Reference Manual

General Concepts

Rubus® OS

Real-Time Operating System
for
Dependable Real-Time Systems

Part 1

Arcticus Systems

Document Version History

Version 4.0 October, 2009

Arcticus Systems AB

Datavägen 9A
Box 530
175 26 JÄRFÄLLA
SWEDEN
Phone +46 8 580 31100
info@arcticus-systems.com
www.arcticus-systems.com

Table of Contents

Figures

1 Introduction

The *Rubus® OS* Reference Manual describes the Application Program Interface specification for the C-Language. The manual is divided into two parts: Part 1 describing the general concepts and a functional overview of Rubus OS. Part 2 contains detailed information about the services provided by Rubus OS.

Rubus® OS is a unique Real Time Operating System specially designed for a broad range of dependable real-time applications. Rubus OS is based on *Basement®*, a distributed real-time architecture developed for vehicle internal use in the automotive industry [BASEMENT95].

The key constituents of the *Basement* concept are:

- Resource sharing (multiplexing) of processing and communication resources,
- A guaranteed real-time service for safety critical applications,
- A best-effort service for non-safety critical applications,
- A communication infrastructure providing efficient communication between distributed devices, and
- A program development methodology allowing resource independent and application oriented development of application software.

Rubus OS supports both static and dynamic scheduling of threads.

To guarantee the real-time behaviour of the safety critical application static scheduling in combination with time-triggered execution is utilised. Dynamic scheduling is utilised manly for non-safety critical application.

The combination of a dynamic and static scheduling enables the design of optimal software architecture.

The Holistic View

The Rubus Concept provides a holistic view of developing dependable real-time applications, in the sense that not only the execution but also the development of application software is considered. In Rubus, the structure and behaviour of application software can be described at a rather high level of abstraction. Such descriptions are independent of the actual hardware on which it will be executed. Tools are provided for mapping abstract descriptions to a particular Rubus system (the resources).

The design of Rubus applications is based on a hardware metaphor, in that Rubus software is built from a set of predefined (or user-defined) software components, which in analogy with hardware circuits are termed *Software Circuits.* The main motivation for using such a metaphor is that it allows a structuring of the software which is conceptually close to hardware design, and thus it will be familiar to engineers. Also, the simple structure increases provability and improves human to human communication concerning designs.

Product Naming

Rubus® originates from Latin meaning a berry named *Rubus Arcticus* [in English *Arctic Raspberry*].

Rubus denotes a real-time architecture as well as a concept aimed for safety critical real time systems.

The company name *Arcticus Systems* originates from this delicious berry *Arctic Raspberry*.

The POSIX Standard

IEEE Standard for Information Technology- Portable Operating System Interface (POSIX).

The design of Rubus OS is based on relevant part of the POSIX Standard 1003. A strong objective has been to design the services with a minimum of overhead and to be “fail safe”. By fail-safe we address detection of normal faults related to the syntax and the semantics of a service.

Terminology

Process (POSIX) [5]: An address space and single thread of control that executes within that address space including required system resources.

Threads according to POSIX: A process may contain several sub processes (or threads). Threads execute in parallel sharing the same address space, variables, data structures and process ID. Related to the C-Program Language, a thread is implemented as a function, but the execution of the function is controlled by the operating system.

Applications in real-time systems can be designed as a set of co-operating sequential *processes* or *threads* (POSIX). A process or thread consists of operations that must be executed in a prescribed order.

A thread can be divided into those that have *hard deadlines*, i.e. meeting their deadlines is critical to the system's operation; and those which have *soft deadlines*, i.e. although a short response time is desirable, occasionally the missing of a deadline can be tolerated.

General Concepts

A real-time system can be seen as a system managing signal flow between sensors and actuators under time constraints, which may be located on different nodes in a network. Considering this, the main objectives of the Rubus OS is to provide services for managing the data signal flow and related computation in a distributed real-time system.

Three categories of run-time services are provided by Rubus OS:

- *Green Run-Time Services*, external event triggered execution (interrupts). The Green Kernel provides the Green Run-Time Services.
- *Red Run-Time Services*, time triggered execution, *mainly* to be used for functions that have *hard real-time requirements*, i.e. meeting their deadlines is critical to the operation. The Red Kernel provides the Red Run-Time Services.
- *Blue Run-Time Services*, internal event triggered execution, to be used for functions that have *hard real-time requirements* as well as *soft real-time requirements*, i.e. meeting their deadlines is not critical to the operation. The Blue Kernel provides the Blue Run-Time Services.

2 Rubus OS Overview

The main objectives of Rubus OS is:

- Support the execution of Time-Triggered Red threads.
- Support the execution of Interrupt-Triggered Green threads.
- Support execution of internal Event-Triggered Blue threads.
- Support communication between different types of threads.
- Support statically allocated resources.
- Support scalability and portability.
- Support runtime analysis.

The ambition when designing Rubus OS has been to design the services with a minimum of overhead and to be "fail safe". By fail-safe we address detection of normal faults related to the syntax and the semantics of a service.

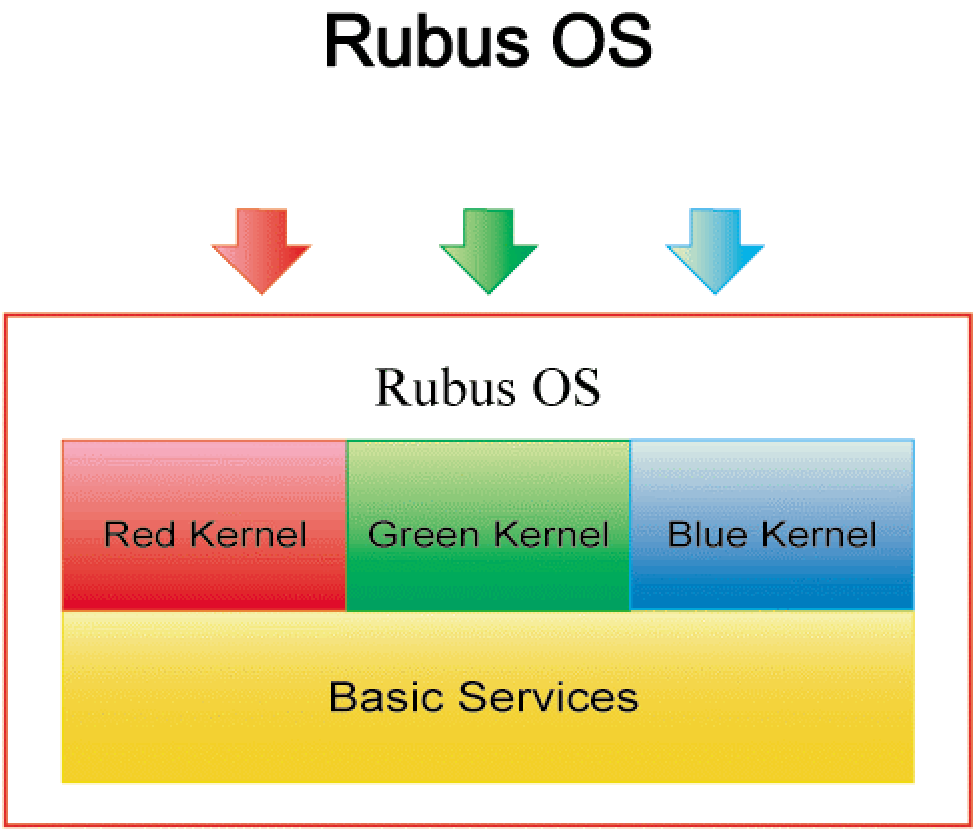

Figure 1 - Rubus OS

The *Red Kernel* manages the execution of time triggered Red threads. The Red threads are off-line scheduled.

