CRITICAL SOFTWARE, SA
EN1/IC2, KM 185,6
BANHOS SECOS, SANTA CLARA
3040-032 COIMBRA, PORTUGAL
TEL. +351.239.801300, FAX. +351.239.801319
WWW.CRITICALSOFTWARE.COM

CRITICAL SOFTWARE, LIMITED
111 NORTH MARKET STREET, SUITE 670
SAN JOSE, CALIFORNIA, USA, 95113
TEL. +1(408) 9711231,
FAX: +1(408) 3513330

# RTEMS 4.5.0 Evaluation Report

## RAMS Call-off Order 2

**Contract Ref.:** *CSW-RAMS-2003-CTR-1306*

*ESTEC/Contract Nº 16582/02/NL/PA*



**Date:** *25/11/2003*
**Pages:** *57*
**State:** *Approved*
**Access:** *See Access List*
**Reference:** DL-RAMS02-01-05
CSW-RAMS-2003-RPT-1334-05

***Partners / Clients:***

# RTEMS 4.5.0 Evaluation Report

# RAMS Call-off Order 2

| Approved Version: 1.31 | | | |
|---|---|---|---|
| **Name** | **Function** | **Signature** | **Date** |
| Ricardo Maia | Project Manager | | 25-11-2003 |
| José Silva | SQA Engineer | | 25-11-2003 |



| Authors and Contributors: | | | |
|---|---|---|---|
| **Name** | **Contact** | **Description** | **Date** |
| Ricardo Barbosa | rbarbosa@criticalsoftware.com | Project Engineer | 09/09/2003 |
| Ricardo Maia | rmaia@criticalsoftware.com | Project Manager | 25/11/2003 |
| João Esteves | jesteves@criticalsoftware.com | Senior Engineer | 26/06/2003 |
| Luís Henriques | lhenriques@criticalsoftware.com | Senior Engineer | 03/10/2003 |
| Diamantino Costa | dcosta@criticalsoftware.com | Review Inspector | 16/09/2003 |



| Access List: |
|---|
| **Internal Access** |
| Project Team Members |
| **External Access** |
| ESA-ESTEC |

| Revision History: | | | |
|---|---|---|---|
| **Version** | **Date** | **Description** | **Author** |
| 0.1 | 27/6/2003 | First Draft | Ricardo Barbosa |
| 0.2 | 07/07/2003 | Update document structure, introduction and revision of the contents. | Ricardo Maia |
| 1.0 | 11/07/2003 | Update after internal review | Ricardo Barbosa |
| 1.1 | 14/07/2003 | Added the Fault Model chapter. | Luís Henriques |
| 1.2 | 9/9/2003 | Changes made after scope redefinition | Ricardo Barbosa |
| 1.3 | 15/09/2003 | Changed according to review. | Luís Henriques |
| 1.4 | 16/09/2003 | Changes imported from the review of DL-RAMS02-02, Issue 2 | D Costa |
| 1.5 | 03/11/2003 | Added the metrics chapter and moved the annexes to a separate document | Luís Henriques |
| 1.6 | 05/11/2003 | Added the summary of the results of the robustness and stress testing, list of problems found and the methodology feedback. | Ricardo Maia |
| 2.0 | 07/11/2003 | Updated after internal review. | Ricardo Maia |
| 2.1 | 25/11/2003 | Updated Results Summary chapter after the Final Presentation at ESTEC | Ricardo Maia |

# Table of Contents

# List of Figures



# List of Tables

# 1. Introduction

## 1.1 Objective

This document presents the results of the evaluation of the Real Time Executive for Multiprocessor Systems (RTEMS) version 4.5.0. This evaluation, performed in the scope of the Call-off Order number 02 under project Software Dependability and Safety Evaluations, ESTEC/Contract Nº 16582/02/NL/PA, consists mainly in trial out of robustness and stress testing techniques.

