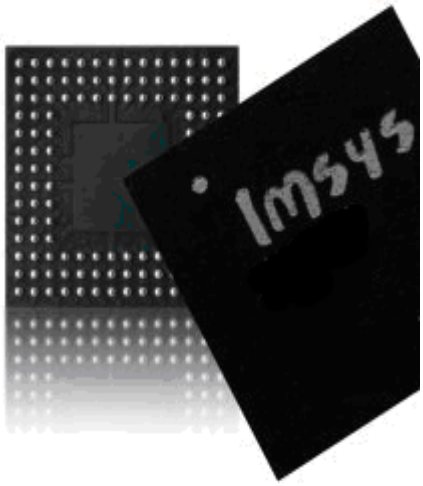
IM3910 Microcontroller - Datasheet

Datasheet

IM3910 Microcontroller

Revision 1.3



Imsys Technologies

Imsys Technologies AB, Sweden
Johanneslundsvägen 3
SE-194 61 Upplands Väsby
Sweden
+46 8 594 110 70 phone
+46 8 591 110 89 fax

Imsys Technologies, USA
9903 North Poetry Ln
Terrell, TX 75160
USA
+1 877 775 1627

838 Henderson Ave
Sunnyvale, CA 94086
USA

VAT:
556453-3247-01

Web:
www.imsystech.com

Copyright

Disclaimer

Imsys Technologies AB makes no warranties for the contents of this document or the accuracy or completeness thereof and disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Imsys Technologies AB reserves the right to revise this document and to make changes without notifications to any person of such changes. The information contained in this document is provided solely for use in connection with Imsys Technologies AB products. This document should not be construed as transferring or granting a license to any intellectual property rights, neither expressed nor implied.

Imsys Technologies AB products have not been designed, tested, or manufactured for use in any application where failure, malfunction, or inaccuracy carries a risk of death, bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft, watercraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Revision History

Document: STO-DEV7028-AG	
Previous revision: 1.2	
Section	**Changes since last revision**
-	Updated version numbers in document to match profile version

Table of Contents

About this document

The IM3910 is a dedicated controller for networked applications in real-time environments. It combines the best features of traditional CISC architectures and the efficient use of hardware resources found in RISC processors. The microcoded architecture is enabled with a Java Virtual Machine, and hence provides the developer a bridge between high-level application development and gate-level execution performance.

The IM3910 integrated circuit comes with a complete software platform including real-time operating system, JVM, flash file system, TCP/IP communication stack, Telnet, FTP and web server. The application designer can mix and match between Java, C and assembler languages to reach optimal performance.

This document describes the software platform for the IM3910 Microcontroller.

1. The Imsys JVM

Imsys Java virtual machine, JVM, is a small and efficient virtual machine for Java2 Micro Edition (J2ME). The implementation is a porting and adaptation of the KVM virtual machine from Sun Microsystems. Many parts of the original KVM code has been modified or completely rewritten to take advantage of the IM3000 processors' unique Java capabilities.

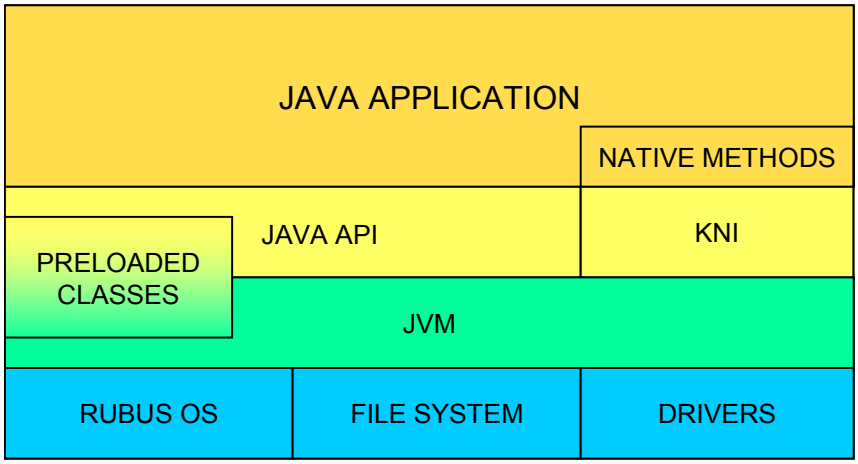


Figure 1: Java Runtime Environment

1.1. Running Java Applications

The JVM is started by the Imsys Shell (ISH) during the system startup. There can in be only one instance of the JVM running at the same time but several Java applications can run simultaneously sharing the same Java heap.

Java applications are started either automatically by ISH at system startup or explicitly by the user from the ISH command line. For more information about ISH see chapter 8.

1.2. Java API

The base of the Java API is the Connected, Limited Device Configuration (CLDC) version 1.0. The API also contains a subset of JDK 1.1.8 and the javax.comm package.

A number of Imsys specific classes that give the Java applications access to system resources are also available.

The different groups of API classes:

- CLDC 1.0
- Subset of JDK 1.1.8
- javax.comm
- Imsys specific classes

1.2.1. Imsys Java API

The Imsys specific Java API contains a number of packages that gives access to system resources not available from the standard APIs.

Table 1: Package se.imsys.comm

Class	Description
CAN	This class gives access to the CAN bus.
I2C	This class implements an I2C bus master interface.
SPI	This class implements a SPI bus master interface.

Table 2: Package se.imsys.net

Class	Description
DNSClient	Contains methods to resolve hosts.
FtpClient	Implements an FTP client.
FtpServer	Implements an FTP server.
HttpServer	Implements a simple web server according to RFC 2616 HTTP 1.1.
SmptClient	This class enables the sending of E-mails through a SMTP server.
TftpClient	Implements a TFTP client.

Table 3: Package se.imsys.ppp

Class	Description
PPP	Contains methods to set up a PPP connection.

Table 4: Package se.imsys.system

Class	Description
DataIO	This class gives low level access to the processor's native I/O bus.
EventLog	Implements an operating system event object that can be used for application profiling.
FileSystem	This class gives access to file system functions such as mounting SD/MMC cards.
Ish	This class gives access to some Imsys Shell resources.
MDIO	This class makes it possible to control the MDIO interface from java.
PortIO	This class gives access to the general purpose I/O ports on the processor.
RTC	This class gives low level access to the RTC.

1.3. Native Bytecode Execution

The Java enabled processors in the IM3000 microcontroller family has beside the standard C/Assembler instruction set a second instruction set, named ISAJ, containing Java bytecodes. This instruction set contains about 85% of the bytecodes in the standard Java bytecode instruction set. These instructions are executed as machine instructions and are therefore very fast and efficiently executed.

See chapter Appendix B for a listing of ISAJ.

The Imsys JVM has no bytecode interpreter. Instead, all bytecodes in ISAJ are executed as native machine instructions and the other bytecodes are trapped by the hardware and emulated in assembler. It is some of the most complex bytecodes, like the different invoke bytecodes, which are emulated.

The execution of some of the emulated bytecodes is enhanced by the use of quickcodes. When an emulated bytecode is executed the first time the JVM saves some information about the execution environment in the actual program location and then replaces the emulated bytecode with a faster quickcode. The next time the program reaches this location, the faster quickcode will be executed.

The processor switches automatically between the different instructions sets.

1.4. K Native Interface - KNI

The K Native Interface (KNI) is designed for J2ME by Sun Microsystems for resource-constraint devices and can be described as a “logical subset” of the Java Native Interface (JNI). KNI enables the programmers to create native Java functions in C and assembler.

Native functions can be used to get access to system functions not available from the Java APIs.

Native functions can also be used to speed up execution in time critical parts of a Java application. Even if most of the Java byte codes are executed as native processor instructions on the IM3000 processors, it is possible to increase the performance by implementing functions in C and assembler. This is due to the fact that some of the Java byte codes are rather complex and therefore more time consuming that equivalent instructions in C, e.g. the Java invoke codes vs. the C subroutine call instruction.