The *Blue Kernel* manages the execution of internal event triggered Blue Threads. The Blue Kernel manages the spare execution time, not utilised by the Red Kernel, for execution of Blue threads. The scheduling of Blue threads is done during run-time.

The *Green Kernel* manages the execution of interrupt triggered Green threads. The Dispatch Unit in the Green Kernel is the target processors interrupt logic and thus its behaviour is target dependent.

The *Basic Services* contains services that are common for the Red, Blue and Green Kernel. These services include, clock and timer services, communication and analysis services.

Each component of Rubus OS consists of a generic part and a Hardware Adaptation Layer (HAL), which contains adaptation for the different supported hardware platforms.

Scheduling Algorithms

Scheduling algorithms are often characterised as being either *static* or *dynamic*.

Static Scheduling

In *static scheduling,* also called *off-line scheduling* or *pre-run-time scheduling***,** the schedule for threads is computed off-line. This approach requires that the major characteristics of the threads in the application are known in advance. It is possible to use pre-run-time scheduling to schedule periodic threads. This consists of computing a schedule off-line for the entire set of threads occurring within a time period that is equal to the least common multiple of the periods of the given set of threads. At run-time the periodic threads are executed in accordance with the pre-computed schedule.

In pre-run-time scheduling, several alternative schedules may be computed off-line for a given time period; each such schedule corresponds to a different *mode* of operation. The different modes of operation or schedules are selected in response to external or internal events.

The pre-run-time scheduling is applicable to the Red Kernel of the Rubus OS

Dynamic Scheduling

In *dynamic scheduling,* also called *on-line scheduling* or *run-time scheduling***,** the schedule is computed on-line as events related to threads occurs, where the scheduler does not assume any knowledge about the major characteristics of threads related to events that have not yet occurred. The advantage of this approach is that it is unnecessary to know the major characteristics of the threads in advance, thus providing flexibility in adapting to changes in the environment. Some important disadvantages are high run-time costs, non-determinism and increased complexity.

A run-time scheduling is providing a more flexible application that can react to levels of activity beyond what was anticipated.

The run-time scheduling is applicable to the Green and Blue kernels of the Rubus OS.

Exploiting Spare Capacity

A processor utilisation of 100% is very unlikely for real-time applications. Various types of relations such as precedence relations and exclusion relations exist between threads as well as resources constraints. These constraints will result in threads waiting for other threads to complete and delays when operating resources. Consequently, a certain amount of spare capacity will be available, regardless of the applied scheduling approach.

The spare capacity in Rubus OS is utilised by the Blue Kernel for applications utilising the Blue-Kernel. In a Red system, not utilising the Blue Kernel the spare capacity is utilised by the Red Idle Thread.

Static Allocation

To support dependability the allocation of resources are done statically pre-run-time. The Rubus OS resources used by the application are allocated using Rubus ICE.

The allocation of a resource is divided into an *attribute* and if applicable a *control block*. The *attribute* contains static information about the resource and is located into ROM. The *control block* contains dynamic information and is located into RAM. The initialisation of the resources control block must be done before using the resources.

Access to the resource in the application is done using a reference to the *attribute* of the resource.

Conventions

The following typographic conventions are employed:

The *italic* font is used for:

- C-Language data types and function names.
- Global external names.
- Cross-references to defined terms and symbolic parameters that are generally substituted with real values by the application.

The `constant-width` (Courier) font is used for:

- C-Language data types and function names within Synopsis subclauses.
- C-Code examples
- References to C-Language header.

Functions and types are named in a structured fashion in order to simplify for the usage of Rubus OS and to avoid conflicting with names occurring in existing packages of various C-Language Run Time Libraries. Another reason is to ensure that the many semantically related functions, types and variables are syntactically connected and have names that can be inferred on the basis of their context.

Function and Type Names

Function and type names have the following syntax:

[<HAL>]<component><object><operation>

<HAL>	Names related to the Hardware Adaptation Layer have the prefix *hal*.
<component>	Relates to the components of Rubus OS. The component Basic Services have the prefix *bs*. The components Red Kernel, Blue Kernel and Green Kernel have the prefix *red* or *r, blue or b* and *green or g,*
<object>	Type object denoted. This element is omitted for names denoting threads.
<operation>	Relates to the operation of the object. The operation = *"_t"* identifies opaque data

types. The operation = *"Attr"* identifies static attributes of an object.

C-Language Headers

The C-Language headers are organised in a directory structure:

	Hal/	Headers related to the target environment
	Basic/	Headers related to Basic Services
Include/	Green/	Headers related to Green Kernel Services
	Red/	Headers related to Red Kernel Services
	Blue/	Headers related to Blue Kernel Services

Symbolic Constants

The header <basic/bs_basic.h> contains basic Rubus OS definitions. Each service returns a value containing information about the operation. In case an error is detected the information about the error is included into the return value or to be found by calling a service:

R_OK	Return code defining successful operation of a service
R_ERROR	Return code defining unsuccessful operation of a service
R_TRUE	Boolean True value 1
R_FALSE	Boolean False value 0
R_TIME_WAIT_INFINITY	Time supervision, wait infinity for an event
R_TIME_NO_WAIT	Time supervision, wait zero time, time value zero
R_BASIC_NSEC	Number of nano seconds per seconds

3 Error Handling

One of the fundamental issues to be considered at designing a dependable real-time system is to support elaborated error handling. Error handling includes:

- Definition of failures
- Detection of error
- Services for error handling

Rubus OS is designed in such a way that vital failures are detected and that the user shall be able to react on the exception in an adequate way. The error handling provides a safe way of managing errors critical to the execution of the application. Rubus OS supports a mechanism for specifying a user defined error handling.

Error Status Values

In general, all services return the error status of the operation, the *return value*. An operation can be successfully or unsuccessfully completed. In case the operation is unsuccessfully completed the *return value* denotes the unsuccessful operation status including the *error number*.

The error status of a service falls into the following categories:

- An operation, which always is successful, the *return value* is omitted.
- An operation which may succeed or fail;

1. A successful operation is indicated by the *return value* containing integer zero or a non-null pointer.
2. An unsuccessful operation is indicated by the *return value* containing the negative value of the error number or a null pointer (R_NULLP).

For each service a set of failures is defined, each failure is defined by a value, the *error number*. The *error numbers* are found in the header <basic/bs_basic.h>.

When an internal error is detected by Rubus OS during runtime a user defined error function is called with parameters defining the error and the erroneous object.

Green Kernel Error Handling

An error detected by the Green Kernel during run-time is notified by calling the user defined function *greenError()*.

The Green Kernel detects the following errors:

Error Number	Error Object	Error Description
`R_ERROR_EXECUTION_TIME`	The Green thread	When a Green thread fails to meet its maximum execution time.
`R_ERROR_FREQUENCY_OVERFLOW`	The Green thread	When a Green thread fails to meet its maximum frequency.

When the Green Kernel detects an error the function *greenError()* is called with parameter as specified in the table. The parameter *error object* of *greenError()* denotes the erroneous object.

Red Kernel Error Handling

An error detected by the Red Kernel during run-time is notified by calling the user defined function *redError()*.

The Red Kernel detects the following errors:

Error Number	Error Object	Error Description
`R_ERROR_DEADLINE_OVERRUN`	The Red thread	When a Red thread fails to meet its deadline.
`R_ERROR_EXECUTION_TIME`	The Red thread	When a Red thread fails to meet its maximum execution time..
`R_ERROR_PREEMPTION_OVERFLOW`	Null-pointer	When the number of pre-emption levels is greater than specified.

