent_HubState Struct Reference

`#include <L1_hub_data_event.h>`

Data Fields

- L1_PacketData * dataPart
- L1_BOOL isSet

7.16.1 Detailed Description

State of a DataEvent-Hub.

Remarks

SPC Data Event state variables

7.16.2 Field Documentation

7.16.2.1 L1_PacketData* dataPart

Data Part of the Data-Event.

7.16.2.2 L1_BOOL isSet

Indicates whether or not the DataEvent-Hub has been signalled.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_data_event.h

7.17 L1_HubNameToID Struct Reference

`#include <L1_types.h>`

Data Fields

- L1_UINT32 id
- char * name
- L1_UINT32 type

7.17.1 Detailed Description

This structures are used to comply with OCG-6, these are currently test implementations.

7.17.2 Field Documentation

7.17.2.1 L1_UINT32 id

The global ID of the Task;

7.17.2.2 char* name

The name given to the Task;

7.17.2.3 L1_UINT32 type

Type ID of the Hub.

The documentation for this struct was generated from the following file:

- include/L1_types.h

7.18 L1_MemoryBlockQueue_HubState Struct Reference

```
#include <L1_hub_memory_block_queue.h>
```

Data Fields

- L1_MemoryBlock * blocks
- const L1_UINT32 blockSize
- L1_List freeBlocks
- L1_BYTE * memory
- L1_UINT32 nbrOfAcquiredBlocks
- const L1_UINT32 nbrOfBlocks
- L1_UINT32 nbrOfUsedBlocks
- L1_List usedBlocks

7.18.1 Detailed Description

Remarks

```
SPC Memory Block Queue fields
```

7.18.2 Field Documentation

7.18.2.1 L1_MemoryBlock* blocks

Array of L1_MemoryBlock elements that will be inserted into the empty buffers list (emptyBuffers There must be at least nbrOfBlocks elements in the array).

7.18.2.2 const L1_UINT32 blockSize

The number of bytes each Block has. This is important during initialisation as well as during operation.

Remarks

```
SPC Memory Block Queue Block size
```

7.18.2.3 L1_List freeBlocks

This is a double linked list of L1_MemoryBlock elements which are currently not in use.

7.18.2.4 L1_BYTE* memory

Pointer to a buffer of size (blockSize * nbrOfBlocks). This memory is assigned to the memory-blocks.

7.18.2.5 L1_UINT32 nbrOfAcquiredBlocks

How many L1_MemoryBlock elements have been given to other Tasks.

7.18.2.6 const L1_UINT32 nbrOfBlocks

How many L1_MemoryBlock elements are assigned to this MemoryBlockQueue-Hub.

Remarks

```
SPC Memory Block Queue size
```

7.18.2.7 L1_UINT32 nbrOfUsedBlocks

Numer of Memory Blocks in the list usedBlocks

7.18.2.8 L1_List usedBlocks

This is a double linked list of L1_MemoryBlock elements which are currently full. Important all these elements must have the same priority, otherwise they will overtake each other.

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_memory_block_queue.h

7.19 L1_MemoryPool_HubState Struct Reference

```
#include <L1_hub_memory_pool.h>
```

Data Fields

- const L1_UINT32 BlockSize
- L1_List FreeMemoryBlockList
- const L1_MemoryBlock * MemoryBlockPool
- const L1_UINT16 NumberOfBlocks
- L1_List OccupiedMemoryBlockList

7.19.1 Detailed Description

The state information of the Memory-Pool Hub.

```
// These are the blocks of memory that get allocated for the user.
L1_BYTE MemoryPool_MP1_Memory[1][1024];

// Data structures that combine the memory block with the management information.
L1_MemoryBlock MemoryPool_MP1_MemoryBlocks[1] =
{
    {
        .Header =
        {
            .ListElement = {NULL,NULL,1},
            // This parameter comes from the system.xml
            .DataSize    = 1024,
        },
        .Data = &MemoryPool_MP1_Memory[0]
    },
};

// The HubState information for the Memroy.
L1_MemoryPool_HubState MemoryPool_MP1_HubState =
{
    // This parameter comes from the system.xml
    .NumberOfBlocks = 1,
    // This parameter comes from the system.xml
    .BlockSize = 1024,
    .MemoryBlockPool = MemoryPool_MP1_MemoryBlocks,
    .FreeMemoryBlockList = { .SentinelElement = {NULL, NULL, 1} },
    .OccupiedMemoryBlockList = { .SentinelElement = {NULL, NULL, 1} }
};
```

Remarks

SPC Memory Pool state variables

7.19.2 Field Documentation

7.19.2.1 const L1_UINT32 BlockSize

Size of the individual blocks in byte. Set by the codegens

7.19.2.2 L1_List FreeMemoryBlockList

List that contains the currently available Free MemoryBlocks.