1.5. Preloaded Classes

To speed up the startup time for Java applications, all classes in CLDC are preloaded. This means that the CLDC classes were loaded at compile time and are linked directly in the JVM. Other classes in the API and user application classes are loaded by the JVM upon demand.

1.6. Preverifier

A class verifier like the one used in JDK are not suitable for small resource-constrained devices. It would require too much memory space and use a lot of CPU power. Therefore Sun has designed a class verifier for KVM that is significantly smaller than the JDK verifier. This verifier requires all subroutines to be inlined (no `jsr` or `ret` instructions are allowed) and the class files should contain a `StackMap` attribute before then can be loaded. This is accomplished by using a post-javac tool named *preverifier*, which inserts this attribute into normal class files.

The *preverifier* is delivered with Imsys Developer and can automatically be started by Imsys Developer after class compilation.

1.7. The Garbage Collector and the Heap

All Java objects created by the Java programs reside in the Java heap which is allocated in the RAM memory when the JVM is initiated. The allocated size of the Java heap can be specified in a system initialization file.

The garbage collector uses a mark-and-sweep algorithm to remove dead (unreferenced) objects from the Java heap. In the mark phase it marks all objects that are referenced to by the running Java threads and then during the sweep phase all unmarked objects are removed. The sweep phase also packs the live objects together to avoid fragmentation of the heap.

It runs as a separate thread and will automatically execute when the heap becomes full. It can also explicitly be run by a Java or C application. The garbage collector can only be interrupted by C threads with higher priority by interrupt routines. No Java threads are able to interrupt the garbage collector.

2. System Software Overview

The System Software consists of a number of tightly integrated components. Each software component implements its own piece of functionality and can provide an interface to other components and to application programs.

They could be logically organized into the following groups:

- Hardware drivers – handle the hardware-related operations and provide an access to hardware resources, such as DMA, interrupts, timers, etc.
- Operating System – supplies a real-time multithreaded operating environment for the rest of the system, including application programs.
- Communication drivers – provide a well-defined access to the facilities of different communication interfaces, such as COM, SPI, I2C.
- File Subsystem – supplies a file-based access to the data stored on different media types, such as FLASH memory, SD/MMC memory cards, RAM-disk.
- Network Subsystem – provides the support for TCP/IP network communications over Ethernet and serial links.
- Shell Environment – provides a UNIX-like command shell as well as Telnet and FTP services.

Below is a description of each System Software component.

3. Hardware Drivers

Hardware drivers are the thin layer of software, which hides the hardware-specific implementation details and provides a simple interface to the hardware resources. They are also responsible for low-level system initialization, such as setting pin directions, initializing DMA controller, installing low-level interrupt handlers and so on.

The System Software includes the system driver, a driver for the base hardware platform, and interface drivers, drivers for different hardware I/O interfaces:

- Ethernet
- Serial
- I2C
- SPI

Hardware drivers are mostly invisible for applications, as applications normally do not have a direct access to the underlying hardware, but uses facilities provided by higher level components, such as an operating system or a file system, to perform an I/O and other tasks.

4. Operating System

The operating system, named Rubus, is a major part of the system software. Along with hardware drivers it provides a simplified interface for using and managing hardware resources as well as an operating environment for system and application tasks.

4.1. Features

The Rubus OS is a real-time multithreaded operating system. It is designed for real-time systems mainly having soft real-time requirements.

The key features of the Rubus OS are:

- Object-based – all system entities are represented as objects.
- Multithreaded – shares processor time between several threads of execution.
- Event-driven – supports two types of event handling – interrupts and services – both can be involved for a processing of a single event.
- Priority-based – provides 256 priority levels, 32 levels for interrupts, 32 levels for services and 192 levels for threads.
- Preemptible – any thread can be preempted by another thread with a higher priority.
- Interruptible – any thread, interrupt or service can be interrupted by another interrupt or service with a higher priority.
- Synchronization – provides various mechanisms for threads synchronization.
- Event Logging – have a built-in event logging facilities.
- Error Handling – have a common error handling mechanism.
- POSIX Compatible – provides a programming interface to its services in compliance with the POSIX 1003.1 Standard.

4.2. Objects

The Rubus OS is not a pure object-oriented operating system, but all entities created within it, such as threads, events, mutexes, message queues are considered as objects. That means that their internal structure is hidden and all manipulations on them can be done only through the well-defined operating system interface.

Each object has a header which contains a set of common attributes, like the name, type of object, status flags, owner and the address of its control block. The control block is the object's body. It holds all data specific to each object type and necessary for the implementation of an object behavior.

For each type of object there is its own creation routine, which allocates and properly initializes the object's header and body and makes the object ready for use. Later objects are addressed by its Object Identifier, an integer value generated by the operating system at a time of object creation. After the object is not needed anymore it should be destroyed to release resources it used.

All objects are accessible either by its name or by its object identifier through the global Object Access Table. There are a number of general purpose object manipulation routines, which are common for all object types. They allow enumerating the object table by the object type, finding an object by its name, getting an object's name by its identifier, creating/destroying objects for event logging. The detailed description of object manipulation routines can be found in the C API Reference Manual.

4.3. Processes and Threads

A Process is an address space, one or more threads of control that execute within that address space and other system resources allocated by these threads during the process life-time.

A Thread is a single sequential flow of control within a process. Threads can be created and terminated dynamically and their execution is managed by a kernel. To each thread a run-time stack area and an executable code is associated. The entry to the code of the thread is specified when the thread is created. Invoking the entry function starts the execution of a thread. When the end statement of this entry function is reached, the thread is implicitly terminated.

Each thread within the Rubus OS is represented by a thread object, which as any other object has the common header and the type-specific body – Thread Control Block (TCB). The TCB contains the thread run-time information, such as thread's state and priority, as well as pointers to the thread's stack and code.

At any point of time a thread can be in one of the following states:

- DEAD - the thread is terminated or not initialized.
- BLOCKED - the thread is blocked, waiting for an event that unblock the thread.
- READY - the thread is ready for execution, waiting to be the running thread.
- RUNNING - the thread is currently the executing thread.

The thread changes its status implicitly, as the result of invoking a system service or as the result of an event.

The priority of a thread defines how "often" and how "soon" the thread will get the processor attention. It can be any value in range 64 – 255. The lower value means the higher priority. The highest and lowest priority levels (64 and 255 respectively) are reserved for system threads. See 4.6 for more information about thread scheduling.

At its startup the Rubus OS creates a System Process, in which context all other threads are created and executed, and two service threads:

- Idle Thread – the lowest priority (255) thread, which occupies the processor when no other threads are ready for execution.
- Kernel Thread – the highest priority (64) thread, which handles the thread cancellation and termination.

The Rubus OS provides a number of services for the thread management, such as thread creation and termination, getting and setting the thread's priority and

error code, retrieving the thread status information. See the C API Reference Manual for more information.

4.4. Interrupts and Services

Interrupt Events and Service Events are events asynchronous to the normal execution flow of the thread. Interrupt events are normally generated by the hardware, while service events are requested by the software.

The Rubus OS supports asynchronous events by introducing the Event Handlers. When an event occurs the processor temporarily suspends the current thread execution and branches to the corresponding event handler in order to service an event. An event handler is not bound to a thread directly. Instead it uses the interrupted thread stack while performing its operations.

The execution of events is priority-based, i.e. the highest priority event is selected for execution at a time. The priority levels 0 – 64 are reserved for events. Interrupt events utilize 32 higher priority levels (0 – 31) and service events uses 32 lower priority levels (32 – 63). The event handler for each level is stored within the Event Vector Table.