When the Red Kernel detects an error the function *redError()* is called with parameter as specified in the table. The parameter *error object* of *redError()* denotes the erroneous object.

The Red stack is not supervised in run-time by the Red Kernel, but the user may do so by calling the function *redStackCheck()*. It is also advisable to control the maximum usage of the Red stack by using the function *redSatus()*.

Blue Kernel Error Handling

An error detected by the Blue Kernel during run-time is notified by calling the user defined function *blueError()*.

The Blue Kernel detects the following errors:

Error Number	Error Object	Error Description
`R_ERROR_INVALID_CONTROL_BLOCK`	Blue thread	A Blue Thread Control Block is corrupt detected at internal operation.
`R_ERROR_PRIORITY`	Blue thread	The priority of a Blue thread is out of range, detected during unblock operation.
`R_ERROR_CORRUPT_RQ`	The Control Block fetched from the Ready Queue	The Ready Queue is corrupt, detected by the Blue Dispatcher.
`R_ERROR_CORRUPT_IRQ`	The running thread	The Blue thread is reincarnated after exit.
`R_ERROR_STACK_INCONSISTENT`	The running thread	The stack of the running Blue thread is corrupt, detected by the Blue Dispatcher.

An error detected by the Blue Kernel during run-time is notified by calling the user defined function *blueError()*.

When the Blue Kernel detects an error the function *blueError()* is called with the *error number* as specified in the table. The parameter *error object blueError()* denotes the erroneous object. The running thread ID can be found by using the service *blueSelf()*.

The Blue Kernel supervises a Blue thread stack in run-time at thread switch, It is also advisable to control the maximum usage of each Blue thread stack space by using the function *blueThreadStatus()*.

4 Hardware Adaptation Layer

The Hardware Adaptation Layer (HAL) contains services that adapt Rubus OS for a specific target processor. By utilising this approach, the target related services is implemented in the most appropriate manner.

Environment Definitions

The header <hal/hal_env.h> contains environment-related definitions to be included into the application. This header is implicitly included into the other headers of Rubus OS. In Rubus OS the environment definitions are part of the Hardware Adaptation Layer, HAL. For each implementation of Rubus OS, headers are adapted to the environment of the implementation. The environment includes both the target processor as well as the selected C-Language environment (C-Language set of tools).

In the header <basic/bs_ctypes.h> the generic Misra 2004 C-Types is defined.

The following definitions are part of header and shall not be modified by the user:

int_t	Signed Integer, environment defined size
uint_t	Unsigned Integer, environment defined size
R_TYPE_ENDIAN	Define target endian format (big or little)
R_TYPE_STACK_PTR	Stack pointer type
R_TYPE_BLUE_THREAD_EXTENSION	Blue Thread Extension type
R_TYPE_GREEN_LEVEL	Green Interrupt Level type
R_LOC_CB	Memory location of Control Blocks to get fast access
R_LOC_HAL_FUNC	Memory location of functions to get fast access
R_LOC_FAR	Memory location of large variables
R_LOC_VAR	Memory location of internal variables to get fast access

The header <hal/hal_cfg.h> contains environment-related definitions used by the Rubus ICE generated configuration files.

5 Message Passing

A real-time application consists of multiple co-operating threads that wish to communicate and co-ordinate their activities. Message passing is a higher-level communication facility and a key facility for construction of high performance real-time applications. In certain cases a global variable will solve the communication, but in most cases the writing and reading of information require atomic operation since the operation is asynchronous events.

Information exchange within a node between Red, Green, and Blue threads can be realized in various ways. If the information is an Atomic Variable, (the size of the information is equal or smaller then the target processors data width), then the reading and writing operations are atomic with regards to the processor.

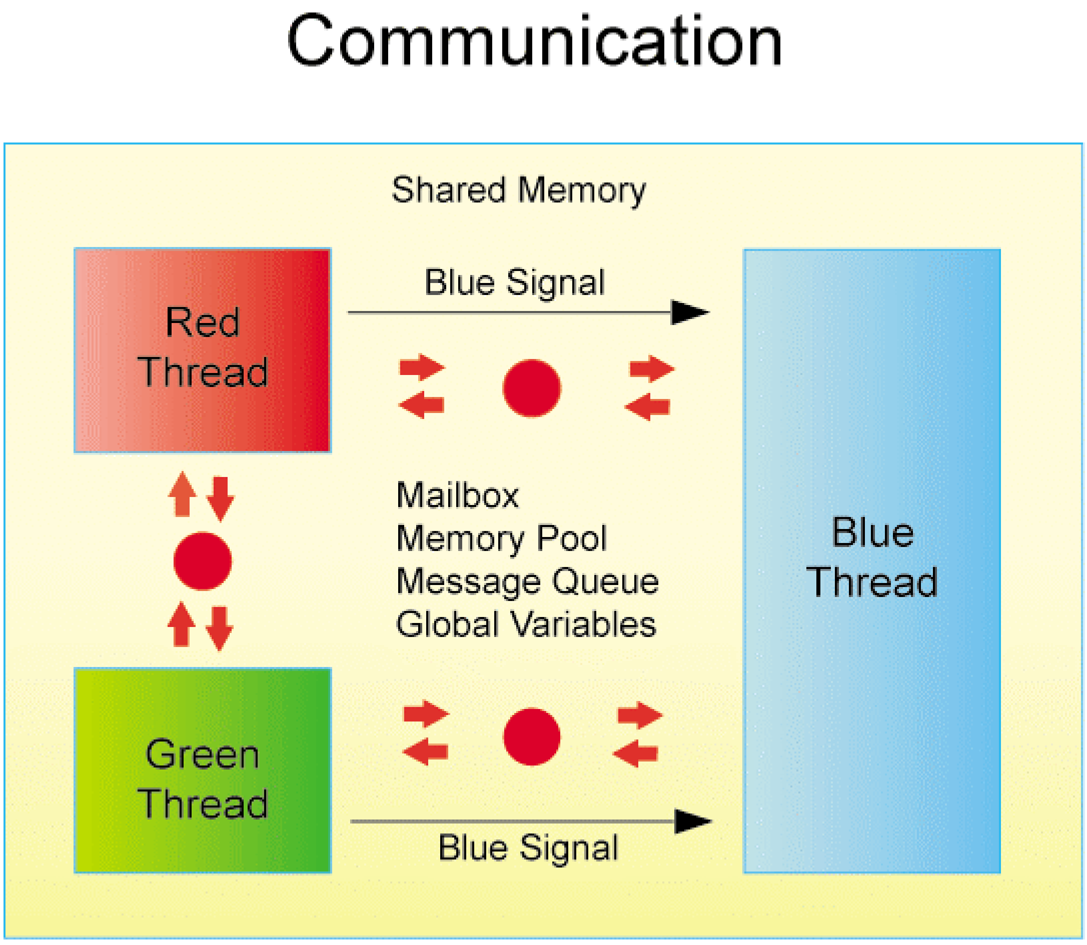

Figure 2 - Communication

If the producer thread generates information with a higher rate than the consumer thread it may require the information to be stored in a queue.

If large amount of data need to be communicated between threads, there is a requirement of dynamic memory management in combination with message passing.

The Basic Service layer contains a set of services for internal communication between Red, Green and Blue threads. For signalling to a Blue thread from a Red or Green thread or communication between Blue threads additional services are available (see Blue Kernel).

The Basic Service layer supports the following services:

- Basic Queue, data passing via a message queue.
- Basic Memory Pool, dynamic memory management of memory pools.
- Basic Mailbox, message passing via a mailbox.