7.19.2.3 const L1_MemoryBlock* MemoryBlockPool

Pointer to an array of elements of type L1_MemoryBlock.

7.19.2.4 const L1_UINT16 NumberOfBlocks

Number of Memory-Blocks in the pool, only used for initialisation. Set by the codegens

7.19.2.5 L1_List OccupiedMemoryBlockList

List that contains the currently occupied MemoryBlocks

The documentation for this struct was generated from the following file:

- include/kernel/hubs/L1_hub_memory_pool.h

7.20 L1_NodeStatusStructure Struct Reference

`#include <L1_kernel_data.h>`

Data Fields

- L1_KernelTicks currentTime
- L1_UINT32 kernelTickFrequencyHz
- L1_UINT32 maxNumberOfPacketsInRxPacketPool
- L1_UINT32 nodePacketCount
- L1_UINT32 numberOfDiscardedRxPackets
- L1_UINT32 numberOfHubs
- L1_UINT32 numberOfIllegalServiceRequests
- L1_UINT32 numberOfTasks
- L1_UINT32 numberOfTimesSemaphoreMaxCountReached

7.20.1 Detailed Description

Information related to the status of the Node.

7.20.2 Field Documentation

7.20.2.1 L1_KernelTicks currentTime

The number of ticks that expired since the Kernel-Task started.

7.20.2.2 L1_UINT32 kernelTickFrequencyHz

The frequency of the Kernel-Tick in Hz.

7.20.2.3 L1_UINT32 maxNumberOfPacketsInRxPacketPool

The maximum number of Packet in the RX-Packet-Pool.

7.20.2.4 L1_UINT32 nodePacketCount

The number of Packets this Node has sent.

7.20.2.5 L1_UINT32 numberOfDiscardedRxPackets

The number of RX-Packets that got discarded because there were not enough Packets in the RX-Packet--Pool.

7.20.2.6 L1_UINT32 numberOfHubs

The number of Hubs this Node manages.

7.20.2.7 L1_UINT32 numberOfIllegalServiceRequests

The number of illegal service requests encountered by the Kernel-Task.

7.20.2.8 L1_UINT32 numberOfTasks

The number of Tasks this Node manages.

7.20.2.9 L1_UINT32 numberOfTimesSemaphoreMaxCountReached

How often a Semaphore-Hub has reached its max-count.

The documentation for this struct was generated from the following file:

- include/kernel/L1_kernel_data.h

7.21 L1_PacketData Struct Reference

```
#include <L1_packet_api.h>
```

Public Member Functions

- L1_BYTE data[L1_PACKET_DATA_SIZE] __attribute__ ((aligned(L1_DATA_ALIGNMENT)))

Data Fields

- L1_UINT32 dataSize
- L1_ListElement ListElement

7.21.1 Detailed Description

Represents the data-part of an L1_Packet.

7.21.2 Member Function Documentation

7.21.2.1 L1_BYTE data [L1_PACKET_DATA_SIZE] __attribute__ ((aligned(L1_DATA_ALIGNMENT)))

The buffer for the data to be stored.

7.21.3 Field Documentation

7.21.3.1 L1_UINT32 dataSize

Remarks

`SPC Data size field` The number of bytes used in the field data.

7.21.3.2 L1_ListElement ListElement

This list element will allow chaining of L1_PacketData elements.