How events are executed is controlled by global and thread-specific hardware registers:

- EIR (Enable Interrupt Register) – global 32-bit register, contains the bit-mask of enabled interrupts.
- PIR (Pending Interrupt Register) – global 32-bit register, contains the bit-mask of pending interrupts.
- PSR (Pending Service Register) – global 32-bit register, contains the bit-mask of pending services.
- FLAGS: DI/EI (Disable Interrupts/Enable Interrupts) – thread-specific flag that temporary disables execution of all interrupts and services.
- SW: PLR (Priority Level Register) – thread-specific register that indicates on which priority level the thread is currently running.

When an interrupt is detected by the hardware, the EIR register is first checked to see whether such interrupt is enabled (1 in the corresponding bit position) or not. If it is enabled, the corresponding bit is set in PIR. When a service is requested by the software the corresponding bit is set in PSR.

If interrupts are not disabled by the DI flag, the highest priority interrupt (or service) is selected and PLR is checked to see if the current priority level is lower than the priority level of the interrupt (or service) to be serviced. If so, the current state is saved onto the interrupted thread stack, PLR is set to the priority level of the interrupt, interrupts are disabled by the DI flag and the execution is transferred to the corresponding event handler. After the event handler executes its last statement, the state is restored and the execution is returned back to the interrupted thread.

If a higher priority interrupt or service occurs during an execution of an event handler, it should be interrupted and a higher priority event handler should be

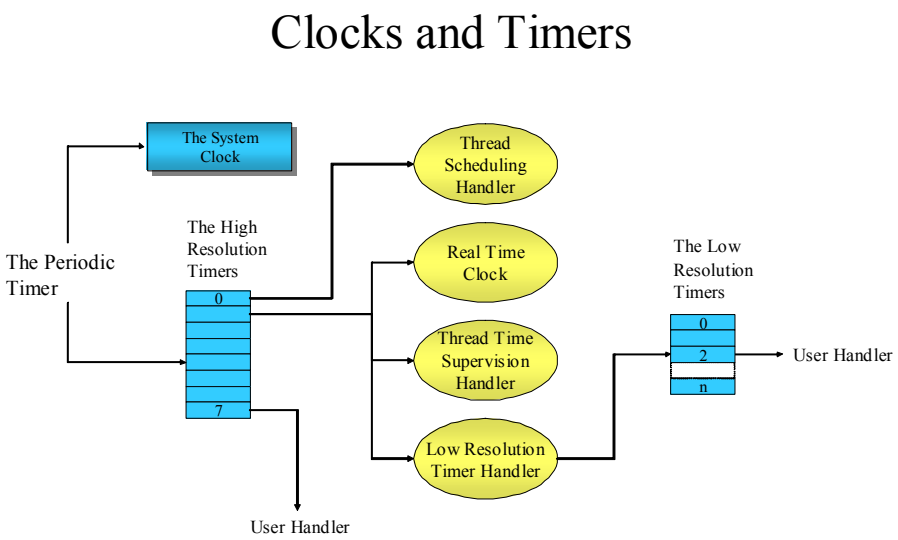
executed. It means that any event handler must enable interrupts as soon as possible to allow higher priority interrupts or services to be serviced.

The Rubus OS is installed as a Service Event Handler on the highest priority level (32) and calls to its interfaces are performed via a request for service.

So, an Interrupt Event Handler shall not utilize Rubus OS services. If an Interrupt Event Handler needs to use them, it shall associate a service event for that purpose and perform necessary operations there.

For detailed information on how to install and remove event handlers, configure periodic events and control interrupts see the C API Reference Manual.

4.5. System Clock and Timers



4.5.1. System Clock

The Rubus OS supports one System Clock measuring relative time. The System Clock is defined to be a clock representing the time for the system, the system time. The amount of time is measured in timer ticks.

The Periodic Timer implemented in hardware constitutes the source of the System Clock. The System Clock is incremented continuously when the periodic timer expires. The System Clock is reset at system start.

The time supervision of threads, thread scheduling and periodic events handling performed by Rubus OS is based on this clock.

4.5.2. Timers

The System Clock supports 8 high-resolution timers. For each timer there is a counter and a reload register associated. When the timer expires it is reloaded with the value of the reload register. If the reload value is not null, it is a periodic timer. The maximum expiration time value is 255 milliseconds (8-bits value) for each timer. Each of these timers has a handler associated that is called when the related timer expire. The handler is called via a subroutine call.

Two of these timers are reserved for internal use. One is used for measuring a thread execution time and implementing a round-robin scheduling policy. Another is for maintaining a Real Time Clock, supervising time-waiting threads and supporting periodic events.

See the C API Reference Manual for a detailed description on how to access system time, use high-resolution timers and periodic events.

4.6. Scheduling Policy and Thread Switching

The Rubus OS is a multithreaded operating system. So it supports more than one thread of execution and divides the processor time between them. It decides when and which thread to run based on the scheduling policy.

The scheduling semantics are defined in terms of Thread Lists. There is one thread list for each priority. Any thread in state READY must belong to one of them. Each non-empty 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. The basic scheduling principles are:

- The thread that is at the head of the highest priority non-empty schedulable list shall be selected to become the RUNNING thread. This thread is then removed from its thread list.
- At a given time, the thread with the highest priority is the one executing. If a thread becomes READY and has higher priority than the running thread then the running thread is temporarily suspended so that the high priority thread may proceed.

4.6.1. Priority Scheduling

Threads scheduled under the Priority Scheduling policy are chosen from a list that is ordered by the time its threads have been on the list, first in – first out.

- When the RUNNING thread becomes a preempted thread, i.e. interrupted by another thread having a higher priority, it becomes the head of the thread list for its priority in state READY.
- When a blocked thread becomes READY, it becomes the tail of the thread list for its priority.
- When the RUNNING thread issues the yield call, the thread becomes the tail of the thread list for its priority in state READY.

4.6.2. Periodic Scheduling

Threads scheduled under the Periodic Scheduling policy are released periodically. The period is the interval between successive unblocks of the thread.

- When released thread becomes a thread in state READY, it becomes the tail of the thread list of its priority.
- A thread scheduled under this policy shall have completed its execution before the next period.

The latest permissible completion time measured from the release time of the thread is called the Deadline. It shall be less or equal the period time.

4.6.3. Asynchronous Scheduling

This policy is often also called Round Robin Scheduling policy.

Threads scheduled under the Asynchronous Scheduling policy are handled according to Priority Scheduling Policy with the additional condition.

- When the RUNNING thread has been executing for a time period of specified length or longer, the thread becomes the tail of the thread list for its priority in state READY.
- The head of the thread list for the priority is removed and becomes the RUNNING thread.

4.6.4. Thread Switching

The Scheduler only selects when and which thread to run according to its scheduling policy rules, but the actual switching of threads is performed by the Dispatcher.

The dispatcher is implemented as the service event handler for a Dispatch Service event and has the lowest possible priority level (63). The scheduler requests the dispatch service in following situations:

- When it detects that the higher priority thread is ready for execution, so the currently running thread should be preempted out.
- It detects that the currently running thread is the subject for an asynchronous scheduling and it has exhausted its time-slice, so it should release the processor for another thread.
- A call to the operating system that should block or yield the current thread execution is issued.

The dispatch service is requested by setting the corresponding bit in PSR register. As soon as the execution priority level defined by PLR register becomes lower than the dispatch level, the dispatch event will take place and its event handler, the dispatcher, will be invoked.

The dispatcher first saves the running thread context information (registers and flags) in the thread's stack. Then it takes the highest priority thread from the ready thread list, restores its context and changes the state of the thread to

RUNNING. After returning from the dispatch service handler the execution is transferred to that thread and it continues from the point where it was stopped.