This document provides an overview of the main activities conducted through out this Call-off Order. It gives an inside view of the RTEMS architecture and some aspects of its design, as well as a description of its main functionalities, presents the methodologies used in robustness and stress testing of the RTEMS and corresponding results and provides some feedback on the methodologies used.

## 1.2 Scope

This report is the deliverable DL-RAMS02-01-05 of the Call-off Order number 02 under project Software Dependability and Safety Evaluations, ESTEC/Contract Nº 16582/02/NL/PA and presents the results of WP210, WP220, WP230, WP240 and WP500.

## 1.3 Audience

This document targets several groups of readers, namely:

- "Software Dependability and Safety Evaluations" team members and in particular the Call-off Order 2 team members.
- Space software staff involved in the development of on-board software.
- Staff involved in the development of RTEMS related software.
- Space software product assurance staff.
- Management and technical ESA/ESTEC staff.

## 1.4 Acronyms

| Acronyms | Description |
|---|---|
| API | Application Programming Interface |
| ASR | Asynchronous Signal Routine |
| BSP | Board Support Package |
| CSW | Critical Software, S.A. |
| COO2 | Call-off Order 2 |
| FIFO | First In First Out |
| ISR | Interrupt Service Routine |
| ITRON | Industrial The Real time Operating system Nucleus |
| LIFO | Last In First Out |
| TCB | Task Control Block |
| POSIX | Portable Operating System Interface |
| RAMS | Reliability, Availability, Maintainability and Safety |
| RTEMS | Real Time Executive for Multiprocessor Systems |
| RTOS | Real Time Operating System |

## 1.5 Document Structure

This document has the following structure:

Chapter 1 introduces the document, as well as the document scope, intended audience and a list of acronyms and references used through out the document.

Chapter 2 provides an overview of RTEMS 4.5.0, the product deployment structure, and the architecture.

Chapter 3 defines the scope of the current evaluation, listing all the RTEMS features subject of test.

Chapter 4 describes the test methodology and in particular the fault/stress model used for defining the robustness and stress test cases.

Chapter 5 presents the test set-up and execution environment.

Chapter 6 summarizes the results obtained in the evaluation, highlighting the main problems found on RTEMS and some potential improvements.

Chapter 7 presents the metrics collected regarding to the coverage of the robustness testing.

Chapter 8 provides some feedback concerning the robustness and stress testing methodologies applied.

## 1.6 References

[1] RTEMS 4.5.0 Robustness Testing Report, DL-RAMS02-02-02, CSW-RAMS-2003-RPT-1335, September 19, 2003, Critical Software, SA

[2] RTEMS 4.5.0 Stress Testing Report, DL-RAMS02-04-02, CSW-RAMS-2003-RPT-1338, October 30, 2003, Critical Software, SA

[3] ESA PSS-05-02 Issue 1 Revision 1, Guide to the user requirements definition phase, March 1995.

[4] RTEMS Real Time Executive for Multiprocessor Systems, www.rtems.com, OAR

[5] RTEMS Development Environment Guide, September 2000, OAR

[6] RTEMS C User's Guide, September 2000, OAR

[7] RTEMS POSIX 1003.1 Compliance Guide, September 2000, OAR

[8] RTEMS Release Notes, May 2000, OAR

[9] RTEMS SPARC Applications Supplement, September 2000, OAR

[10] ORK-ERC32-SW Technical Specification, Software Requirement Specification, STADY-D2.2-2002, November 11, 2002, ESTEC Contract nr. 15751/02/NL/LvH, Critical Software, SA

[11] Automated Robustness Testing of Off-the-Shelf Software Components, June 1998, 28th Fault Tolerant Computing Symposium, in press, Kropp, N., Koopman, P. & Siewiorek, D.

[12] SLOCCount, http://www.dwheeler.com/sloccount

# 2. RTEMS Description

## 2.1 RTEMS Overview

A study was completed in 1988, within the Research, Development and Engineering Center, U.S Army Missile Command, which compared the various aspects of the Ada83 programming language as they relate to the application of Ada code in distributed and/or multiple processing systems. The conclusions driven from that study had a major impact on the way the Army developed since then application software for embedded applications. One of the conclusions of this study was that the Ada83 programming language does not adequately support multiprocessor environments, although it provides multi-tasking mechanisms.