The documentation for this struct was generated from the following file:

- include/kernel/L1_packet_api.h

7.22 L1_TaskControlRecord Struct Reference

`#include <L1_task_api.h>`

Data Fields

- const L1_TaskAbortFunction AbortHandler
- L1_TaskArguments Arguments
- L1_TaskContext * Context
- L1_KeyedList CriticalSectionWaitingList
- const L1_TaskFunction EntryPoint
- L1_Priority IntrinsicPriority
- L1_BOOL isSuspended
- L1_ListElement ListElement
- L1_Packet *const RequestPacket
- const L1_TaskID TaskID
- L1_InputPort *const TaskInputPort
- L1_TaskStatus TaskState

7.22.1 Detailed Description

L1_TaskControlRecord is a structure that represents a Task

7.22.2 Field Documentation

7.22.2.1 const L1_TaskAbortFunction AbortHandler

Pointer to the function that represents the Abort-Handler, i.e. the function that will be called when the Task caused a fatal CPU-Exception.

This function shall cleanup the Task-State to ensure that the Task can be restarted. This means releasing off Resources the Task may have acquired during it's runtime.

7.22.2.2 L1_TaskArguments Arguments

Arguments to the Task-Entry-Point function identified by the field EntryPoint.

Remarks

```
SPC Task arguments
```

7.22.2.3 L1_TaskContext* Context

Pointer to the Platform dependent Context of the Task.

7.22.2.4 L1_KeyedList CriticalSectionWaitingList

Remarks

```
SPC Resource Priority Inheritance
SPC Returning the Priority back to the original value
```

7.22.2.5 const L1_TaskFunction EntryPoint

Pointer to the function that represents the Task-Entry-Point, i.e. the function that will be called when the Task gets scheduled.

7.22.2.6 L1_Priority IntrinsicPriority

Remarks

```
SPC Task priority
SPC Returning the Priority back to the original value
```

7.22.2.7 L1_BOOL isSuspended

Remarks

`SPC Task states` Indicates whether or not the Task is currently suspended.

7.22.2.8 L1_ListElement ListElement

Remarks

```
SPC Priority based scheduling
SPC Task priority
```

7.22.2.9 L1_Packet* const RequestPacket

Pointer to the L1_Packet which represents the Request-Packet of the Task.

Remarks

```
SPC Request Packet
SPC Task priority
```

7.22.2.10 const L1_TaskID TaskID

The ID assigned to the Task.

7.22.2.11 L1_InputPort* const TaskInputPort

Identifies the Input Port of the Task. (Driver Input Port or Task Input Port.)

Remarks

```
SPC Dedicated input Port
SPC Input Port
```

7.22.2.12 L1_TaskStatus TaskState

State of the Task, identifies whether the Task is started or inactive.

The documentation for this struct was generated from the following file:

- include/kernel/L1_task_api.h

7.23 L1_TaskNameToID Struct Reference

```
#include <L1_types.h>
```

Data Fields

- L1_UINT32 id
- char * name

7.23.1 Detailed Description

This structures are used to comply with OCG-6, these are currently test implementations.

7.23.2 Field Documentation

7.23.2.1 L1_UINT32 id

The global ID of the Task;

7.23.2.2 char* name

The name given to the Task;

The documentation for this struct was generated from the following file:

- include/L1_types.h

7.24 L1_WLM_State Struct Reference

```
#include <L1_workload_monitoring.h>
```

Data Fields

- volatile L1_UINT32 currentLoopCount
- volatile L1_UINT32 previousLoopCount
- L1_TimeStamp t0
- L1_TimeStamp t1
- volatile L1_UINT32 terminationLoopCount
- volatile L1_UINT32 workloadPeriodCount
- const L1_UINT32 workloadPeriodLength

7.24.1 Detailed Description

Remarks

`SPC Workload Monitoring` The integration time defines how often a new workload measurement will be available and the amount of time over which the execution of the idle task will be tracked. Workload monitor state.

7.24.2 Field Documentation

7.24.2.1 volatile L1_UINT32 currentLoopCount

This will contain the current workload in units of 0.1%. Current count of workload loops in the idleTask.

7.24.2.2 volatile L1_UINT32 previousLoopCount

Previous workload count.

7.24.2.3 L1_TimeStamp t0

Starting time of the measurement period, as L1_TimeStamp

7.24.2.4 L1_TimeStamp t1

End time of the measurement period, as L1_TimeStamp

7.24.2.5 volatile L1_UINT32 terminationLoopCount

Workload count at which the Idle-Loop shall terminate. This is used during the calibration run.

