OpenComRTOS-Suite Manual and API Manual

1.4

Altreonic NV
Gemeentestraat 61 Bus 1
3210 Linden
Belgium

http://www.altreonic.com

Contents

Part I

OpenComRTOS Fundamentals

Introduction

This document is intended as a manual that describes the use of OpenComRTOS, a network centric Real-time Operating System (RTOS) for developing embedded real-time applications. However, OpenComRTOS is more than that. OpenComRTOS 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 OpenComRTOS 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 OpenComRTOS has several unique properties that can make a big difference when developing embedded applications. We name some of the most important ones:

- Scalability: OpenComRTOS applications can be redeployed, mostly by recompilation of the application source code, from very small single microcontroller systems to target systems with a large number of distributed heterogeneous processing Nodes.
- Extensible. OpenComRTOS 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. OpenComRTOS 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 1 Kbyte while a fully featured distributed implementation only takes up about 5 Kbytes.
- 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, OpenComRTOS 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.

As one can see, OpenComRTOS 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 (OpenComRTOS-VE) supporting automatic code generation and visual tracing and debugging.

Scope

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

Chapter 1

General Concepts

1.1 Background of OpenComRTOS

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

In OpenComRTOS 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, OpenComRTOS 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.

OpenComRTOS 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 OpenComRTOS 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, OpenComRTOS 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

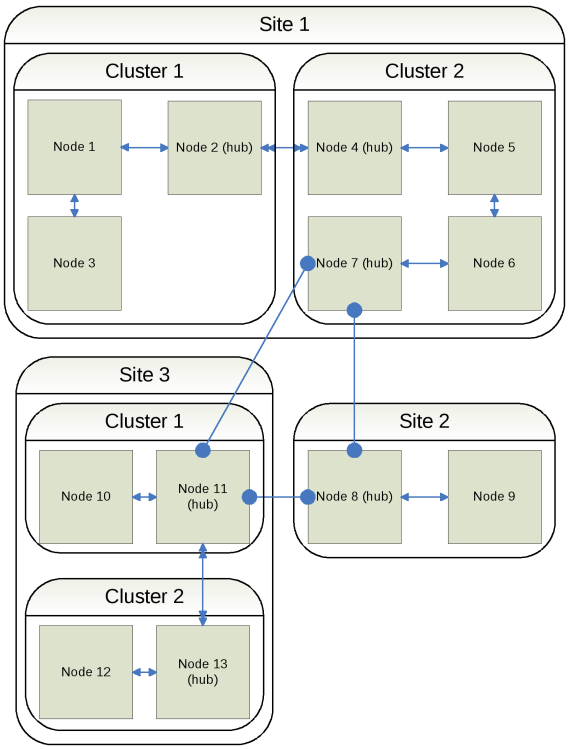


Figure 1.1: Generic structure of a distributed computing system

in a transparent way.

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 OpenComRTOS.

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 OpenComRTOS

OpenComRTOS 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 tunctional 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. L1 services will run on top of an application specific L2 communication layer.

OpenComRTOS 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 OpenComRTOS 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 OpenComRTOS 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

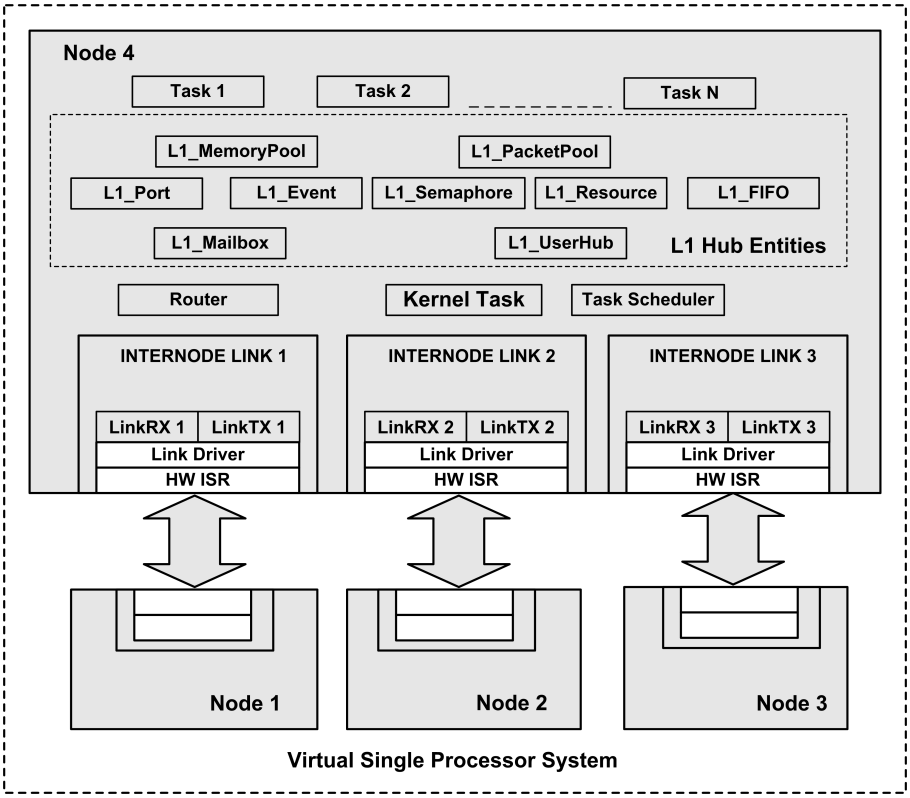


Figure 1.2: Logical structure of a distributed OpenComRTOS system

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 OpenComRTOS 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.

OpenComRTOS 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 Getd from an intermediate Entity. This effectively isolates Tasks from each other and increases the scalability of the system. At the application view OpenComRTOS 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 OpenComRTOS to reach the destination Node. The router itself is invisible to the application programmer. As all OpenComRTOS services are by default “distributed”, the routing is void when routing between local Tasks.

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.), OpenComRTOS has a Kernel with a Task scheduler.

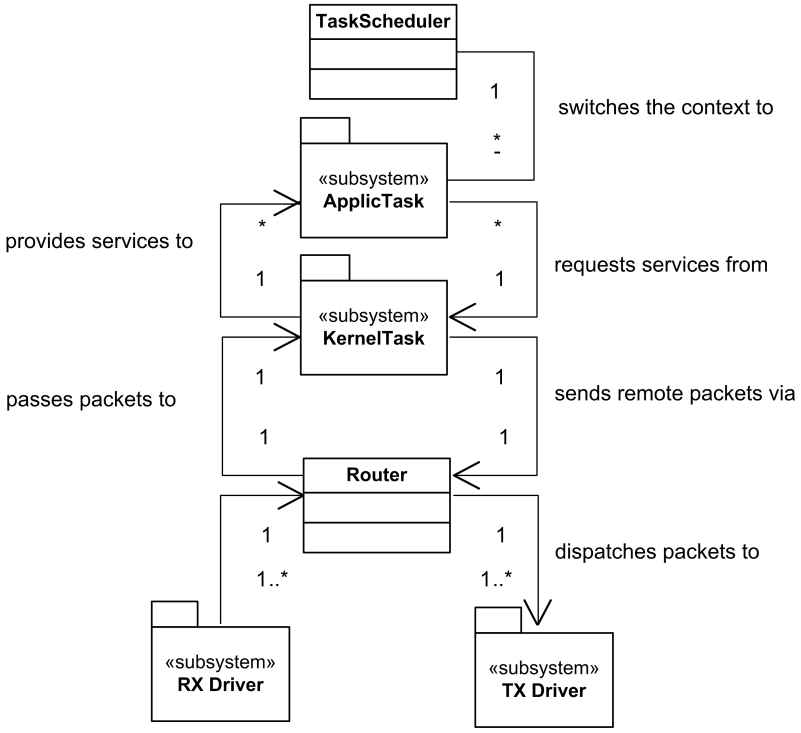


Figure 1.3: Functional relationship between entities of the distributed system

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.

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

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

- A Task scheduler that switches the CPU context between Tasks
- (One or more) Tasks that request services from the Kernel (using a Packet, but that may be hidden)
- The Kernel that provides these services. When one of the Tasks is remote, it passes on the service request to 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 ISR)

The relations are represented in Figure 1.3.

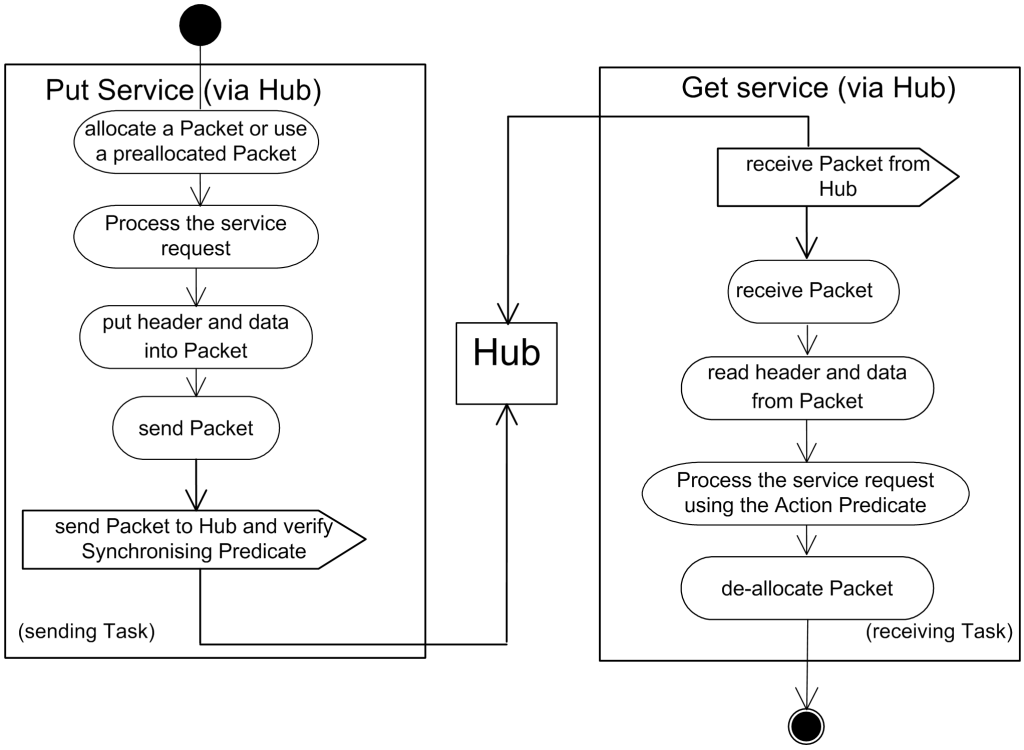


Figure 1.4: Generic scenario of a service request using the Hub entity

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 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.

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 sec-

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
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 available	Task enters WAIT state
Memory Pool	Put	Memory block available	Task reschedules, block returned to Pool

Table 1.1: Interactions between Tasks and Hubs

ond 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 OpenComRTOS 1.4.

In most cases the put request is performed by one Task while the get request is performed by another Task. However, as the interaction is through Hubs, it can as well be that e.g. driver Tasks or hardware specific ISRs 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 OpenComRTOS 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, OpenComRTOS 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 OpenComRTOS can be adapted to become another 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 OpenComRTOS 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 OpenComRTOS, 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 OpenComRTOS 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

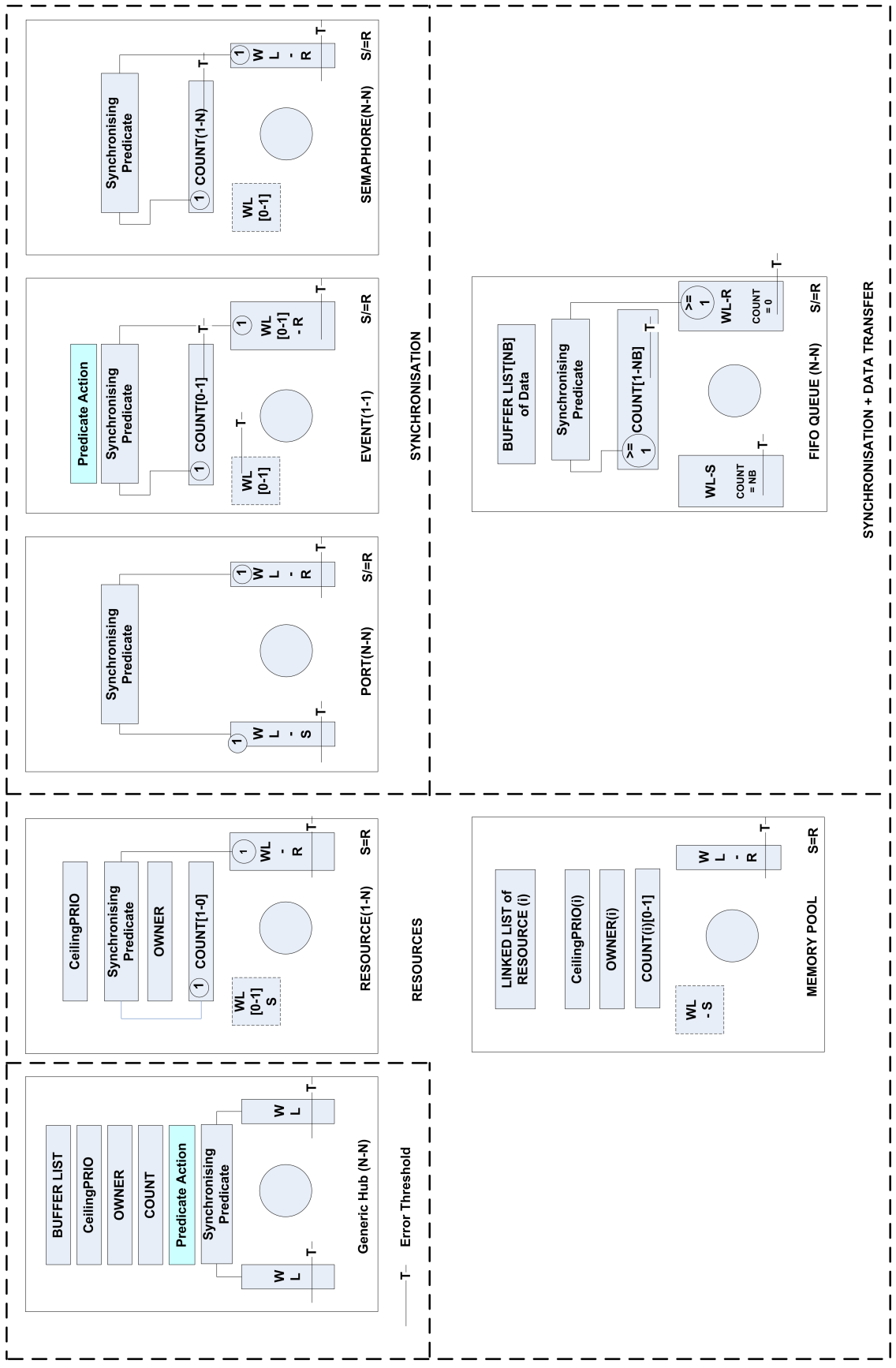


Figure 1.5: Graphical representation of the different Hub-types

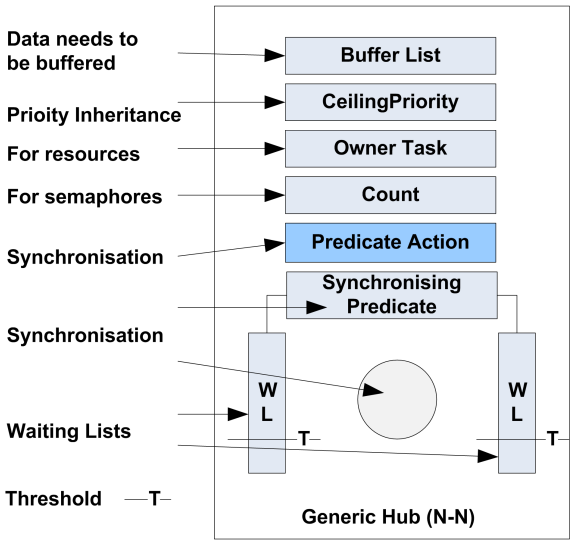


Figure 1.6: General conept of the generic Hub

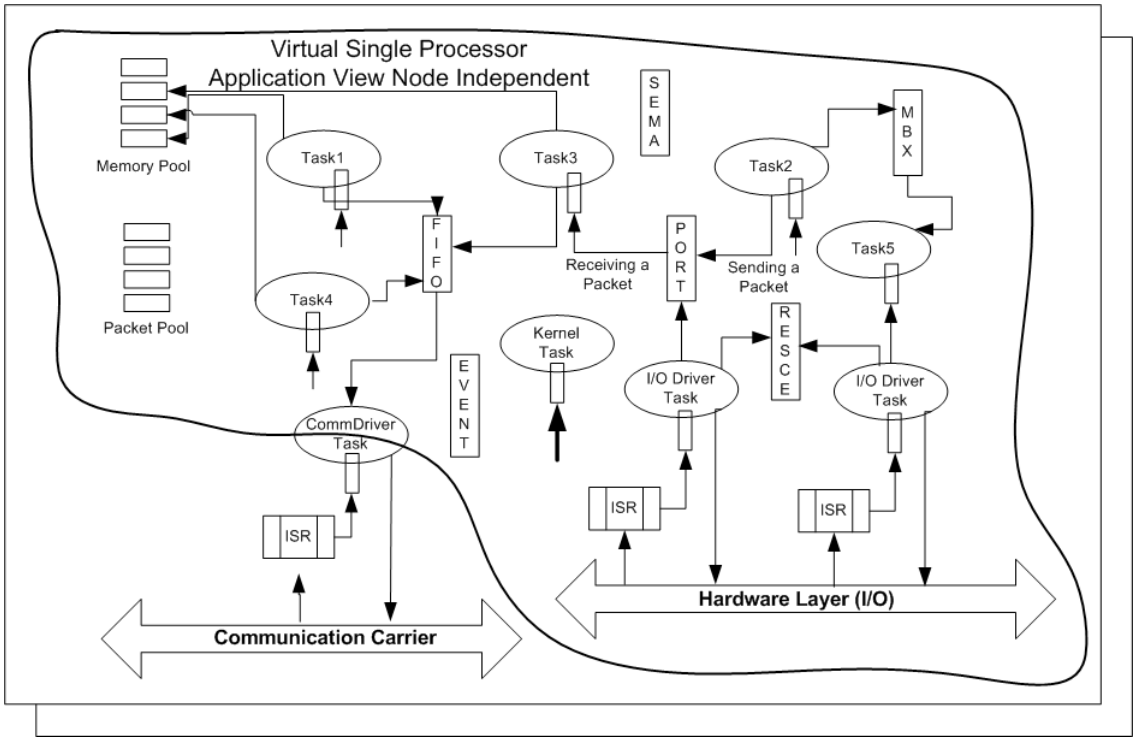


Figure 1.7: Possible distribution of Entities, involved in Task Interaction

(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 OpenComRTOS, 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 OpenComRTOS are a pragmatic superset of CSP. Although the OpenComRTOS 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

OpenComRTOS provides topology independent interaction between Tasks. All services, except when dictated otherwise by hardware dependencies, are from the application's Task point of view independent of the location in the network of Nodes. This applies e.g. to Task management services as well as to the L1 interaction Entities. The link hardware layer may implement the communication very differently from one Platform to another.

While in OpenComRTOS Tasks can interface directly with the hardware via Interrupt Service Routines, most often 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. OpenComRTOS defines dedicated Tasks, called Link Driver Tasks, that implements the OpenComRTOS system level communication protocol. Of course, in general, hardware will be accessed through a combination of an ISR and a Driver Task, but then a hardware and application specific protocol will be used.

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

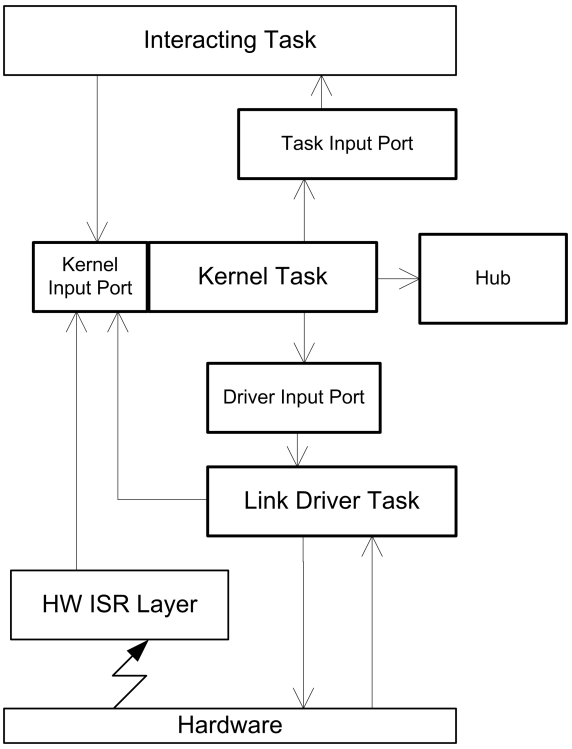


Figure 1.8: Interactions between HW, ISR Layer, Driver Task and application Tasks.

Chapter 2

Functional Design of the L1 Layer

Figure 2.1 presents the functional model of the OpenComRTOS Layer L1.

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

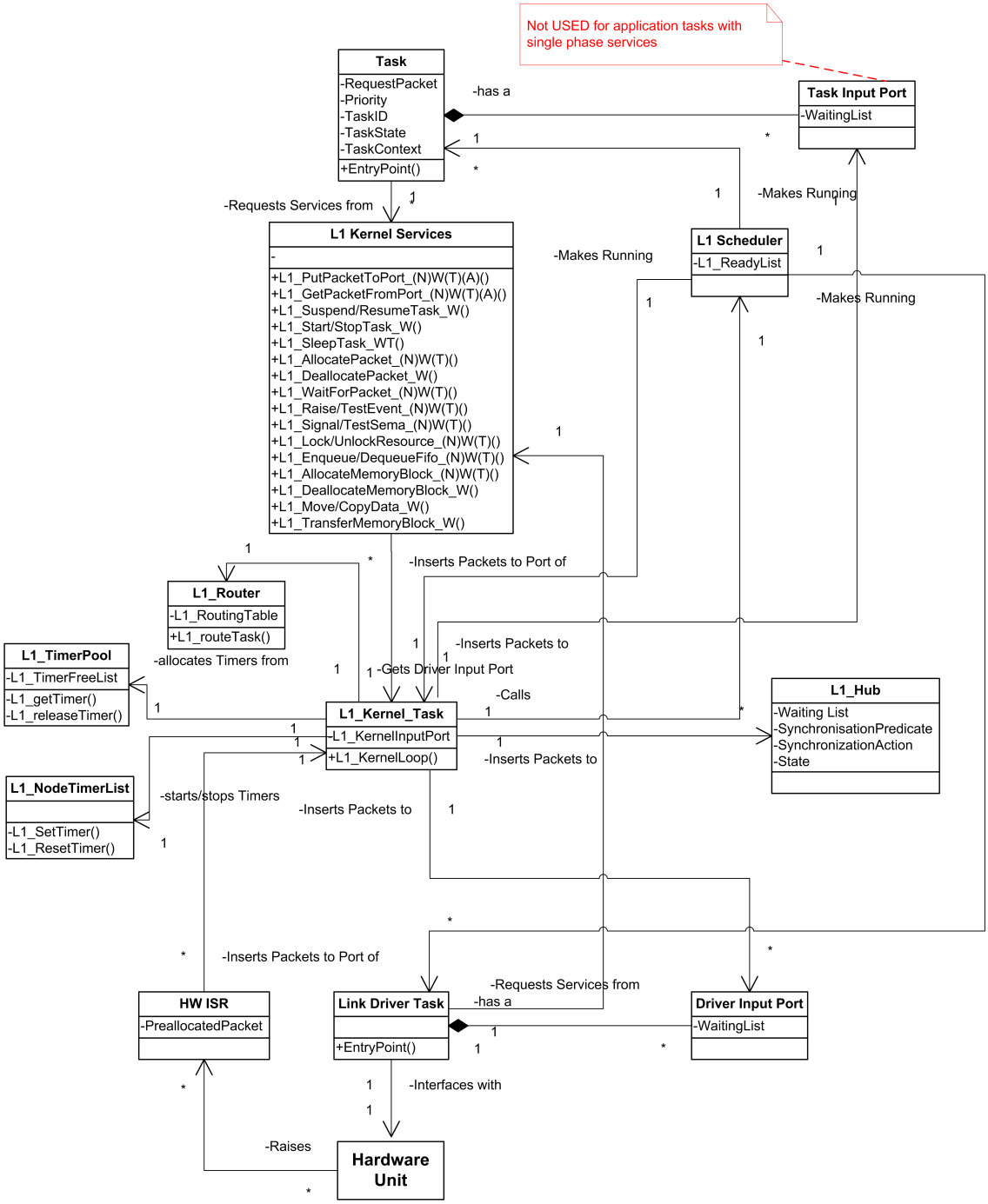


Figure 2.1: Functional model of OpenComRTOS

- 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 OpenComRTOS 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 OpenComRTOS:

- 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, OpenComRTOS 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 OpenComRTOS, the interacting Tasks interchange Packets of a fixed size. The "fixed size" of a Packet means that the physical size of Packet is always the same 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 OpenComRTOS 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, OpenComRTOS 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. A similar mechanism can be used to

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, OpenComRTOS 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 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 OpenComRTOs 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 OpenComRTOS, 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

OpenComRTOS 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

OpenComRTOS 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.