The Guidance and Control Directorate began a software development effort to address this and some other problems driven from this study. A project to develop an experimental real-time kernel begun in order to eliminate these major drawbacks of the Ada83 programming language mentioned above.

The Real Time Executive for Multiprocessor Systems (RTEMS) (at the time called Real Time Executive for Missile Systems) is a real time executive that provides a high performance environment for embedded critical and military applications including the following features:

- Multitasking capabilities;
- Homogeneous and heterogeneous multiprocessor systems support;
- Event-driven, priority based, preemptive scheduling;
- Optional rate monotonic scheduling;
- Intertask communication and synchronisation;
- Priority Inheritance mechanisms;
- Responsive interrupt management;
- Dynamic memory allocation;
- High level of user configurability.

RTEMS is free software and most of its source code can be redistributed and/or modified under the terms of the GNU General Public License (version 2 or later) as published by the Free Software Foundation. RTEMS has been implemented in both Ada and C programming languages. Since the release implemented in Ada seams to be unavailable, the scope of this report is limited to the release implemented in C, although, throughout the documentation, it can be observed that both releases are similar in terms of functionalities.

## 2.2 RTEMS Foundations

RTEMS was developed based on strong concepts in order to make a self contained, highly versatile software component. In order to make an evaluation on RTEMS and since no documentation regarding requirements or detailed design is available, some assumptions were made regarding these topics after performing an analysis on the source code and available documentation. These assumptions are presented next:

- ***Reusability*** - RTEMS is designed as a reusable software component;

- ***Portability*** – RTEMS is designed to minimize the use of non-portable code. It isolates all hardware dependencies (processor and target dependent code) from the rest of the source code, allowing as much common source code as possible to be shared across multiple processors and targets;

- ***Modularity*** – RTEMS is designed to encourage the development of modular components;

- ***Reliability*** – Although is not documented, RTEMS is intended to be used in systems with high integrity requirements, in this particular case, it is intended to be used as on-board software in ESA space missions. As a consequence, requirements from ESA standards, namely ECSS-Q-80 are considered in this evaluation.

## 2.3 Architecture

This section describes the architecture of RTEMS. As a reference, the RTEMS API is used to explain the RTEMS architecture. A more detailed description of the RTEMS API itself is given in the following section.

### 2.3.1 RTEMS External Architecture

One of the goals of RTEMS was to provide a bridge between two critical layers of real time systems, namely, the project dependent application and the target hardware. Figure 1 shows the application architecture of RTEMS.

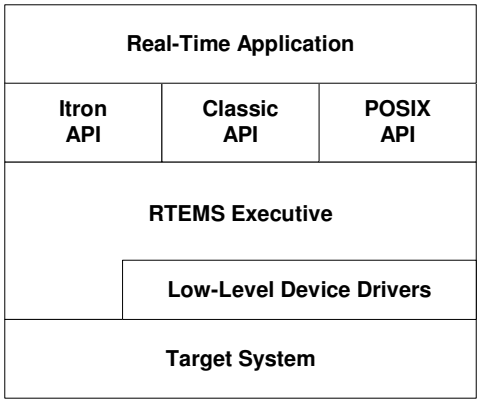


Figure 1. RTEMS Application Architecture

The directory structure presented before can directly be mapped into the presented architecture. The **`score`** and **`sapi`** are the directories that contain the code for the *"**RTEMS Executive**"* layer. The ITRON API implementation is located in the **`itron`** directory, the Classic API implementation is located in the **`rtems`** and the POSIX API is divided in two directories, namely the **`posix`** and **`libc`**, in the Cygnus NEWLIB and in gcc.

This release also provides some applications for testing some features of the executive. These applications are mainly located in the **`tests`** directory. A web server is also made available in this release and is located in the **`libnetworking`** directory.