The Rubus OS provides system services that affect thread scheduling and switching. When the thread is created it is possible to specify which scheduling policy to apply for that thread, the time-slice interval (for asynchronously scheduled threads) and the initial priority of the thread. The priority can also be changed by the thread during its run-time. It is also possible to temporary lock (or disable) the preemption of the calling thread. If the preemption is locked the scheduling of running thread is prohibited until either the calling thread is blocked or the preemption is unlocked. The thread can also yield (or relinquish) the processor until it again becomes the head of its thread list.

See the C API Reference Manual for detailed description of these services.

4.7. Synchronization Mechanisms

One of the primary functions of any synchronization mechanism is to prevent data corruption. A principal cause of data corruption is a lack of synchronization between two or more concurrent threads of execution that manipulate the same data structures. Another function is to provide an ordering in execution of several threads doing the common job.

The Rubus OS provides a number of synchronization mechanisms with different behavior and semantic. The synchronization mechanism in terms of Rubus OS is an object and a set of services to manipulate with that object. Below is an overview of all supported mechanisms. The detailed description of threads synchronization services can be found in the C API Reference Manual.

4.7.1. Signals

Signals provide the ability to pause execution of the thread waiting for a selected set of events. This allows applications to be written in an event-driven style similar to a state machine.

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 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 acceptance. 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 disabled from delivery to a thread. If the receiving thread has disabled delivery of the signal, the signal remains pending on the thread until the thread accept delivery. Each thread has a signal mask that defines the set of signals currently disabled 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 disabled at creation.

When the thread issues a signal wait call specifying the set of signals it is waiting for, it is immediately put into a sleep state, if none of specified signals is already pending for that thread. Later, another thread can issue a signal send

call for that thread, thus waking it up and making ready for execution. The signal wait call can also be a subject of timeout, so the thread issuing this call will stay blocked until either it receives a signal or until the specified timeout interval expires. The thread can also examine at any time if any signals are already pending for it and clear them if necessary.

4.7.2. Mutexes

A Mutex is a short name for a Mutual Exclusion Semaphore. It is mostly used to protect a shared resource which allows only an exclusive access at a time. Before performing any manipulation on this resource a thread has to synchronize its access with other threads by locking the mutex. After the thread has finished with the resource it has to release or unlock the mutex, thus allowing other threads to access the resource.

A mutex is an object having the following attributes:

- A Mutex is a type of binary semaphore.
- A Mutex has an owner. The owner is the thread performing the lock operation.
- A Mutex can be locked and unlocked. Only the owner can perform unlocking.
- A Mutex uses a priority inheritance protocol to avoid priority inversion.

A thread becomes the owner of a mutex when control returns successfully from a lock operation with the mutex identifier as the argument. A thread remains the owner until the unlock operation returns successfully.

The mutex lock operation can either be blocking or not and the blocking operation can be the subject of timeout. So if a thread tries to lock the mutex, which is owned by other thread, it will either immediately get an error (in the case of non-blocking call) or it will be placed into a wait state. The thread will stay blocked until the mutex becomes unlocked or until the specified timeout interval expires.

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 that a lower priority thread blocks a higher priority thread, the mutex supports a priority inversion algorithm as the Priority Inheritance Protocol.

The Priority Inheritance Protocol assures that the thread owning a resource will execute at the higher of either its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread. That means, when a thread tries to lock a mutex which is locked and the thread decide to wait for the unlocking, the thread owning the mutex shall inherit the calling threads priority, if it is higher than the owner's priority.

A thread may own several mutexes at the same time. However, this may lead to a deadlock. If a thread owns mutex M1 and tries to lock mutex M2 owned by another thread, which in turn tries to lock M1, this will cause a deadlock situation.

4.7.3. Semaphores

A Semaphore is used to guard the shared resource by limiting the number of threads than can access it simultaneously. When a semaphore is created, an initial state of the semaphore has to be provided, which specifies the number of threads that can concurrently access the resource it protects. It can be used to provide mutual exclusion (similar to mutex objects) by specifying that only one thread should be allowed to access to the shared resource at any point in time.

A semaphore is defined as an object that has an integral value and a set of blocked thread associated with it. If the value is positive or zero, then the set of blocked threads 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 thread that attempts to lock it again will block or be informed that it is not possible to perform the operation.

A semaphore is a simplified synchronization mechanism in respect to mutex. It can be used in a context where the locking and unlocking may be controlled from different threads and no support to avoid priority inversion is required.

A semaphore is an object having the following attributes:

- A Semaphore is a type of counting semaphore.
- A Semaphore has no owner.
- A Semaphore can be locked and unlocked.
- A Semaphore does not use a priority inheritance protocol to avoid priority inversion.

A thread that wants to access a critical resource has to wait on the semaphore that guards the resource. The semaphore wait operation can be blocking or not and the blocking operation can be the subject of timeout. If a thread issues a semaphore wait call and the thread limit is not reached yet (the semaphore count value is positive) the thread will proceed immediately. Otherwise it will get an error (in the case of non-blocking call) or it will go into a wait state and will stay blocked until either one or more threads release the semaphore or until a specified timeout interval expires.

4.7.4. Monitors

A Monitor supports two kind of thread synchronization: mutual exclusion and cooperation. Mutual exclusion enables multiple threads to independently work on shared data without interfering with each other. Cooperation enables threads to work together towards a common goal.

A Monitor is an object having the following attributes:

- A Monitor is a type of binary semaphore.
- A Monitor has an owner. The owner is the thread performing the lock operation.
- A Monitor can be locked and unlocked. Only the owner can perform unlocking.

In its mutual exclusion a monitor is almost identical to a mutex, but with no priority inheritance.

Another form of monitor is called “wait and notify” monitor. A thread that currently owns the monitor can suspend itself inside the monitor by issuing a monitor wait call. When a thread executes a wait, it releases the monitor and enters the wait state. The thread will stay blocked until another thread issues a monitor notify call. When a thread executes a notify operation, it continues to own the monitor until it releases the monitor explicitly, either by executing a monitor wait or by unlocking it. After the notifying thread has released the monitor, waiting threads become ready and try to reacquire the monitor.

4.7.5. Message Queues

The message passing facility of the Rubus OS allows threads to communicate through message queues. It provides services for opening and closing a message queue send a message to a message queue and receive a message from a message queue.

A Message Queue 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 dynamically allocated.
- A Message Queue has no owner. Different threads can do open and close operations.

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.

When the message queue is empty the thread may wish to wait for a message to arrive. The Receive Waiting List contains all threads waiting for the arrival of a message. When the message is placed into the message queue the thread at the head of the list becomes unblocked. The ordering of the Receive Waiting List is priority based – the thread with the highest priority is inserted at the head of the list.

When the message queue is full the thread may wish to wait for a free message to become available. The Send Waiting List contains all threads waiting for a free space in the message queue. When this happens the thread at the head of the list becomes unblocked. The ordering of the Send Waiting List is priority based – the thread with the highest priority is inserted at the head of the list.

4.8. 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

Error Detection. Rubus OS is designed in such a way that failures are detected and the user shall be able to react on them in an adequate way.

Error Handling. Rubus OS supports a mechanism for specifying user defined error handlers. An error detected during run-time is notified by calling one of installed handlers.

The error handling in Rubus OS is layered. The layers are:

- Thread Layer. A thread may or may not have an error handler assigned. When a thread is running and an error is detected the assigned Thread Error Handler is invoked. If no handler is assigned the process level is checked.
- Process Layer. A process may or may not have an error handler assigned. When an error is detected and the running thread has no assigned handler the Process Error handler is invoked. If no error handler is assigned the System Error Handler is called.
- System Layer. A default System Error Handler shall always be installed. An error not directly connected to the running thread shall cause a call of this handler as well as if no Thread or Process Error Handler is installed.

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 and the error number is set into the object *errno*.

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. A successful operation is indicated by the return value containing integer zero or a non-null pointer. An unsuccessful operation is indicated by the return value containing -1 or a null pointer, accompanied by setting errno to the value of error number.