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 Getd 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 OpenComRTOS 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

OpenComRTOS 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 Getd Packets in the Kernel Port. The Kernel will deliver the Packets to the appropriate local Ports and Task Input Ports, or route the Packets to the LinkTX of the appropriate Inter-node Links (i.e. to a Driver Input Port) as applicable.

A Link Driver Task will implement the following behaviour:

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

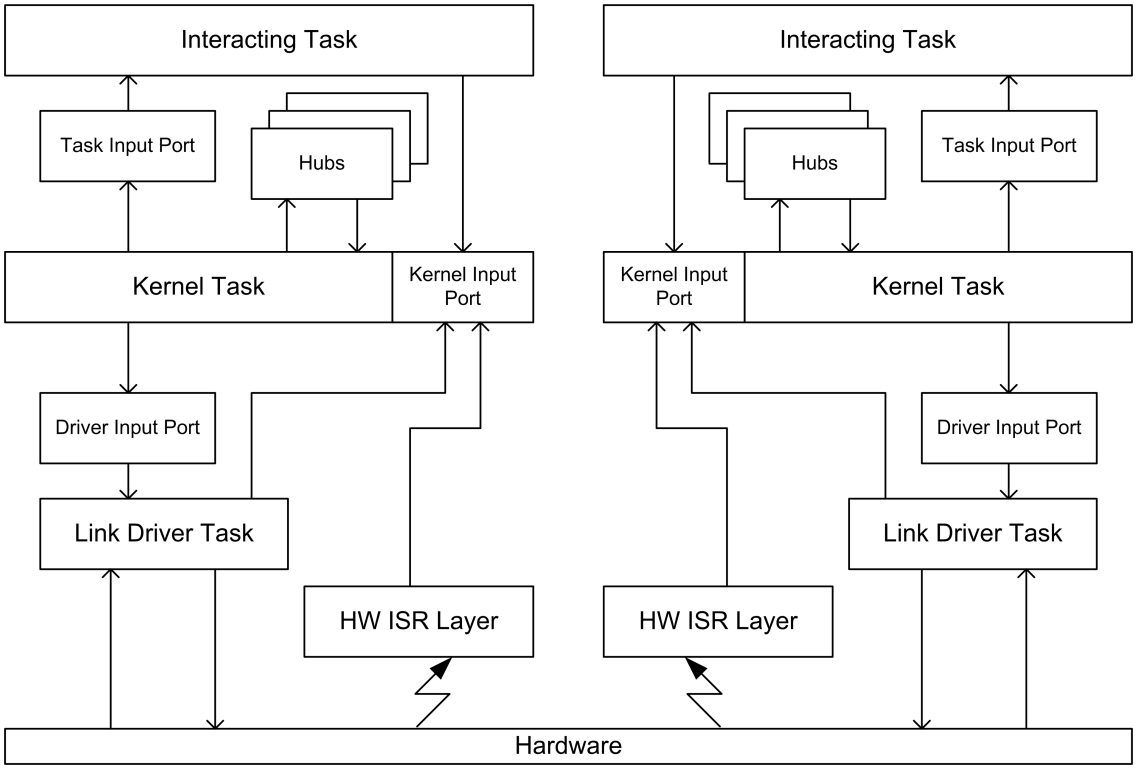


Figure 2.2: Communication between Inter-node Links and Tasks

- 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, OpenComRTOS has adopted the following mechanism:

An inter-node Link is addressed by the Input Port of the Driver Task that is driving the link.

When a Task calls a 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

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 OpenComRTOS, 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 OpenComRTOS Layer L0 is to request the services from the Kernel, which runs as a separate Task.

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

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

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

OpenComRTOS defines the following:

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

Each service of the Kernel is provided as a dedicated function call, exported to other SW layers as a part of the Kernel API, Section 10.1 (page 115) lists the complete API provided by the Kernel.

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, OpenComRTOS 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.

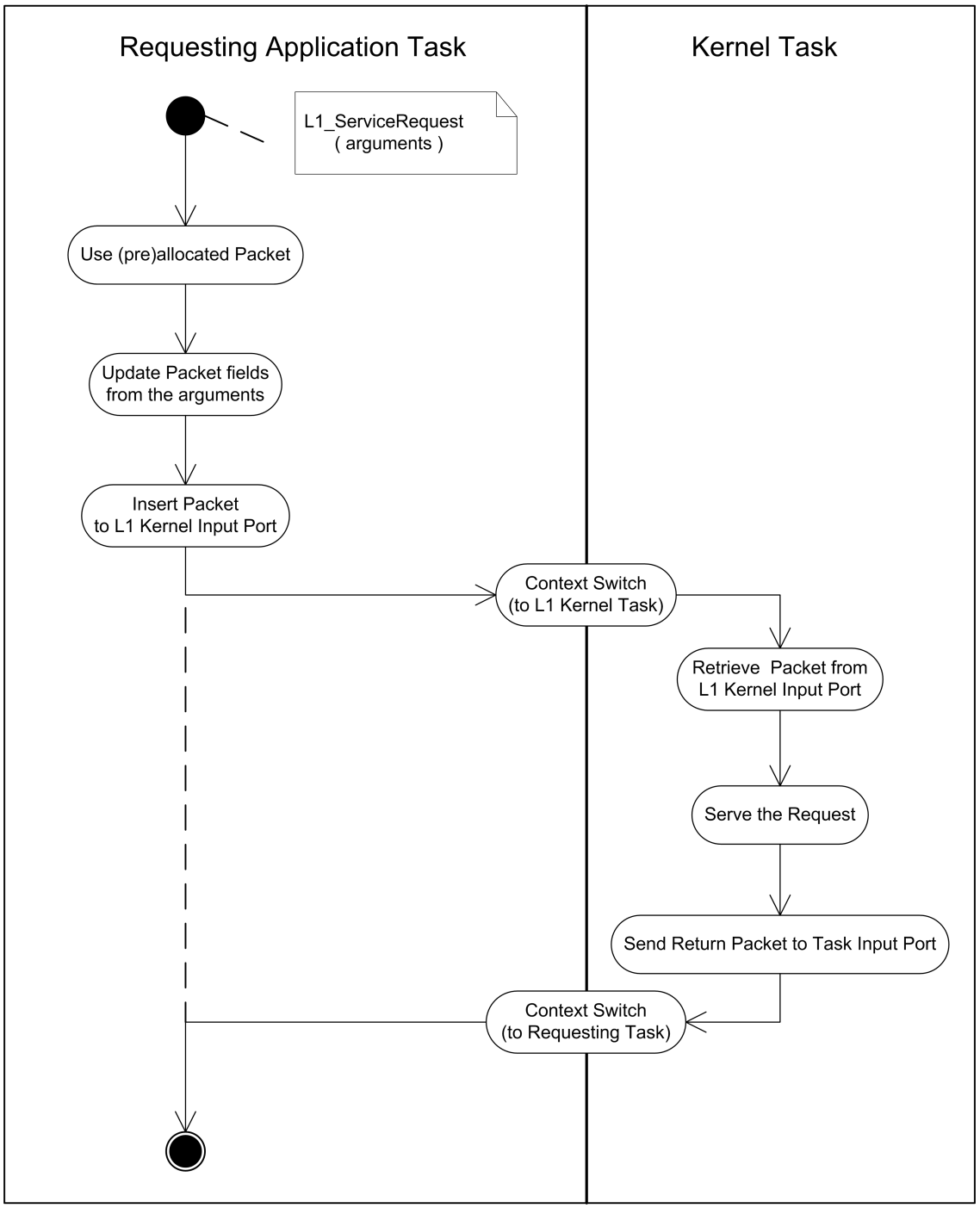


Figure 2.3: Template scenario of the serving of a request to the Kernel

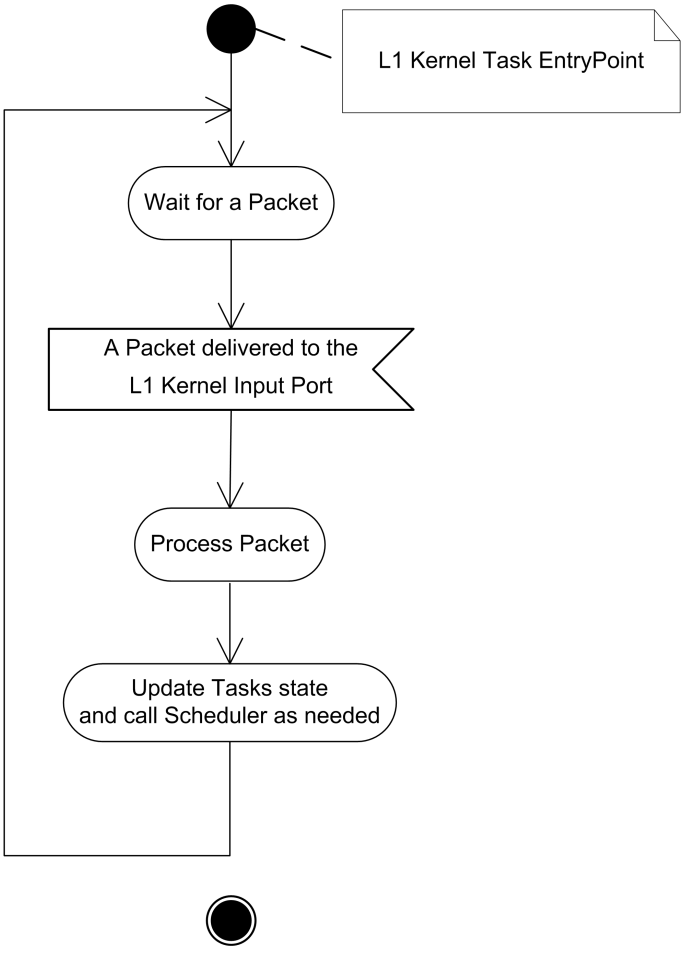


Figure 2.4: The Kernel Loop

2.3.4 Logical view of the Scheduler

For providing multi-tasking OpenComRTOS 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

OpenComRTOS is one of the few formally developed real time operating systems. This rigorous formalism has two benefits. First of all: good performance. This good performance is reflected in the small code size and the fast execution speed. This manual will guide you through the installation process of OpenComRTOS-Suite 1.4 which includes an OpenComRTOS win32 port and the corresponding examples. After guiding you through the installation process, the manual explains how to build the provided examples.

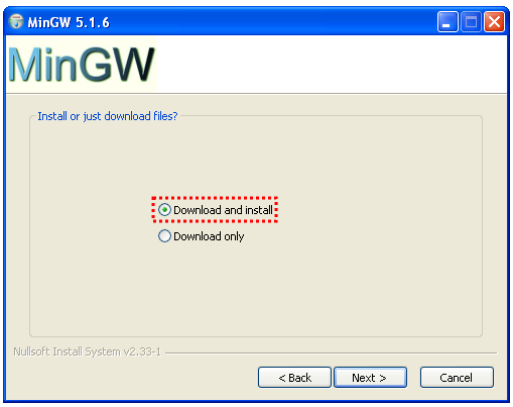
3.1 OpenComRTOS-Suite Installation Instructions

This section details how to setup the OpenComRTOS-Suite-1.4.3.x, which requires the MinGW toolchain, and the CMake build system. These instructions assume that you have received an USB key from Altreonic, if this is not the case you can obtain all necessary software from the Internet, the links are provided in the Bibliography.

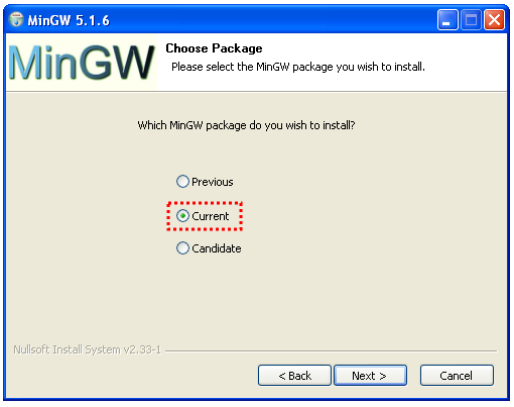
3.1.1 MinGW Tool-chain for Windows

MinGW [1] is a GNU GCC 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. The enclosed USB key contains version 5.1.6. of MinGW, to install it follow these steps:

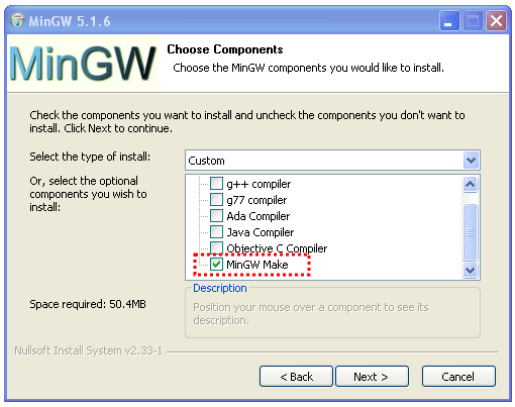
1. Start the installation process by executing:
   `OpenComRTOS_Suite_1.4\Win32\MinGW32\MinGW-5.1.6.exe`
   contained in the USB key you received from us.
2. When the installer asks whether to “Download and install” or to “Download only” (Figure 3.1a) select “Download and install”. The necessary files have been downloaded previously, and the installer will use these files, instead of downloading them again.
3. When the installer queries you which package of MinGW to install (Figure 3.1b), select “Current”.
4. In the component selection screen (Figure 3.1c) select to install “MinGW Make”. This component is an essential part of the OpenComRTOS build system.



(a) The MinGW installer operation selection screen



(b) The MinGW installer package selection screen.



(c) The MinGW installer component selection screen.

Figure 3.1: MinGW installer screens

3.1.2 Adding MinGW to the System Binary Search Path

The MinGW installer does not add the binary directory of MinGW ("`;c:\MinGW\bin`") to the System Binary Search Path of MS-Windows. This section explains the necessary steps to achieve this for MS-Windows XP, the procedure is similar for MS-Windows Vista and MS-Windows 7. Follow these steps to add the MinGW tools to the system wide Binary Search Path:

1. Add the binary path of MinGW to the PATH: Open the System Properties (right click on "My Computer" and select "Properties"), see Figure 3.2a.

2. There select the tab labeled "Advanced", in which you click on the Button labeled "Environment Variables" , see Figure 3.2b.

3. In the list box "System Variables" select the variable "Path" and click on the button labeled "Edit" (you can also double click on the list entry) , see Figure 3.2c.

4. In the dialogue "Edit System Variable" (see Figure 3.2d) add the following to the end of the Edit Field labeled " Variable value": "`;c:\MinGW\bin`". Be careful not to delete the previous value of "Path" because otherwise MS-Windows will not work correctly any longer.

3.1.3 Installing the SVM Toolchain

To compile OpenComRTOS Tasks that can be run inside the Save Virtual Machine for C (SVM), it is necessary to install a toolchain that is capable to produce pure ARM-Thumb-1 binaries. The SVM build system supports the arm-none-eabi-gcc as compiler and linker. The provided USB Memory Key contains the CodeSourcery toolchain for ARM, called "Sourcery G++ Lite toolchain for ARM EABI" [2]. To install the toolchain execute the file:

```
OpenComRTOS_Suite_1.4\Win32\arm-2009q1-161-arm-none-eabi.exe
```

During the installation apply the following settings:

- In the 'Choose Install Set': chose the Minimal installation, see Figure 3.3b.
- In the 'Add to Path?': chose 'Modify PATH for all users', see Figure 3.3a.

This completes the setup of the toolchain necessary to compile tasks for the SVM.

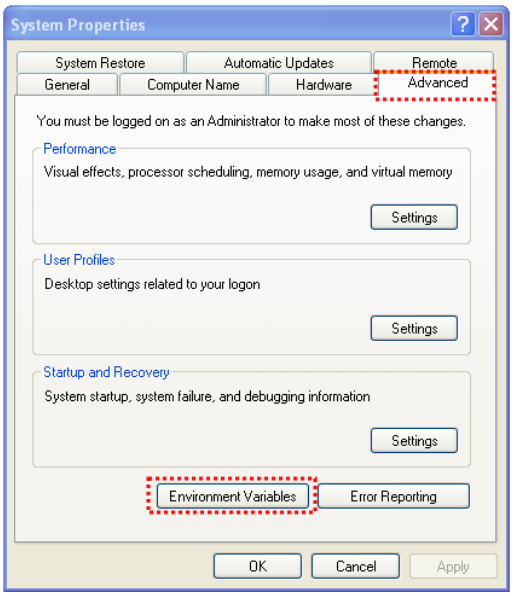
3.1.4 CMake Build System

OpenComRTOS uses the CMake build system [3] (version 2.6 or better) to build itself and applications using it. The following steps guide you through the installation process:

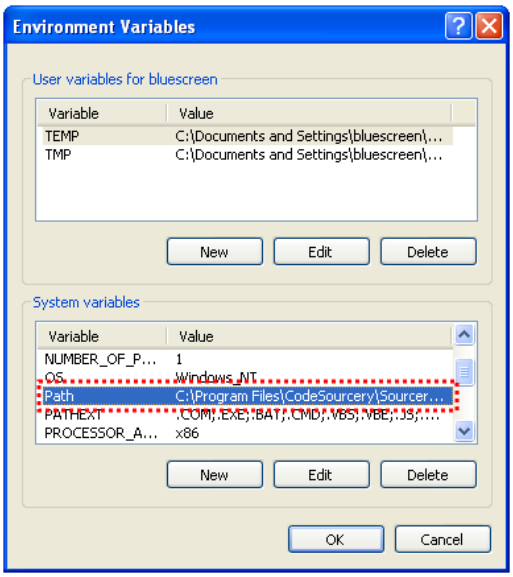
1. Start the installation process by executing:
   `OpenComRTOS_Suite_1.4\Win32\cmake-2.6.4-win32-x86.exe`
   from the enclosed USB key.

2. In the screen "Install Options" select "Add CMake to the system PATH for all users" (see Figure 3.4). This adds the CMake binary directory to the System Binary Search Path, which is necessary in order for the OpenComRTOS build system to be able to use CMake.

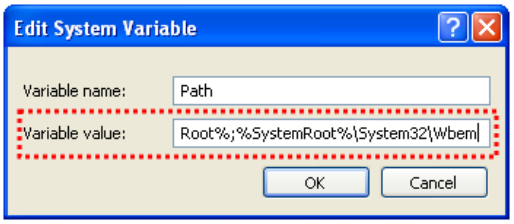
(a) Opening the System Properties Dialogue



(b) Opening the Environment Variables Dialogue

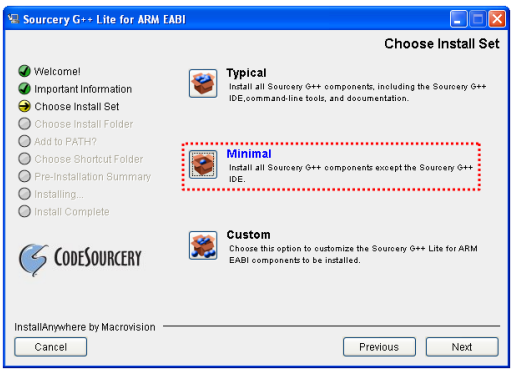


(c) Opening the Dialogue to modify the variable Path

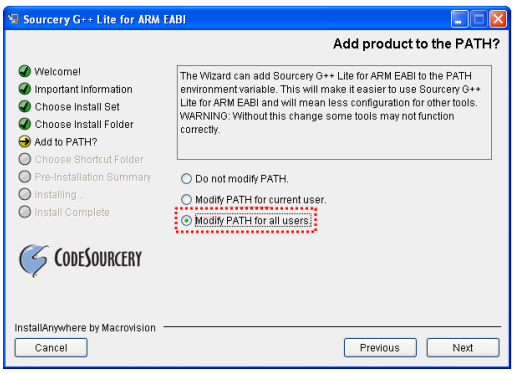


(d) Modifying the value of the variable

Figure 3.2: Setting the System Binary Search Path



(a) Setting the Minimal Installation Option



(b) Modifying the PATH for all users

Figure 3.3: Code Sourcery Installer Settings

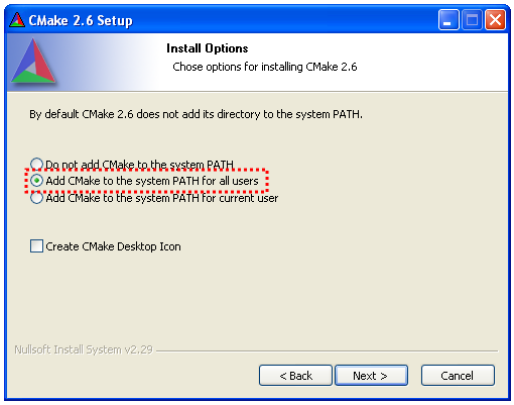


Figure 3.4: Adding CMake to the System Binary Search Path

3.1.5 Installing the OpenComRTOS-Suite

The OpenComRTOS-Suite installation image is available on the included USB key. To install it, execute: "`OpenComRTOS_Suite_1.4\Win32\OpenComRTOS-Suite-1.4.3.xmsi`", where '`x`' is a number representing the patch-level of the MSI. After this step the Altreonic OpenComRTOS-Suite 1.4 including the OpenComRTOS Kernel Images for Win32 is installed.

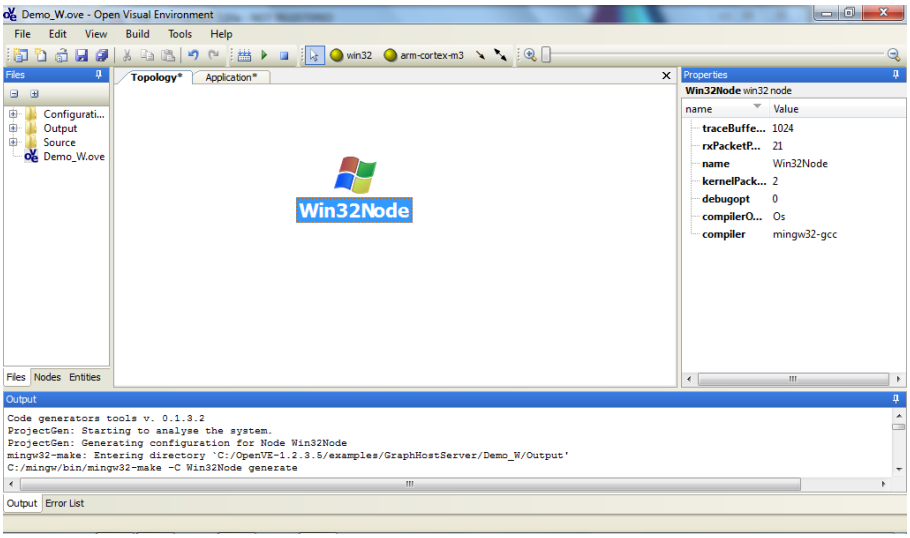


Figure 3.5: Topology of the Demo_W example

3.1.6 Installing an additional OpenComRTOS Kernel Image

The OpenComRTOS-Suite supports many other targets than just MS-Windows. The additional targets get shipped in form of a so called Kernel-Image, which is a tar.gz-file containing the libraries, metamodels, examples and documentaiton for the given target. To install the kernel image, extract it, using for instance 7-zip[1], and then copy its contents (directories: examples and targets) into the OpenComRTOS-Suite directory. During the copy operation you'll be prompted that the kernel image contains files that are already present. These files can be safely overwritten, as these are common files shared among all Kernel-Images. *Warning: It is your responsibility to ensure that you are not mixing different versions of kernel images!*.

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\win32`' below the OpenComRTOS-Suite installation directory.

1. Start OpenVE:
   In the Start Menu of MS-Windows select open the group 'OpenComRTOS-Suite-1.4.3.x' (x representing the patch-level of OpenVE). Inside this group click on the entry labeled 'OpenVE' to start OpenVE.