### 2.3.2 RTEMS Internal Architecture

The internal architecture for RTEMS can be viewed as a set of layers that work closely with each other to provide the set of services to the real time applications. The executive interface presented to the application is formed by grouping *directives* (API calls) into logical sets called resource managers. Scheduling, dispatching and object management is

provided by the executive core, which depends only on a small set of CPU dependent routines.

Next sections provide and overview of the two APIs that were evaluated in the scope of this Call-off Order.

#### *2.3.2.1 RTEMS Classic API*

Figure 2 illustrates this organization for the RTEMS Classic API.

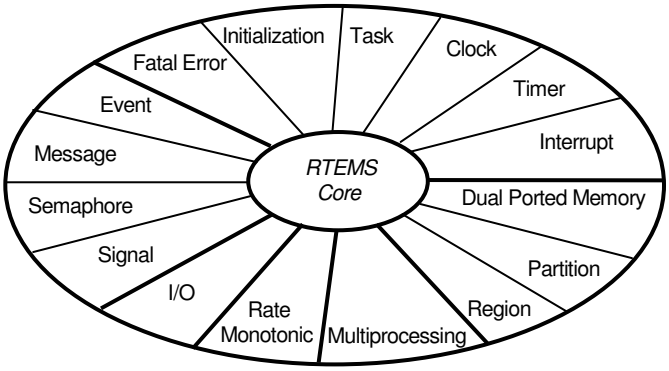


Figure 2. RTEMS Classic API Internal Architecture

The Classic API provides seventeen resource managers. Next, a short description of each manager is presented. In section 6 a more detailed description is provided since each one of these managers provides an interface to be used by the real time applications:

- **Initialization**: This manager is responsible for initiating and shutting down RTEMS;
- **Task**: This manager provides a comprehensive set of directives to create, delete and administer tasks;
- **Clock:** This manager provides support for time of day and other related capabilities;
- **Timer:** This manager provides support for timer facilities;
- **Interrupt**: This manager provides support for connecting functions to hardware interrupt vectors;
- **Dual ported memory:** This manager provides a mechanism for converting addresses between internal and external representations for multiple dual-ported memory areas;
- **Partition:** This manager provides facilities to dynamically allocate memory in fixed-size units;
- **Region:** This manager provides facilities to dynamically allocate memory in variable-size units;
- **Multiprocessing:** This manager provides the facilities for supporting multiprocessor environments composed of both homogeneous and heterogeneous mixtures of processors and target boards;
- **Rate monotonic:** This manager provides facilities to implement tasks which execute in a periodic fashion (Provides facilities to manage the execution of periodic tasks);

- **I/O:** This manager provides a well defined mechanism for accessing device drivers and structured methodology for organizing device drivers;

- **Signal:** This manager provides the capabilities required for asynchronous communication;

- **Semaphore:** This manager utilizes the standard Dijkstra counting semaphores to provide synchronization and mutual exclusion capabilities;

- **Message:** This manager provides communication and synchronization capabilities using RTEMS message queues;

- **Event:** This manager provides a high performance method of intertask communication and synchronization: an event flag is used by a task (or ISR) to inform another task of the occurrence of a significant situation;

- **Fatal error:** This manager processes all fatal or irrecoverable errors;

- **User extensions:** This manager allows the application developer to augment the executive by allowing them to supply extension routines which are invoked at critical system events;

#### *2.3.2.2 POSIX API*

The RTEMS POSIX API presents nineteen managers. Figure 3 show an overview of the RTEMS POSIX API based architecture.