Basic Queue Services

A Basic Queue is configured to contain a maximum number of objects of a specified size. The number of entries and the object size is specified in Rubus ICE. The queue is constructed as a First In First Out (FIFO) queue. An object is inserted into the queue by copying the contents of the object to the memory area reserved in the queue. An object is released from the queue by copying the contents in the queue to the memory area given as an address by the user.

Basic Memory Pool Services

Services for management of memory buffers organised in memory pools are included in Rubus OS. A Basic Memory Pool consists of a list of free buffers of specified size. The number of buffers and the buffer size is specified in Rubus ICE. There are services for allocation of a buffer from a memory pool and to free a buffer to a memory pool. The numbers of memory pools as well as the numbers of the buffers of a pool are unlimited.

The memory pools can be utilised for communication, see Basic Mailbox

At initialisation of a Basic Memory Pool the list of the free buffers are build from the memory space statically allocated to the pool. Each buffer contains a reference to its memory pool.

Basic Mailbox Services

A mailbox consists of references to messages sent to the mailbox. The references are FIFO sorted. The mailbox can be empty or contain an unlimited number of references. This requires that each message must reserve space for a message header. The message header contains a message type field. When receiving a message from a mailbox, either the first message in the mailbox is received, or, a message of a specified type.

The required message header is declared in <basic/bs_mbox.h>

Full duplex communication requires two mailboxes, on for each direction.

The Basic Memory Pool may be used to allocate message buffers. The producing thread allocates a buffer from a pool, fill the information into the buffer and send the buffer pointer to a Mailbox. The consumer thread receives the buffer pointer from the Mailbox, reads the information and restores the buffer into the Memory Pool.

6 Clocks and Timers

The mechanism to measure the passage of time is called a clock. A timer is associated with a clock, the timer expires when the associated clock reaches or exceeds the specified time. In case a timer operates with relative time, the timer expires when the specified interval, as measured by the associated clock, has passed.

Timers

Two types of timer are required for a system supporting real-time:

- One shot
- Periodic

For both types of timers, initial time expiration is specified as a relative time value.

A one shot timer is a timer that is armed with an initial expiration time relative to the current time. The timer expires once, and then is disarmed.

A periodic timer is a timer that is armed with an initial expiration time, relative to the current time, and a repetition interval. When the initial expiration occurs, the timer is reloaded, with the repetition interval and continues counting.

Clocks

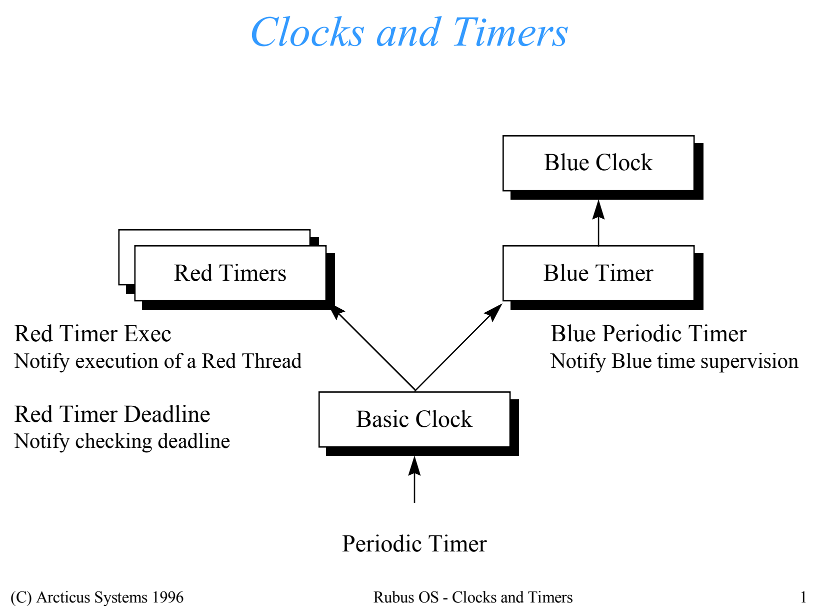

Figure 3 - Clocks and Timers

In Rubus there are two clocks; the *Basic Clock* and the *Blue Clock*.

The Basic Clock is the clock of on which the Red and Blue timers are based. The Blue Clock is based on the Blue timer and supports time supervision by the Blue Kernel. In principle the resolution of Basic Clock is the smallest fraction of the Red and Blue timers. The resolution of all Red and Blue timers is user defined.

Basic Clock

The Basic Clock is defined to be a system clock representing the real-time clock for the system, the *system time*. This clock represents the amount of time in timer ticks (*jiffies*). *Jiffiy* is a user defined time sort.

The header <basic/bs_basic.h> defines the structures and types used by the Clock and Timer services. The structure *bsPosixTime_t* specifies a single time value containing a member defining number of seconds and the second member number of nanoseconds. This structure can represent the amount of time in seconds and nano seconds since Epoch (January 1, 1970)

In a distributed environment, the Basic Clock in each node represents the system time and is maintained by the communication services and in case one node is operating locally, the Basic Clock may not represent the system time.

Rubus OS provides services to convert between POSIX Time representation and jiffies. (See *bsJiffiesToTv()* and *bsTvToJiffies()*).

Blue Clock

The Blue Clock is incremented by the Blue Kernel Thread. The invocation of the Blue Kernel Thread is done from each Red Schedule according to a user-defined prescaler that constitutes the Blue timer. Thus, the resolution of the Blue Clock is related to the active Red Schedule and its Blue prescaler. Since the time management services in the Blue Kernel (e.g. *blueSleep()*) are based on the Blue Clock, the resolution of these services is also dependent on the active Red Schedule and its Blue prescaler.

Basic Periodic Timer

In Rubus OS the different clocks are based on one Basic Periodic Timer source, which the user must supply. This Basic Periodic Timer is normally a processor supported programmable timer generating an interrupt when the timer expires.

Since this Basic Periodic Timer is the time source for the Red Kernel, it is important that the resolution of the timer is related to the application requirements. Selecting a too high resolution may cause unnecessary overhead. The resolution must be specified in Rubus ICE in order for Rubus ICE to generate correct code.

On targets supporting interrupt priority levels, the interrupt priority level of the Basic Periodic Timer is the level at which the Red Kernel is invoked, pre-empting interrupts with lower priority. The priority level at which the Red application will execute is implementation defined. The priority level for the Red application must be less then the priority level of the Timer interrupt. It is important to select an appropriate priority level for the Periodic Timer and the Red application based on the application requirements. The priority levels must be specified in Rubus ICE in order for Rubus ICE to generate correct code.

Basic Timer Interrupt Handling

Since the Basic Periodic Timer is related to the users environment, a user provided set of functions to enable and disable the interrupt request of the Basic Periodic Timer is provided.

The set of functions contains *halBsTimerStart()* function to initiate and start the Basic Periodic Timer as well as start the timers for the measurement of execution time and event logging. Also the *halBsTimerStop()* function to stop the timers should be included into this set.

The set also contains *halBsTimerDisable()* function to disable interrupt associated to the Basic Periodic Timer and *halBsTimerEnable()* function to enable interrupt associated to the timer.

This set of functions is utilised during execution of the Red Kernel and allows other interrupt requests to be processed. When the Red Dispatcher launches execution of a Red thread the interrupt is enabled.

User Supplied Basic Timer Interrupt Handling

In respect to the design of Basic Periodic Timer Interrupt handling, the user shall supply an ISR for the Basic Periodic Timer Interrupt. In some implementations the user sets the entry to the function *halBsTimerIntrEntry()* into the interrupt vector depending on the timer device selected.

Red Timer

The Red Timer measures the time during a period, the length of current Red Schedule, it is incremented by one for each Periodic Timer tick. The timer is reloaded with zero when the timer expires (end of current Red Schedule). The Red Timer is used to supervise the Red Threads deadline.