7.24.2.6 volatile L1_UINT32 workloadPeriodCount

Counter for the number of ticks that still need to occur till the end of the measurement period.

7.24.2.7 const L1_UINT32 workloadPeriodLength

Over how many ticks the workload shall be calculated.

The documentation for this struct was generated from the following file:

- include/kernel/L1_workload_monitoring.h

Part IV

Stdio Host Service

Chapter 8

Module Index

8.1 Modules

Here is a list of all modules:

Chapter 9

Module Documentation

9.1 Stdio Host Server

Modules

- Stdio Host Server Component Description

Functions

- L1_ReturnCode Shs_putChar_W (L1_HubID shs, L1_BYTE charValue)
- L1_ReturnCode Shs_getChar_W (L1_HubID shs, L1_BYTE ∗pChar)
- L1_ReturnCode Shs_putInt_W (L1_HubID shs, L1_INT32 intValue, L1_BYTE format)
- L1_ReturnCode Shs_getInt_W (L1_HubID shs, L1_INT32 ∗pInt)
- L1_ReturnCode Shs_putString_W (L1_HubID shs, const char ∗str)
- L1_ReturnCode Shs_getString_W (L1_HubID shs, L1_UINT32 maxLength, char ∗pStr, L1_UINT32 ∗pRealLength)
- L1_ReturnCode Shs_openFile_W (L1_HubID shs, const char ∗fileName, const char ∗mode, L1_GlobalPointer ∗fileHandle)
- L1_ReturnCode Shs_closeFile_W (L1_HubID shs, L1_GlobalPointer fileHandle)
- L1_ReturnCode Shs_writeToFile_W (L1_HubID shs, L1_GlobalPointer fileHandle, L1_BYTE ∗buffer, L1_UINT32 toWrite, L1_UINT32 ∗pWritten)
- L1_ReturnCode Shs_readFromFile_W (L1_HubID shs, L1_GlobalPointer fileHandle, L1_BYTE ∗buffer, L1_UINT32 toRead, L1_UINT32 ∗pRead)
- L1_ReturnCode DumpTraceBuffer_W (L1_HubID ServerInputPort)

9.1.1 Detailed Description

9.1.2 Function Documentation

9.1.2.1 L1_ReturnCode DumpTraceBuffer_W (L1_HubID *ServerInputPort*)

Temporarily stops the tracing and meanwhile sends the content of the trace buffer to the StdioHostServer specified in the parameter ServerInputPort.

Parameters

ServerInput-Port	address of the Stdio Host Server Input port.

Returns

L1_ReturnCode:

- RC_OK: Dumping the trace buffer was completed successfully.
- RC_FAIL: Operation failed.

9.1.2.2 L1_ReturnCode Shs_closeFile_W (L1_HubID *shs*, L1_GlobalPointer *fileHandle*)

Closes a file previously opened using the function Shs_openFile().

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
fileHandle	is the file-handle previously acquired from the Stdio Host Server using the function Shs_openFile().

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.3 L1_ReturnCode Shs_getChar_W (L1_HubID *shs*, L1_BYTE * *pChar*)

Retrieves one Character from the Stdio Host Server console. The retrieved character is returned to the user in the character value at pChar.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
pChar	is the Pointer to a variable of type L1_BYTE which should hold the retrieved character.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.4 L1_ReturnCode Shs_getInt_W (L1_HubID *shs*, L1_INT32 * *pInt*)

Retrieves an integer from the Stdio Host Server console. The retrieved integer is returned to the user in the character value at pInt.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
pInt	is the pointer to a variable of type int which should hold the retrieved integer value.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.5 L1_ReturnCode Shs_getString_W (L1_HubID *shs*, L1_UINT32 *maxLength*, char * *pStr*, L1_UINT32 * *pRealLength*)

Retrieved a string value from the Stdio Host Server.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
maxLength	is the the number of characters the buffer (pStr) can hold.
pStr	is the pointer to character array which should be filled with the retrieved string.
pRealLength	is the pointer to an integer which will hold number of returned characters, including the terminating zero.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.6 L1_ReturnCode Shs_openFile_W (L1_HubID *shs*, const char * *fileName*, const char * *mode*, L1_GlobalPointer * *fileHandle*)

Opens a file on the Stdio Host Server file system.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
fileName	is the name of a file to open.
mode	is the mode in which the file should be opened. It can contain 1 or 2 symbols.
fileHandle	is the pointer to file handle associated with the opened file. This handle in generated by the StdioHostService.