2. Open the 'Demo_W' project in OpenVE:
   In the menu-bar click on 'File' and then on 'Open Project' to open the 'Open Project' dialogue of OpenVE. Now navigate to the folder '`examples\GraphHostServer\Demo_W\Demo_W.ove`'' below the OpenVE installation directory (usually `c:\OpenComRTOS-Suite-1.4.3.x`). 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 Win32 Node, similar to the one shown in Figure 3.5. 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

[1] http://www.7-zip.org

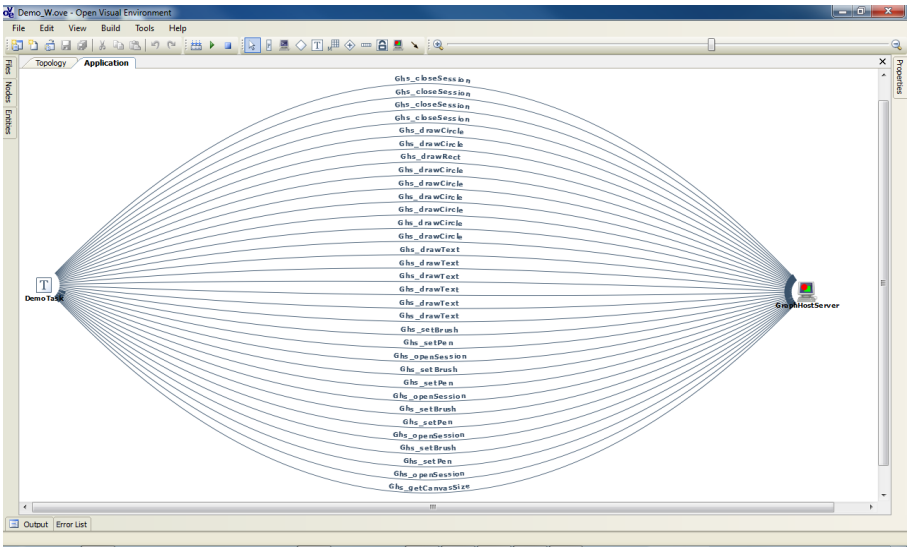


Figure 3.6: Application Diagram of the Demo_W example

the Window. Left click on the Win32 node to display its properties. Now check whether or not the property 'Compiler' refers to the compiler to use for Win32 Nodes on your system (The MinGW compiler which you installed in Lecture 1)[2].

4. Take a look at the Application Diagram for the example (Figure 3.6). 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.

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:\OpenComRTOS-Suite-1.4.3.x\examples\GraphHostServer\Demo_W\Output\-bin\win32_node.exe`

3.3 Summary

This document started by giving the installation instructions for OpenComRTOS-Suite 1.4 and its dependencies. The second part of this document detailed how to build the supplied examples.

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

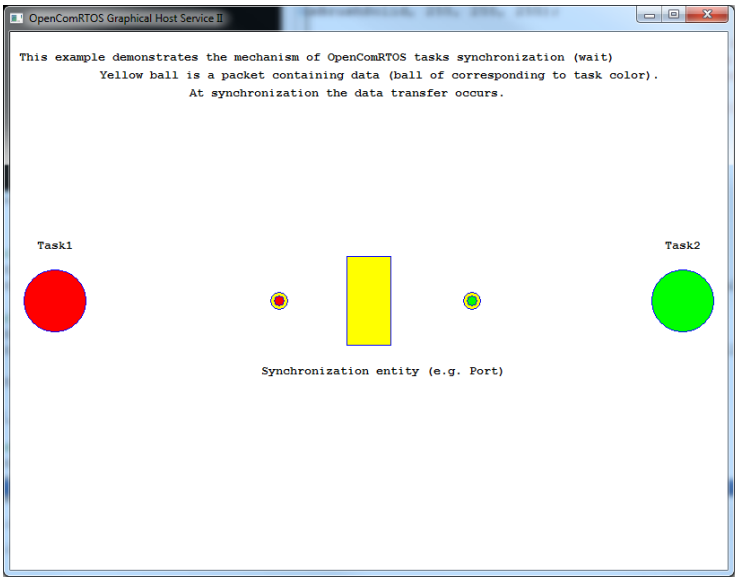


Figure 3.7: Screenshot of the running Demo_W example

Chapter 4

Installing ARM Cortex M3

Introduction

OpenComRTOS is one of the few formally developed real time operating systems. This rigorous formalism has two benefits. First of all: good performance. This good performance is reflected in the small code size and the fast execution speed. The second benefit is the usability. Due to the formal development, the user interface is clean and the usability is high. Nevertheless, before OpenComRTOS can be used on the ARM target, some setup work is necessary.

The following section presents a step by step manual which guides the user from the 'OpenComRTOS bundle for ARM and Win32' towards a working development environment.

This document is a step by step guide on how to bring up the OpenComRTOS LM3S6965 Evaluation kit. Section 4.2 covers the setup of the LM3S6965 Evaluation Board from Luminary Micro. While Section 4.3 details how to build a heterogeneous system consisting of an ARM -Node and a Win32-Node. Finally, Section 4.4, details how to retrieve trace information from an ARM node.

4.1 OpenComRTOS-Suite Installation Instructions

This section details how to setup the OpenComRTOS-Suite-1.4.3.x, which requires the MinGW toolchain, and the CMake build system. These instructions assume that you already installed OpenComRTOS-Suite 1.4 for MS-Windows, as described in Section 3.1.5. Furthermore, it assumes that you have installed the CodeSourcery Toolchain for arm-none-eabi, as detailed in 3.1.3.

4.1.1 Installing the OpenComRTOS Kernel Image for ARM-Cortex-M3

If you acquired the ARM-Cortex-M3 version of the OpenComRTOS-Suite then the USB-Stick that was supplied to you contains a directory called '`OpenComRTOS_Suite_1.4\ARM_Cortex_M3`'. This directory contains all the additional files that are availalble for the ARM-Cortex-M3 port. The OpenComRTOS kernel image is contained in the file '`OpenComRTOS_1.4.3.3_arm-cortex-m3.tar.gz`'. This image contains not only the binary distribution of OpenComRTOS for ARM-Cortex-M3, but also a number of examples for evaluation purposes. Section 4.3 explains how to build the examples and the specialties of the ARM-Cortex-M3 port. The installation instructions given in Section 3.1.6, on page 39 apply.

4.2 Setup of the LM3S6965 Development Board

This section details all the necessary steps to setup the LM3S6965 board under MS-Windows. After this section, you will have all necessary tools on your system to compile programs and configure the LM3S6965 board.

4.2.1 FTDI Driver Installation

In order to use the Lumniary Micro LM3S6965 board under MS-Windows it is necessary to install the device driver for the included FTDI chip

1. Attach the evaluation board to the PC using the provided USB cable.
2. When MS-Windows prompts you which driver to install instruct it to search in the directory: '`Luminary_FTDI_Driver`'.

After MS-Windows has completed the driver installation you should see the following devices in the Device Manager of MS-Windows (also see Figure 4.1):

- Ports (COM & LTP)
  - Stellaris Virtual COM Port (COMX) 'X' is the Com port number assigned to the RS232 port provided by the ARM board.
- Universal Serial Bus controllers
  - Stellaris Evaluation Board A
  - Stellaris Evaluation Board B

If any of these devices is not properly installed, you have to manually install the drivers by selecting 'Update Driver Software...' from the properties menu of the device (right click on the device).

4.2.2 Installing the LM Flash Programmer

In order to be able to download a program to the LM3S6965 evaluation board, it is necessary to install the Luminary Micro Flash Programmer. A Microsoft Installer Image containing the Luminary Micro Flash programmer (`LMFlashProgrammer.msi`) is available on the USB Memory key. To install simply executed the file, and follow the displayed instructions.

This completes the setup of the Luminary Micro LM3S6965 Evaluation Kit and the necessary toolchain.

4.3 Building and Running a Heterogeneous System consisting of an ARM Node and the Win32 Node

This section guides you through the necessary steps to run an already existing project using the provided ARM kit. Section 4.3.1, explains the general prepartions of an ARM example with a special focus on linking the ARM-Node with a Win32-Node RS232 link technology. While section 4.3.2 explains how to use TCP-IP over Ethernet to link ARM-Node with a Win32-Node.

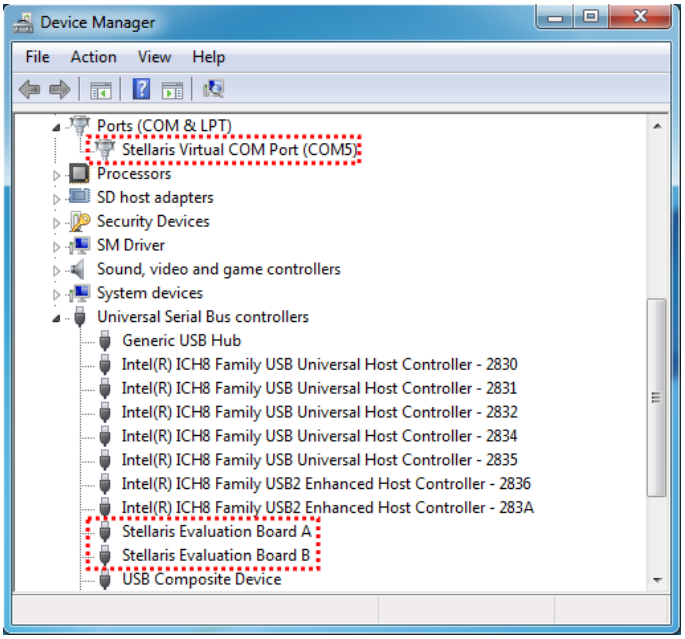


Figure 4.1: MS-Windows device manager with the three Stellaris Devices highlighted

4.3.1 Semaphore Loop using RS232 link Technology

These instructions concentrate on one the Semaphore-Loop example but are applicable to the other ARM-Examples as well:

1. Open the project: '`c:\OpenComRTOS-Suite-1.4.3.x\ArmExamples\Semaphore\Semaphore.ove`'. This will open the topology diagram (Figure 4.2).

   In the Topology check: Win32Node Compiler, ArmNode Compiler;

2. Adjust the topology to the concrete execution environment:
   The provided examples were developed and tested in an environment different from where you will execute them. It is therefore necessary to adjust them to your environment. There are three things that need to be checked:

   (a) Compiler of 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 Win32 node to display its properties. Now check whether or not the property 'Compiler' refers to the compiler to use for Win32 Nodes on your system[1].

   (b) Compiler of the ArmNode: 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. Right click on the ARM node to display its properties. Now check whether or not the property 'Compiler' refers to the compiler to use for Win32 Nodes on your system. If not adjust it[2].

   (c) RS232 Device of the Win32Node: To adjust this to the correct setting, right-click in the Topology view, on the node labeled 'Win32Node', to open the properties menu. Inside this menu, click on 'Node Configuration', to open the Node Configuration dialogue, there select the tab labelled 'Link Ports', see Figure 4.3, which displays the configured Link Ports for this node.

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

[2] If you followed the instructions given in Tutorial 1, the compiler should be set to "`arm-none-eabi-gcc.exe`".

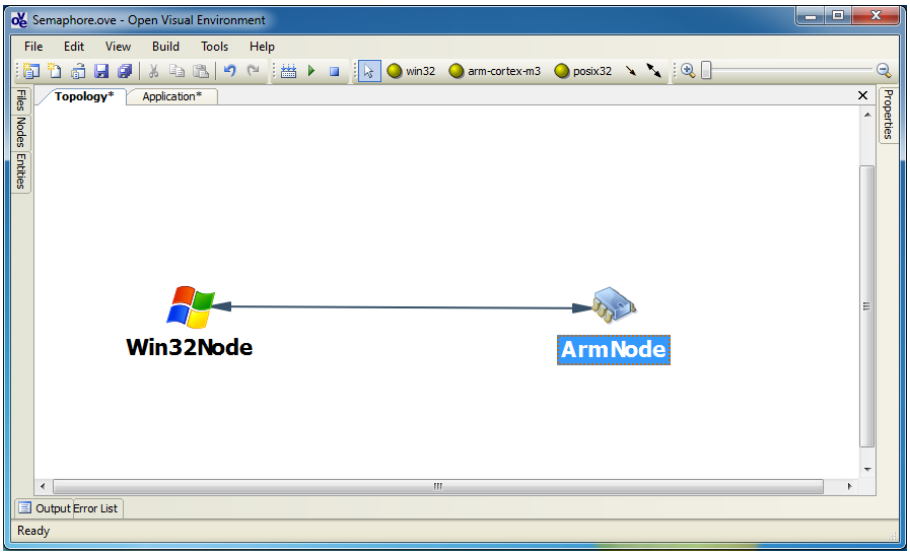


Figure 4.2: Topology of the Semaphore Example

On top you will see a combo-bos which lists all configured link ports for this node. Select the one named 'UartPort' and check that the property 'PortName' corresponds with the COM port assigned to the Stellaris UART device.

3. Build the project:
   Compile the example application using the menu-item 'Build' form the 'Build' menu. The build run should end with: "Build successful".

4. Execute the project:

   (a) Download the program to the ARM board:
   To download the generated binary to the LM3S6965 Evaluation Board use the 'Luminary Micro Flash Programmer' utility. In the tab 'Configuration' select 'LM3S6965 Ethernet Evaluation Board' (Figure 4.4a) and in the tab 'Program' (Figure 4.4b) select the previously generated bin-file for the ARM node :
   `c:\OpenComRTOS-Suite-1.4.3.x\ArmExamples\Semaphore\Output\bin\-ARM_Node.bin`

   After flashing the ARM board, press the RESET button on the board.

   (b) Execute the executable which represents the Win32 node of the system:
   `c:\OpenComRTOS-Suite-1.4.3.x\ArmExamples\Semaphore\Output\bin\-win32_node.exe`

Now the ARM-Node and the Win32-Node will synchronize their connection and then start to run the example, until you terminate it.

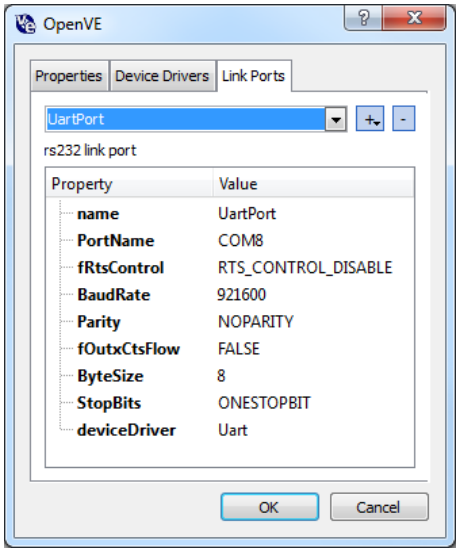
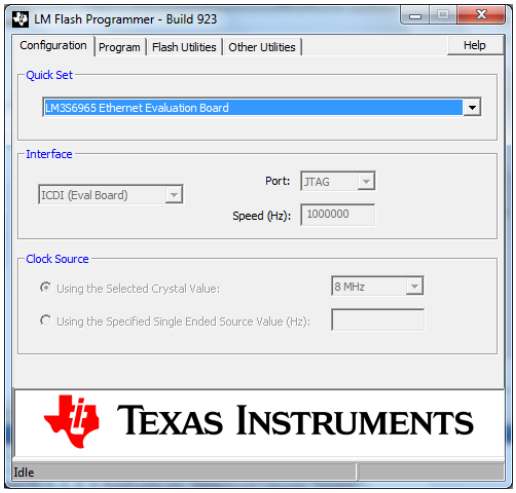
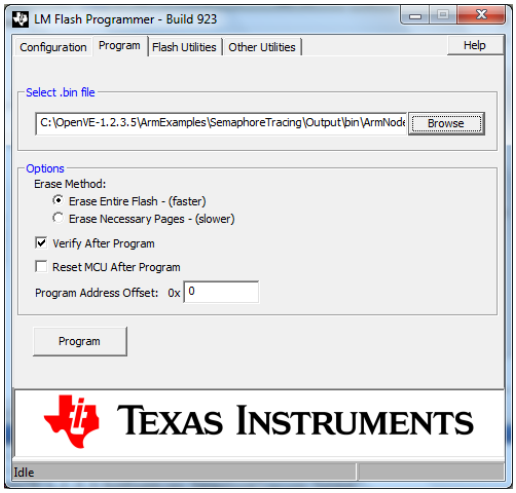


Figure 4.3: Setting the Link Port Parameters for the Win32 UART driver

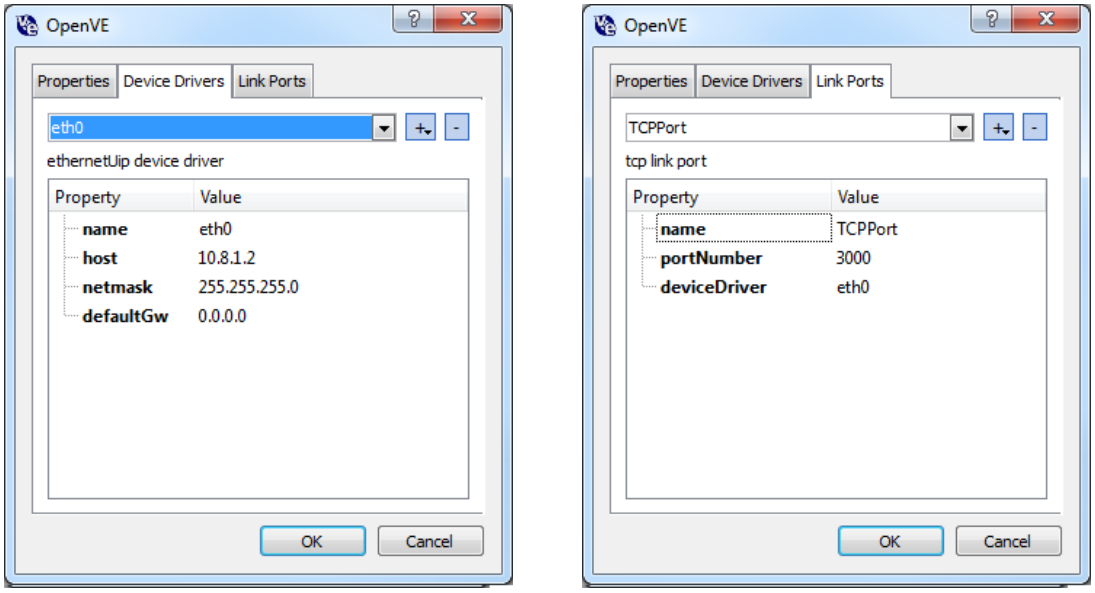


(a) Configuration Tab



(b) Program Tab

Figure 4.4: Luminary Micro Flash Programmer



(a) Device Driver Tab (b) Link Port Tab

Figure 4.5: Node Configuration of the ArmNode

4.3.2 Semaphore Loop using TCP-IP over Ethernet link Technology

This section details how to run the Semaphore Loop example, between an ARM-Node and a Win32-Node using TCP-IP over Ethernet.

1. Open the project: '`c:\OpenComRTOS-Suite-1.4.3.x\ArmExamples\Semaphore_TCPIP\-Semaphore.ove`'. This will open the topology diagram (Figure 4.2).This will open the topology diagram. In the Topology check: Win32Node Compiler, ArmNode Compiler, as detailed in Section 4.3.1.

2. Adjust the topology to the concrete execution environment:
   The TCP-IP configuration of both the ArmNode and the Win32Node need to be adjusted to comply to the local settings. For this purpose use open the Link Port Editor used previously and adjust the settings if necessary. Figures 4.5a – 4.5b show the default settings of the ArmNode and the Win32Node.

3. Build the project:
   Compile the example application using the menu-item 'Build' form the 'Build' menu. The build run should end with: "Build successful".

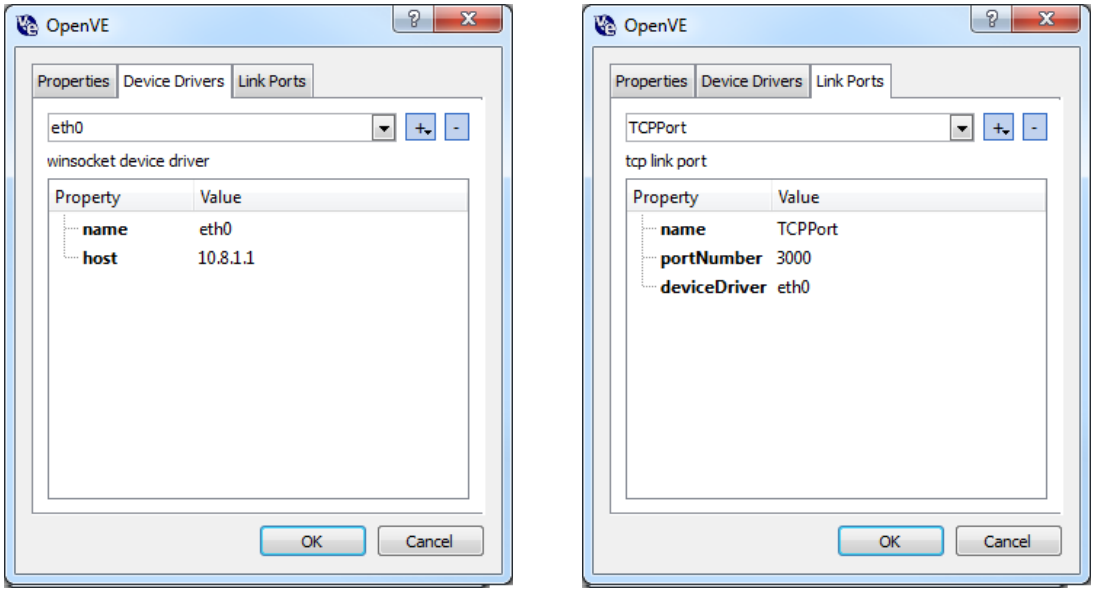
4. Execute the project:

   (a) Connect ArmNode and Win32 using an ethernet cable.

   (b) Download the program to the ArmNode.

   (c) Execute the executable which represents the Win32Node of the system.

   Now the ARM-Node and the Win32-Node will synchronize their connection and then start to run the example, until you terminate it.



(a) Device Driver Tab

(b) Link Port Tab

Figure 4.6: Node Configuration of the Win32Node

4.4 OpenComRTOS Tracing

Before reading this section, it is necessary to first understand how to work with OpenVE, please refer to its manual for this before reading on.

This section first gives an introduction to tracing in Open ComRTOS. Section 4.4.2 explains how to enable the tracing mode of OpenComRTOS, which collects runtime information within a Node. Section 4.4.3 explains the retrieval of these information from a Node and the storage in a File. Section 4.4.3.1 explains the additional changes done to the Semaphore-Example to make it generate traces, and how to display the retrieved tracefile using the OpenTracer application.

4.4.1 Tracing in OpenComRTOS

The OpenComRTOS kernel has a tracing mode, in which it collects the following events:

- Scheduling Events — which task ran at what time.
- Service Requests by Tasks — at what time did a task issue a specific service request. Naturally, the nature of the service request is captured as well.
- Hub interactions — when did the kernel perform an interaction with a hub.

While running in the tracing mode OpenComRTOS continuously collects these events and stores them in an internal buffer.

4.4.2 How to enable tracing

To enable the tracing mode you have to set node specific properties (node-properties). To set a node-property, open the node-property pane by double clicking on a node in the topology-diagram, your OpenVE

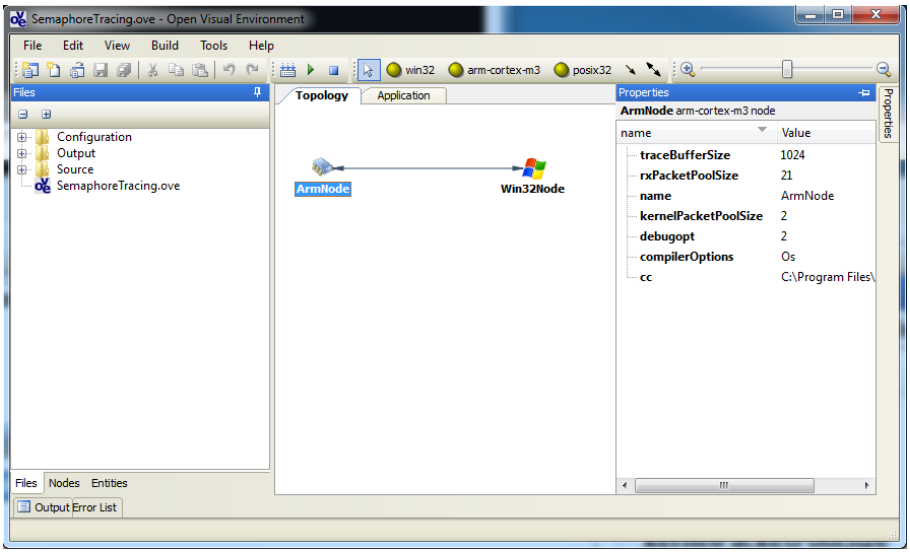


Figure 4.7: OpenVE with open Property Pane

window should now look similar to Figure 4.7.

There are two node-properties relevant to tracing:

- `debugopt` must be set to 2 — `debugopt` defines the debug-mode a node runs in. The default value of this property is '0'. To enable tracing "debugopt" must be set to '2'.

- `traceBufferSize` — `traceBufferSize` defines how many past events get recorded. It defaults to '1024', its upper limit is defined by the amount of memory available on the Node.

The Node now collects trace-information in its trace-buffer, but these trace information are not yet available to the OpenTracer application. For this, it first needs to be written to a file, i.e. the trace-buffer needs to be dumped, only then, the OpenTracer application can interpret the trace. The following section explains how to retrieve trace information from a Node to generate a trace-file.