Execution Time Measurement and Supervision

Rubus OS provides services to measure and supervise the execution time of Red and Green Threads. These services require a high-resolution hardware timer to work. The user sets the resolution of this timer and it must be specified in Rubus ICE in order for Rubus ICE to generate correct code and calculate the execution times correctly. To integrate execution time measurement and supervision with Rubus OS please see *halBsExecTimeStart()* and *halBsExecTimeStop()*.

7 Analysis Services

The resource allocation of the application and the Rubus OS services is done pre-run-time requiring the user to have full control of the allocated resources. To facilitate this approach feedback information about the run-time behaviour is required.

Rubus ICE contains services for resource allocation, facilities for fetching information and analysis of the real-time information.

All appropriate resource allocation parameters are monitored in run-time by Rubus OS as maximum stack usage for each thread, execution time for Red threads and for interrupts and peek usage of a resource as queues,

To facilitate a trace of the run-time behaviour of the target system an Event Log is supported.

A communication protocol, the Rubus XHP protocol (please see the document Rubus XHP describing the protocol) is provided to communicate between Rubus ICE and the Rubus application running on the target. Using this communication link the run-time information may be displayed in Rubus ICE.

Communication Protocol Services

The communication protocol is defined to handle fast half duplex communication, the host (Rubus ICE) as the master and the target(s) connected on the bus as slaves.

The size of the protocol buffer in the target is user defined and set in Rubus ICE.

In the target, the execution of the protocol is implemented either as a Blue thread or by polling. It is user defined which method to use. In a system containing both Red and Blue services, it is advisable to use a Blue thread to handle the protocol. In a Red system, the polling approach is applicable.

Event Log Services

Services for logging Basic, Red, Green and Blue run-time events are included in Rubus OS. Since each event is logged with a time stamp, this service requires a high-resolution hardware timer to work. The user must supply this timer and set its resolution to an appropriate value. This resolution must be specified in Rubus ICE. The hardware timer is adapted to Rubus OS using the *halBsFreeRunTimerGet()* function

The size of the Event Log buffer needed to store the events is defined in Rubus ICE. To increase log performance the size of the buffer is rounded according to the formula $2^n - 1$ where n is the number of log entries, for example, 127, 255, 511 and so on. Each log event stores an id, an event and a timestamp in the buffer. The amount of bytes the log buffer requires is dependent on the target. For example; for a 32-bit target each log event requires 8 bytes.

Rubus ICE contains functions for defining which object to log and to start and stop the logging. The content of the Event Log Buffer is transferred to the host via the Rubus XHP protocol. Rubus ICE contains services for presentation of the log information.

The user may control when to start and stop logging from the application program in order to log a certain state of the application. To support this the following services are available: *bsLogReset()*, *bsLogSet()*, *bsLogStart()* and *bsLogStop()*.

8 Green Kernel

Interrupt Handling

Interrupts are asynchronous events related to the normal program execution of threads. As one extreme, interrupts should be avoided in a dependable real-time system since interrupts cause non-determines behaviour of the system. Due to the current design of micro-controller and a common way of designing hardware interrupts are a reality and must be supported.

In the area of real-time systems, devices typically have very simple interface that can be or should be operated without a full-fledged driver within the operating systems. What is needed is the ability to access the control and status register of the interface, the ability to capture the interrupts that are generated, and the application program handles the interrupts.

The design of the interrupt handling in Rubus OS is based on our requirement to simplify porting to various types of processors as well as various programming environments.

The design of Rubus OS is such that locking of interrupts are done only in situations, which require atomic operation due to interrupt handling.

In Rubus OS the *Green Kernel* manages interrupt processing.

Rationale

The *Green Kernel* manages the execution of *Green Threads* (interrupt handlers). The Dispatch Unit in the Green Kernel is the target processors interrupt logic and thus its behaviour is target dependent. A Green thread pre-empts execution of Red and Blue threads in relation to the processors interrupt logic.

The adherent code of a Green thread is the entry function of the thread. Invoking the entry function of the thread starts the execution. When the end statement of this entry function is reached, the thread is terminated implicitly. The entry function may have an optional argument that can be used to contain the state variables of the thread.

A Green thread can be classified to use Blue services as sending a signal to a Blue thread. Such thread body shall contain entry and exit functions to manage the Blue pre-emption control. The entry and exit function calls are generated by Rubus ICE.

Measurement and supervision of maximum, minimum or average execution time (WCET/BSET/ASET) of a Green thread is an option. If the execution time of a Green Thread exceeds the maximum execution time set the Green Error handler will be called. A user-supplied timer measures the execution time.

A Green Thread is defined by its attributes and its entry function.

A *Green Thread* in Rubus OS is an object having the following attributes:

- A Green Thread consists of program code and may have associated a memory space containing run-time information.
- A Green Thread has the processors attention according to target processors interrupt logic.
- A Green Thread object is statically allocated (as all objects in Rubus OS).
- A Green Thread has an interrupt level associated.

- Maximum, minimum or average execution time, to be supervised in run-time as an option.

Green Interrupt Control

Green Atomic Operations are defined such that Green interrupts are locked during the atomic operation. The atomic operation is used internally by Rubus OS during a sequence of operations, which must be atomic.

For processors supporting several interrupt levels the Green Ceiling Priority Level is calculated by Rubus ICE based on the priority level specified for all the Green threads as well as for the Basic Timer. The ceiling priority is equal to the highest specified level.

A set of functions is provided to control the interrupt locking and unlocking. The function *greenIntrDisable()* or *greenIntrSuspend()* locks interrupts by setting the interrupt level of the processor to the Green Ceiling Priority Level and *greenIntrEnable()* function unlock interrupts.

The function *greenIntrSuspend()* locks also interrupts by setting the interrupt level of the processor to the Green Ceiling Priority Level also returning current interrupt level and *greenIntrResume()* function sets the interrupt level to the specified value. This set of functions is utilised as well internally by Rubus OS.

Blue Notification

The Green to Blue communication in Rubus OS utilises the signal concept and the services for handling interrupt notifications are in conformance to the services for handling signals.

If the Green thread associated to the interrupt has enabled Blue services in Rubus ICE, a signal can be sent from the Green thread to the user code (the body of a Blue thread) for notification of the interrupt.

9 The Red Kernel

The Red Kernel contains services related to time driven execution. The Red Kernel contains services to select a Red Schedule and to manage execution of Red Threads. The HAL layer contains services that need to be adapted to the user's environment (Basic Timer interrupt control) (See Hal Services).

Red Thread

Rationale

A thread is a single sequential flow of control within a process (POSIX). A *Red Thread* is a single sequential flow of control within the *Red Process*. The Red Kernel manages the execution of Red Threads. Red Threads are allocated statically pre-run-time.

A Red application consists of a number of *Red Threads* grouped into a *Red Schedule.* Several schedules may coexist in a system. The Red Kernel contains services for switching between different schedules in run-time.

The adherent code of a Red thread is the entry function of the thread. Invoking the entry function of the thread starts the execution.

Execution of Red thread is defined as Single Shot Execution (SSX). Execution of each function begins at its the entry point and is completed at reaching its end statement the thread becomes then terminated implicitly. A Red thread is not allowed to block its execution such as waiting for a resource to be free.

The entry function may have an optional argument that can be used to contain the state variables of the thread, since all Red Threads share the same stack.

Measurement and supervision of maximum, minimum or average execution time (WCET/ BSET/ASET) of a Red Thread is an option. If the execution time of a Red Thread exceeds the maximum execution time or is below the minimum execution time set, the Red Error handler will be called. A user-supplied timer measures the execution time.