Returns

L1_ReturnCode

- RC_OK: The request was successful.
- RC_FAIL: The request failed.

9.1.2.7 L1_ReturnCode Shs_putChar_W (L1_HubID *shs*, L1_BYTE *charValue*)

Writes one character value onto the console associated with the Stdio Host Server.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
charValue	is the character to write onto the console.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.8 L1_ReturnCode Shs_putInt_W (L1_HubID *shs*, L1_INT32 *intValue*, L1_BYTE *format*)

This function outputs an integer (intValue) into the console associated with the Stdio Host Server. The output format (octal, decimal, hexa-decimal) must be specified using the character format.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
intValue	is the integer to output into the console.
format	is the character specifying in which format the integer should be written onto the console. The following are permitted: • 'o' – Octal output • 'd' – Decimal output • 'x' – Hexa-decimal output.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.9 L1_ReturnCode Shs_putString_W (L1_HubID *shs*, const char * *str*)

Prints the string str with onto the console, only length characters are written on to the console.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
str	is the C-string to write onto the console.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.10 L1_ReturnCode Shs_readFromFile_W (L1_HubID *shs*, L1_GlobalPointer *fileHandle*, L1_BYTE * *buffer*, L1_UINT32 *toRead*, L1_UINT32 * *pRead*)

Reads from a file opened by the server.

Parameters

shs	- is the ID of the ShsInputPort of the Stdio Host Server.
fileHandle	- is the file-handle previously acquired from the Stdio Host Server using the function Shs_openFile().
buffer	is the pointer to the location where the retrieved data should be stored.
toRead	- how many bytes should be read from the file.
pRead	- how many bytes were actually retrieved from the file.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.1.2.11 L1_ReturnCode Shs_writeToFile_W (L1_HubID *shs*, L1_GlobalPointer *fileHandle*, L1_BYTE * *buffer*, L1_UINT32 *toWrite*, L1_UINT32 * *pWritten*)

This function writes the number of bytes (toWrite) of the byte array at buffer into the file indicated by fileHandle.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
fileHandle	is the file-handle previously acquired from the Stdio Host Server using the function Shs_openFile().
buffer	- the pointer to the first byte of the memory block to be written to the file.
toWrite	- the number of bytes to be written into the file.
pWritten	- pointer to an unsigned integer which contain the number of bytes that were actually written.

Returns

L1_ReturnCode

- RC_OK: The request was successful
- RC_FAIL: The request failed.

9.2 Stdio Host Server Component Description

Figure 9.1: Stdio Host Server Icon.

The StdioHostServer allows Tasks to access the screen and the file system of a Win32 or Posix32 Node. This Component can only be mapped to Windows32 or Posix32 Nodes.

The Stdio Host Server Component has the following attributes:

- node: The name of the Node to which the Stdio Host Server instance is mapped.
- name: Name of the Stdio Host Server instance.
- SHSCeilingPriority: The ceiling priority of the Stdio Host Server instance.

Part V

Graphical Host Service

Chapter 10

Data Structure Index

10.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 11

File Index

11.1 File List

Here is a list of all files with brief descriptions:

Chapter 12

Data Structure Documentation

12.1 GhsBrush Struct Reference

`#include <GhsTypes.h>`

Data Fields

- GhsColour colour
- GhsBrushStyle style

12.1.1 Detailed Description

Defines the type describing a Brush as used by the Graphical Host Service.

12.1.2 Field Documentation

12.1.2.1 GhsColour colour

12.1.2.2 GhsBrushStyle style

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

12.2 GhsColour Struct Reference

`#include <GhsTypes.h>`

Data Fields

- L1_BYTE r
- L1_BYTE g
- L1_BYTE b

12.2.1 Detailed Description

Defines how Colours are represented in the Graphical Host Server Data structures.

12.2.2 Field Documentation

12.2.2.1 L1_BYTE b

Blue component

12.2.2.2 L1_BYTE g

Green component

12.2.2.3 L1_BYTE r

Red component

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

12.3 GhsPen Struct Reference

```
#include <GhsTypes.h>
```

Data Fields

- GhsColour colour
- L1_UINT32 lineWidth
- GhsPenStyle style

12.3.1 Detailed Description

Defines the type describing a Pen as used by the Graphical Host Service.