4.4.3 How to retrieve a trace

An embedded Node has usually no file system available which could be used to store a trace. Instead, OpenComRTOS Nodes can transfer the contents of the trace-buffer to a StdioHostServer which will then write the retrieved trace information into a file for the OpenTracer application.

1. Add a StdioHostServer to the application diagram and place it on a Node of type Win32. The added StdioHostServer will be referred to as 'Shs' in this example. A StdioHostServer is a task which offers a range of stdio functionalities to embedded Nodes, such as the ARM Node. One of which is to receive the contents of a trace-buffer and write it onto a disk.

2. Add the instruction `DumpTraceBuffer(Shs)` to one of the tasks. This is the actual instruction which will transfer the contents of the trace-buffer to the StdioHostServer with the name 'Shs'. The retrieved trace information are then written to a file with the extension '`trace1`' (in the following this file will be referred to as trace1-file).

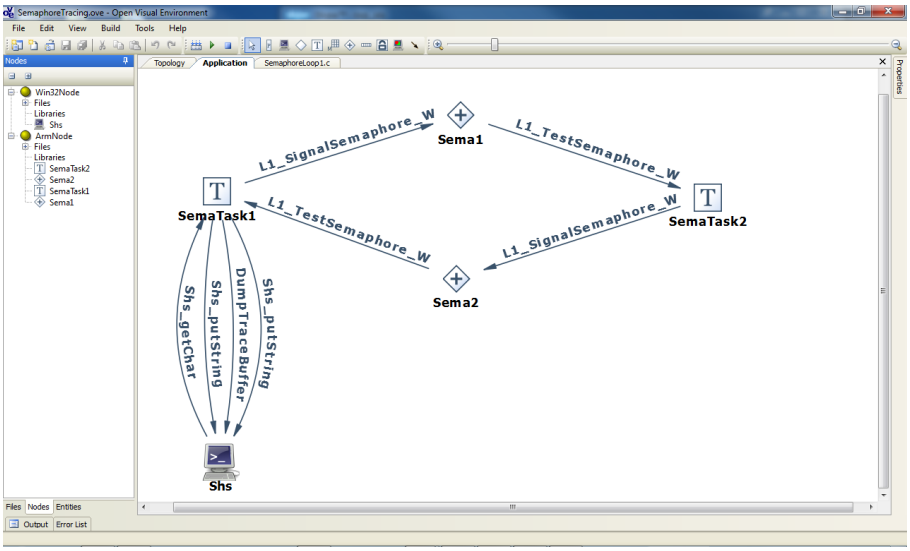


Figure 4.8: Tracing enabled Application Diagram

4.4.3.1 Extending the Semaphore-Example with Tracing

The project 'SemaphoreTracing' is a tracing enabled version of the previously shown Semaphore-Example. In addition to the changes explained in Sections 4.4.2 and 4.4.3 the following has been changed:

1. All Tasks and Hubs of the Semaphore Loop get mapped onto the ARM-Node.
2. A StdioHostServer has been added to the Application, and mapped onto the Win32 node. Using the StdioHostServer, the ARM node can write the contents of its trace buffer directly onto the disk of the Win32Node. This is means there is no need for special hardware to retrieve trace information.
3. In one of the two tasks of the Semaphore-Loop add a for-loop which which lets the loop execute for 500 times.
4. Before writing out the trace-buffer a message is displayed on the StdioHostServer announcing that the dumping of the trace takes place.
5. The task calls the function `DumpTraceBuffer()` to write the contents of the trace buffer into a file;
6. After the dumping of the trace is completed another message is displayed which asks the user to press enter to continue the execution. This is necessary because otherwise the system would continuously overwrite the trace file it generated.

Depending on your application you may need to perform similar changes. Figure 4.8 shows the resulting Application Diagram for the Semaphore Example.

To test it, follow the instructions given in Section 4.3. To display the retrieved trace open the trace-file (extension 'trace') with OpenTracer, this should give you an output similar to the one shown in Figure 4.9.

4.5 Summary

This guide covered the installation process of the OpenComRTOS Suite ARM Evaluation Kit. It covered the installation of the necessary software packages in Sections 4.2 and 4.1. This was followed in Section

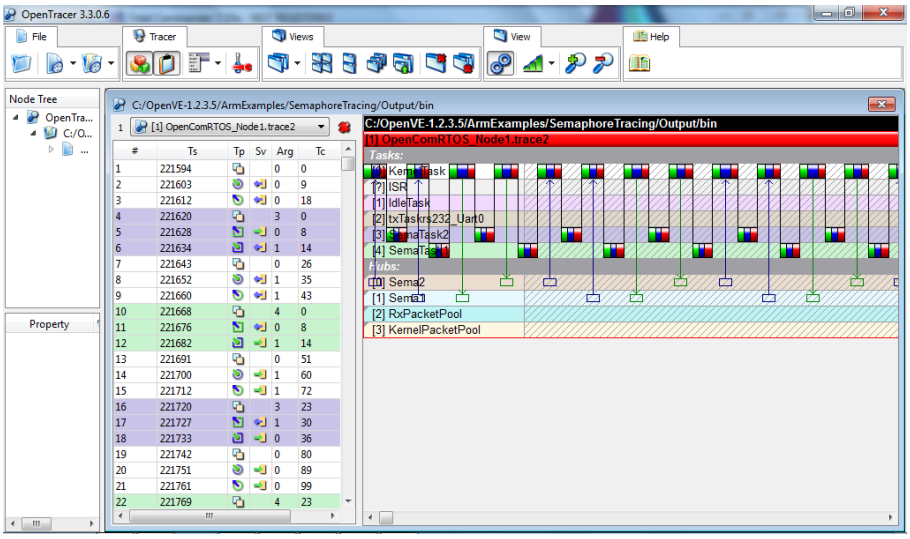


Figure 4.9: OpenTracer displaying the retrieved Trace

4.3 with an explanation on how to work with heterogeneous systems consisting of an ARM-Cortex-M3 Node and a Win32 Node. Finally, Section 4.4 explained how tracing works in OpenComRTOS, as well as how to enable the tracing mode, retrieve a trace from the ARM evaluation kit and the necessary steps to display it using OpenTracer.

Chapter 5

Installing NXP-Coolflux

Introduction

OpenComRTOS is one of the few formally developed real time operating systems. This rigorous formalism has two benefits. First of all: good performance. This good performance is reflected in the small code size and the fast execution speed. This manual will help you to install and evaluate the OpenComRTOS Kernel Image for NXP-CoolFlux for OpenComRTOS-Suite 1.4. After this, the manual explains how to build the provided examples and gives details of the individual examples.

5.1 OpenComRTOS-Suite Installation Instructions

This section details how to setup the OpenComRTOS-Suite-1.4.3.x, which requires the MinGW toolchain, and the CMake build system. These instructions assume that you already installed OpenComRTOS-Suite 1.4 for MS-Windows, as described in Section 3.1.5. Furthermore, it assumes that you have installed the necessary toolchain to build software for the NXP-CoolFlux. OpenComRTOS for NXP-CoolFlux was built using version: cf6-c1.2-08R1.1 of the Target compilers. If this is not the then please do so before proceeding.

5.1.1 Installing the OpenComRTOS Kernel Image for NXP-CoolFlux

This image contains not only the binary distribution of OpenComRTOS for NXP-CoolFlux, but also a number of examples for evaluation purposes. Section 5.2 explains how to build the examples and the specialties of the NXP-CoolFlux port. The installation instructions given in Section 3.1.6, on page 39 apply.

5.2 Examples

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\coolflux'` below the OpenComRTOS-Suite installation directory.

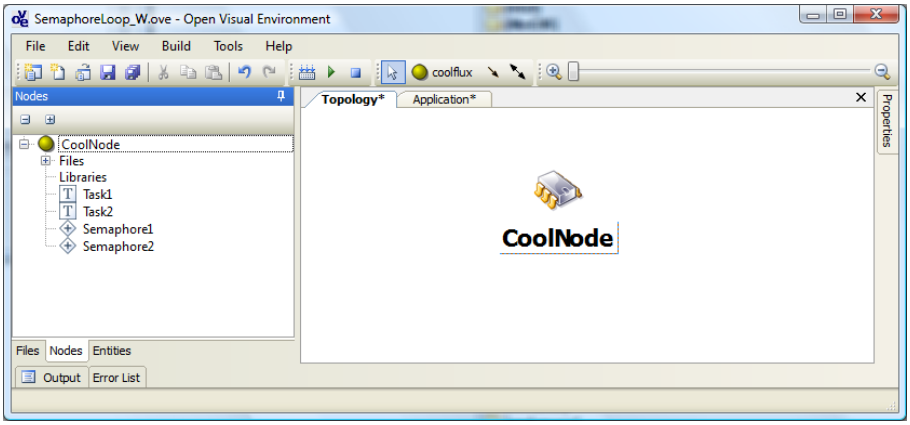


Figure 5.1: Topology of the SemaphoreLoop_W example

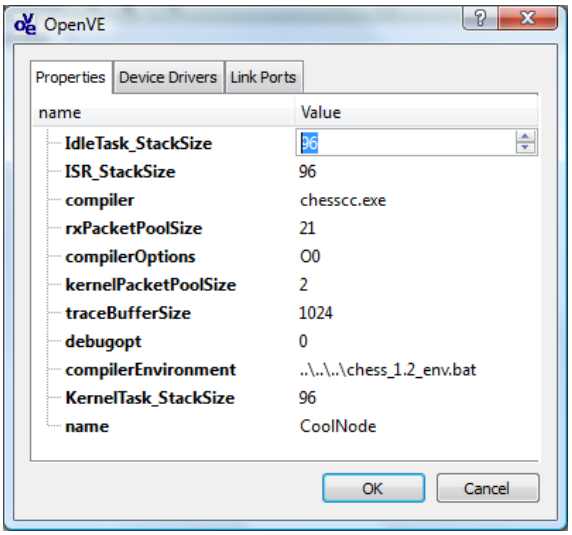


Figure 5.2: Node Properties of a NXP-CoolFlux Node

5.2.1 Loading and building a NXP-CoolFlux Example with OpenVE

This section explains how to build the SemaphoreLoop_W example of the provided NXP-CoolFlux examples. To build this example follow these steps:

1. Start OpenVE:
   In the Start Menu of MS-Windows select open the group 'OpenComRTOS-Suite-1.4.3.x' (x representing the patch-level of OpenVE). Inside this group click on the entry labelled 'OpenVE' to start OpenVE.

2. Open the 'SemaphoreLoop_W' project in OpenVE:
   In the menu-bar click on 'File' and then on 'Open Project' to open the 'Open Project' dialogue of OpenVE. Now navigate to the folder '`examples\coolflux\SemaphoreLoop_W\SemaphoreLoop_W.ove`' below the OpenVE installation directory (usually `c:\OpenComRTOS-Suite-1.4.3.x`). There, select the file '`SemaphoreLoop_W.ove`' and click on the button labeled 'open' to open the project. You should now see a topology consisting of a NXP-CoolFlux Node, similar to the one shown in Figure 5.1. The topology diagram is a graphical representation of the project-topology.

3. Check the Node-Properties for the node labelled 'CoolNode': Open the 'Node Configuration Dia-

logue', by opening the properties menu of the Node labeled 'CoolNode', using a right click, and then selecting the menu entry 'Node Configuration', there select the tab labeled 'Properties', see Figure 5.2. Every Node of type NXP-CoolFlux has the following properties, which might have to be adjusted depending on the actual project:

- name — Specifies the name of the Node. This name is use when mapping entities to nodes, and also while building there will be a directory in the Output directory with the same name, where all the files specific to this Node get stored.
- `KernelTask_StackSize` — Stack size in words for the Kernel Task, the default value is 96 words.
- `IdleTask_StackSize` — Stack size in words for the Idle Task, the default value is 96 words.
- `ISR_StackSize` — Stack size in words for the ISR handling, the default value is 96 words. This is currently unused, as the ISR at the moment does not use a separate stack.
- `compilerEnvironment` — This is an optional script that should be called before building, to setup the environment for the chosen toolchain. All NXP-CoolFlux examples call the script located in the directory `examples\coolflux`. The script was generated during the installation of the Target Toolchain. Please replace it with a script fitting your installation.
- `compiler` — Name of the compiler to use.
- `compilerOptions` — OpenComRTOS usually comes compiled with different compiler optimizations applied. For NXP-CoolFlux this not the case, there is only the option `O0` available.
- `debugopt` — Specifies the tracing level. The current version of the NXP-CoolFlux port does not support tracing at the moment. Thus leave it at its default value of `0`.
- `kernelPacketPoolSize` — How many L1_Packets to provide in the Kernel Packet Pool. This setting is only relevant to Multi-Node (MP) versions of OpenComRTOS, thus irrelevant to the present NXP-CoolFlux port.
- `rxPacketPoolSize` — How many L1_Packets to provide in the Receiver Packet Pool. This setting is only relevant to Multi-Node (MP) versions of OpenComRTOS, thus irrelevant to the present NXP-CoolFlux port..

4. Take a look at the Application Diagram for the example (Figure 5.3). 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.

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". The resulting elf-file has been copied into the directory `Output\bin` below the original project directory. It can directly be loaded into the NXP-CoolFlux simulator.

   Follow these instructions to build all examples detailed in the following subsections.

5.2.2 Example: SemaphoreLoop_W

This example represents a simple Semaphore loop, where two tasks signal each other using two semaphores, without any interrupts being enabled. It is our standard test to determine the time it takes to perform a context switch on the given target. One loop represents a total of eight context switches. Figure 5.3 shows the application diagram of the example.

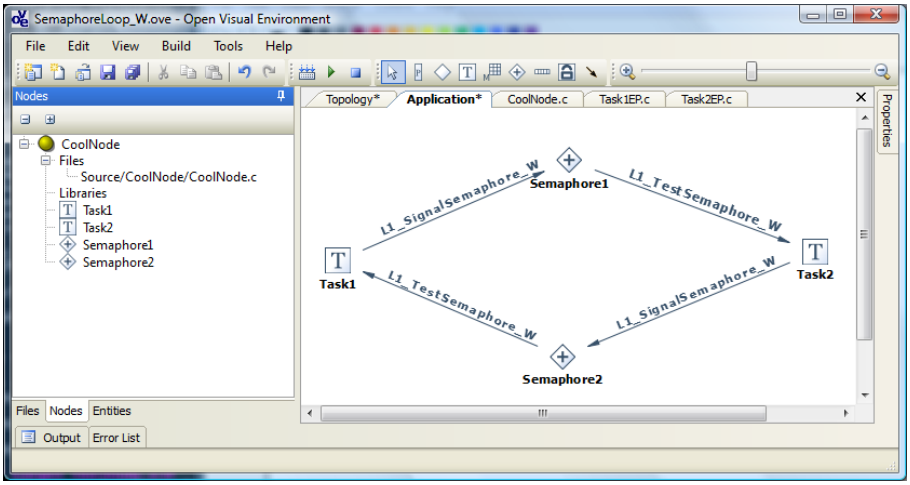


Figure 5.3: Application Diagram of the SemaphoreLoop_W example

5.2.2.1 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 4
  - Stack size of each Task: 96 words
- Memory consumption:
  - PMEM: 1785 words
  - XMEM: 786 words
  - YMEM: 1 word
  - XYMEM: 1 word
- Loop time: 3826 cycles

5.2.3 Example: PortLoop_W

This example represents a simple Port loop, where two tasks signal each other using two Ports, without any interrupts bein enabled. The difference between a Port and a Semaphore-Loop is that with Port loop it is also possible to move data from one Task to another Task. However, this test was performed without moving any data between the Tasks. Figure 5.4 shows the application diagram of the example.

5.2.3.1 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 4
  - Stack size of each Task: 96 words
- Memory consumption:
  - PMEM: 1759 words
  - XMEM: 784 words

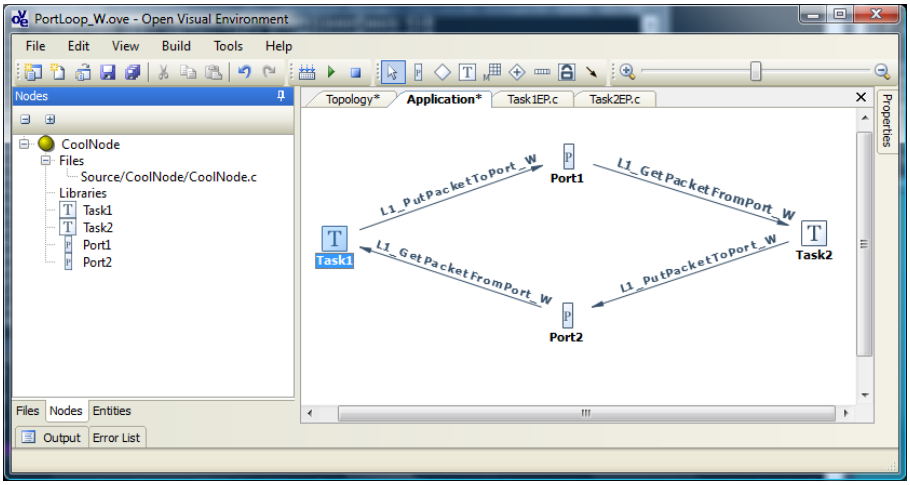


Figure 5.4: Application Diagram of the PortLoop_W example

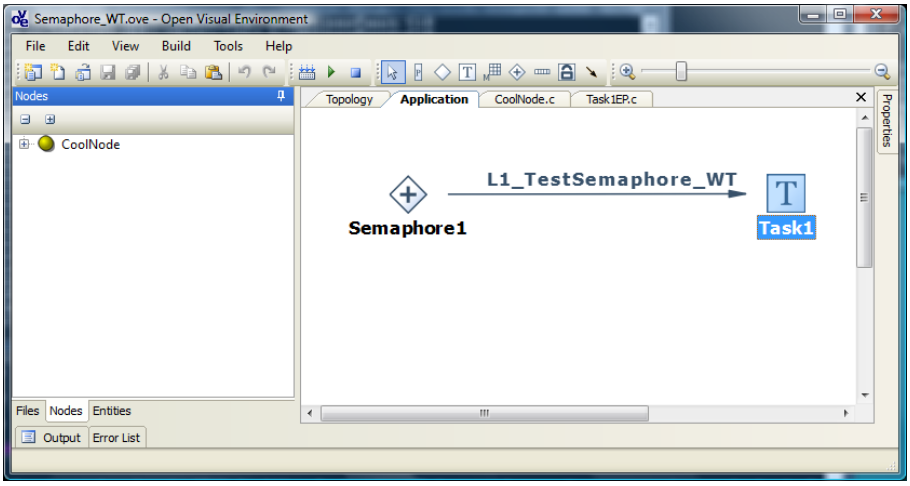


Figure 5.5: Application Diagram of the Semaphore_WT example

  - YMEM: 1 word
  - XYMEM: 1 word

- Loop time: 3871 cycles

5.2.4 Example: Semaphore_WT

This example demonstrates that the _WT services of the NXP-CoolFlux port are working. It does this by having a Task waiting with a timeout of 10 ticks, for a Semaphore to become signaled. This invokes the pTimer implementation to wait for 10 external interrupts on interrupt pin 1. Once the timeout has expired, the Task gets rescheduled and then tries again to test the Semaphore. Figure 5.5 shows the application diagram of the example.

To try out the ISR in the simulator this example provides two input files for the NXP-CoolFlux simulator, which are located in the directory: `simulator`, below the project directory. The file `irq-input.txt` contains the sequence in which interrupts will be generated, the file `irq-timing.txt` contains the duration of the interrupts. For the interrupts to be generated by the simulator, it is necessary to instruct the

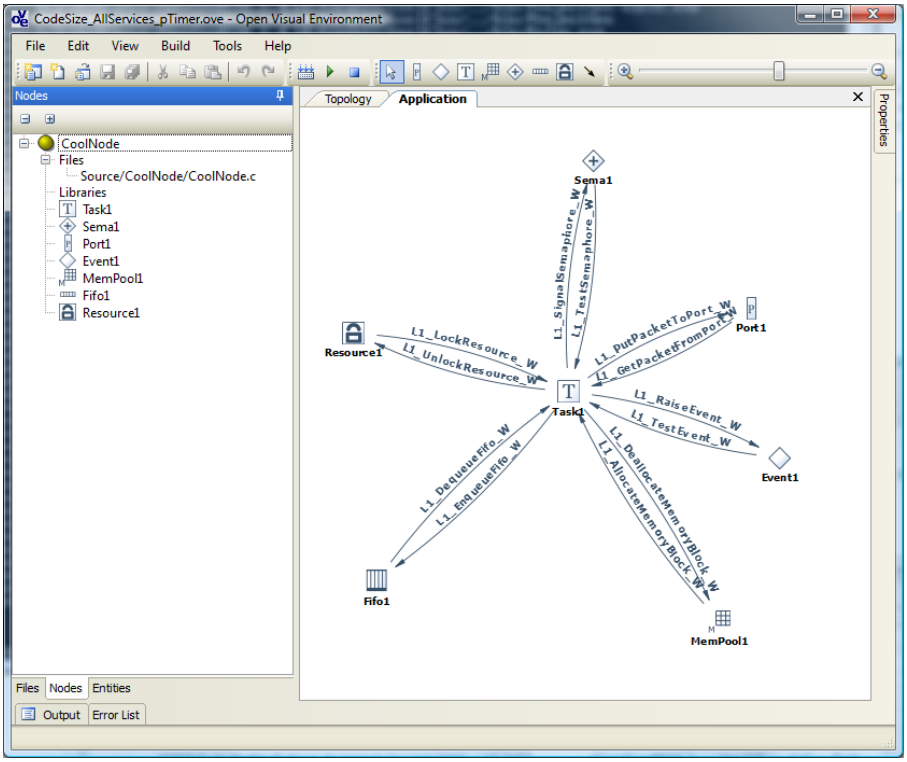


Figure 5.6: Application Diagram of the CodeSize_AllServices_pTimer example

simulator to read from the file at every cycle.

5.2.4.1 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 3
  - Stack size of each Task: 96 words
  - pTimer ISR enabled
- Memory consumption:
  - PMEM: 2001 words
  - XMEM: 613 words
  - YMEM: 1 word
  - XYMEM: 1 word

5.2.5 Example: CodeSize_AllServices

The purpose of this example is to determine how much memory is necessary when enabling all services available in OpenComRTOS. The example itself does not perform any useful operation, it is purely meant for code-size estimation. This version of the example does not include the pTimer ISR. Figure 5.6 shows the application diagram the the CodeSize_AllServices_pTimer, which is identical to the application diagram of this example.

5.2.5.1 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 3
  - Stack size of each Task: 96 words
- Memory consumption:
  - PMEM: 2066 words
  - XMEM: 721 words
  - YMEM: 1 word
  - XYMEM: 1 word

5.2.6 Example: CodeSize_AllServices_pTimer

The purpose of this example is to determine how much memory is necessary when enabling all services available in OpenComRTOS. The example itself does not perform any useful operation, it is purely meant for code-size estimation. This version of the example does include the pTimer ISR. Figure 5.6 shows the application diagram of this example.

5.2.6.1 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 3
  - Stack size of each Task: 96 words
  - pTimer ISR enabled
- Memory consumption:
  - PMEM: 2301 words
  - XMEM: 721 words
  - YMEM: 1 word
  - XYMEM: 1 word

5.2.7 Example: InterruptLatencyMeasurement

For a developer of real time systems it is very interesting to know how long it takes after an IRQ until the ISR respectively the Task get executed. This example allows one to measure these two values within the simulator. In this example a custom ISR gets inserted into the system which signals the Event `Event1`. Task `Task1` waits for this event to be signaled, and thus gets woken up once the ISR signaled the Event. By placing breakpoints at the right places one can use the simulator to determine the interrupt latencies. Figure 5.6 shows the application diagram the the InterruptLatencyMeasurement example.

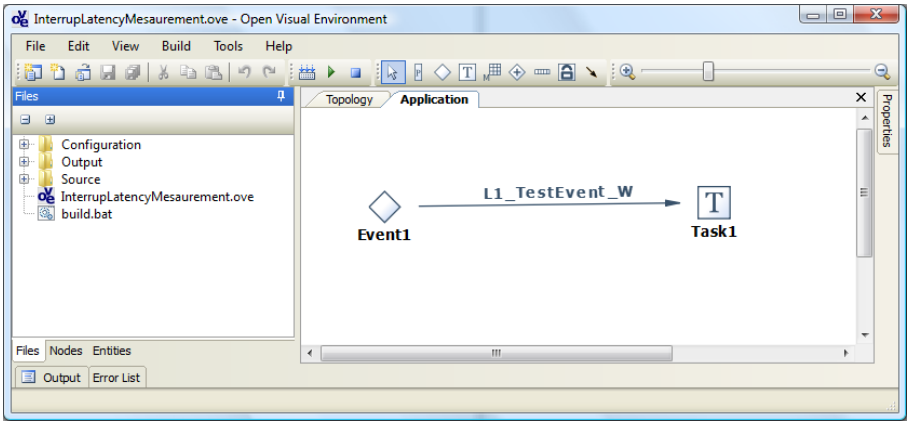


Figure 5.7: Application Diagram of the InterruptLatencyMeasurement example

5.2.7.1 Building and running the Example

This example uses a custom ISR, which is not supported by OpenvE at the present moment of time. Due the fact that NXP-CoolFlux is a Harvard architecture it is also not possible to hook the ISR dynamically without paying a performance penalty. Therefore, this example is preprepared, to be easily built by hand. For this purpose the example directory contains the file `build.bat` which will execute the necessary steps to build the example. The result of the build process is an elf file in the directory `Output\bin`.