A Red Thread is defined by its basic attribute. A Red Schedule consists of instances of Red Threads.

A *Red Thread* in Rubus OS is an object having the following attributes:

- A Red Thread consists of program code and may have associated a memory space containing run-time information.
- A Red Thread is part of a Red Schedule, statically allocated.
- A Red Thread has the processors attention according to a Red Schedule.
- A Red Thread object is statically allocated (as all objects in Rubus OS).
- A Red Thread cannot be blocked from execution.
- Maximum, minimum or average execution time, to be supervised in run-time as an option.

A Red Thread utilises Red Run-Time Services.

Red Schedule

Rationale

A *Red Schedule* is composed of sets of Red Threads instances. Each set contains a number of Red Threads having the same *release time*, thus the execution of all the Red threads included into a set are done in sequence. The Red Schedule defines the running behaviour of all the Red threads included. A Red Schedule is designed statically pre-run-time. The execution of a Red Schedule is repeated with the *period time* specified.

All the time values in a Red Schedule are given in a number of timer ticks. The user defines the period time for the timer tick.

The *deadline* of a Red Thread instance defines when the execution of the Red thread shall be completed. The *Red Kernel* supervises the *release time* and the *deadline* at run-time. Both *release time* and *deadline* are specified relative the beginning of the schedule. The *Red Schedule Timer* measures the time during a period and is reloaded with zero when the timer expires. All the time values within a Red Schedule must be <= the period time.

If the deadline is missed or the maximum or minimum execution time is exceeded the Red Error handler is called (see *redErrorHandler()*).

The Red Kernel supports pre-emption between threads. A Red Schedule can be composed of Red threads, which pre-empts another thread, the release time of a thread may occur during execution of another. The maximum pre-emption level is target dependent.

All Red threads share the same stack to minimise the memory need of an application. When pre-emption is selected this requires, of course, that the thread which pre-empts another thread must finish its execution before the other thread can be resumed again.

The pre-run-time scheduler shall manage precedence relations and exclusion relations between Red threads. The Red Schedule is designed such that the executions of the threads are ordered according to the precedence relationship between the threads.

In a Red & Blue system, periodic notification of the Blue Kernel shall be included in all Red schedules.

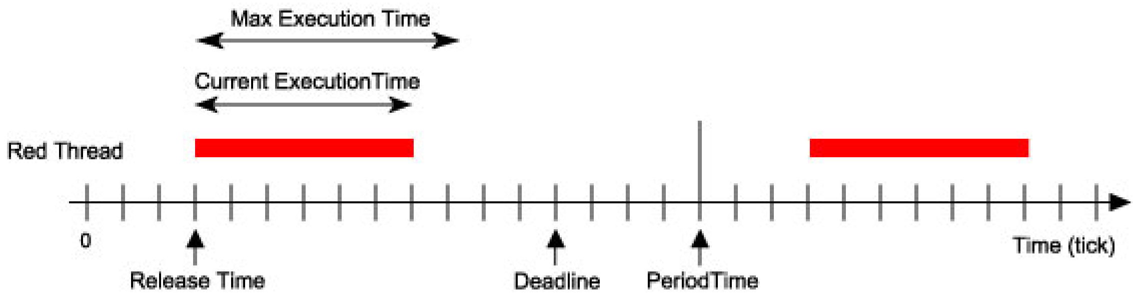

Figure 4 - Red Schedule

10 The Blue Kernel

Rationale

The *Blue Kernel* contains services for management of Blue threads and the execution of the threads, services for co-operation between threads such as synchronisation and message passing, memory management.

The Blue Kernel manages the spare execution time, not utilised by the Red Kernel, for execution of Blue threads. The scheduling of Blue threads is done run-time based on the selected Scheduling Policy.

The Blue Kernel may be seen as managing the background process, the *Blue Process*, or to be part of the Blue Process.

The Blue Kernel requires for its operation two internal Blue threads, the *Blue Kernel Thread* and the *Blue Idle Thread*.

The Basic Layer contains the primitives related to a basic configuration of the Blue Kernel. Services for management of Blue threads, handling signals including interrupts, the Blue Clock and time supervision of Blue threads. The Basic Layer contains the following services:

- Management of Blue Thread.
- Management of Blue Signals.
- Management of the selected scheduling policy.
- Management of Blue Clock and Timer.

The Co-operation Layer consists of a set of services for synchronisation of resources and communication between Blue threads. These services are optimal and are aimed at more complex operations than the Basic Layer services. The Co-operation Layer consists of the following services:

- Management of Mutex.
- Management of Message Queues.

Blue Thread

Rationale

A Blue thread is a single sequential flow of control within a process (POSIX). In Rubus OS a *Blue Thread* is a single sequential flow of control within the *Blue Process*. The Blue Kernel in the Blue Executive manages execution of Blue Threads. Blue Threads can be created and terminated dynamically during run-time.

Each thread has its attribute statically defined, pre run-time. To each Blue thread a control block called the *Blue Thread Control Block* is associated, containing run-time information defined in the attribute of the thread. Blue threads may have its own run-time stack, called the *Blue Thread Run-Time Stack* or share a common Blue stack, called *Blue Shared Run-Time Stack.* The memory areas for these entities are allocated statically (pre-run-time).

The code associated with a Blue thread is called the *Blue Thread Code Body*. The entry to the code body of a Blue thread is defined into the Blue Thread Attribute. Invoking the entry function defined in the attribute of the thread starts the execution of a created Blue thread. When the end statement of this entry function is reached, the thread is blocked, waiting for the *Activation Signal*.

A *Blue Thread* in Rubus OS is an object having the following attributes:

- A Blue Thread consists of program code, utilising a run-time stack and memory space containing the run-time information.
- A Blue Thread object is statically allocated (as all objects in Rubus OS).
- A Blue Thread gains the processor attention according to scheduling policy and the priority of the thread.
- A Blue Thread can be temporary suspended from being involved in the scheduling.

Blue Thread Types

Two kinds of Blue threads are supported one of traditional type that have its own stack and one that share a common Blue stack.

By sharing a common stack among a set of Blue threads minimise the stack memory space and simplifies the estimation of stack size.

Own Stack

A Blue thread of type _R_THREAD_TYPE_STACK may be blocked waiting for an event and the thread code body may contain an infinite loop. Such Blue thread may utilise all Blue services.

Share Stack

A Blue thread of type _R_THREAD_TYPE_NO_STACK shall not contain any service calls that may block the execution of the code or contain an infinite loop.

Execution of such a Blue thread is defined as Single Shot Execution (SSX) named *Blue SSX Thread*. Such a Blue thread is not allowed to block its execution such as waiting for a resource to be free.

Blue Thread Scheduling

A Blue thread can be in one of the following state:

- *Dormant*, the thread is not alive, not created or is terminated.
- *Ready*, the thread is unblocked and is waiting to be the running thread.
- *Blocked*, the thread is blocked, waiting for a signal.
- *Running*, the thread has got the processors attention according to scheduling policy and the priority of the thread.
- *Pre-empted,* the thread is pre-empted by another thread of higher priority.
- *Suspended*, the thread is temporary not involved in the scheduling, is alive.

Scheduling Policy

The scheduling semantics described are defined in terms of a conceptual model which contains a set of *Blue Thread Lists*. There is, conceptually, one Blue Thread List for each priority. Any Blue thread in state *Ready* may be on any Blue Thread List. Each non-empty Blue Thread List is ordered, and contains a head and a tail.

The purpose of a scheduling policy is to define the allowable operations on this set of lists. An associated scheduling policy and priority shall control each Blue thread.