12.3.2 Field Documentation

12.3.2.1 GhsColour colour

12.3.2.2 L1_UINT32 lineWidth

12.3.2.3 GhsPenStyle style

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

12.4 GhsRect Struct Reference

`#include <GhsTypes.h>`

Data Fields

- L1_UINT32 left
- L1_UINT32 top
- L1_UINT32 right
- L1_UINT32 bottom

12.4.1 Detailed Description

This structure represents a rectangle.

12.4.2 Field Documentation

12.4.2.1 L1_UINT32 bottom

12.4.2.2 L1_UINT32 left

12.4.2.3 L1_UINT32 right

12.4.2.4 L1_UINT32 top

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

Chapter 13

File Documentation

13.1 src/include/GraphicalHostService/GhsTypes.h File Reference

```
#include <L1_api.h>
```

Enumerations

- enum GhsBrushStyle { GhsBrushSolid = 1, GhsBrushDiagonal }
- enum GhsPenStyle { GhsPenSolid = 1 }

13.1.1 Enumeration Type Documentation

13.1.1.1 enum GhsBrushStyle

Defines the different styles a brush can have.

Enumerator

GhsBrushSolid

GhsBrushDiagonal Not Implemented yet.

13.1.1.2 enum GhsPenStyle

Defines the different styles a pen can have.

Enumerator

GhsPenSolid

13.2 src/include/GraphicalHostService/GraphicalHostClient.h File Reference

```
#include <GraphicalHostService/GhsTypes.h>
```

Functions

- L1_ReturnCode Ghs_openSession_W (L1_HubID ghsInputPort, L1_UINT32 ∗pSessionID)
- L1_ReturnCode Ghs_closeSession_W (L1_HubID ghsInputPort, L1_UINT32 sessionId)
- L1_ReturnCode Ghs_getServerVersion_W (L1_HubID ghsInputPort, L1_UINT32 ∗pServerVersion)
- L1_ReturnCode Ghs_setPen_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, GhsPenStyle penStyle, L1_BYTE lineWidth, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_ReturnCode Ghs_setBrush_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, GhsBrushStyle brushStyle, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_ReturnCode Ghs_drawLine_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x1, L1_UINT32 y1, L1_UINT32 x2, L1_UINT32 y2)
- L1_ReturnCode Ghs_drawRect_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x1, L1_UINT32 y1, L1_UINT32 x2, L1_UINT32 y2)
- L1_ReturnCode Ghs_drawCircle_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x, L1_UINT32 y, L1_UINT32 r)
- L1_ReturnCode Ghs_drawText_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT16 x, L1_UINT16 y, char ∗text)
- L1_ReturnCode Ghs_setTextColour_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_ReturnCode Ghs_setCanvasSize_W (L1_HubID ghsInputPort, L1_UINT32 width, L1_UINT32 height)
- L1_ReturnCode Ghs_getCanvasSize_W (L1_HubID ghsInputPort, L1_UINT32 ∗width, L1_UINT32 ∗height)

13.2.1 Function Documentation

13.2.1.1 L1_ReturnCode Ghs_closeSession_W (L1_HubID *ghsInputPort,* L1_UINT32 *sessionId*)

Closes a previously opened session on the Graphical Host Server.

Parameters

ghsInputPort	Input Port of the Graphical Host Server where to close the session.
sessionId	ID of the session to close

Returns

L1_ReturnCode

- RC_OK the session could be closed.
- RC_FAIL the session could not be closed.