To try out the ISR in the simulator this example provides two input files for the NXP-CoolFlux simulator, which are located in the directory: `simulator`, below the project directory. The file `irq-input.txt` contains the sequence in which interrupts will be generated, the file `irq-timing.txt` contains the duration of the interrupts. For the interrupts to be generated by the simulator, it is necessary to instruct the simulator to read from the file at every cycle.

5.2.7.2 Example setup and Measurement Results

- Example Configuration:
  - Number of Tasks: 3
  - Stack size of each Task: 96 words
  - One ISR.
- Memory consumption:
  - PMEM: 1988 words
  - XMEM: 660 words
  - YMEM: 1 word
  - XYMEM: 1 word
- Interrupt Latency Measurement results:
  - cycle 1997 — IRQ got signaled.
  - cycle 2003 — ISR breakpoint reached;
  - cycle 2112 — First useful instruction can be executed, after the ISR has stored the context of the interrupted task. This is the point in time that we define as IRQ to ISR latency.
  - cycle 2902 — Task gets scheduled again, due to the Event having been signaled by the ISR. This is the point in time we define as IRQ to ISR latency.

- Resulting Latency:
  1. IRQ to ISR latency — 115 cycles
  2. IRQ to Task latency — 910 cycles

5.3 Summary

This document started by giving the installation instructions for OpenComRTOS-Suite 1.4 and its dependencies. This was followed by instructions on how to install the NXP-CoolFlux evaluation Kernel-Image. In the second part this document detailed how to build the supplied examples and shortly gave an introduction to each example, which included code-size and performance figures where applicable.

Part III

Usage Tutorials

Chapter 6

Howto Use the Open System Inspector

Introduction

This tutorial explains the procedure to use the Open System Inspector to inspect the system state, during runtime of the Semaphore_W_MP example. In this example two nodes: Win32Node1 and Win32Node2 execute a semaphore-loop distributed between them.

1. Start OpenVE
2. Open the project located at: `examples\win32\Semaphores\Semaphore_W_MP`.
3. Save the project as a new project called Semaphore_W_SVM. To do this follow these steps:
   (a) Go to the main menu: File → 'Save Project As'. This opens the corresponding dialogue, shown in Figure 6.1.
   (b) In the text-field labelled 'Name:' insert the new name: 'Semaphore_W_MP_OSI'.
   (c) Press on the button labelled 'Finish'. This will copy the current project at the new location; close the current project; and open the newly created project.
4. To be able to connect with the OpenSI-GUI to the system an OSIRelayComponent has to be added to the application diagram. This component acts as an interface between the OpenSI-GUI and the OpenComRTOS environment, by relaying messages between the two worlds. When adding it to the application diagram use the following properties:

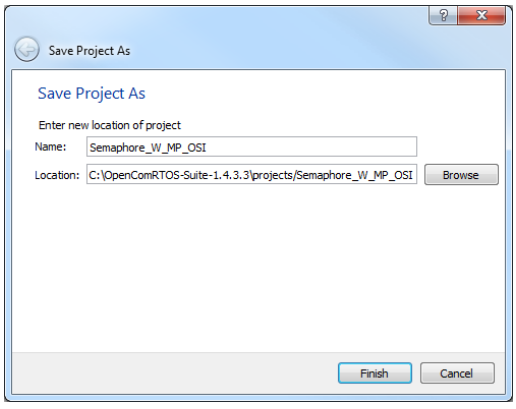


Figure 6.1: The Save Project As Dialogue from OpenVE

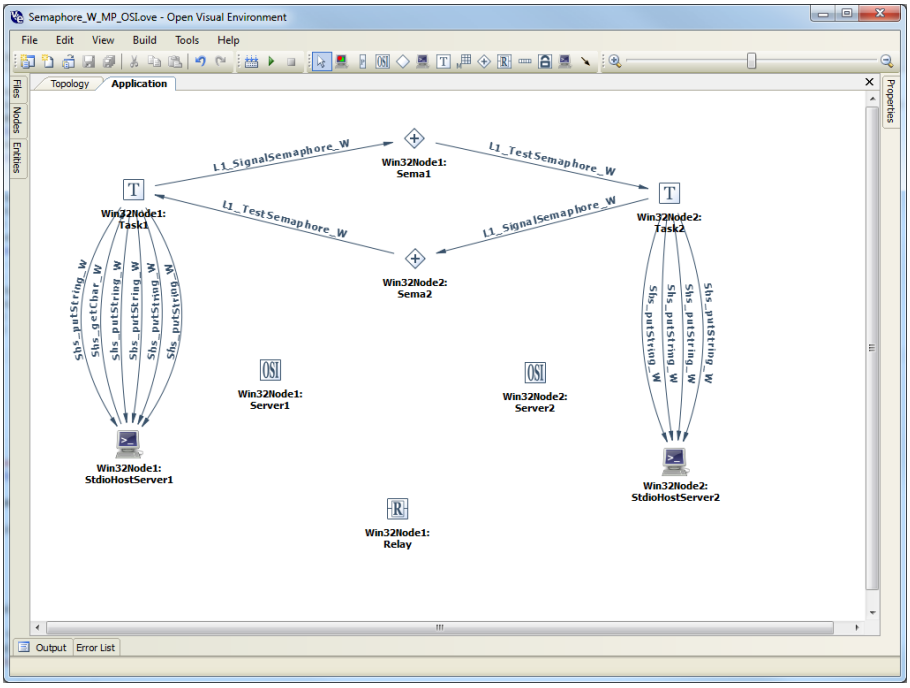


Figure 6.2: Application Diagram with Open System Inspector componenents

- node: Win32Node1
- name: Relay
- portNumber: 4040 (default)
- RelayStackSize: 170 (default)

Please note, that the OSIRelayComponent is at the moment only available for the Win32-Port of OpenComRTOS.

5. In oder to read out values from each Node it is necessary to add an OSIServerComponent on each Node. The OSIServerComponent acts as an endpoint with which the OpenSI-GUI can communicate. Please add the following OSIServerComponents to the system:

   (a) OSIServerComponent for Win32Node1:

   - node: Win32Node1
   - name: Server1
   - OSIStackSize: 512 (default)
   - OSICeilingPriority: 32 (default)

   (b) OSIServerComponent for Win32Node2

   - node: Win32Node2
   - name: Server2
   - OSIStackSize: 512 (default)
   - OSICeilingPriority: 32 (default)

   After this step your application diagram should look similar to the one shown in Figure 6.2.

6. Now build the project using either the build button in the toolbar, or by using the main-menu: Build → Build.

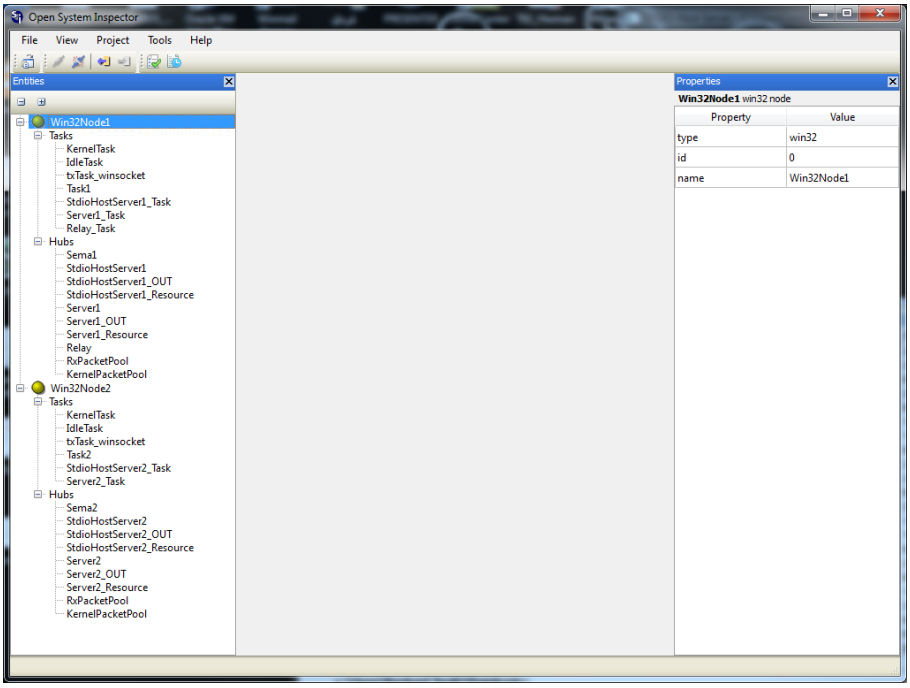


Figure 6.3: Open System Inspector Connected to the Project

7. Afer the build process completed successfully run the project, either by clicking on the run button in the toolbar or using the main-menu: Build → Run.

8. Start the OpenSI-GUI from the Start menu.

9. Open the project that has been created during this tutorial, by using the main-menu: File → Open Project. The file to open is the OpenVE-Project file (extension 'ove').

10. Now the OpenSI-GUI must connect to the project, for this use the main-menu: Project → Connect to Project. Afterwards you should see a window similar to the one shown in Figure 6.3.

11. To acquire information about the current state of any entity simply open its properties menu. For tasks there is also the possibility to suspend and resume their execution. These are dangrous operations, because suspending kernel, driver or Open System Inspector tasks can render the system unable to run. Keep in mind that the Open System Inspector is in practice nothing more than a normal user task. Figure 6.4 shows the current state of the entity Task1, which is part of the Semaphore-Loop, this information was acquired by using the properties menu entry: 'Get information about Entity'.

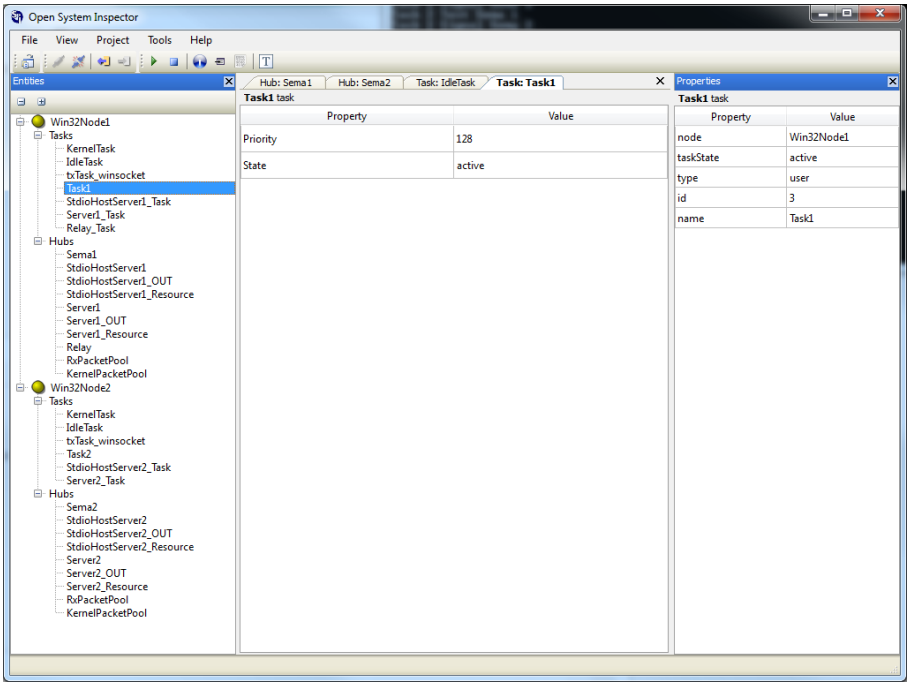


Figure 6.4: Open System Inspector Displaying the current state of Task1

Part IV

OpenComRTOS

Chapter 7

Module Index

7.1 Modules

Here is a list of all modules:

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

Chapter 9

Module Documentation

9.1 The OpenComRTOS Hub Concept

L1_Hub is a data structure representing a generic Hub.

The architecture defines the logical view of a Hub as one that has two waiting lists: the Get Request Waiting List and the Put Request Waiting List. From the design point of view there is no need to operate with two waiting lists for all L1 services. Port Hub, Event, Semaphore, Resource, FIFO, Memory Pool only have one that is used in the implementation as at any given point in time, there can either be only Get request(s), or only Put request(s) in the waiting lists or the waiting lists are empty.

Inserting or removing an element in the waiting list must be an atomic operation. This is guaranteed as only the Kernel Task performs operations on a Hub.

The Synchronization Predicate determines whether a Put or Get request synchronizes according to the State of the Hub and the content of the WaitingList(s). If synchronization can occur, the appropriate Synchronization Action is called to perform the synchronization and to update the state. These two functions are also called the Synchonisation Predicate of the Hub, and are dependent on the HubType.

The Synchronization Predicate is a function that takes as arguments:

- the Hub to retrieve the WaitingLists and the State
- the Packet, i.e. the L1 service request, that arrives in the Hub

The Synchronization Predicate returns:

- TRUE, if the newly arrived Packets can synchronize, The Synchronization Action is then called.
- FALSE otherwise. The Packet will then be inserted into the WaitingList.

When synchronization occurs, the Packet is returned to the Requesting Task. Note that upon an execution of a Synchronization Action, the Synchronization Predicate may need to be re-evaluated as the (Packet in the other) WaitingList could become enabled for synchronization.

9.2 Port Hub

Functions

- L1_Status L1_PutPacketToPort_W (L1_HubID port)

- L1_Status L1_GetPacketFromPort_W (L1_HubID port)
- L1_Status L1_PutPacketToPort_WT (L1_HubID port, L1_Timeout timeout)
- L1_Status L1_GetPacketFromPort_WT (L1_HubID port, L1_Timeout timeout)
- L1_Status L1_PutPacketToPort_NW (L1_HubID port)
- L1_Status L1_GetPacketFromPort_NW (L1_HubID port)

9.2.1 Detailed Description

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

9.2.2 Hub Description

The Port Hub, has the following properties, see also the section The OpenComRTOS Hub Concept :

- State: void
- Synchronisation Predicate upon L1_PutPacketToPort_{W,WT,NW}:
  - Predicate: GetWaitingList is not Empty -Action: Exchange Data between Put Packet and first waiting Get Packet, and return first waiting Get Packet or initiate the data transfer with the parameters specified in the packet
- Synchronisation Predicate upon L1_GetPacketFromPort_{W,WT,NW}:
  - Predicate: PutWaitingList is not Empty
  - Action: Exchange Data between Get and first waiting Put Packet, and return first waiting Put Packet or initiate the data transfer with the parameters specified in the packet
- Invariant(s): both waiting lists are empty, or only one waiting list contains waiting requests, i.e.
  - length (put waiting list) <> 0 implies length (get waiting list) = 0
  - length (get waiting list) <> 0 implies length (put waiting list) = 0

9.2.3 Example

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

9.2.3.1 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.

9.2.4 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);
        }
    }
}
```

9.2.5 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
      Port1: ");
            Shs_putChar_W(StdioHostServer1, ch);
            Shs_putChar_W(StdioHostServer1, '\n');
            L1_UnlockResource_W(StdioHostServer1Res);
        }else
        {
            Shs_putString_W(StdioHostServer1, "Error: Could not acquire a symbol
      from Port1, terminating. \n");
        }
    }
}
```

9.2.6 Function Documentation

9.2.6.1 L1_Status L1_GetPacketFromPort_NW (L1_HubID *port*)

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

port	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.

9.2.6.2 L1_Status L1_GetPacketFromPort_W (L1_HubID *port*)

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

port	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.

9.2.6.3 L1_Status L1_GetPacketFromPort_WT (L1_HubID *port,* L1_Timeout *timeout*)

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

port	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.

9.2.6.4 L1_Status L1_PutPacketToPort_NW (L1_HubID *port*)

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

port	of type L1_HubID, which identifies the Port.

Returns

L1_Status:

- 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

9.2.6.5 L1_Status L1_PutPacketToPort_W (L1_HubID *port*)

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

port	of type L1_HubID, which identifies the Port.

Returns

L1_Status:

- 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

9.2.6.6 L1_Status L1_PutPacketToPort_WT (L1_HubID *port,* L1_Timeout *timeout*)

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

port	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_Status:

- 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

9.3 Event Hub Operations

Functions

- L1_Status L1_RaiseEvent_W (L1_HubID event)
- L1_Status L1_TestEvent_W (L1_HubID event)
- L1_Status L1_RaiseEvent_WT (L1_HubID event, L1_Timeout timeout)
- L1_Status L1_TestEvent_WT (L1_HubID event, L1_Timeout timeout)
- L1_Status L1_RaiseEvent_NW (L1_HubID event)
- L1_Status L1_TestEvent_NW (L1_HubID event)

9.3.1 Detailed Description

The Event Hub, has the following properties, see also the section The OpenComRTOS Hub Concept :

L1_Event is a data structure representing a logical Event, which is a specific instantiation of an L1_Hub.

- State:
  - L1_BOOL: isSet (True or False)
- Synchronisation Action upon L1_RaiseEvent_{W,WT,NW}:
  - Predicate: isSet == False
  - Action: isSet := True, Packet := PutPacket inserted in waiting list
- Synchronisation Action upon L1_TestEvent_{W,WT,NW}:
  - Predicate: isSet == True
  - Action: isSet := False, GetPacket := Packet removed from waiting list

- Invariant(s):
  - both waiting lists are empty, or only one waiting list contains waiting requests, i.e.
    - length (put waiting list) <> 0 implies length (Get waiting list) = 0
    - length (get waiting list) <> 0 implies length (Put waiting list) = 0

- content of waiting lists are dependent on the current state, i.e.
  - isSet = True implies length (Get waiting list) = 0
  - isSet = False implies length (Put waiting list) = 0

- Notes: One example of a user defined Event could be to copy the data from the PutPacket. In this case the data must be copied into the Hub when the Event is raised. Note also that the Boolean condition can be any well formed logical expression that evaluates to true or false.

9.3.2 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.

9.3.2.1 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.

9.3.3 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_W(StdioHostServer1Res);
        }
    }
}
```

9.3.4 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_W(StdioHostServer1Res);
        }
    }
}
```

9.3.5 Function Documentation

9.3.5.1 L1_Status L1_RaiseEvent_NW (L1_HubID *event*)

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

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

Returns

L1_Status:

- 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

9.3.5.2 L1_Status L1_RaiseEvent_W (L1_HubID *event*)

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

Parameters:

Parameters

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

Returns

L1_Status:

- 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

9.3.5.3 L1_Status L1_RaiseEvent_WT (L1_HubID *event,* L1_Timeout *timeout*)

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

event	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_Status:

- 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

9.3.5.4 L1_Status L1_TestEvent_NW (L1_HubID *event*)

This service tests an Event. Returns immediately.

Parameters

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

Returns

L1_Status, 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

9.3.5.5 L1_Status L1_TestEvent_W (L1_HubID *event*)

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

Parameters

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

Returns

L1_Status, 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

9.3.5.6 L1_Status L1_TestEvent_WT (L1_HubID *event,* L1_Timeout *timeout*)

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

Parameters

event	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_Status, 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

9.4 Semaphore Hub Operations

Functions

- L1_Status L1_SignalSemaphore_W (L1_HubID semaphore)
- L1_Status L1_TestSemaphore_W (L1_HubID semaphore)
- L1_Status L1_SignalSemaphore_WT (L1_HubID semaphore, L1_Timeout timeout)
- L1_Status L1_TestSemaphore_WT (L1_HubID semaphore, L1_Timeout timeout)
- L1_Status L1_SignalSemaphore_NW (L1_HubID semaphore)
- L1_Status L1_TestSemaphore_NW (L1_HubID semaphore)

9.4.1 Detailed Description

L1_Semaphore is a data structure representing a Semaphore, which is a specific instantiation of a L1_Hub. The Semaphore Hub, has the following properties, see also the section The OpenComRTOS Hub Concept :

- State:
  - L1_UINT16 Count
- Synchronisation Predicate upon L1_SignalSemaphore_{W,WT,NW}:
  - Predicate: True, i.e. always succeeds (assuming Count $<$ MaxInt).
  - Action: Count := Count + 1
- Synchronisation Predicate upon L1_TestSemaphore_{W,WT,NW}:
  - Predicate: Count $>$ 0
  - Action: Count := Count -1
  - Note: when a predicate holds, the other predicate also has to be (re)evaluated. This evaluation and synchronization can be combined in the implementation.
- Invariant(s):
  - Put waiting list is empty.
  - count $<>$ 0 implies length (Get waiting list) = 0
- Notes: Because the Predicate always holds when signaling a semaphore, no data can be transferred via the PutPacket.

9.4.2 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.

9.4.2.1 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.

9.4.3 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");
        }
    }
}
```

9.4.4 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");
        }
    }
}
```

9.4.5 Function Documentation

9.4.5.1 L1_Status L1_SignalSemaphore_NW (L1_HubID *semaphore*)

Signals a semaphore, i.e. increases the semaphore count. This call returns immediately.

Parameters:

Parameters

semaphore	is the L1_HubID which identifies the Semaphore, that the calling Task wants to signal

Returns

L1_Status:

- 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

9.4.5.2 L1_Status L1_SignalSemaphore_W (L1_HubID *semaphore*)

Signals a semaphore, i.e. increases the semaphore count. This call waits until it could increment the Semaphore count.

Parameters:

Parameters

semaphore	the L1_HubID which identifies the Semaphore, that the calling Task wants to signal

Returns

L1_Status:

- 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

9.4.5.3 L1_Status L1_SignalSemaphore_WT (L1_HubID *semaphore,* L1_Timeout *timeout*)

Signals a semaphore, i.e. increases the semaphore count. This service waits until it either could increment the semaphore count or the timout expired.

Parameters:

Parameters

semaphore	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_Status:

- 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

9.4.5.4 L1_Status L1_TestSemaphore_NW (L1_HubID *semaphore*)

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

semaphore	identifies the Semaphore, that the calling Task wants to test.

Returns

L1_Status

- 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

9.4.5.5 L1_Status L1_TestSemaphore_W (L1_HubID *semaphore*)

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

semaphore	identifies the Semaphore-Hub, that the calling Task wants to test.

Returns

L1_Status

- 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

9.4.5.6 L1_Status L1_TestSemaphore_WT (L1_HubID *semaphore,* L1_Timeout *timeout*)

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

semaphore	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_Status, 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

9.5 Resource Hub Operations

Functions

- L1_Status L1_LockResource_W (L1_HubID resource)
- L1_Status L1_UnlockResource_W (L1_HubID resource)
- L1_Status L1_LockResource_WT (L1_HubID resource, L1_Timeout timeout)
- L1_Status L1_UnlockResource_WT (L1_HubID resource, L1_Timeout timeout)
- L1_Status L1_LockResource_NW (L1_HubID resource)
- L1_Status L1_UnlockResource_NW (L1_HubID resource)

9.5.1 Detailed Description

The Semaphore Hub, has the following properties, see also the section The OpenComRTOS Hub Concept. L1_Resource is a data structure representing a logical Resource, which is a specific instantiation of a L1_-Hub.

- State:
  - L1_Bool: Locked
  - L1_TaskID: OwningTask
  - L1_Priority: CeilingPriority
- Synchronisation Predicate upon L1_LockResource_{W,WT,NW}:
  - Precondition: Current Task != OwningTask
  - Predicate: not Locked:
    - Action: (Locked := TRUE), OwningTask := Packet->RequestingTaskID
  - Predicate: Locked:
    - Action: (Locked := TRUE), insert Request Packet in waiting list, apply priority inheritance if priority (OwningTask) > priority (Requesting Task)
- Synchronisation Predicate upon L1_UnlockResource_{W,WT,NW}
  - Precondition: Packet->RequestingTaskID == OwningTask
  - Predicate: Locked AND OwningTask == Packet->RequestingTaskID
  - Action: Locked := FALSE, OwningTask := None, apply LockResource Action on next waiting Task in waiting list.
- Invariant(s):
  - Locked = False implies length (get waiting list) = 0
- Notes:

- Locked attribute may be redundant in the implementation.
- It is an application design error if the Synchronization Predicate is not valid for the get waiting list (A task should only Unlock when it has Locked the resource)
- A task must not request to lock a resource it already has locked.
- When a predicate holds, the other predicate also has to be (re)evaluated. This evaluation and synchronization can be combined in the implementation.

9.5.2 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.

9.5.2.1 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 acess to the console.
- StdioHostServer1Res: A Resource Hub to ensure that a second task does not interfere with console access.