Associated with each policy is a priority range. The number of priorities is limited. The numeric value 0 defines the lowest priority and the highest numeric value defines the highest priority. Each Blue thread has a priority associated specified into the attribute of the Blue thread.

The thread that is defined at the head of the highest priority non-empty schedulable list shall be selected to become the *Running* thread, regardless of its associated policy. This thread is then removed from its Thread List.

The *pre-emptive scheduling* strategy shall be implemented. At a given time, the Blue thread with the highest priority is the one executing. If a thread becomes in state *Ready* and has higher priority than of the running thread then the running thread is temporarily suspended so that the high priority thread may proceed.

FIFO Scheduling Policy

The scheduling policy defined in Rubus OS for Blue thread scheduling is the FIFO Scheduling Policy.

Blue threads scheduled under the FIFO Scheduling Policy are chosen from a Blue Thread List that is ordered by the time its Blue thread have been on the list, first in first out.

The Blue thread that is defined at the head of the highest priority non-empty schedulable list shall be selected to become the running Blue thread in state *Running*.

When a running Blue thread becomes a pre-empted Blue thread, it becomes the head of the Blue Thread List for its priority in state *Ready.*

When a blocked Blue thread becomes a Blue thread in state *Ready*, it becomes the tail of the Blue Thread List for its priority.

When a running Blue thread issues the yield function, the thread becomes the tail of the Blue Thread List for its priority in state *Ready*.

Round Robin Scheduling Policy

Threads scheduled under the Round Robin Scheduling Policy are handled according to FIFO Scheduling Policy with the additional condition that when a running thread has been executing for a time period of specified length or longer, the thread becomes the tail of the Blue Thread List for its priority in state *Ready* and the head of the Blue Thread List of the priority is removed and becomes a running thread.

Blue Kernel Mode

Blue Kernel Mode is defined such that re-scheduling of threads is disabled (pre-emption). Interrupts are not locked out during execution in Blue Kernel Mode, but a signal send from a Green thread will be registered and will be served when the Blue Kernel leaves the Blue Kernel Mode. The Blue Kernel Mode covers internal operations to define a stable internal state.

Blue Kernel Thread

The *Blue Kernel Thread* manages time supervision and is resumed periodically when the Blue Timer expires by sending a signal to the Blue Kernel Thread. The Blue Kernel Thread copies the Basic Clock to the Blue Clock at the time the Blue Kernel is processing the Blue Timer notification. This thread is assigned the highest priority. Rubus ICE builds the attribute and the control block of the Blue Kernel Thread. The user controls the stack size.

Blue Idle Thread

The *Blue Idle Thread* is running when no other Blue thread is ready for execution. This thread consumes resuming processor capacity not used by the Red Kernel. The thread is assigned the lowest priority. The Blue Idle Thread is normally used for background processing as monitoring of the system. This thread shall not utilise any blocking services. The Resource Designer of Rubus ICE builds the attribute and the control block of the Blue Idle Thread. The user controls the stack size and supplies the code body of the thread. The user supplies the function that constitutes the code body of the Blue Idle Thread.

Blue Thread Management

Rationale

A Blue thread is a single sequential flow of control within a process (POSIX). In Rubus OS a *Blue Thread* is a single sequential flow of control within the *Blue Process*. The Blue Kernel in the Blue Executive manages execution of Blue Threads. Blue Threads can be created and terminated dynamically during run-time.

Blue Signals

Rationale

Signal services are included into the Basic Layer of the Blue Kernel. Signals provide the ability to pause execution waiting for a selected set of events and handle the first event that occurs. This allows applications to be written in an event-directed style similar to a state machine. Signals are the fundamental mechanism in Rubus OS utilised to synchronise activities and to signal events within the Rubus OS. Signals also provide a compact set of services to be used in a minimal version of Rubus OS.

A signal is said to be *generated* for (or sent to) a thread when the event that cause the signal first occurs. A signal is said to be *delivered* to a thread when the appropriate action for the thread and signal is taken. A signal is said to be *accepted* by a thread when the signal is selected and returned by the signal wait function. Each signal can be considered to have a lifetime beginning with *generation* and ending with *delivery.*

During the time between the generation of the signal and its delivery or acceptance, the signal is said to be *pending*.

A signal can be *blocked* from delivery to a thread. If the receiving thread has blocked delivery of the signal, the signal remains pending on the thread until the thread unblocks delivery.

A signal is delivered to a thread, which has indicated interest in handling the signal. Interest is expressed by being blocked waiting for the signal which upon arrival unblocks the signal.

Signals are always related to a Blue thread. Each Blue thread has a signal mask that defines the set of signals currently *blocked* from delivery to it and a signal mask that defines the set of signals currently *pending* on the thread.

All signals of a thread are defined *blocked* at creation.

The implementation does not support queued signals.

A set of signals is reserved for the internal use of the Blue Kernel and may not be used by the user. This set of signals is defined by the macro R_SIGNAL_RESERVED in the header <blue/b_signal.h>.

Blue Mutex

Rationale

The mutexes and condition variables are the synchronisation mechanisms between threads within one process specified in POSIX. Much experience with semaphores shows that there are two distinct uses of synchronisation: locking, which is typically of short duration, and waiting, which is typically of long or unbounded duration. In Rubus OS, condition variable synchronisation mechanisms are not supported.

A Mutex is a short name for a Mutual Exclusion Semaphore.

A *Mutex* in Rubus OS is an object having the following attributes:

- A Mutex is a type of binary semaphore.
- A Mutex has an owner. The owner is the Blue thread performing the lock operation.
- A Mutex can be locked and unlocked. Only the owner can perform unlocking.
- A Mutex uses the Priority Ceiling Protocol to avoid priority inversion.
- A Mutex object is statically allocated (as all objects in Rubus OS).

The identifier to an initialised Mutex is the address to the Mutex Control Block, defined in the attribute of the Mutex.

Priority Inversion

Priority inversion occurs when a higher priority thread is forced to wait for the completion of a lower priority thread. Such a situation is in conflict to a strict priority based scheduling. To avoid unbounded priority inversion, the Mutual Exclusion semaphore supports the Priority Ceiling Protocol.

Priority Ceiling Protocol

Priority Ceiling Protocol guarantees that a lower priority thread blocks a higher priority thread only once. Each Mutex is assigned a priority, called the ceiling priority. The ceiling priority of a Mutex is the minimum priority level at which the critical section guarded by the Mutex is executed. The ceiling priority shall be set to a priority higher than or equal to the highest priority of all the threads that may lock the Mutex.

When a thread locks a Mutex, its priority is automatically raised to the ceiling priority for that Mutex. At releasing the Mutex the priority of the thread is set to its previous value.

If a thread locks several Mutex in sequence, the unlocking sequence shall be in LIFO order (Last In First out) to preserve the execution priority correct. The Mutex locked last shall be the one first unlocked.

To define the ceiling priority requires that the user do a static analyse of all threads that may own the Mutex to find the highest priority among these threads and assign this priority level as the ceiling priority of the Mutex.

Thread Termination

A thread possessing a Mutex may be terminated without releasing the Mutex. This situation may cause unexpected results when other threads are blocked waiting for the Mutex.

In the current version of Rubus OS, unlike in POSIX, termination of a Blue thread owning a Mutex causes the Mutex to stay locked that may cause unexpected results. As safety critical real-time systems are normally static configured, termination of a thread is rare.

Ownership

A Mutex requested by a thread is linked to the thread requesting the semaphore. A Blue thread that becomes the *owner* of a Mutex is said to have *acquired* the Mutex and the Mutex is said to have become *locked*. One thread locking a Mutex owns the Mutex and must also be the one to unlock it. At any point in time discernible to the application, there shall be at most one owner of a Mutex.