The object *errno* is a logical object and is included into the context of a thread. It can be acquired via a dedicated system service call (see the C API Reference Manual). The value of the object *errno* is defined only after a call for a service for which it is explicitly stated to be set and until it is changed by the next service call. It should only be examined when it is indicated as valid by a return value of a service.

4.9. Event Logging

The Rubus OS provides an event logging facility. During run-time the operating system maintains the log of important system events, such as interrupts, thread switches, operations on synchronization objects and so on. Each log entry contains an event descriptor and a timestamp, i.e. what happened, when and in which order.

This mechanism is extensible, so user can append its own event object into the logging system and put a log entry each time the event occurs. See the C API Reference Manual for details.

The event logging can help to simplify and speed up the process of applications development and debugging, performance analyzing and tuning, searching for problems or bottlenecks. The Event Log can be viewed and examined from the

Imsys Developer IDE, where it is displayed in the convenient graphical form. See the Imsys Developer help for more information.

4.10. System Initialization and Startup

Before the operating system can do its job it must be initialized and started. The Rubus OS provides special routines for its initialization and startup.

The Initialization routine performs the following tasks:

- Initializes the High Resolution Timer block and installs two handlers: Thread Scheduling Handler and Time Supervising Handler
- Initializes the Interrupt Vector Table and installs a handler for the Timer Interrupt, which will process interrupts generated by the timer block when one of its timers expires. Installs Kernel Service handler and Dispatch Service handler.
- Installs a system-wide error handler specified as an initialization parameter.
- Initializes the Event Log with the buffer specified as an initialization parameter.
- Initializes the Object Access Table.
- Initializes thread lists.
- Creates and initializes System Process, Kernel Thread and Idle Thread.

The Start routine starts execution of the previously created System Process. The thread assigned the highest priority will be executed first. From now interrupts and services are enabled. Upon successful completion this routine shall NOT return.

5. Communication Drivers

Communication drivers are high-level drivers that provide a well-defined access to the facilities of different communication interfaces, such as COM, SPI, I2C. Each communication driver tightly cooperates with the corresponding hardware interface driver, which in turn interacts with the underlying hardware.

All operations supported by any communication driver can be divided into the following types: initialization, configuration, input-output. They are briefly described below. For more detailed information see the C API Reference Manual.

5.1. COM

Prior performing any I/O operations on COM interface it should be initialized. The initialization routine should be called ones. In order to use COM-port for I/O operations an application first should open it. All further references to that port are done through a handle returned by an open call. After the port is not used anymore the application should close it and release the system resources.

The actual I/O is performed via read or write routines, which take the port handle, the data buffer and its size as parameters. The write operation is buffered, so a flush operation is also provided. The COM driver supports a lot of routines for COM-port configuration, control and management. All of them require the port to be opened first.

5.2. SPI

Prior performing any I/O operations on SPI interface it should be initialized and configured. The initialization routine initializes the hardware interface by setting up corresponding port pins. A set of configuration routines allow to configure an operating mode and to set a bit rate. The operating mode is configured by selecting a clock phase, a clock polarity, a bit order and by configuring a slave select line.

An I/O is done though the bi-directional xmit routine, which takes the data buffer and its size as arguments. The data will always be sent and received simultaneously. The buffer content sent on the MOSI line will be replaced by the data received on the MISO line.

5.3. I2C

The configuration and the initialization of the I2C interface are very simple. First the port pin set is selected from two options: a standard set (default) or an alternate set. Then an initialization routine is called which initializes port pins to their idle state.

An I/O is performed though a set of read and write routines. They allow reading or writing a one byte or multiple bytes. All of these routines take a slave device address and a delay constant as arguments and depending on its type a data value or a data buffer and its size.

6. File Subsystem

The File Subsystem is an integral part of the System Software, which provides a file-based access to the data stored on different media types, such as FLASH memory devices, SD/MMC memory cards, RAM-disks.

The file subsystem has a layered architecture. Layers interact with each other though a well-defined interface.

Application Layer. Normally to perform file-related tasks applications use file I/O functions from a standard C library. This is the upper-most layer.

Common API Layer. The layer below is the Common API that hides the differences and implementation details of underlying file systems and provides a single and consistent interface to the application layer.

File System Layer. The next layer is the file system implementation. Two file systems are supported: SAFE – an efficient and failsafe file system for internal flash devices and FAT – a traditional file system for a PC-compatible media.

Media Type Driver Layer. The next layer is a high-level driver for each general media type that shares common properties. This driver handles issues of FAT maintenance, wear-leveling, etc.

Physical Device Driver Layer. The bottom layer is the low-level physical device driver, which does the translation between the high-level driver and physical device hardware.

6.1. SAFE File System

The SAFE is an efficient and failsafe file system. It was designed primarily for internal flash devices, but can also be used on RAM-disks. The file system key features are:

- Power Fail Safety – the system may be stopped and restarted at any point of time and no data will be lost. The previous completed state of the file system will be restored.
- Dynamic Wear – the file system allocates the least used blocks from those available thus maximizing the flash device lifetime.
- Multithreaded – the file system supports simultaneous assess to files from multiple threads.
- Long File Names – the file system supports file names of almost unlimited length. The file name handling is efficient - it is build from a chain of small blocks and that blocks are allocated and added automatically.
- Multiple Volumes – the file system supports multiple volumes.
- Multiple Open Files – the file system allows multiple files to be opened simultaneously on a volume or on different volumes.
- Unicode File Names – the file system supports Unicode file names.

6.2. FAT File System

The FAT is a traditional file system for a PC-compatible media, such as MMC/SD flash memory cards. It can also be used on RAM-disks. The file system key features are:

- Multithreaded – the file system supports simultaneous assess to files from multiple threads.
- Long File Names – the file system supports file names of up to 260 bytes length.
- Multiple Volumes – the file system supports multiple volumes.
- Multiple Open Files – the file system allows multiple files to be opened simultaneously on a volume or on different volumes.
- Unicode File Names – the file system supports Unicode file names.
- Caching – the file system supports FAT caching, write caching and directory caching.
- PC-compatibility – FAT created partitions are Windows XP compatible.

6.3. File I/O API

An application can access files though the well-known file I/O functions of the Standard C library. See the C API Reference Manual for details.

6.4. Volume Mounting

Before accessing any files stored on the media, the file system should “mount” it, i.e. made it available to the system. The mounted media is represented in the system as a volume, which has a 0-based number and a drive letter starting with ‘A’. Any file on this volume identified by its full file name, which includes the drive letter, the full path, and the file name.

For each type of supported media (flash, RAM-disk, MMC/SD) there is a corresponding mount routine. See the C API Reference Manual for details.

6.5. Initialization

The file system must be initialized before using any of its functions. An initialization routine is provided for that purpose. It should be executed before issuing any other file system call. See the C API Reference Manual for details.

7. Network Subsystem

The Network Subsystem, as a part of the System Software is responsible for the networking support and network communications. Its major part is TCP/IP protocol stack software, which provides the support for TCP/IP environment and TCP/IP communications over Ethernet and serial links.

The TCP/IP stack software was designed to be fully reentrant and multithreaded safe, as well as scalable and highly configurable. It is compliant with standards and has a reach application programming interface.

7.1. TCP/IP Stack Architecture

In general, functional architecture of the TCP/IP stack is the same as in any other TCP/IP implementations. It comprises four major layers: link layer, network layer, transport layer and application layer. These layers interact with each other via well-defined interfaces.

A Link layer provides an abstract view of a network interface device (or adapter) and isolates its details from upper layers. Link layer includes device drivers for real or virtual (pseudo) hardware and implementations of underlying physical network protocols. Currently, TCP/IP link layer software provides following components: driver for internal loopback adapter (pseudo-device), driver for Imsys Ethernet Adapter and implementations of loopback and Ethernet II protocols.