9.5.3 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_W(StdioHostServer1Res);
       }
}
```

9.5.4 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_W(StdioHostServer1Res);
       }
}
```

9.5.5 Function Documentation

9.5.5.1 L1_Status L1_LockResource_NW (L1_HubID *resource*)

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

Parameters

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

Returns

L1_Status 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

9.5.5.2 L1_Status L1_LockResource_W (L1_HubID *resource*)

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

Parameters

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

Returns

L1_Status 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

9.5.5.3 L1_Status L1_LockResource_WT (L1_HubID *resource*, L1_Timeout *timeout*)

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

Parameters

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

Returns

L1_Status

- 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

9.5.5.4 L1_Status L1_UnlockResource_NW (L1_HubID *resource*)

Unlocks a logical Resource. This service returns immediately, independent whether or not it could decrement the semaphore count.

Parameters

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

Returns

L1_Status

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

Precondition

- None

Postcondition

- Calling task ready

9.5.5.5 L1_Status L1_UnlockResource_W (L1_HubID *resource*)

Unlocks a logical Resource. This service waits until is could unlock the resource!

Parameters

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

Returns

L1_Status

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

Precondition

- None

Postcondition

- Calling task ready

9.5.5.6 L1_Status L1_UnlockResource_WT (L1_HubID *resource,* L1_Timeout *timeout*)

Unlocks a logical Resource. This service waits until is could either unlock the resource, or the timeout expired.

Parameters

resource	identifies the Resource-Hub, that the calling Task wants to unlock
timeout	the number of system ticks the call should wait while trying to enqueue the packet.

Returns

L1_Status

- RC_OK service successful (the resource was released)
- RC_FAIL service failed (the resource could not be unlocked)
- RC_TO the timeout expired.

Precondition

- None

Postcondition

- Calling task ready

9.6 FIFO Hub Operations

Functions

- L1_Status L1_EnqueueFifo_W (L1_HubID fifo)
- L1_Status L1_DequeueFifo_W (L1_HubID fifo)
- L1_Status L1_EnqueueFifo_WT (L1_HubID fifo, L1_Timeout timeout)
- L1_Status L1_DequeueFifo_WT (L1_HubID fifo, L1_Timeout timeout)
- L1_Status L1_EnqueueFifo_NW (L1_HubID fifo)
- L1_Status L1_DequeueFifo_NW (L1_HubID fifo)

9.6.1 Detailed Description

The Resource Hub, has the following properties, see also the section The OpenComRTOS Hub Concept. L1_FIFO is a data structure representing a L1_FIFO buffer, which is a specific instantiation of a L1_Hub.

- State:
  - const L1_UINT16: Size // number of fixed size data blocks in FIFO
  - L1_UINT16: Count
  - L1_List: Buffer
- Synchronisation Predicate upon L1_EnqueueFIFO:
  - Predicate: Count < Size
  - Action: Count := Count + 1, retrieve Data from Packet, and insert Data at end of List/Buffer
- Synchronisation Predicate upon L1_DequeueFIFO:
  - Predicate: ount <> 0
  - Action: Count := Count – 1, retrieve Data from (first element) of List/Buffer and exchange with Packet, rerun L1_EnqueueFIFO I any waiting Task in the Put Waiting List
- Invariant(s):
  - both waiting lists are empty, or only one waiting list contains waiting requests, i.e.
    - length (put waiting list) <> 0 implies length (get waiting list) = 0
    - length (get waiting list) <> 0 implies length (put waiting list) = 0
  - content of waiting lists are dependent on the current state, i.e.
    - Count <> 0 implies length (get waiting list) = 0
    - length (Put waiting list) <> 0 implies Count == Size
    - Count <> Size implies length (put waiting list) = 0

9.6.2 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.

9.6.2.1 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 acess to the console.
- StdioHostServer1Res: A Resource Hub to ensure that a second task does not interfere with console access.

9.6.3 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_W(StdioHostServer1Res);
            }else
            {
                Shs_putString_W(StdioHostServer1,
                                "A symbol is not put by Task1 into FIFO1\n");
            }
        }
    }
}
```

9.6.4 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 F
      IFO to fill up\n\n");
        L1_UnlockResource_W(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_W(StdioHostServer1Res);
            }else
            {
                L1_LockResource_W(StdioHostServer1Res);
                Shs_putString_W(StdioHostServer1,
                                "A symbol is not read by Task2 from FIFO1\n");
                L1_UnlockResource_W(StdioHostServer1Res);
            }
        }
    }
}
```

9.6.5 Function Documentation

9.6.5.1 L1_Status L1_DequeueFifo_NW (L1_HubID *fifo*)

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

fifo	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_Status

- 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

9.6.5.2 L1_Status L1_DequeueFifo_W (L1_HubID *fifo*)

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

fifo	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_Status

- 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

9.6.5.3 L1_Status L1_DequeueFifo_WT (L1_HubID *fifo,* L1_Timeout *timeout*)

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

fifo	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_Status

- 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

9.6.5.4 L1_Status L1_EnqueueFifo_NW (L1_HubID *fifo*)

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

fifo	the L1_HubID which identifies the FIFO-Hub.

Returns

L1_Status

- 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

9.6.5.5 L1_Status L1_EnqueueFifo_W (L1_HubID *fifo*)

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

fifo	identifies the FIFO-Hub to use.

Returns

L1_Status

- 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

9.6.5.6 L1_Status L1_EnqueueFifo_WT (L1_HubID *fifo,* L1_Timeout *timeout*)

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

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

Returns

L1_Status

- 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

9.7 Memory Pool Hub Operations

Functions

- L1_Status L1_AllocateMemoryBlock_W (L1_HubID memoryPool, L1_BYTE ∗∗memoryBlock, L1_-UINT16 size)
- L1_Status L1_DeallocateMemoryBlock_W (L1_HubID memoryPool, void ∗memoryBlock)
- L1_Status L1_AllocateMemoryBlock_WT (L1_HubID memoryPool, void ∗∗memoryBlock, L1_-UINT16 size, L1_Timeout timeout)
- L1_Status L1_AllocateMemoryBlock_NW (L1_HubID memoryPool, void ∗∗memoryBlock, L1_-UINT16 size)

9.7.1 Detailed Description

The Resource Hub, has the following properties, see also the section The OpenComRTOS Hub Concept.

L1_MemoryPool is a data structure representing a list of memory Resources, managed by a specific instantiation of a L1_Hub.

- State of every memory block in the pool:
  - Bool: Locked
  - L1_UINT16: size > 0 AND ((2∗∗N-1) < size < (2∗∗N))
  - TaskID: OwningTask
  - L1_Priority: CeilingPriority // not implemented
- Synchronisation Predicate upon L1_Allocate_MemoryBlock:
  - Predicate: not Locked AND size available blocks >= requested size AND size available blocks==2∗∗N
    - Action: Locked := TRUE, OwningTask := Packet->RequestingTaskID
  - Predicate: Locked OR size available blocks < requested size OR size available blocks /= 2∗∗N
    - Action: insert Request Packet in waiting list, (priority inheritance is not used)
- Synchronisation Predicate upon L1_Deallocate_MemoryBlock:
  - Predicate: Locked AND OwningTask == Packet->RequestingTaskID

  - Action: Locked := FALSE, OwningTask := None, apply LockResource Action on next waiting Tasks in waiting list. (whole list myst be checked until the Predicate holds)

- Invariant(s):
  - If a task is waiting and there is a free block of large enough size, the waiting task will get a block allocated

- Notes:
  - Locked attribute may be redundant in the implementation.
  - A task can request to lock another memory block while holding a block
  - The current release implements the memory pool as list of equally sized blocks, defined at design time.

9.7.2 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.

9.7.2.1 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.

9.7.2.2 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_W(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");
}
```

9.7.3 Function Documentation

9.7.3.1 L1_Status L1_AllocateMemoryBlock_NW (L1_HubID *memoryPool*, void ** *memoryBlock*, L1_UINT16 *size*)

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

L1_Status

- 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.

9.7.3.2 L1_Status L1_AllocateMemoryBlock_W (L1_HubID *memoryPool*, L1_BYTE ** *memoryBlock*, L1_UINT16 *size*)

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

L1_Status

- 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.

9.7.3.3 L1_Status L1_AllocateMemoryBlock_WT (L1_HubID *memoryPool,* void ** *memoryBlock,* L1_UINT16 *size,* L1_Timeout *timeout*)

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

L1_Status

- 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.

9.7.3.4 L1_Status L1_DeallocateMemoryBlock_W (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

L1_Status:

- 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

9.8 Task Management Operations

Functions

- L1_Status L1_StartTask_W (L1_PortID task)
- L1_Status L1_StopTask_W (L1_PortID task)
- L1_Status L1_SuspendTask_W (L1_PortID task)
- L1_Status L1_ResumeTask_W (L1_PortID task)
- L1_Status L1_WaitTask_WT (L1_Timeout timeout)

9.8.1 Detailed Description

OpenComRTOS offers the following operations to manage Tasks.

9.8.2 Function Documentation

9.8.2.1 L1_Status L1_ResumeTask_W (L1_PortID *task*)

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

Parameters

task	ID of the Task to be resumed.

Returns

L1_Status:

- 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.

9.8.2.2 L1_Status L1_StartTask_W (L1_PortID *task*)

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_Status:

- 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)

9.8.2.3 L1_Status L1_StopTask_W (L1_PortID *task*)

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_Status:

- 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.

9.8.2.4 L1_Status L1_SuspendTask_W (L1_PortID *task*)

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_Status:

- 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.

9.8.2.5 L1_Status L1_WaitTask_WT (L1_Timeout *timeout*)

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_Status:

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

Precondition

- None

Postcondition

- Calling task ready.

9.9 Base Variable types

Typedefs

- typedef unsigned char L1_BYTE

- typedef int L1_INT32
- typedef short int L1_INT16
- typedef unsigned int L1_UINT32
- typedef unsigned short L1_UINT16
- typedef L1_BYTE L1_BOOL

Variables

- const L1_BYTE L1_BYTE_MIN = 0x0
- const L1_BYTE L1_BYTE_MAX = 0xFF
- const L1_UINT16 L1_UINT16_MIN = 0x0
- const L1_UINT16 L1_UINT16_MAX = 0xFFFF
- const L1_UINT32 L1_UINT32_MIN = 0x0
- const L1_UINT32 L1_UINT32_MAX = 0xFFFFFFFF
- const L1_INT16 L1_INT16_MIN = (-32768)
- const L1_INT16 L1_INT16_MAX = 32767
- const L1_INT32 L1_INT32_MIN = (-2147483647 - 1)
- const L1_INT32 L1_INT32_MAX = 2147483647

9.9.1 Typedef Documentation

9.9.1.1 typedef L1_BYTE L1_BOOL

L1_BOOL is a basic integer type sufficient to represent the values: L1_TRUE and L1_FALSE. (size depends on target)

9.9.1.2 typedef unsigned char L1_BYTE

L1_BYTE is a 8-bit unsigned integer type.

See also

L1_BYTE_MIN
L1_BYTE_MAX

9.9.1.3 typedef short int L1_INT16

L1_INT16 is a 16-bit signed integer type.

See also

L1_INT16_MIN
L1_INT16_MAX

9.9.1.4 typedef int L1_INT32

INT32 is a 32-bit signed integer type.

See also

L1_INT32_MIN
L1_INT32_MAX

9.9.1.5 typedef unsigned short L1_UINT16

UINT16 is a 16-bit unsigned integer type.

See also

L1_UINT16_MIN
L1_UINT16_MAX

9.9.1.6 typedef unsigned int L1_UINT32

UINT32 is a 32-bit unsigned integer type.

See also

L1_UINT32_MIN
L1_UINT32_MAX

9.9.2 Variable Documentation

9.9.2.1 const L1_BYTE L1_BYTE_MAX = 0xFF

The maximal value of a L1_BYTE variable.

9.9.2.2 const L1_BYTE L1_BYTE_MIN = 0x0

The minimal value of a L1_BYTE variable.

9.9.2.3 const L1_INT16 L1_INT16_MAX = 32767

The maximal value of a L1_INT16 variable.

9.9.2.4 const L1_INT16 L1_INT16_MIN = (-32768)

The minimal value of a L1_INT16 variable.

9.9.2.5 const L1_INT32 L1_INT32_MAX = 2147483647

The maximal value of a L1_INT32 variable.

9.9.2.6 const L1_INT32 L1_INT32_MIN = (-2147483647 - 1)

The minimal value of a L1_INT32 variable.

9.9.2.7 const L1_UINT16 L1_UINT16_MAX = 0xFFFF

The maximal value of a L1_UINT16 variable.

9.9.2.8 const L1_UINT16 L1_UINT16_MIN = 0x0

The minimal value of a L1_UINT16 variable.

9.9.2.9 const L1_UINT32 L1_UINT32_MAX = 0xFFFFFFFF

The maximal value of a L1_UINT32 variable.

9.9.2.10 const L1_UINT32 L1_UINT32_MIN = 0x0

The minimal value of a L1_UINT32 variable.

9.10 Types related to Timer Handling

Typedefs

- typedef L1_UINT32 L1_Timeout
- typedef L1_UINT32 L1_Time

Variables

- const L1_UINT32 L1_Time_MIN = 0x0
- const L1_UINT32 L1_Time_MAX = 0xFFFFFFFF

9.10.1 Typedef Documentation

9.10.1.1 typedef L1_UINT32 L1_Time

This datatype is used to represent the number of expired ticks.

See also

L1_Time_MIN
L1_Time_MAX

9.10.1.2 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

9.10.2 Variable Documentation

9.10.2.1 const L1_UINT32 L1_Time_MAX = 0xFFFFFFFF

The maximal value of a L1_Time variable.

9.10.2.2 const L1_UINT32 L1_Time_MIN = 0x0

The minimal value of a L1_Time variable.

Chapter 10

File Documentation

10.1 include/L1_api_apidoc.h File Reference

```
#include <L1_types.h>
#include <kernel/L1_port_api.h>
#include <kernel/L1_packet_api.h>
#include <kernel/L1_task_api.h>
#include <kernel/L1_kernel_types.h>
#include <kernel/L1_kernel_api.h>
#include <L1_hal_asm.h>
#include <kernel/L1_hub_api.h>
```

Defines

- #define OCR_VERSION 0x01040303
- #define theServicePacket (L1_CurrentTaskCR->RequestPacket)

Functions

- L1_UINT32 L1_GetVersion (void)
- int L1_runOpenComRTOS (L1_UINT32 NodeNumberOfTasks, L1_UINT32 NodeNumberOfHubs)
- L1_Status L1_PutPacketToPort_W (L1_HubID port)
- L1_Status L1_GetPacketFromPort_W (L1_HubID port)
- L1_Status L1_PutPacketToPort_WT (L1_HubID port, L1_Timeout timeout)
- L1_Status L1_GetPacketFromPort_WT (L1_HubID port, L1_Timeout timeout)
- L1_Status L1_PutPacketToPort_NW (L1_HubID port)
- L1_Status L1_GetPacketFromPort_NW (L1_HubID port)
- L1_Status L1_RaiseEvent_W (L1_HubID event)
- L1_Status L1_TestEvent_W (L1_HubID event)
- L1_Status L1_RaiseEvent_WT (L1_HubID event, L1_Timeout timeout)
- L1_Status L1_TestEvent_WT (L1_HubID event, L1_Timeout timeout)
- L1_Status L1_RaiseEvent_NW (L1_HubID event)

- L1_Status L1_TestEvent_NW (L1_HubID event)
- L1_Status L1_SignalSemaphore_W (L1_HubID semaphore)
- L1_Status L1_TestSemaphore_W (L1_HubID semaphore)
- L1_Status L1_SignalSemaphore_WT (L1_HubID semaphore, L1_Timeout timeout)
- L1_Status L1_TestSemaphore_WT (L1_HubID semaphore, L1_Timeout timeout)
- L1_Status L1_SignalSemaphore_NW (L1_HubID semaphore)
- L1_Status L1_TestSemaphore_NW (L1_HubID semaphore)
- L1_Status L1_LockResource_W (L1_HubID resource)
- L1_Status L1_UnlockResource_W (L1_HubID resource)
- L1_Status L1_LockResource_WT (L1_HubID resource, L1_Timeout timeout)
- L1_Status L1_UnlockResource_WT (L1_HubID resource, L1_Timeout timeout)
- L1_Status L1_LockResource_NW (L1_HubID resource)
- L1_Status L1_UnlockResource_NW (L1_HubID resource)
- L1_Status L1_EnqueueFifo_W (L1_HubID fifo)
- L1_Status L1_DequeueFifo_W (L1_HubID fifo)
- L1_Status L1_EnqueueFifo_WT (L1_HubID fifo, L1_Timeout timeout)
- L1_Status L1_DequeueFifo_WT (L1_HubID fifo, L1_Timeout timeout)
- L1_Status L1_EnqueueFifo_NW (L1_HubID fifo)
- L1_Status L1_DequeueFifo_NW (L1_HubID fifo)
- L1_Status L1_AllocateMemoryBlock_W (L1_HubID memoryPool, L1_BYTE ∗∗memoryBlock, L1_-UINT16 size)
- L1_Status L1_DeallocateMemoryBlock_W (L1_HubID memoryPool, void ∗memoryBlock)
- L1_Status L1_AllocateMemoryBlock_WT (L1_HubID memoryPool, void ∗∗memoryBlock, L1_-UINT16 size, L1_Timeout timeout)
- L1_Status L1_AllocateMemoryBlock_NW (L1_HubID memoryPool, void ∗∗memoryBlock, L1_-UINT16 size)
- L1_Status L1_StartTask_W (L1_PortID task)
- L1_Status L1_StopTask_W (L1_PortID task)
- L1_Status L1_SuspendTask_W (L1_PortID task)
- L1_Status L1_ResumeTask_W (L1_PortID task)
- L1_Status L1_WaitTask_WT (L1_Timeout timeout)

10.1.1 Define Documentation

10.1.1.1 #define OCR_VERSION 0x01040303

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

This number is loosely associated with the OpenVE version number.

10.1.1.2 #define theServicePacket (L1_CurrentTaskCR->RequestPacket)

This is the pointer to the preallocated service packet of the current task.

10.1.2 Function Documentation

10.1.2.1 L1_UINT32 L1_GetVersion (void)

This service returns the Kernel version of OpenComRTOS.

Returns

L1_UINT32, the value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

10.1.2.2 int L1_runOpenComRTOS (L1_UINT32 *NodeNumberOfTasks*, L1_UINT32 *NodeNumberOfHubs*)

This function starts the execution of the OpenComRTOS kernel.

Parameters

NodeNumberOfTasks	is the number of tasks on the given node.
NodeNumberOfHubs	the number of Hubs on this node.

Returns

This function does not return.

10.2 include/L1_types_apidoc.h File Reference

```
#include <L1_hal_types.h>
```

Defines

- #define L1_FALSE 0U
- #define L1_TRUE 1U
- #define L1_GLOBALID_SIZE 32
- #define L1_GLOBALID_MASK 0xFFFFFF00U

Typedefs

- typedef unsigned char L1_BYTE
- typedef int L1_INT32
- typedef short int L1_INT16
- typedef unsigned int L1_UINT32
- typedef unsigned short L1_UINT16
- typedef L1_BYTE L1_BOOL
- typedef L1_BYTE L1_Priority
- typedef void * L1_TaskArguments
- typedef L1_UINT32 L1_Timeout
- typedef L1_UINT32 L1_Time
- typedef void(* L1_TaskFunction)(L1_TaskArguments Arguments)
- typedef L1_UINT32 L1_TaskID
- typedef L1_UINT32 L1_PortID
- typedef L1_UINT32 L1_HubID

Enumerations

- enum L1_ServiceID {

  L1_SID_START_TASK, L1_SID_SUSPEND_TASK, L1_SID_RESUME_TASK, L1_SID_STOP_-TASK,

  L1_SID_WAIT_TASK, L1_SID_AWAKE_TASK, L1_SID_RETURN, L1_SID_ANY_PACKET,

  L1_SID_SEND_TO_HUB, L1_SID_RECEIVE_FROM_HUB, L1_SID_IOCTL_HUB, L1_CHANGE_-PRIORITY,

  L1_SID_CHANGE_PACKET_PRIORITY }
- enum L1_ServiceType {

  L1_SERVICE, L1_EVENT, L1_SEMAPHORE, L1_PORT,

  L1_RESOURCE, L1_FIFO, L1_PACKETPOOL, L1_MEMORYPOOL }
- enum L1_HubControlType { L1_IOCTL_HUB_OPEN }
- enum L1_Status { RC_OK, RC_FAIL, RC_TO }

Variables

- const L1_BYTE L1_BYTE_MIN
- const L1_BYTE L1_BYTE_MAX
- const L1_INT32 L1_INT32_MIN
- const L1_INT32 L1_INT32_MAX
- const L1_INT16 L1_INT16_MIN
- const L1_INT16 L1_INT16_MAX
- const L1_UINT32 L1_UINT32_MIN
- const L1_UINT32 L1_UINT32_MAX
- const L1_UINT16 L1_UINT16_MIN
- const L1_UINT16 L1_UINT16_MAX
- const L1_UINT32 L1_Time_MIN
- const L1_UINT32 L1_Time_MAX

10.2.1 Define Documentation

10.2.1.1 #define L1_FALSE 0U

10.2.1.2 #define L1_GLOBALID_MASK 0xFFFFFF00U

10.2.1.3 #define L1_GLOBALID_SIZE 32

10.2.1.4 #define L1_TRUE 1U

10.2.2 Typedef Documentation

10.2.2.1 typedef L1_UINT32 L1_HubID

L1_HubID is a type that represents an identifier of a Hub on a Node. L1_HubID is a system wide identifier represented by a 32 bit datastructure divided in the following 8bit fields: LocalHubID, NodeID, SiteID, ClusterID.

10.2.2.2 typedef L1_UINT32 L1_PortID

L1_PortID is a type that represents an Task Input Port identifier of a Task. L1_PortID is a system wide identifier represented by a 32 bit data structure divided in the following 8bit fields: LocalTaskID, NodeID, SiteID, ClusterID.

10.2.2.3 typedef L1_BYTE L1_Priority

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.

10.2.2.4 typedef void* L1_TaskArguments

Argument to a Task Entry Point.

10.2.2.5 typedef void(* L1_TaskFunction)(L1_TaskArguments Arguments)

L1_TaskFunction is a pointer to a function with one input parameter of type L1_TaskArguments. The function, pointed to by L1_TaskFunction is used as an entry point to start a Task.

10.2.2.6 typedef L1_UINT32 L1_TaskID

L1_EntityAddress is an abstract type that represents an identifier of an Entity. L1_EntityAddress is a system wide address represented by a 32 bit data structure with the following 8bit fields: LocalEntityID, NodeID, SiteID, ClusterID. In practice at L1 we will only find EntityAdresses for Tasks en HubID and the context will allow to distinguish between them. In this context we call them L1_TaskID and L1_-HubID. L1_ TaskID is a type that represents an identifier of a Task .L1_TaskID is a system wide identifier represented by a 32 bit data structure divided in the following 8bit fields: LocalTaskID, NodeID, SiteID, ClusterID.

10.2.3 Enumeration Type Documentation

10.2.3.1 enum L1_HubControlType

This enumeartion lists all IO Control Messages that can be sent to a Hub.

Enumerator:

L1_IOCTL_HUB_OPEN IO Control Message to initialise a Hub. This is used by the Kernel when initialising the Hubs.

10.2.3.2 enum L1_ServiceID

This enumerates the different service identifiers available in OpenComRTOS.

Enumerator:

L1_SID_START_TASK Service identifier for starting a task

L1_SID_SUSPEND_TASK Service identifier for suspension of a task

L1_SID_RESUME_TASK Service identifier for resumption of a task

L1_SID_STOP_TASK Service identifier for stopping a task

L1_SID_WAIT_TASK Service identifier for putting a task temporarily in waiting state (timeout).

L1_SID_AWAKE_TASK Service identifier for notifying the kernel of a timer expiry, i.e. timeout event (typically only used by timer HW ISR).

L1_SID_RETURN Service identifier for returning from a service request.

L1_SID_ANY_PACKET Service identifier for receiving any packet (only used for WaitForPacket).

L1_SID_SEND_TO_HUB Service identifier for sending an L1_Packet to a Hub (also referred to as putting). The type of Hub gets encoded according to the enumeration L1_ServiceType.

See also

L1_ServiceType.

L1_SID_RECEIVE_FROM_HUB Service identifier for receiving an L1_Packet from a Hub (also referred to as getting). The type of Hub gets encoded according to the enumeration L1_ServiceType.

See also

L1_ServiceType.

L1_SID_IOCTL_HUB Service identifier to control the Hub.

See also

L1_HubControlType

L1_CHANGE_PRIORITY Service identifier to boost the priority of a Task. Used to boost the priority of a Task in distributed systems.

Warning

This is only used by the Kernel iteself, it is not meant to be used by applications.

L1_SID_CHANGE_PACKET_PRIORITY Service identifier to boost the priority of a Packet. Used to boost the priority of the Request Packet of a Task in distributed systems.

Warning

This is only used by the Kernel iteself, it is not meant to be used by applications.

10.2.3.3 enum L1_ServiceType

L1_ServiceType is an enumeration type used to identify the L1-Services, provided by the Kernel.

Enumerator:

L1_SERVICE Service identifier for a generic service.

L1_EVENT Service identifier for an event.

L1_SEMAPHORE Service identifier for a semaphore.

L1_PORT Service identifier for a port.

L1_RESOURCE Service identifier for a resource.

L1_FIFO Service identifier for a FIFO buffer.

L1_PACKETPOOL Service identifier for a packet pool.

L1_MEMORYPOOL Service identifier for a memory pool.

10.2.3.4 enum L1_Status

L1_Status is an enumeration type used to specify the result of a service request (success, failure, etc.).

Enumerator:

RC_OK Return code for a successful request

RC_FAIL Return code for a failed request

RC_TO Return code for a failed request after the timeout expired.

10.3 src/kernel/L1_types.c File Reference