A Blue thread becomes the owner of a Mutex when control returns successfully from a *lock operation* with the Mutex identifier as the argument.

A Blue thread remains the owner until the *unlock operation* returns successfully.

Blue Semaphore

Rationale

Semaphores are a high-performance process synchronisation mechanism in POSIX. In Rubus semaphores are used for synchronisation between Blue threads.

When a semaphore is created, an initial state for the semaphore has to be provided. This value is a non-negative integer. Negative values are not possible since they indicate the presence of blocked Blue thread.

A semaphore is defined as an object that has an integral value and a set of blocked Blue thread associated with it. If the value is positive or zero, then the set of blocked Blue thread is empty. otherwise, the size of the set is equal to the absolute value of the semaphore value. When a semaphore value is less than or equal to zero, any Blue thread that attempts to lock it again will block or be informed that it is not possible to perform the operation.

A semaphore may be used to guard access to any resource accessible by more than one schedulable task. It is a global entity and not associated with any particular Blue thread. A Blue thread that wants access to a critical resource has to wait on the semaphore that guard the resource. When the semaphore is locked on behalf of a Blue thread, it knows that it can utilise the resource without interference by any other cooperating Blue thread. When the Blue thread finishes its operation of the resource, leaving it in a well defined state, it post the semaphore, indicating that some other Blue thread may now obtain the resource associated with the semaphore.

A Semaphore is a simplified synchronisation mechanism in respect to Blue Mutex, to be used in a context where the locking and unlocking may be controlled from different threads and no support to avoid priority inversion is convenient.

A *Semaphore* in Rubus OS is an object having the following attributes:

- A Semaphore is a type of binary semaphore.
- A Semaphore has no owner.
- A Semaphore can be locked and unlocked.
- A Semaphore object is statically allocated (as all objects in Rubus OS).

The identifier to an initialised Semaphore is the address to the Semaphore Control Block, defined in the attribute of the Semaphore.

Thread Termination

A thread possessing a Semaphore may be terminated without releasing the Semaphore. This situation may cause unexpected results when other threads are blocked waiting for the Semaphore.

In the current version of Rubus OS, unlike in POSIX, termination of a Blue thread owning a Semaphore causes the Semaphore to stay locked an may cause unexpected results. As safety critical real-time systems are normally static configured, termination of a thread is rare.

Ownership

A Semaphore has no owner and can be locked by one thread and unlocked by an other.

Blue Message Queue

Rationale

In Rubus OS, Blue Message Queues provide a facility for communicating between Blue threads. By definition, Blue threads work in a shared memory space, but there are a number of advantages to supporting the message queue concept in POSIX for threads. By copying the message to and from the queue, the requirement of dynamic buffer management is eliminated, at least in respect to message passing. Static allocation of messages leads to a robust design; that is a resource adequate design related to message passing. The negative side is that the message queue have to be dimensioned in respect to the maximum number of messages queued. On the other hand, a safety critical application must be designed resource adequate.

Another advantage of the message queue concept is that an implementation supporting MMU can utilise the message queue concept for communication between memory segments.

A *Message Queue* in Rubus OS is an object having the following attributes:

- A Message Queue contains messages of a fixed size.
- A Message Queue supports copying messages to and from the queue.
- A Message Queue object is statically allocated (as all objects in Rubus OS).
- A Message Queue has no owner. Different threads can do open and close operations.
- A Message Queue has no message prioritisation mechanism, messages are handled FIFO.

The identifier to an initialised Message Queue is the address to the attribute of the Message Queue.

Fixed Size of Message

The interface imposes a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis.

The purpose of the fixed size is to increase the ability of the system to optimise the implementation. With fixed size of messages and fixed numbers of messages, the message queue, its descriptor, and buffers can be pre-allocated. This eliminates significant amount of checking for errors and boundary conditions. Additionally, a highly tuned implementation can optimise data copying to maximise performance. Finally, with a restricted range of message sizes, an implementation is better able to provide deterministic operations.

Prioritisation of Messages

Message prioritisation permits the application to determine the order in which messages are received. The major purpose of having priorities in message queues is to avoid priority inversions in the message system that is where a high-priority message is delayed behind one or more lower-priority messages. The prioritisation does add additional overhead to the message operations in those cases it is actually utilised.

The Message Queue services in Rubus OS are optimised for the FIFO operation of messages.

Receive Waiting List

When the Message Queue is empty, the user may wish to wait for a message to arrive. The Receive Waiting List contains all Blue threads waiting for the arrival of a message. The Blue thread at the head of the list is unblocked at the arrival of a message to the Message Queue. The ordering of the Receive Waiting List is priority based; the thread with the highest priority is inserted at the head of the list.

Send Waiting List

When the Message Queue is full, the user may wish to wait for a free message to arrive.

The Message Queue services in Rubus OS do not support a Send Waiting List. If the Message Queue is full, the sending cannot be completed successfully.

Ownership

The Message Queue has no owner. Different threads can do open and close operations. In accordance with the message queue having no owner, there is no action taken at termination of a Blue thread, which has opened a Message Queue.

11 References

[Lap92] J.C. Laprie. "Dependability: Basic Concepts and terminology" in Dependable Computing and Fault-Tolerant Systems. Vol 5, Springer Verlag 1992.

[BASEMENT95] Hans A. Hansson, Harold W. Lawson, Mikael Strömberg, Sven Larsson. BASEMENT, a Distributed Real-Time Architecture for Vehicle Applications. Mars 1995.

[IFAC95] C. Eriksson, K-L Lundbäck, H. Lawson, "A Real-Time Kernel Integrated with an Off-Line Scheduler", Accepted to 3rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control, Ostend-Belgium, May 95.

Basement

Basement is a vehicle internal real-time architecture developed in the Vehicle Internal Architecture (VIA) project, within the Swedish Road Transport Informatics Programme. The objective has been to design a platform that meets the stringent demands of the automotive industry. *Basement* is a pilot for future vehicle internal distributed real-time systems. As such, it is required to provide

- A communication infrastructure, allowing cost-effective communication between physically distributed units.
- An execution platform for application software, providing *guaranteed* services for safety critical applications, while providing *acceptable* response times for non safety critical applications.
- Resource sharing, i.e. *Basement* should allow multiple vehicle internal applications to efficiently share (multiplex) communication infrastructure as well as computing resources (processors).
- A priori predictability, i.e. it should be possible to off-line (before runtime) determine if sufficient resources are available to guarantee required behaviour (e.g. that deadlines of safety critical applications are always met).
- Reliability, i.e. the probability of a system failure should be very low (in the range of 10^{-8} faults/hour/car).
- Facilities for communication with vehicle external equipment.
- Open interfaces, i.e. the interfaces, connectors, and communication protocols should be precisely defined. This is to allow different vendors to develop *Basement* compatible equipment, and to facilitate the integration of components from different vendors in a *Basement* system.
- An application development environment and methodology, providing engineers with an application oriented interface, as well as tools for efficient development and integration of applications.
- An architecture which allows large product series to be implemented at a very low cost.
- Simplicity, both in terms of minimal run-time overhead (i.e. minimal amount of *non-productive code),* and in terms of a simple and intuitive method for application development. This simplicity facilitates validation and formal proof of correctness.

Basement has been designed to meet all of the above requirements.

The *Basement* real-time architecture is based on the collective efforts of the *Basement* design team, with members from the following organisations: Mecel AB, Arcticus Systems AB, Swedish Institute of Computer Science, Lawson Förlag & Konsult AB, Chalmers Univ. of Technology (Dept. of Computer Engineering), and Uppsala University (Dept. of Computer Systems).