A Network layer provides the basic packet transmission service as well as addressing and routing services. It uses the link layer interfaces to communicate with network devices. The TCP/IP network layer software includes an implementation of Internet Protocol (IP), as the major protocol for packet transmission, and an implementation of Address Resolution Protocol (ARP) as the protocol for the remote host's hardware address discovery. Note that architecturally ARP is implemented as the network layer module, while it's functionally belongs rather to the link layer than to the network layer.

A Transport layer provides for applications a number of various transport services to exchange data over network. It uses network layer interfaces to request an address, routing and control information, a packet transmission and so on. The most common transports are: a datagram transport, which provides mechanisms for unreliable datagram exchange, and a stream transport, which provides reliable and sequential data transfer. The TCP/IP software implements User Datagram Protocol (UDP) as datagram transport and Transmission Control Protocol (TCP) as stream transport. It also implements Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP) as "raw" transport modules. Any application can send and receive data in ICMP and IGMP format. An additional transport module called "raw wildcard" transport allows applications to access packet transmission services of the IP module, i.e. to send and receive raw IP datagrams.

An Application layer provides unified access to the transport layer features, independently of underlying transport protocol's semantics and hides each

protocol implementation details. The TCP/IP software includes reach protocol-independent application programming interface, compatible with BSD sockets. All network applications using this API also belong to the application layer.

7.2. Application Layer Components

The TCP/IP software also includes a number of application layer components. They implement application layer protocols which are parts of a TCP/IP suite, such as Dynamic Host Configuration Protocol and Domain Name System.

7.2.1. DHCP Client

The DHCP Client plays an important role in the system initialization. To operate in the TCP/IP environment the system needs to know its IP address, network mask and, optionally, a default gateway. This information can be configured manually or distributed through the DHCP server. So, the DHCP client dynamically obtains these parameters from the DHCP server and automatically configures TCP/IP stack software thus eliminating the need for manual configuration. In the absence of a DHCP server on the network the system is connected to, the DHCP client uses a so-called Zero-Config protocol to select an IP address from a well-known address range and negotiate this selection with other nodes on the network.

7.2.2. DNS Client

Any node on the TCP/IP network is identified by its unique IP address. But it is more convenient to use for identification a human-readable name instead of an IP address. The node name to the node address translation is performed by a DNS client. In order to perform this translation the DNS client interacts with one or more DNS servers, which store and maintain a distributed global name-to-address mapping database.

7.3. BSD Sockets API

An application can access communication facilities of the network subsystem through the well-known BSD Sockets application programming interface. This interface is based on the concept of a socket – a communication endpoint – which is quite similar to the file descriptor. See the C API Reference Manual for details.

7.4. Network Initialization

Before communication facilities can be used, the network subsystem must be initialized and configured. Normally it is done during the initialization of the shell environment. The shell startup code reads configuration files and initializes the TCP/IP stack software. When the target system does not require the shell to be running it can initialize and configure the network subsystem by using a network initialization routine. It will do the following:

- If TCP/IP configuration parameters were not provided, it tries to read them from a permanent storage such as on-board flash memory.
- It initializes the link layer by calling each network interface driver's entry point.
- It initializes the network layer by invoking IP and ARP initialization routines.
- It initializes the transport layer by calling each transport module initialization routine.
- It configures the internal loopback interface.
- It either configures the Ethernet interface manually if configuration parameters were specified, or requests the DHCP client to configure it dynamically.
- Finally it sets the hostname and parameters for a name resolution client.

This routine has to be called after the Rubus OS has been initialized and started.

8. Shell Environment

The shell, called ISH (Imsys Shell), is responsible for the high-level system initialization, for execution of various servers and for providing a UNIX-like command interface.

8.1. High-Level Initialization

The shell startup code completes initialization and configuration of the system performing the following tasks:

- Initializes the serial port interface
- Registers stdout and stderr functions
- Reads the Hardware Identification String from the flash memory
- Reads and processes a shell configuration file ish.ini.
- Reads and processes a system configuration file system.ini
- Initializes and starts a Serial server
- Configures a hostname
- Initializes and configures TCP/IP stack software
- Starts FTP and Telnet servers
- Sets the Time Zone and Daylight Saving
- Reads and executes startup file startup.ini

See the ISH User Manual for more information about configuration files and their parameters.

8.2. Communication Servers

The shell includes Serial server, Telnet server and FTP server. The shell starts them automatically during startup, if specified in the system configuration file. Otherwise they can be started and stopped manually using startserver and stopserver commands.

The Serial server provides the command-line user interface over a serial channel. The serial port number and port parameters can be specified though environment variables.

The Telnet server provides the command-line user interface over a TCP/IP communication channel. The TCP port number and the server priority can be specified though the environment variable.

The FTP server provides a remote assess to the local file system. It supports both get and put operations. The number of simultaneous connections and the server priority can be specified through environment variables.

8.3. Command Interface

The shell provides a UNIX-like command interface over serial or TCP/IP channels. It operates as any other command interpreter, i.e. reads its standard input, detects a command, executes it and output the result into its standard output.

For information about supported commands see ISH User Manual.

Appendix A. Java API Summary

The following classes from JDK 1.1.8 are supported.

- java.io.BufferedInputStream
- java.io.BufferedOutputStream
- java.io.BufferedReader
- java.io.BufferedWriter
- java.io.File
- java.io.FileDescriptor*
- java.io.FileInputStream*
- java.io.FilenameFilter
- java.io.FileNotFoundException
- java.io.FileOutputStream*
- java.io.FilterInputStream
- java.io.FilterOutputStream
- java.io.Serializable
- java.io.SyncFailedException
- java.lang.Double
- java.lang.Float
- java.lang.Number
- java.lang.IllegalAccessError
- java.lang.InternalError
- java.lang.LinkageError
- java.lang.NoClassDefFoundError
- java.lang.Number
- java.lang.VirtualMachineError
- java.net.BindException
- java.net.ConnectException
- java.net.ContentHandler
- java.net.ContentHandlerFactory
- java.net.DatagramPacket*
- java.net.DatagramSocket*
- java.net.FileNameMap
- java.net.HttpURLConnection
- java.net.InetAddress*

- java.net.MalformedURLException
- java.net.MulticastSocket
- java.net.NoRoutetoHostException
- java.net.ProtocolException
- java.net.ServerSocket*
- java.net.Socket*
- java.net.SocketException
- java.net.UnknownHostException
- java.net.UnknownServiceException
- java.net.URL
- java.net.URLConnection
- java.net.URLStreamHandler
- java.net.URLStreamHandlerFactory
- java.util.Enumeration
- java.util.EventListener
- java.util.EventObject
- java.util.Locale*
- java.util.Properties*
- java.util.StringTokenizer*
- java.util.TooManyListenersException
- javax.comm.CommDriver
- javax.comm.CommPort
- javax.comm.CommPortIdentifier
- javax.comm.CommPortOwnershipListener
- javax.comm.NoSuchPortException
- javax.comm.PortInUseException
- javax.comm.SerialPort
- javax.comm.SerialPortCommDriver
- javax.comm.SerialPortEvent
- javax.comm.SerialPortEventListener
- javax.comm.SerialPortImplementation
- javax.comm.UnsupportedCommOperationException

*) Differs from JDK 1.1.8

Appendix B. ISAJ Summary

The following table lists the instructions in the ISAJ instructions set. It also shows how the instructions are implemented, as native instructions or emulated.