```
#include <L1_types.h>
```

Variables

- const L1_BYTE L1_BYTE_MIN = 0x0
- const L1_BYTE L1_BYTE_MAX = 0xFF
- const L1_UINT16 L1_UINT16_MIN = 0x0
- const L1_UINT16 L1_UINT16_MAX = 0xFFFF
- const L1_UINT32 L1_UINT32_MIN = 0x0
- const L1_UINT32 L1_UINT32_MAX = 0xFFFFFFFF
- const L1_INT16 L1_INT16_MIN = (-32768)
- const L1_INT16 L1_INT16_MAX = 32767
- const L1_INT32 L1_INT32_MIN = (-2147483647 - 1)
- const L1_INT32 L1_INT32_MAX = 2147483647
- const L1_UINT32 L1_Time_MIN = 0x0
- const L1_UINT32 L1_Time_MAX = 0xFFFFFFFF

Part V

Stdio Host Service

Chapter 11

File Index

11.1 File List

Here is a list of all files with brief descriptions:

Chapter 12

File Documentation

12.1 src/include/StdioHostService/StdioHostClient.h File Reference

```
#include <kernel/L1_trace_api.h>
#include <StdioHostService/TraceHostClient.h>
#include <L1_types.h>
```

Defines

- #define SHS_VERSION 0x01000304
- #define ShsGetVersion() ((L1_UINT32) SHS_VERSION)

Functions

- L1_Status Shs_putChar_W (L1_HubID shs, L1_BYTE charValue)
- L1_Status Shs_getChar_W (L1_HubID shs, L1_BYTE *pChar)
- L1_Status Shs_putInt_W (L1_HubID shs, L1_INT32 intValue, L1_BYTE format)
- L1_Status Shs_getInt_W (L1_HubID shs, L1_INT32 *pInt)
- L1_Status Shs_putFloat_W (L1_HubID shs, float floatValue, L1_BYTE prec)
- L1_Status Shs_getFloat_W (L1_HubID shs, float *pFloat)
- L1_Status Shs_putString_W (L1_HubID shs, const char *str)
- L1_Status Shs_getString_W (L1_HubID shs, L1_UINT32 maxLength, char *pStr, L1_UINT32 *pRealLength)
- L1_Status Shs_openFile_W (L1_HubID shs, const char *fileName, const char *mode, L1_UINT32 *fileHandle)
- L1_Status Shs_closeFile_W (L1_HubID shs, L1_UINT32 fileHandle)
- L1_Status Shs_writeToFile_W (L1_HubID shs, L1_UINT32 fileHandle, L1_BYTE *buffer, L1_UINT32 toWrite, L1_UINT32 *pWritten)
- L1_Status Shs_readFromFile_W (L1_HubID shs, L1_UINT32 fileHandle, L1_BYTE *buffer, L1_UINT32 toRead, L1_UINT32 *pRead)

12.1.1 Define Documentation

12.1.1.1 #define SHS_VERSION 0x01000304

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

This number is loosly associated with the OpenVE version number.

12.1.1.2 #define ShsGetVersion( ) ((L1_UINT32) SHS_VERSION)

This service returns the Kernel version of OpenComRTOS.

Returns

L1_UINT32, the value of is formatted the following way:
- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

12.1.2 Function Documentation

12.1.2.1 L1_Status Shs_closeFile_W (L1_HubID *shs,* L1_UINT32 *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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.2 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.3 L1_Status Shs_getFloat_W (L1_HubID *shs*, float * *pFloat*)

Retrieves a float value from the Stdio Host Server console. The retrieved float value is returned to the user as a pointer.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
pFloat	is the pointer to a variable of the float type which holds the retrieved float value.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.4 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.5 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.6 L1_Status Shs_openFile_W (L1_HubID *shs*, const char * *fileName*, const char * *mode*, L1_UINT32 * *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_Status

- RC_OK: The request was successful.
- RC_FAIL: The request failed.

12.1.2.7 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.8 L1_Status Shs_putFloat_W (L1_HubID *shs*, float *floatValue*, L1_BYTE *prec*)

This function outputs a float value (floatValue) into the console associated with the Stdio Host Server. The precision of output of a float value must be specified using the prec parameter.

Parameters

shs	is the ID of the ShsInputPort of the Stdio Host Server.
floatValue	is the float value to output into the console.
prec	is the character specifying the precision of output of a float value onto the console.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.9 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.10 L1_Status 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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.11 L1_Status Shs_readFromFile_W (L1_HubID *shs*, L1_UINT32 *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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.1.2.12 L1_Status Shs_writeToFile_W (L1_HubID *shs*, L1_UINT32 *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_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

12.2 src/include/StdioHostService/TraceHostClient.h File Reference

```
#include <L1_types.h>
```

Functions

- L1_Status DumpTraceBuffer_W (L1_HubID ServerInputPort)

12.2.1 Function Documentation

12.2.1.1 L1_Status 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_Status:

- RC_OK: Dumping the trace buffer was completed successfully.
- RC_FAIL: Operation failed.

Part VI

Graphical Host Service

Chapter 13

Data Structure Index

13.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 14

File Index

14.1 File List

Here is a list of all files with brief descriptions:

Chapter 15

Data Structure Documentation

15.1 GhsBrush Struct Reference

`#include <GhsTypes.h>`

Data Fields

- GhsColour colour
- GhsBrushStyle style

15.1.1 Detailed Description

Defines the type describing a Brush as used by the Graphical Host Service.

15.1.2 Field Documentation

15.1.2.1 GhsColour colour

15.1.2.2 GhsBrushStyle style

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

15.2 GhsColour Struct Reference

`#include <GhsTypes.h>`

Data Fields

- L1_BYTE r
- L1_BYTE g
- L1_BYTE b

15.2.1 Detailed Description

Defines how Colours are represented in the Graphical Host Server Data structures.

15.2.2 Field Documentation

15.2.2.1 L1_BYTE b

Blue component

15.2.2.2 L1_BYTE g

Green component

15.2.2.3 L1_BYTE r

Red component

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

15.3 GhsPen Struct Reference

`#include <GhsTypes.h>`

Data Fields

- GhsColour colour
- L1_UINT32 lineWidth
- GhsPenStyle style

15.3.1 Detailed Description

Defines the type describing a Pen as used by the Graphical Host Service.

15.3.2 Field Documentation

15.3.2.1 GhsColour colour

15.3.2.2 L1_UINT32 lineWidth

15.3.2.3 GhsPenStyle style

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

15.4 GhsRect Struct Reference

```
#include <GhsTypes.h>
```

Data Fields

- L1_UINT32 left
- L1_UINT32 top
- L1_UINT32 right
- L1_UINT32 bottom

15.4.1 Detailed Description

This structure represents a rectangle.

15.4.2 Field Documentation

15.4.2.1 L1_UINT32 bottom

15.4.2.2 L1_UINT32 left

15.4.2.3 L1_UINT32 right

15.4.2.4 L1_UINT32 top

The documentation for this struct was generated from the following file:

- src/include/GraphicalHostService/GhsTypes.h

Chapter 16

File Documentation

16.1 src/include/GraphicalHostService/GhsTypes.h File Reference

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

Enumerations

- enum GhsBrushStyle { GhsBrushSolid = 1, GhsBrushDiagonal }
- enum GhsPenStyle { GhsPenSolid = 1 }

16.1.1 Enumeration Type Documentation

16.1.1.1 enum GhsBrushStyle

Defines the different styles a brush can have.

Enumerator:

GhsBrushSolid

GhsBrushDiagonal Not Implemented yet.

16.1.1.2 enum GhsPenStyle

Defines the different styles a pen can have.

Enumerator:

GhsPenSolid

16.2 src/include/GraphicalHostService/GraphicalHostClient.h File Reference

```
#include <GraphicalHostService/GhsTypes.h>
```

Functions

- L1_Status Ghs_openSession_W (L1_HubID ghsInputPort, L1_UINT32 ∗pSessionID)
- L1_Status Ghs_closeSession_W (L1_HubID ghsInputPort, L1_UINT32 sessionId)
- L1_Status Ghs_getServerVersion_W (L1_HubID ghsInputPort, L1_UINT32 ∗pServerVersion)
- L1_Status Ghs_setPen_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, GhsPenStyle penStyle, L1_BYTE lineWidth, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_Status Ghs_setBrush_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, GhsBrushStyle brushStyle, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_Status Ghs_drawLine_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x1, L1_UINT32 y1, L1_UINT32 x2, L1_UINT32 y2)
- L1_Status Ghs_drawRect_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x1, L1_UINT32 y1, L1_UINT32 x2, L1_UINT32 y2)
- L1_Status Ghs_drawCircle_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT32 x, L1_UINT32 y, L1_UINT32 r)
- L1_Status Ghs_drawText_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_UINT16 x, L1_UINT16 y, char ∗text)
- L1_Status Ghs_setTextColour_W (L1_HubID ghsInputPort, L1_UINT32 ghsSession, L1_BYTE r, L1_BYTE g, L1_BYTE b)
- L1_Status Ghs_setCanvasSize_W (L1_HubID ghsInputPort, L1_UINT32 width, L1_UINT32 height)
- L1_Status Ghs_getCanvasSize_W (L1_HubID ghsInputPort, L1_UINT32 ∗width, L1_UINT32 ∗height)

16.2.1 Function Documentation

16.2.1.1 L1_Status 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_Status

- RC_OK the session could be closed.
- RC_FAIL the session could not be closed.

16.2.1.2 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.3 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.4 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.5 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.6 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.7 L1_Status 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_Status

- RC_OK, *pVersion contains the version number of the server.
- RC_FAIL, operation failed, *pVersion is set to zero.

16.2.1.8 L1_Status 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_Status

- 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().

16.2.1.9 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.10 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.11 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.2.1.12 L1_Status 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_Status

- RC_OK the request was successful.
- RC_FAIL the request was not successful.

16.3 src/include/GraphicalHostService/GraphicalHostService.h File Reference

```
#include <GraphicalHostService/GraphicalHostClient.h>
```

Defines

- #define GHS_VERSION 0x01000303

16.3.1 Define Documentation

16.3.1.1 #define GHS_VERSION 0x01000303

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

This number is loosly associated with the OpenVE version number.

Part VII

Open System Inspector Service

Chapter 17

Data Structure Index

17.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 18

File Index

18.1 File List

Here is a list of all files with brief descriptions:

Chapter 19

Data Structure Documentation

19.1 _union_Hubs::_struct_L1_Event_ Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- char isSet

19.1.1 Field Documentation

19.1.1.1 char isSet

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.2 _union_Hubs::_struct_L1_Fifo_ Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- short int size
- short int count
- short int head
- short int tail

19.2.1 Field Documentation

19.2.1.1 short int count

19.2.1.2 short int head

19.2.1.3 short int size

19.2.1.4 short int tail

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.3 _union_Hubs::_struct_L1_PacketPool_ Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- short int size

19.3.1 Field Documentation

19.3.1.1 short int size

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.4 _union_Hubs::_struct_L1_Resource_ Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- L1_BOOL locked
- unsigned int taskID
- unsigned char ceilingPrioriry
- unsigned char boostedPrioriry

19.4.1 Field Documentation

19.4.1.1 unsigned char boostedPrioriry

19.4.1.2 unsigned char ceilingPrioriry

19.4.1.3 L1_BOOL locked

19.4.1.4 unsigned int taskID

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.5 _union_Hubs::_struct_L1_Semaphore_ Struct Reference

```
#include <OpenSystemInspectorClient.h>
```

Data Fields

- short int count

19.5.1 Field Documentation

19.5.1.1 short int count

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.6 _union_Hubs Union Reference

```
#include <OpenSystemInspectorClient.h>
```

Data Structures

- struct _struct_L1_Event_
- struct _struct_L1_Fifo_
- struct _struct_L1_PacketPool_
- struct _struct_L1_Resource_
- struct _struct_L1_Semaphore_

Data Fields

- struct _union_Hubs::_struct_L1_Fifo_ Fifo
- struct _union_Hubs::_struct_L1_Event_ Event

- struct _union_Hubs::_struct_L1_Semaphore_ Semaphore
- struct _union_Hubs::_struct_L1_Resource_ Resource
- struct _union_Hubs::_struct_L1_PacketPool_ PacketPool

19.6.1 Field Documentation

19.6.1.1 struct _union_Hubs::_struct_L1_Event_ Event

19.6.1.2 struct _union_Hubs::_struct_L1_Fifo_ Fifo

19.6.1.3 struct _union_Hubs::_struct_L1_PacketPool_ PacketPool

19.6.1.4 struct _union_Hubs::_struct_L1_Resource_ Resource

19.6.1.5 struct _union_Hubs::_struct_L1_Semaphore_ Semaphore

The documentation for this union was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.7 hubInfoStruct Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- L1_ServiceType type
- Hubs hub

19.7.1 Detailed Description

Structure for saving information about the requested hub

19.7.2 Field Documentation

19.7.2.1 Hubs hub

19.7.2.2 L1_ServiceType type

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.8 reqType Struct Reference

`#include <OpenSystemInspectorClient.h>`

Data Fields

- enum SERVICE req
- L1_HubID id
- L1_UINT16 objId
- L1_BYTE ∗ address

19.8.1 Field Documentation

19.8.1.1 L1_BYTE∗ address

19.8.1.2 L1_HubID id

19.8.1.3 L1_UINT16 objId

19.8.1.4 enum SERVICE req

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

19.9 taskInfoStruct Struct Reference

```
#include <OpenSystemInspectorClient.h>
```

Data Fields

- home bluescreen workspace OpenComRTOS Suite_API Manual downloads osd include OpenSystemInspector OpenSystemInspectorClient h home bluescreen workspace OpenComRTOS Suite_API Manual downloads osd include OpenSystemInspector OpenSystemInspectorClient h unsigned char priority
- unsigned char state

19.9.1 Detailed Description

Structure for saving information about the requested task

19.9.2 Field Documentation

19.9.2.1 home bluescreen workspace OpenComRTOS Suite_API Manual downloads osd include OpenSystemInspector OpenSystemInspectorClient h home bluescreen workspace OpenComRTOS Suite_API Manual downloads osd include OpenSystemInspector OpenSystemInspectorClient h unsigned char priority

19.9.2.2 unsigned char state

The documentation for this struct was generated from the following file:

- include/OpenSystemInspector/OpenSystemInspectorClient.h

Chapter 20

File Documentation

20.1 include/OpenSystemInspector/OpenSystemInspectorClient.h File Reference

```
#include <L1_types.h>
#include "socketconnection.h"
#include "OpenSystemInspectorServer.h"
#include <stdlib.h>
```

Data Structures

- struct taskInfoStruct
- union _union_Hubs
- struct _union_Hubs::_struct_L1_Fifo_
- struct _union_Hubs::_struct_L1_Event_
- struct _union_Hubs::_struct_L1_Semaphore_
- struct _union_Hubs::_struct_L1_Resource_
- struct _union_Hubs::_struct_L1_PacketPool_
- struct hubInfoStruct
- struct reqType

Defines

- #define getTaskById(OSIPort, TaskID, TaskStruct) sendPacketToServer((L1_HubID)(OSIPort),GetTaskByID,GetTaskBy
  Return,(id_type)(TaskID),(taskInfoStruct∗)(TaskStruct))
- #define getHubById(OSIPort, HubID, HubStruct) sendPacketToServer((L1_HubID)(OSIPort),GetHubByID,GetHubByI
  Return,(id_type)(HubID),(hubInfoStruct∗)(HubStruct))
- #define getTCB(OSIPort, TaskID) sendPacketToServer((L1_HubID)(OSIPort),GetTCB,GetTCB_-
  Return,(id_type)(TaskID),NULL)
- #define getLocalHub(OSIPort, HubID) sendPacketToServer((L1_HubID)(OSIPort),GetLocalHub,GetLocalHub_-
  Return,(id_type)(HubID),NULL)
- #define getPreallocatedPacket(OSIPort, TaskID) sendPacketToServer((L1_HubID)(OSIPort),GetPreallocatedPacket,GetP
  Return,(id_type)(TaskID),NULL)

- #define getOSIVersion(OSIPort, version) sendPacketToServer((L1_HubID)(OSIPort),GetOSIVersion,GetOSIVersion_-Return,0,(L1_UINT32∗)(version));
- #define getPPSize(OSIPort, HubID, size) sendPacketToServer((L1_HubID)(OSIPort),GetPacketPoolSize,GetPacketPool
Return,0,(L1_UINT32∗)(size));
- #define osiLock(Tld) L1_LockResource_W(Tld+2);
- #define osiUnlock(Tld) L1_UnlockResource_W(Tld+2);

Typedefs

- typedef union _union_Hubs Hubs

Functions

- void ∗ sendPacketToServer (L1_HubID OSIPort, enum SERVICE service, enum SERVICE serviceRet, id_type id, void ∗structPtr)
- L1_Status stopTasks (L1_HubID OSIPort, id_type id)
- L1_Status startTasks (L1_HubID OSIPort, id_type id)
- int getReadyList (L1_HubID OSIPort, id_type ∗readyList)
- int peek (L1_HubID OSIPort, L1_BYTE ∗address, L1_UINT16 length, L1_BYTE ∗result)
- L1_Status poke (L1_HubID OSIPort, L1_BYTE ∗address, L1_UINT16 length, L1_BYTE ∗data)
- void OSIClient_ISR (L1_HubID OSIClientPort)
- void OSIClient_entrypoint (osiSocketConnection ∗currentConnection)

20.1.1 Define Documentation

20.1.1.1 #define getHubById(*OSIPort, HubID, HubStruct*) sendPacketToServer((L1_-HubID)(OSIPort),GetHubByID,GetHubByID_Return,(id_-type)(HubID),(hubInfoStruct∗)(HubStruct))

Function for getting structure with information about current state of hub

Parameters

OSIPort	- an ID of OSI Server entity
HubID	- Local Hub ID information of interest hub
HubStruct	- Pointer to a structure which is at the output will contain information about the requested hub

Returns

If information getting corrected returned RC_OK, otherwise RC_FAIL

20.1.1.2 #define getLocalHub(*OSIPort, HubID*) sendPacketToServer((L1_-HubID)(OSIPort),GetLocalHub,GetLocalHub_Return,(id_type)(HubID),NULL)

20.1.1.3 #define getOSIVersion(*OSIPort, version*) sendPacketToServer((L1_-HubID)(OSIPort),GetOSIVersion,GetOSIVersion_Return,0,(L1_UINT32∗)(version));

20.1.1.4 #define getPPSize(*OSIPort, HubID, size*) sendPacketToServer((L1_-HubID)(OSIPort),GetPacketPoolSize,GetPacketPoolSize_Return,0,(L1_UINT32∗)(size));

20.1.1.5 #define getPreallocatedPacket(*OSIPort, TaskID*) sendPacketToServer((L1_-HubID)(OSIPort),GetPreallocatedPacket,GetPreallocatedPacket_Return,(id_-type)(TaskID),NULL)

20.1.1.6 #define getTaskById(*OSIPort, TaskID, TaskStruct*) sendPacketToServer((L1_HubID)(OSIPort),GetTaskByID,GetTaskByID_Return,(id_-type)(TaskID),(taskInfoStruct∗)(TaskStruct))

Function for getting structure with information about current state of task

Parameters

OSIPort	- an ID of OSI Server entity
TaskID	- Local task ID information of interest task
TaskStruct	- Pointer to a structure which is at the output will contain information about the requested task

Returns

If information getting corrected returned RC_OK, otherwise RC_FAIL

20.1.1.7 #define getTCB(*OSIPort, TaskID*) sendPacketToServer((L1_-HubID)(OSIPort),GetTCB,GetTCB_Return,(id_type)(TaskID),NULL)

20.1.1.8 #define osiLock(*Tld*) L1_LockResource_W(Tld+2);

20.1.1.9 #define osiUnlock(*Tld*) L1_UnlockResource_W(Tld+2);

20.1.2 Typedef Documentation

20.1.2.1 typedef union _union_Hubs Hubs

20.1.3 Function Documentation

20.1.3.1 int getReadyList (L1_HubID *OSIPort,* id_type ∗ *readyList*)

Function for getting Ready List information

Parameters

OSIPort	- an ID of OSI Server entity
readyList	- pointer on output array where ID's will be placed

Returns

If task stopped successfully returned RC_OK, otherwise RC_FAIL

20.1.3.2 void OSIClient_entrypoint (osiSocketConnection * *currentConnection*)

20.1.3.3 void OSIClient_ISR (L1_HubID *OSIClientPort*)

20.1.3.4 int peek (L1_HubID *OSIPort*, L1_BYTE * *address*, L1_UINT16 *length*, L1_BYTE * *result*)

Function for reading data from given address with given length

Parameters

OSIPort	- an ID of OSI Server entity
address	- pointer on memory address from where data will be read
length	- number of byes for reading
result	- pointer on data where will be placed read symbols

Returns

If data was read successfully returned amount of copied bytes, otherwise return -1

20.1.3.5 L1_Status poke (L1_HubID *OSIPort*, L1_BYTE * *address*, L1_UINT16 *length*, L1_BYTE * *data*)

Function for writting data to given address with given length

Parameters

OSIPort	- an ID of OSI Server entity
address	- pointer on memory address where data will be writed
length	- number of byes for writting
data	- pointer on data which will be copied

Returns

If data was wrote successfully returned RC_OK, otherwise - RC_FAIL

20.1.3.6 void* sendPacketToServer (L1_HubID *OSIPort*, enum SERVICE *service*, enum SERVICE *serviceRet*, id_type *id*, void * *structPtr*)

20.1.3.7 L1_Status startTasks (L1_HubID *OSIPort*, id_type *id*)

Function for starting tasks by local id

Parameters

OSIPort	- an ID of OSI Server entity
id	- Local task ID

Returns

If task started successfully returned RC_OK, otherwise RC_FAIL

20.1.3.8 L1_Status stopTasks (L1_HubID *OSIPort,* id_type *id*)

Function for stopping tasks by local id

Parameters

OSIPort	- an ID of OSI Server entity
id	- Local task ID

Returns

If task stopped successfully returned RC_OK, otherwise RC_FAIL

20.2 include/OpenSystemInspector/OpenSystemInspectorServer.h File Reference