13.2.1.2 L1_ReturnCode Ghs_drawCircle_W (L1_HubID *ghsInputPort,* L1_UINT32 *ghsSession,* L1_UINT32 *x,* L1_UINT32 *y,* L1_UINT32 *r*)

Draws a circle defined by the centre point (x,y) and the radius r. The circle will be filled with the brush defined by setBrush() and the surrounding line will be drawn with the pen specified for the ghsSession.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID for the session to draw in.
x	The X part of the centre point.
y	The Y part of the centre point.
r	The radius of the circle.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.3 L1_ReturnCode Ghs_drawLine_W (L1_HubID *ghsInputPort*, L1_UINT32 *ghsSession*, L1_UINT32 *x1*, L1_UINT32 *y1*, L1_UINT32 *x2*, L1_UINT32 *y2*)

Draws a line between the points x1,y1 and x2,y2, using the pen specified for the given ghsSession.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID for the session to draw in
x1	The X part of the first point.
y1	The Y part of the first point.
x2	The X part of the second point.
y2	The Y part of the second point.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.4 L1_ReturnCode Ghs_drawRect_W (L1_HubID *ghsInputPort*, L1_UINT32 *ghsSession*, L1_UINT32 *x1*, L1_UINT32 *y1*, L1_UINT32 *x2*, L1_UINT32 *y2*)

Draws a rectangle defined by the points (x1,y1) and (x2,y2). The rectangle will be filled with the brush defined by setBrush() and the surrounding line will be drawn with the pen specified for the ghsSession.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID for the session to draw in.
x1	The X part of the first point.
y1	The Y part of the first point.
x2	The X part of the second point.
y2	The Y part of the second point.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.5 L1_ReturnCode Ghs_drawText_W (L1_HubID *ghsInputPort,* L1_UINT32 *ghsSession,* L1_UINT16 *x,* L1_UINT16 *y,* char * *text*)

This function draws the string s at the position (x,y) onto the canvas.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID for the session to draw in.
x	The X part of the first point.
y	The Y part of the first point.
text	The string to draw onto the canvas.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.6 L1_ReturnCode Ghs_getCanvasSize_W (L1_HubID *ghsInputPort,* L1_UINT32 * *width,* L1_UINT32 * *height*)

This functions gets the size of the canvas the Graphical Host Server provides.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
width	This parameter returns the horizontal size (x-axis) of the canvas in pixel.
height	This parameter returns the vertial size (y-axis) of the canvas in pixel.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.7 L1_ReturnCode Ghs_getServerVersion_W (L1_HubID *ghsInputPort,* L1_UINT32 * *pServerVersion*)

Queries the Graphical Host Server for its version number.

Parameters

ghsInputPort	Port to which to send the query to.
pServerVersion	After this function returns successfully, the L1_UINT32 which this pointer points to contains the version number of the Graphical Host Server.

Returns

L1_ReturnCode

- RC_OK, *pVersion contains the version number of the server.
- RC_FAIL, operation failed, *pVersion is set to zero.

13.2.1.8 L1_ReturnCode Ghs_openSession_W (L1_HubID *ghsInputPort,* L1_UINT32 * *pSessionID*)

Opens a session with the graphical host server indicated by ghsInputPort.

Parameters

ghsInputPort	Input Port of the Graphical Host Server where to close the session.
pSessionID	the L1_UINT32 variable this pointer points to will contain the sessionID of the newly opened session. This ID has to be used whenever trying to communicate with the Graphical Host Server.

Returns

L1_ReturnCode

- RC_OK the session could be created.
- RC_FAIL the session could not be created.

Warning

Once a Task does not want to interact with a Graphical Host Service any longer, do not forget to close the session using the Function Ghs_closeSession_W().

13.2.1.9 L1_ReturnCode Ghs_setBrush_W (L1_HubID *ghsInputPort,* L1_UINT32 *ghsSession,* GhsBrushStyle *brushStyle,* L1_BYTE *r,* L1_BYTE *g,* L1_BYTE *b*)

Sets the fill color for the given. ghsSession

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID of the session for which to set the brush.
brushStyle	The style of the brush to use.
r	Red component of the color to set.
g	Green component of the color to set.
b	Blue component of the color to set.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.10 L1_ReturnCode Ghs_setCanvasSize_W (L1_HubID *ghsInputPort,* L1_UINT32 *width,* L1_UINT32 *height*)

This functions sets the size of the canvas the Graphical Host Server provides.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
width	This specifies the horizontal size (x-axis) of the canvas in pixel.
height	This specified the vertial size (y-axis) of the canvas in pixel.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.11 L1_ReturnCode Ghs_setPen_W (L1_HubID *ghsInputPort*, L1_UINT32 *ghsSession*, GhsPenStyle *penStyle*, L1_BYTE *lineWidth*, L1_BYTE *r*, L1_BYTE *g*, L1_BYTE *b*)