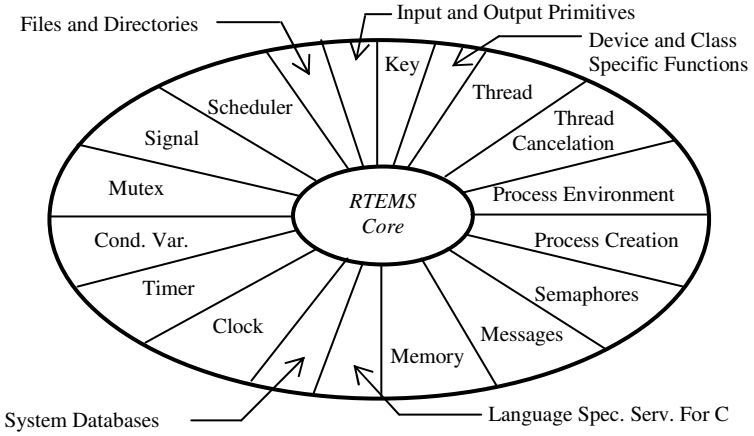


Figure 3. RTEMS POSIX API Internal Architecture

All managers provide their functionalities according to the IEEE 1003.1b standard.

- **Process Creation and Execution Manager[1]:** This manager provides functionalities for process creation and termination;

- **Process Environment Manager[1]:** This manager provides functionalities for process management;

[1] RTEMS implements a single process, multithreaded environment. Because of this, some of these functionalities regarding process management are not supported by RTEMS, more specifically, all process management related functions are not implemented by RTEMS, with the exception of one, the _exit() function;

- **Files and Directories Manager:** This manager provides functionalities for file management;
- **Input and Output Primitives Manager:** This manager provides functionalities for I/O management;
- **Device and Class Specific functions Manager:** This manager provides functionalities for device management;
- **Language Specific Services for the C Programming Language Manager:** this manager provides functionalities for specific features present on the C programming language like acquiring or releasing ownership of file streams;
- **System Databases Manager**: This manager provides functionalities for system database management;
- **Memory Manager**: This provides functionalities for memory management;
- **Message Passing Manager**: This manager provides functionalities for communication and synchronisation using message queues;
- **Semaphores Manager**: This manager provides functionalities for semaphore management;
- **Thread Manager**: This manager provides the functionalities for thread management;
- **Thread Cancellation Manager:** This manager provides functionalities for thread cancellation;
- **Clock Manager**: This manager provides the functionalities for clock management like getting time or setting time;
- **Timer Manager**: This manager provides functionalities for timer management;
- **Key Manager**: This manager provides the functionalities for key management like key creation or deletion. Key values are opaque objects used to locate thread-specific data;
- **Condition Variable Manager**: This manager provides functionalities for conditional variables management like condition variable initialisation or signalling;
- **Mutex Manager**: This manager provides functionalities for mutex variables management like mutex priority setting or initialisation;
- **Signal Manager**: This manager provides functionalities for signal management like signal clearing or sending;
- **Scheduler Manager**: This manager provides functionalities for managing the scheduling options of the executive;

The implementation of the POSIX API is divided into three parts: RTEMS core, namely the files available in the `posix` and `libc` directories; Cygnus NEWLIB and gcc.

## 2.4 RTEMS Executive Core

The executive core is responsible for the low level system management. It implements several handlers to be used by the API's to perform each specific function. These handlers are presented next:

- **Message Handler –** this handler is responsible for message passing inside the core;
- **Mutex Handler –** this handler is responsible for mutex objects management inside the core;
- **Semaphore Handler –** this handler is responsible for semaphore management inside the core;
- **Time of Day Handler -** this handler is responsible for time management inside the core;
- **Heap Handler -** this handler is responsible for heap memory management inside the core;
- **Internal Error Handler -** this handler is responsible for managing internal errors that occur inside the core;
- **ISR Handler** - this handler is responsible for ISR management inside the core;
- **Multiprocessing Communications Interface Handler -** this handler is responsible for multiprocessing environment management inside the core;
- **Object Handler -** this handler is responsible for object management inside the core;
- **Thread Handler -** this handler is responsible for thread management inside the core;
- **Thread Queue Handler -** this handler is responsible for thread queues management inside the core;
- **Watchdog Handler -** this handler is responsible for software watchdog management inside the core;
- **Workspace Handler -** this handler is responsible for workspace management inside the core.

The core was not designed to be used directly by the user's application, although no restrictions are imposed, at either compilation or linking time.