Table 5: ISAJ Summary

Hex	bytecode	Implementation
00	nop	Native
01	aconst_null	Native
02	iconst_m1	Native
03	iconst_0	Native
04	iconst_1	Native
05	iconst_2	Native
06	iconst_3	Native
07	iconst_4	Native
08	iconst_5	Native
09	lconst_0	Native
0a	lconst_1	Native
0b	fconst_0	Native
0c	fconst_1	Native
0d	fconst_2	Native
0e	dconst_0	Native
0f	dconst_1	Native
10	bipush	Native
11	sipush	Native
12	ldc	Emulated
13	ldc_w	Emulated
14	ldc2_w	Emulated
15	iload	Native
16	lload	Native
17	fload	Native
18	dload	Native
19	aload	Native
1a	iload_0	Native
1b	iload_1	Native
1c	iload_2	Native
1d	iload_3	Native
1e	lload_0	Native
1f	lload_1	Native
20	lload_2	Native
21	lload_3	Native
22	fload_0	Native
23	fload_1	Native
24	fload_2	Native

Hex	bytecode	Implementation
25	fload_3	Native
26	dload_0	Native
27	dload_1	Native
28	dload_2	Native
29	dload_3	Native
2a	aload_0	Native
2b	aload_1	Native
2c	aload_2	Native
2d	aload_3	Native
2e	iaload	Native
2f	laload	Native
30	faload	Native
31	daload	Native
32	aaload	Native
33	baload	Native
34	caload	Native
35	saload	Native
36	istore	Native
37	lstore	Native
38	fstore	Native
39	dstore	Native
3a	astore	Native
3b	istore_0	Native
3c	istore_1	Native
3d	istore_2	Native
3e	istore_3	Native
3f	lstore_0	Native
40	lstore_1	Native
41	lstore_2	Native
42	lstore_3	Native
43	fstore_0	Native
44	fstore_1	Native
45	fstore_2	Native
46	fstore_3	Native
47	dstore_0	Native
48	dstore_1	Native
49	dstore_2	Native
4a	dstore_3	Native
4b	astore_0	Native
4c	astore_1	Native
4d	astore_2	Native
4e	astore_3	Native
4f	iastore	Native
50	lastore	Native
51	fastore	Native

Hex	bytecode	Implementation
52	dastore	Native
53	aastore	Native
54	bastore	Native
55	castore	Native
56	sastore	Native
57	pop	Native
58	pop2	Native
59	dup	Native
5a	dup_x1	Native
5b	dup_x2	Native
5c	dup2	Native
5d	dup2_x1	Native
5e	dup2_x2	Native
5f	swap	Native
60	iadd	Native
61	ladd	Native
62	fadd	Native
63	dadd	Native
64	isub	Native
65	lsub	Native
66	fsub	Native
67	dsub	Native
68	imul	Native
69	lmul	Native
6a	fmul	Native
6b	dmul	Native
6c	idiv	Native
6d	ldiv	Native
6e	fdiv	Native
6f	ddiv	Native
70	irem	Native
71	lrem	Native
72	frem	Emulated
73	drem	Emulated
74	ineg	Native
75	lneg	Native
76	fneg	Native
77	dneg	Native
78	ishl	Native
79	lshl	Native
7a	ishr	Native
7b	lshr	Native
7c	iushr	Native
7d	lushr	Native
7e	iand	Native

Hex	bytecode	Implementation
7f	land	Native
80	ior	Native
81	lor	Native
82	ixor	Native
83	lxor	Native
84	iinc	Native
85	i2l	Native
86	i2f	Native
87	i2d	Native
88	l2i	Native
89	l2f	Native
8a	l2d	Native
8b	f2i	Native
8c	f2l	Native
8d	f2d	Native
8e	d2i	Native
8f	d2l	Native
90	d2f	Native
91	i2b	Native
92	i2c	Native
93	i2s	Native
94	lcmp	Native
95	fcmpl	Native
96	fcmpg	Native
97	dcmpl	Native
98	dcmpg	Native
99	ifeq	Native
9a	ifne	Native
9b	iflt	Native
9c	ifge	Native
9d	ifgt	Native
9e	ifle	Native
9f	if_icmpeq	Native
a0	if_icmpne	Native
a1	if_icmplt	Native
a2	if_icmpge	Native
a3	if_icmpgt	Native
a4	if_icmple	Native
a5	if_acmpeq	Native
a6	if_acmpne	Native
a7	goto	Native
a8	jsr	Emulated
a9	ret	Emulated
aa	tableswitch	Native
ab	lookupswitch	Native

Hex	bytecode	Implementation
ac	ireturn	Emulated
ad	lreturn	Emulated
ae	freturn	Emulated
af	dreturn	Emulated
b0	areturn	Emulated
b1	return	Emulated
b2	getstatic	Emulated
b3	putstatic	Emulated
b4	getfield	Emulated, Quickcode
b5	putfield	Emulated, Quickcode
b6	invokevirtual	Emulated, Quickcode
b7	invokespecial	Emulated, Quickcode
b8	invokestatic	Emulated, Quickcode
b9	invokeinterface	Emulated, Quickcode
ba	*unused*	-
bb	new	Emulated
bc	newarray	Emulated
bd	anewarray	Emulated
be	arraylength	Emulated
bf	athrow	Emulated
c0	checkcast	Emulated
c1	instanceof	Emulated
c2	monitorenter	Emulated
c3	monitorexit	Emulated
c4	wide	Native
c5	multianewarray	Emulated
c6	ifnull	Native
c7	ifnonnull	Native
c8	goto_w	Native
c9	jsr_w	Emulated

Appendix C. C API Functions Summary

Table 6: Rubus Function Summary

Functions	Description
__errno	Get current error.
event_call	Call a service event handler.
event_install	Install an event handler.
event_remove	Remove an event handler.
event_request	Request a service event handler.
mq_close	Close message queue.
mq_getattr	Retrieves status information and attributes of the message queue.
mq_open	Open a message queue.
mq_receive	Receive a message from a message queue.
mq_send	Send a message to a message queue.
mq_timedreceive	Receive a message from a message queue.
mq_timedsend	Send a message to a message queue.
nanosleep	High resolution sleep.
os_init	Initialize the operating system.
os_logstart	Start event log.
os_logstop	Stop event log.
os_start	Start process scheduling.
os_stop	Stop process scheduling.
os_userevent_put	Puts an event.
os_userobject_destroy	Destroy event log object.
os_userobject_init	Create event log object.
pthread_attr_destroy	Destroy thread attributes object.
pthread_attr_init	Initialize thread attributes object.
pthread_attr_setschedparam	Set schedparam attribute.
pthread_attr_setschedpolicy	Set schedpolicy attribute.
pthread_attr_setstacksize	Set stack attribute.
pthread_cancel	Cancel thread.
pthread_create	Create a new thread.
pthread_equal	Get thread ID.
pthread_exit	Terminate thread.
pthread_getschedparam	Get scheduling parameters.
pthread_getspecific	Get thread-specific data.
pthread_getstatus	Get thread status.
pthread_key_create	Create thread-specific data key.
pthread_key_delete	Delete thread-specific data key.
pthread_mutex_destroy	Destroy mutex.
pthread_mutex_init	Initialize mutex.
pthread_mutex_lock	Lock mutex.
pthread_mutex_timedlock	Lock mutex.
pthread_mutex_trylock	Lock mutex.

Functions	Description
pthread_mutex_unlock	Unlock mutex.
pthread_mutexattr_destroy	Destroy mutex attributes object.
pthread_mutexattr_init	Initialize mutex attributes object.
pthread_self	Get thread ID.
pthread_setschedparam	Set scheduling parameters.
pthread_setspecific	Set thread-specific data.
pthread_yield	Yield thread.
sem_destroy	Destroy an unnamed semaphore.
sem_getvalue	Get the value of a semaphore.
sem_init	Initialize an unnamed semaphore.
sem_open	Initialize and open a named semaphore.
sem_post	Unlock a semaphore.
sem_timedwait	Lock a semaphore.
sem_trywait	Lock a semaphore.
sem_wait	Lock a semaphore.
signal_clear	Clear pending signals.
signal_send	Send a signal to a thread.
signal_timedwait	Wait for signals with timeout.
signal_wait	Wait for signals.