Sets the pen to use for the drawing operations in this session.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID of the session for which to set the pen.
penStyle	Value of the enumeration GhsPenStyle defining what pen to use.
lineWidth	Width of the line in pixel.
r	Red component of the color to set.
g	Green component of the color to set.
b	Blue component of the color to set.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.2.1.12 L1_ReturnCode Ghs_setTextColour_W (L1_HubID *ghsInputPort*, L1_UINT32 *ghsSession*, L1_BYTE *r*, L1_BYTE *g*, L1_BYTE *b*)

This function sets the colour with which text will be drawn.

Parameters

ghsInputPort	Address of the Graphical Host Server to send this request to.
ghsSession	SessionID for the session to draw in.
r	Red component of the color to set.
g	Green component of the color to set.
b	Blue component of the color to set.

Returns

L1_ReturnCode

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

13.3 src/include/GraphicalHostService/GraphicalHostService.h File Reference

```
#include <GraphicalHostService/GraphicalHostClient.h>
```

Macros

- #define GHS_VERSION 0x01010303

13.3.1 Macro Definition Documentation

13.3.1.1 #define GHS_VERSION 0x01010303

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23–16: Minor Version
- 15–8 : Patch-level
- LSByte: Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release

Part VI

Appendix

Bibliography

[1] Mingw-w64 - for 32 and 64 bit windows. https://sourceforge.net/projects/mingw-w64/.

[2] Cmake — cross platform make. http://www.cmake.org/cmake/resources/software.html.

Bibliography

[1] Mingw-w64 - for 32 and 64 bit windows. https://sourceforge.net/projects/mingw-w64/.

[2] Cmake — cross platform make. http://www.cmake.org/cmake/resources/software.html.

Glossary

Abort Handler A Task specific function that gets invoked when a Task gets aborted due to a processor exeption.. 23

Black Board An L1 Entity that represents a safe global data structure. It avoids that reading Tasks can retrieve the data during it being updated. 10

Ceiling Priority An attribute to a resource that defines the maximum priority it may boost a Task to in case of a priority inheritance operation. 21, 22

Cluster An ensemble of Nodes. 5

Context switch The process of swapping Task-specific information usually associated with CPU registers during Task scheduling.. 25

Data Event An L1 Entity that combines data transfer and event notification. 10

Event A (binary) Event Entity to synchronise a single task with another task or a specific hardware peripheral through it's driver task. 10

FIFO queue An L1 Entity used to pass fixed size data in a buffered way between tasks. 10

Hub The generic L1 entity of VirtuosoNext used to implement all L1 entities.. 5–7, 9–12, 15, 17, 21

Inter-node Link Point to point communication system between two nodes. It can be virtualised when the communication medium is shared.. 7

Memory Block Queue An L1 Entity that is used to transfer blocks of memory without having to copy the complete memory block. 10

Memory Pool An L1 Entity providing exclusive ownership to memory blocks with a predefined size. 10

Node A processing device in a network containing at least a CPU and its local memory.. 3, 5–8, 12, 15, 17, 19

Platform Hardware system with CPU, specific peripherals and development support.. 15

Port An L1 Entity used to synchronise and communicate between Tasks using Packets. 10

Priority A task attribute used by the scheduler to activate the tasks in the ready list in order of their respective priority. 19–23, 25, 26, 29

Priority inheritance A term used in the context of the priotity based scheduling to reduce the blocking time by tasks that have taken ownership of a resource entity. 21

Priority Inversion Happens when a high priority Task has to wait for a low priority Task to release a resoruce. In fact the priority of the high priority task gets lowered to the priority of the low priority Task which holds the resource. Priority Inheritance is used to overcome this problem. 22

Resource An L1 Entity used to provide exclusive access to a logical resource. 10

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. 25

Semaphore An L1 Entity used to synchronize tasks based upon counting Event to sychronise between multiple tasks or hardware peripherals through it's driver task. 10

Task Active RTOS Entity: a function with its private workspace. 3, 5–12, 15–17, 19–26, 29

Acronyms

ISR Interrupt Service Routine. 9, 12, 15, 16, 23, 24, 26

RTOS Real-time Operating System. 3–5, 7, 12, 15

Index