```
#include "OpenSystemInspectorService.h"
```

Enumerations

- enum SERVICE {
  GetReadyList = 1, RunAllTasks, StopAllTasks, GetTaskByID,
  GetHubByID, Peek, Poke, GetTCB,
  GetLocalHub, GetPreallocatedPacket, GetOSIVersion, ShowHexTCB,
  ShowHexLocalHub, GetPacketPoolSize, GetHubByID_Return, GetTaskByID_Return,
  GetReadyList_Transmit, GetReadyList_Return, Peek_Return, Poke_Return,
  GetLocalHub_Return, GetTCB_Return, GetPreallocatedPacket_Return, GetOSIVersion_Return,
  GetPacketPoolSize_Return }

Functions

- void sendTaskInfo (L1_Packet ∗Packet, id_type id)
- void sendHubInfo (L1_Packet ∗Packet, id_type id)
- void returnReadyList (L1_HubID OSIPort, L1_Packet ∗Packet)
- void OSIEntryPoint (L1_HubID OSIServerPort)

20.2.1 Enumeration Type Documentation

20.2.1.1 enum SERVICE

Enumerator:

GetReadyList

RunAllTasks

StopAllTasks

GetTaskByID

GetHubByID

Peek

Poke

GetTCB

GetLocalHub

GetPreallocatedPacket

GetOSIVersion

ShowHexTCB

ShowHexLocalHub

GetPacketPoolSize

GetHubByID_Return

GetTaskByID_Return

GetReadyList_Transmit

GetReadyList_Return

Peek_Return

Poke_Return

GetLocalHub_Return

GetTCB_Return

GetPreallocatedPacket_Return

GetOSIVersion_Return

GetPacketPoolSize_Return

20.2.2 Function Documentation

20.2.2.1 void OSIEntryPoint (L1_HubID *OSIServerPort*)

20.2.2.2 void returnReadyList (L1_HubID *OSIPort,* L1_Packet * *Packet*)

20.2.2.3 void sendHubInfo (L1_Packet * *Packet,* id_type *id*)

20.2.2.4 void sendTaskInfo (L1_Packet * *Packet,* id_type *id*)

comment

20.3 include/OpenSystemInspector/OpenSystemInspectorService.h File Reference

Defines

- #define OSI_VERSION 0x01000303

Typedefs

- typedef L1_BYTE id_type

20.3.1 Define Documentation

20.3.1.1 #define OSI_VERSION 0x01000303

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

This number is loosly associated with the OpenVE version number.

20.3.2 Typedef Documentation

20.3.2.1 typedef L1_BYTE id_type

Part VIII

Save Virtual Machine for C

Chapter 21

Safe Virtual Machine for C (SVM)

The Safe Virtual Machine for C (SVM) is a small (∼3kB code on an ARM-Cortex-M3) virtual machine that is able to interpret ARM-Thumb-1 instruction set binaries. The SVM consists of two parts:

1. SVM Host Server
2. SVM-Platform

21.1 Introduction

21.2 SVM Host Server

This is the actual virtual machine that can be used within an application. Each virtual machine can execute one OpenComRTOS Task. You can have multiple SVM Host Server instances in your system.

21.2.1 Properties

The user can modify the following properties of the SVM Host Service Component:

- name: The name of the component, this must be unique in the whole system. The default node name is 'svm'.
- bufferSize: The size of the program buffer in 32bit words. It has a default value of 5000 32bit words, but the size depends on the concrete application.
- VM_Task_StackSize: The amount of stack space allocated for the virtual machine task. The default value is 512, which is a safe bet, but it can be reduced depending on target CPU the SVM is executing on.
- SVM_Supervisor_StackSize: The amount of stack space allocated for the supervisor task. The default value is 512, which is a safe bet, but it can be reduced depending on target CPU the SVM is executing on.
- SVMCeilingPriority: The priority of that the Resource-Hub embedded in the SVM can at most boost its owner task to in case of a priority inheritance operation taking place.

21.3 SVM-Platform

This is a virtual node on which the tasks get mapped that should be compiled for the SVM Host Server. The output of the compilation is a bin-file in the Output/bin folder.

21.3.1 Properties

The SVM-Platform has used and unused properties. The unused properties will be removed in a future version, once the build system has been adjusted to work without them. The used properties are:

- name: The name of the node that represents the SVM-Platform. This is used to assign nodes to it.
- compiler: The compiler to use, by default the SVM-Platform uses the compiler arm-none-eabi-gcc. If your compiler is not in the executable search path you can set the correct path to it here.
- compilerOptions: The task that gets compiled using the SVM-Platform can be compiled with the following compilation options:
  - O0: No optimisations
  - O3: Speed optimisations
  - Os: Size optimisations, this is the default value.

The unused properties are:

- rxPacketPoolSize: This defines how many L1_Packets are present in the Node global RX packet pool. This pool is used by the link drivers to acquire local packets for packets they received over their link.
- kernelPacketPoolSize: Number of L1_Packets in the Kernel packet pool. This pool is used by the kernel to send messages to other nodes. Typical examples where this used is for implementing priority inheritance for non local tasks.
- traceBufferSize: the number of events the node can remember when run in tracing mode (dbugopt==1 or dbugopt==2).
- debugopt: This defines the tracing mode in which the node is run, the following modes are available:
  - 0: No trace information to be generated
  - 1: Limited tracing
  - 2: Full tracing

21.4 Tutorial

This tutorial explains the steps to run one Task of the Semaphore_W_SP example inside an instance of the SVN.

1. Start OpenVE
2. Open the project located at:

   `(examples\win32\Semaphores\Semaphore_W_SP)`

3. Save the project as a new project called Semaphore_W_SVM. To do this follow these steps:

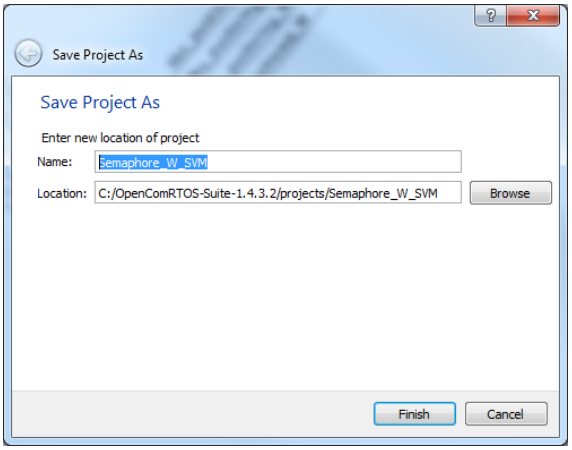


Figure 21.1: The 'Save Project As' Dialogue

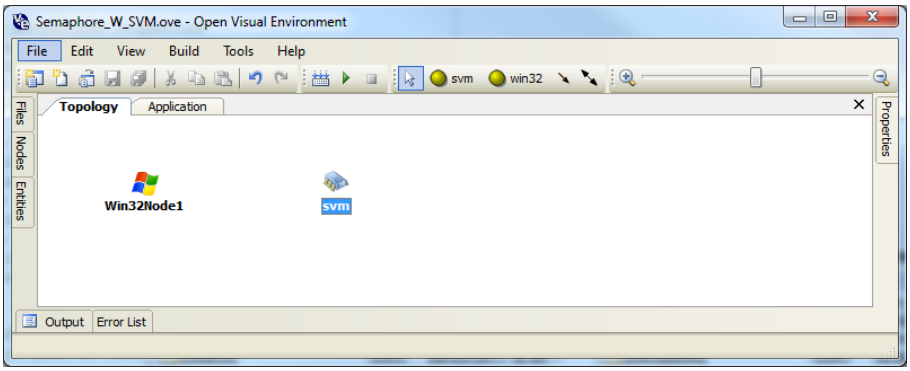


Figure 21.2: Extended Topology with SVM-Platform Node

(a) Go to the main menu: File -> 'Save Project As'. See Figure 21.1 for an illustration of the the 'Save Project As' Dialogue.

(b) In the text-field labeled 'Name:' insert the new name: 'Semaphore_W_SVM'.

(c) Press on the button labeled 'Finish'. This will copy the current project at the new location; close the current project; and open the newly created project. Figure 21.1 show the 'Save Project As' Dialogue.

4. Open to the topology diagram and add a node of type 'svm' to the topology diagram. The diagram should then look similar to the one shown in Figure 21.2.

5. Add an SvmComponent to the Application diagram, with the following properties:

   - node: Win32Node1
   - name: SvmHostService
   - bufferSize: 5000 (means 5000 32Bit words program buffer)
   - SVMCeilingPriority: 32
   - VM_Task_StackSize: 512
   - SVM_Supervisor_StackSize: 512 Afterwards the application diagram should look similar to the one shown in Figure 21.3. Now the project contains an SVM-Platform and an SVM-Component. What is still missing are the following things:
   - A Task that gets run in the SVM-Component

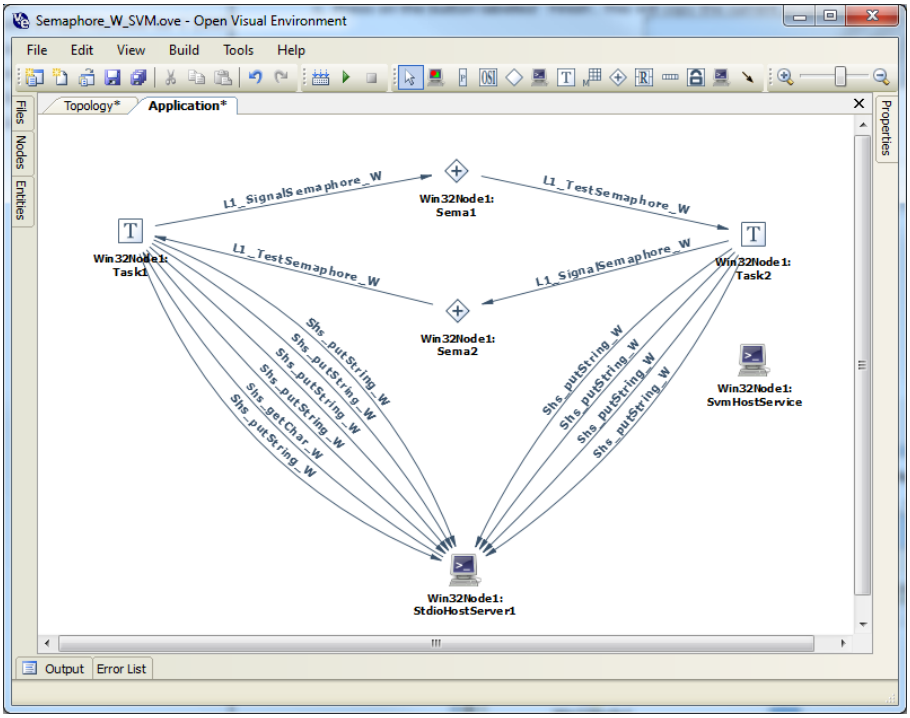


Figure 21.3: Application Diagram with SVM-Component

- A Task that loads the binary image of the Task that gets run in the SVM-Component into the SVM-Component and then starts the SVM-Component.

6. Map Task2 onto the previously created SVM-Platform node. To do this follow these steps:

   (a) Double click on the icon representing ‘Task2’, this will open the properties menu.

   (b) Change the property ‘node’ from ‘Win32Node1’ to ‘svm’ This causes the build system to compile the task for the SVM, the resulting binary image is named according to the following naming scheme: ${Task Name}.bin. Thus in our example here the file will be called ‘Task2.bin’.

7. Add a new Task to the Application Diagram, setting the following properties:

   - node: Win32Node1
   - name: LoaderTask
   - priority: 128
   - arguments: NULL
   - status: L1_Started
   - stackSize: 512
   - entryPoint: Create a new entry-point, by clicking on the plus button, call this new entry-point: ‘LoaderTaskEP’. The application diagram should now look similar to the one shown in Figure 21.4.

8. The newly created LoaderTask must now be filled with the corresponding code to load the binary image of Task2 (contained in the file Task2.bin), and then start the SVM-Execution:

   (a) Open the source code that represents the Task Entry Point of the LoaderTask, by opening the properties menu of task (right click on the icon that represents the Task) and then clicking on ‘Go to LoaderTaskEP’. This will open the source code representing the LoaderTask.

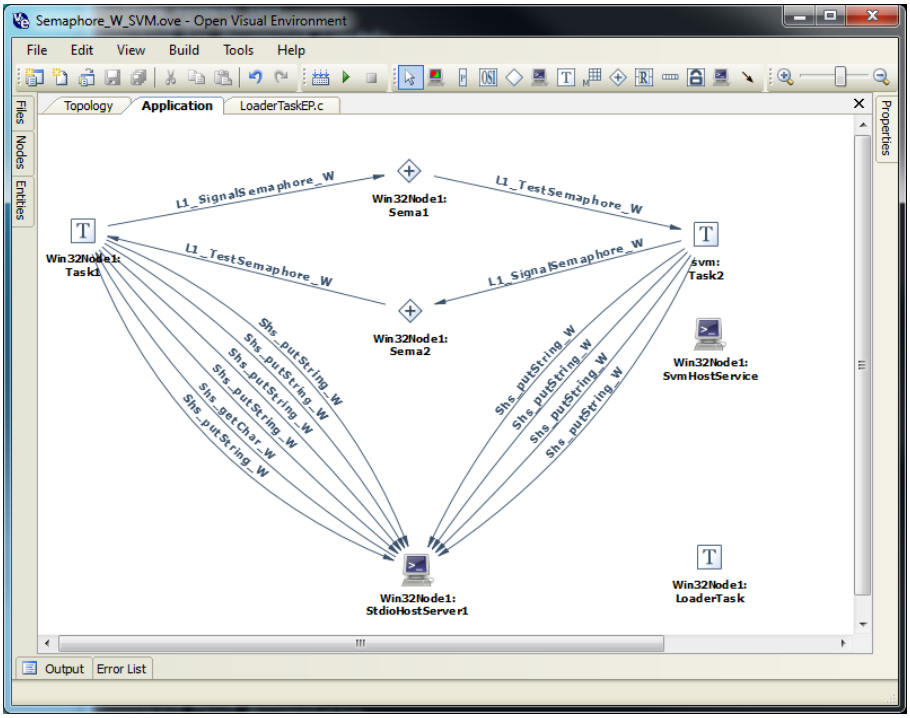


Figure 21.4: Application Diagram with Loader Task

(b) Include the file SvmService/SvmClient.h to ensure that the compiler finds the functions to control the SVM-Component.

(c) Delete the while(1){} loop

(d) Add the following lines into the body of the loader task:

- Svm_loadTaskFromFile(SvmHostService, StdioHostServer1, "Task2.bin"); This instructs the SVM-Component, called SvmHostService, to open the file 'Task2.bin' using the Stdio Host Server called StdioHostServer1. Then it loads the contents into the its program memory, before closing the file again.
- Svm_startTask(SvmHostService); Instructs the SVM-Component, called SvmHostService, to start executing the previously loaded task.

(e) The file LoaderTaskEP.c should look now as follows:

```
#include <L1_api.h>
#include <L1_nodes_data.h>

#include<SvmService/SvmClient.h>

void LoaderTaskEP(L1_TaskArguments Arguments)

    Svm_loadTaskFromFile(SvmHostService, StdioHostServer1,
                         "Task2.bin");
    Svm_startTask(SvmHostService);
```

9. Now you can build the program and execute it by pressing the run-button in the toolbar.

Chapter 22

Data Structure Index

22.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 23

File Index

23.1 File List

Here is a list of all files with brief descriptions:

Chapter 24

Data Structure Documentation

24.1 Svm_errorDescription Struct Reference

`#include <SvmServer.h>`

Data Fields

- SvmErrorCode errorCode

24.1.1 Field Documentation

24.1.1.1 SvmErrorCode errorCode

The error code.

The documentation for this struct was generated from the following file:

- include/SvmService/SvmServer.h

24.2 Svm_taskArguments Struct Reference

`#include <SvmServer.h>`

Data Fields

- L1_HubID SvmServerInputPort
- L1_HubID SvmServerOutputPort
- SvmHsSync * vmState
- L1_Packet clientInterfacePacket
- L1_Packet vmInterfacePacket
- L1_Packet * nextClientPacket
- L1_UINT32 bytesReceived

24.2.1 Field Documentation

24.2.1.1 L1_UINT32 bytesReceived

Number of bytes that were received already.

24.2.1.2 L1_Packet clientInterfacePacket

This L1_Packet is used to asynchronously wait for input from the SvmClient(s).

24.2.1.3 L1_Packet* nextClientPacket

This is used to return a client packet that was received while waiting for a Packet from the VM-Task.

24.2.1.4 L1_HubID SvmServerInputPort

24.2.1.5 L1_HubID SvmServerOutputPort

server output port

24.2.1.6 L1_Packet vmInterfacePacket

This L1_Packet gets used to asynchronously wait for input from the VM-Task.

24.2.1.7 SvmHsSync* vmState

This is a structure shared between the VM and the SVM-HS Tasks.

The documentation for this struct was generated from the following file:

- include/SvmService/SvmServer.h

24.3 Svm_vmTaskArguments Struct Reference

`#include <SvmServer.h>`

Data Fields

- SvmHsSync * vmState

24.3.1 Field Documentation

24.3.1.1 SvmHsSync* vmState

This is a structure shared between the VM and the SVM-HS Tasks.

The documentation for this struct was generated from the following file:

- include/SvmService/SvmServer.h

24.4 SvmHsSync Struct Reference

`#include <SvmServer.h>`

Data Fields

- SvmRequest request
- SvmState state
- L1_HubID requestIssued
- L1_HubID stateChanged
- L1_UINT32 * programBuffer
- L1_UINT32 programBufferSize
- L1_UINT32 registers [20]
- SvmErrorCode errorCode

24.4.1 Detailed Description

This structure is used by the SVM-HS to control and synchronise with the Virtual-Machine Task.

24.4.2 Field Documentation

24.4.2.1 SvmErrorCode errorCode

Contains the last error code the VM generated.

24.4.2.2 L1_UINT32* programBuffer

program memory

24.4.2.3 L1_UINT32 programBufferSize

Number of 32bit words the program memory can store.

24.4.2.4 L1_UINT32 registers[20]

Represents the Register set of the emulated CPU.

24.4.2.5 SvmRequest request

Holds the request for the VM.

Warning

Only the SVM-HS is allowed to write to this variable.

24.4.2.6 L1_HubID requestIssued

This is the HubID of the Event-Hub which is used by the SVM-HS to signal to the VM-Task that a new request has been sent, currently only used for the request SVM_START_VM.

24.4.2.7 SvmState state

Holds the current state of the VM.

Warning

Only the VM is allowed to modify this varialbe.

24.4.2.8 L1_HubID stateChanged

This is the HubID of the Event-Hub which is used by the VM-Task to signal to the SVM-HS that its state has changed.

The documentation for this struct was generated from the following file:

- include/SvmService/SvmServer.h

Chapter 25

File Documentation

25.1 include/SvmService/SvmClient.h File Reference

```
#include <L1_api.h>
#include <kernel/L1_memcpy.h>
#include <driver/linkcommunication.h>
#include <SvmService/SvmServer.h>
```

Functions

- L1_Status Svm_loadTask (L1_HubID vsp, L1_BYTE ∗program, L1_UINT32 length)
- L1_Status Svm_loadTaskFromFile (L1_HubID vsp, L1_HubID shsServer, char ∗filename)
- L1_Status Svm_startTask (L1_HubID vsp)
- L1_Status Svm_stopTask (L1_HubID vsp)
- L1_Status Svm_clearMemory (L1_HubID vsp)
- L1_Status Svm_getErrorInfo (L1_HubID vsp, Svm_errorDescription ∗info)
- L1_Status Svm_getState (L1_HubID vsp, SvmState ∗state)

25.1.1 Function Documentation

25.1.1.1 L1_Status Svm_clearMemory (L1_HubID *vsp*)

Clear program memory of the VM. This is synonim to 'unload task'.

Parameters

vsp	The ID of the Svm-Server which should process this request.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

25.1.1.2 L1_Status Svm_getErrorInfo (L1_HubID *vsp*, Svm_errorDescription * *info*)

Getting information about last error inside the virtual program.

Parameters

vsp	The ID of the Svm-Server which should process this request.
info	Pointer to an Svm_errorDescription structure, which will be updated with the description of the error.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

25.1.1.3 L1_Status Svm_getState (L1_HubID *vsp*, SvmState * *state*)

Retrieves the current state of the SVM.

Parameters

vsp	The ID of the Svm-Server which should process this request.
state	Pointer to a variable of type SvmState, after this call returns successfully the variable will contain the state of the SVM. stored

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

25.1.1.4 L1_Status Svm_loadTask (L1_HubID *vsp*, L1_BYTE * *program*, L1_UINT32 *length*)

Load binary image of task to the memory of VM.

Parameters

vsp	is the ID of the SvmInputPort of the VSP host server
program	A character array holding the image of the Task to be loaded.
length	Size of the Image in bytes.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

25.1.1.5 L1_Status Svm_loadTaskFromFile (L1_HubID *vsp*, L1_HubID *shsServer*, char * *filename*)

Loads a binary image from a file using a Stdio-Host-Server.

Parameters

vsp	The ID of the Svm-Server which should executed the image.
shsServer	The ID of the Stdio-Host-Server from which to load the image.
filename	The name of the file that contains the image.

25.1.1.6 L1_Status Svm_startTask (L1_HubID *vsp*)

Starts already loaded task.

Parameters

vsp	The ID of the Svm-Server which should process this request.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed. Task was not loaded yet.

25.1.1.7 L1_Status Svm_stopTask (L1_HubID *vsp*)

Stops running task.

Parameters

vsp	The ID of the Svm-Server which should process this request.

Returns

L1_Status

- RC_OK: The request was successful
- RC_FAIL: The request failed.

25.2 include/SvmService/SvmServer.h File Reference

```
#include <L1_types.h>
```

Data Structures

- struct Svm_errorDescription
- struct SvmHsSync
- struct Svm_taskArguments
- struct Svm_vmTaskArguments

Defines

- #define SVM_COMMAND_NTOH(cmd) ntoh32(cmd)
- #define SVM_COMMAND_HTON(cmd) hton32(cmd)
- #define SVM_COMMAND_LENGTH 4
- #define SVM_STATE_LENGTH 4

Typedefs

- typedef L1_UINT32 SVM_COMMAND_TYPE
- typedef L1_UINT32 SVM_STATE_TYPE
- typedef enum SVM_REQUEST SvmRequest
- typedef enum SVM_STATE SvmState

Enumerations

- enum SvmErrorCode { SVM_NO_ERROR = 1, SVM_ERROR_PROCESS_SWI, SVM_ERROR_-UNKNOWN_INSTRUCTION }
- enum SVM_PROTOCOL {

  SVM_LOAD_TASK, SVM_LOAD_TASK_SESSION_START, SVM_LOAD_TASK_SESSION_-STOP, SVM_START_TASK,

  SVM_STOP_TASK, SVM_CLEAR_MEMORY, SVM_GET_ERROR_INFO, SVM_GET_STATE }
- enum SVM_REQUEST { SVM_NO_REQUEST = 1, SVM_START_VM, SVM_STOP_VM, SVM_-SUSUPEND_VM }
- enum SVM_STATE { SVM_VM_STOPPED = 1, SVM_VM_RUNNING, SVM_VM_SUSPENDED, SVM_VM_ERROR }

25.2.1 Define Documentation

25.2.1.1 #define SVM_COMMAND_HTON(*cmd*) hton32(cmd)

25.2.1.2 #define SVM_COMMAND_LENGTH 4

25.2.1.3 #define SVM_COMMAND_NTOH(*cmd*) ntoh32(cmd)

25.2.1.4 #define SVM_STATE_LENGTH 4

25.2.2 Typedef Documentation

25.2.2.1 typedef L1_UINT32 SVM_COMMAND_TYPE

25.2.2.2 typedef L1_UINT32 SVM_STATE_TYPE

25.2.2.3 typedef enum SVM_REQUEST SvmRequest

These are the requests that the SVM-HostServer Task can send to the Virtual-Machine Task.

25.2.2.4 typedef enum SVM_STATE SvmState

These are the states the Virtual-Machine Task can be in.

25.2.3 Enumeration Type Documentation

25.2.3.1 enum SVM_PROTOCOL

Enumerator:

SVM_LOAD_TASK loading program to the server

SVM_LOAD_TASK_SESSION_START loading program to the server(session start)

SVM_LOAD_TASK_SESSION_STOP loading program to the server(session stop)

SVM_START_TASK start virtual task

SVM_STOP_TASK stop virtual task

SVM_CLEAR_MEMORY clear program memory of the VM(unload task)

SVM_GET_ERROR_INFO get info about error in the virtual program

SVM_GET_STATE get current state of the VM

25.2.3.2 enum SVM_REQUEST

These are the requests that the SVM-HostServer Task can send to the Virtual-Machine Task.

Enumerator:

SVM_NO_REQUEST Indicates that there is currently no requst being sent.

SVM_START_VM Indicates that the Interpreter should be started.

SVM_STOP_VM Indicates that the Interpreter should bes stopped.

SVM_SUSUPEND_VM Indicates that the Interpreter should suspend the Task it executes.

25.2.3.3 enum SVM_STATE

These are the states the Virtual-Machine Task can be in.

Enumerator:

SVM_VM_STOPPED The VM is stopped, this is the initial state. The VM changes into this state after receiving the SVM_STOP_VM request.

SVM_VM_RUNNING The VM is running, the VM changes into this state after receiving the SVM_-START_VM request, and a packet in its input port.

SVM_VM_SUSPENDED The VM has suspended the Task it was executing. The VM changes into this state after receiving the SVM_SUSPEND_VM request.

SVM_VM_ERROR The VM has detected an error during its execution.

25.2.3.4 enum SvmErrorCode

Structure that stores the error description. It passed inside an L1_Packet from the VM-Task to the Supervisor-Task.

Enumerator:

SVM_NO_ERROR

SVM_ERROR_PROCESS_SWI

SVM_ERROR_UNKNOWN_INSTRUCTION

25.3 include/SvmService/SvmService.h File Reference

```
#include <SvmService/SvmClient.h>
```

Defines

- #define SVML_HOST_SERVICE_HEADER
- #define SVMHS_VERSION 0x00020301

25.3.1 Define Documentation

25.3.1.1 #define SVMHS_VERSION 0x00020301

The L1_UINT32 value of is formatted the following way:

- MSByte: Major Version of the Kernel
- 23--16: Minor Version
- 15--8 : Release status:
  - 0: Alpha
  - 1: Beta
  - 2: Release Candidate
  - 3: Public Release
- LSByte: Patch-level

25.3.1.2 #define SVML_HOST_SERVICE_HEADER

Part IX

Appendix

Bibliography

[1] Mingw minimalist gnu for windows. http://www.mingw.org/. 33

[2] Sourcery g++ lite 2009q1-161 for arm eabi. http://www.codesourcery.com/sgpp/lite/arm/portal/release830. 35

[3] Cmake — cross platform make. http://www.cmake.org/cmake/resources/software.html. 35

Glossary

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

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 OpenComRTOS used to implement all L1 entities.. 5–7, 10, 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

ISR Interrupt Service Routine. 9, 12, 15, 23, 24, 26

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, 5–10, 12, 15, 17, 19–26, 29

Acronyms

RTOS Real-time Operating System. 3, 5, 7, 12

Index