Table 7: COM Function Summary

Functions	Description
comClose	Closes port.
comConfig	Change port settings.
comDataAvailable	Test if data available.
comFlush	Empty the port.
comGetBaudrate	Get current baudrate.
comGetBit	Get port signal.
comGetFlowControlMode	Get flow control mode.
comGetInputBufferSize	Get input buffer size.
comGetOutputBufferSize	Get current output buffer size.
comGetReceiveFraming	Get framing character.
comInit	Initializes the COM driver.
comOpen	Opens a port.
comPortExists	Tests if port exists.
comRead	Read data.
comSendBreak	Sends break.
comSetBaudrate	Set baudrate.
comSetBit	Set port signal.
comSetFlowControlMode	Set flow control mode.
comSetInputBufferSize	Set input buffer size.
comSetIrDAMode	Enable or disable IrDA mode.
comSetOutputBufferSize	Set output buffer size.
comSetReceiveFraming	Enable or disable receive framing.
comWaitData	Wait for data.
comWrite	Write data.

Table 8: SPI Function Summary

Functions	**Description**
spiGetBitRate	Gets SPI bitrate.
spiInit	Initializes SPI.
spiSetBitOrder	Sets SPI data bit order.
spiSetBitRate	Sets SPI bitrate.
spiSetPhase	Sets SPI clock phase.
spiSetPolarity	Sets SPI clock polarity.
spiSetSlaveSelect	Sets SPI slave select line.
spiXmit	Communicates over SPI.

Table 9: I2C Function Summary

Functions	Description
i2cInit	Initialize the I2C port pins to their idle state.
i2cPinSet	Initialize the I2C port pins to their idle state.
i2cRead	Read a block of data from the specified address.
i2cRead1	Read one byte of data from the specified address.
i2cWrite	Write a block of data to the specified address.
i2cWrite1	Write one byte of data to the specified address.
i2cWriteRead	First write to, and then read from the same address.

Table 10: TCP/IP Function Summary

Functions	Description
accept	Accepts an incoming connection on a listenning socket.
bind	Assigns a local protocol address to a socket
closesocket	Closes a socket.
connect	Initiates a connection on a socket
getpeername	Retrieves the name of the peer to which a socket is connected.
getsockname	Retrieves the local name for a socket.
getsockopt	Retrieves a socket option.
listen	Places a socket into a state in which it can receive incoming connections.
recv	Receives data from a connected socket.
recvfrom	Receives data from a socket with its source address.
send	Sends data on a connected socket.
sendto	Sends data on a socket to a specific destination.
setsockopt	Sets a socket option.
shutdown	Shuts down a connection on a socket.
socket	Creates an endpoint for communication and returns its descriptor.

Table 11: Standard Library Function Summary

Functions	Description
abort	Causes abnormal program termination.
abs	Computes the absolute value of an integer.
asctime	Converts date and time to a string.
assert	Aborts the program if assertion is false.
atexit	Registers a function to be called at program termination.
atof	Convert a string to a double.
atoi	Convert a string to an integer.
atol	Convert a string to a long.
bsearch	Binary search of a sorted array.
calloc	Allocate memory and set content to zero.
chdir	Change working directory.
chdrive	Change current drive.
clearerr	Clears indicators on a stream.
clock	Determine processor time.
close	Close file.
closedir	Close a directory stream.
creat	Create file.
ctime	Transform date and time to ASCII.
difftime	Calculates time difference
div	Compute quotient and remainder of an integer division.
exit	Terminate program.
fclose	Close a stream.
feof	Test end-of-file indicator on a stream.
ferror	Checks the stream error status.
fflush	Flush a stream.
fgetc	Get a byte from a stream.
fgetpos	Get current file position information.
fgets	Get a string from a stream.
fopen	Opens a stream.
fprintf	Print formatted data to a stream.
fputc	Put a byte on a stream.
fputs	Put a string on a stream.
fread	Binary stream input.
free	Free memory area.
freopen	Open a stream.
fscanf	Read formatted data from a stream.
fseek	Reposition a stream.
fsetpos	Set current file position.
ftell	Return a file offset in a stream.
ftime	Return date and time.
fwrite	Binary output to a stream.
getc	Get a byte from a stream.
getchar	Get a byte from a stdin stream.
getcwd	Get current working directory.

Functions	Description
getdrive	Get current drive number.
getenv	Get an environment variable.
gets	Read string from stdin.
gmtime	Converts a time value to a broken-down UTC time.
isalnum	Test for an alphanumeric character.
isalpha	Test for an alphabetic character.
iscntrl	Test for a control character.
isdigit	Test for a digit.
isgraph	Test for a printable character except space.
islower	Test for a lower-case character.
isprint	Test for a printable character including space.
ispunct	Check if a character is printable but not space or alphanumeric.
isspace	Check if a character is white space.
isupper	Check if a character is upper case.
isxdigit	Check if a character is a hexadecimal digit.
labs	Calculate the absolute value of a long.
ldiv	Compute quotient and remainder of an integer division.
localtime	Convert a time value to a broken-down local time.
longjmp	Do a long jump.
lseek	Move the read/write file offset
malloc	Allocates a memory block.
mblen	Get the number of bytes in next multibyte character.
mbstowcs	Convert a multibyte string to a wide character string.
mbtowc	Convert a multibyte string to a wide character.
memchr	Search for a character in memory.
memcmp	Compare memory areas.
memcpy	Copy memory area.
memmove	Copy memory area.
memset	Fill memory with a constant byte.
mkdir	Create directory.
mktime	Convert a broken-down time structure into a time stamp.
open	Open file.
opendir	Open directory.
perror	Prints system error message.
printf	Formatted print.
putc	Write character to stream.
putchar	Write character to `stdout`.
putenv	Change or add an environment variable.
puts	Writes a string to `stdout`.
qsort	Sorts an array.
rand	Get a pseudo-random number.
read	Read file.
readdir	Read directory.
remove	Remove file.

Functions	Description
rename	Rename a file.
rewind	Rewind a stream to the beginning.
rmdir	Remove directory.
scanf	Scan *stdin* according to a specific format.
setbuf	Sets buffer of a buffered I/O stream.
setenv	Change or add an environment variable.
setjmp	Set return point for a longjump.
setvbuf	Sets buffer of a buffered I/O stream.
sprintf	Print formatted output.
srand	Set up the random generator.
sscanf	Scan a string for arguments.
stat	Get file status.
strcat	Concatenate two strings.
strchr	String scanning operation.
strcmp	Compare two strings.
strcoll	String compare using the current locale.
strcpy	Copy a string.
strerror	Returns string describing error code.
strftime	Format date and time.
strlen	Get string length.
strncat	Concatenate a string with part of another.
strncmp	Compare part of two strings.
strncpy	Copy part of a string.
strpbrk	Scan string for byte.
strrchr	String scanning operation.
strspn	Get length of a substring.
strstr	Find a substring.
strtod	Convert string to a double-precision number.
strtok	Scan string for token.
strtol	Convert a string to a long.
strtoul	Convert a string to a unsigned long.
strxfrm	String transformation.
time	Get time of day.
tmpnam	Create a unique file name.
tolower	Convert a character to lower case.
toupper	Convert a character to upper case.
ungetc	Push back a character to a stream.
unsetenv	Remove an environment variable.
vfprintf	Format output of a stdarg argument list
vfscanf	Format input of a stdarg list.
vprintf	Formatted print to stdout.
vscanf	Scan stdin for a specific format.
vsprintf	Formatted print.
vsscanf	Scan string for a specific format.
wcstombs	Convert a wide character string to a multibyte string.

Functions	Description
wctomb	Convert a wide character to a multibyte sequence.
write	Write file.