As shown in Figure 1, the executive core manages all CPU specific features and low level device drivers. In this particular case, the CPU specific features belong to the SPARC/ERC32 processor. Features like low power mode or number of register windows are defined at this level.

The executive core also contains configuration structures. RTEMS system configuration is made through a file that contains all of the configuration tables required by an RTEMS application, including the CPU specific table.

## 2.5 API Description

Depending on the API (e.g. RTEMS Classic API is compliant with Real Time Executive Interface Definition (RTEID) and POSIX API is based on the standard IEEE 1003.1b) RTEMS provides specific managers. This section provides a relatively detailed description of the Classic RTEMS API manager's functionalities.

In this release, RTEMS presents three distinct API's. The first API was developed with compliance with the RTEID, by Motorola with technical input from Software Components Group. This is also mentioned on the documentation as the Classic RTEMS API. The second is a POSIX 1003.1b based API, whose implementation is based on a single process multithreaded environment. The third one is an Industrial The Real Time Operating System Nucleus[2] (ITRON) based API. All of these interfaces are mainly implemented in C programming language, with some Assembly code used in some CPU dependent files. Only Classic and POSIX APIs were subjected to this evaluation. The next section presents some details of the Classic RTEMS API (For an API directive description, refer to [6]). Description of the POSIX API can be found on POSIX and Cygnus NEWLIB documentation.

### 2.5.1 Classic API

As stated in section 2.3.2.1 and shown in Figure 2, Classic API is divided in seventeen resource managers, logically organised, each one with a group of specific directives according to their characteristics. One particular aspect regarding this API is that RTEMS excludes all unused managers from the run time environment. This way no unused code is placed in the run time.

These managers are described in the following subsections.

#### *2.5.1.1 Initialisation Manager*

This manager is responsible for initiating and shutting down RTEMS. Initiating RTEMS involves creating and starting all configured initialisation tasks, and for invoking the initialisation routine for each user-supplied device driver. For multiprocessor environments, this manager is responsible for the initialisation of the inter-processor communications layer.

Initialisation tasks are the mechanism by which RTEMS transfers initial control to the user's application. A typical initialisation task will create and start the static set of application tasks. Initialisation tasks which only perform initialisation should delete themselves upon completion to free resources for other tasks. RTEMS does not automatically delete the initialisation tasks. These tasks are defined in the User Initialisation Tasks Table and are automatically created and started by RTEMS as part of its initialisation sequence.

System Initialisation Task is responsible for initialising all device drivers. After device driver initialisation in a single processor system, this task will delete itself. In multiprocessor environments, the system initialisation task does not delete itself after initialising the device drivers. Instead, it transforms itself into the Multiprocessor Server which initialises the Multiprocessor Communications Interface Layer, verifies multiprocessor system consistency and processes all requests from the remote nodes.

The Idle Task is a task that consists in an infinite loop and will be pre-empted when any other task is made ready to execute. This happens because this task has the lowest priority in the system.

If a fatal error occurs during the initialisation, a Fatal Error Manager directive will be called to deal with the situation.

After the BSP completes its initialisation, the following sequence is performed by the Initialisation Manager:

- Initialise internal RTEMS variables;

[2] The ITRON specification defines a highly flexible operating system architecture designed specifically for application in embedded systems, see http://tron.um.u-tokyo.ac.jp/TRON/ITRON/ for more details.

- Allocate system resources;
- Create and start the System Initialisation Task;
- Create and start the Idle Task;
- Create and start the User Initialisation Task;
- Initialise Multitasking.

The RTEMS shutting down directive is called by the application when multitasking is to be ended and control is to be returned to the BSP. RTEMS will resume its execution when the RTEMS initialisation directive is called again by the BSP.

#### *2.5.1.2 Task Manager*

The task manager provides a comprehensive set of directives to manage and administer tasks.

A task in RTEMS is the smallest thread of execution which can compete on its own for system resources.

RTEMS defines a data structure called Task Control Block (TCB) which is used to keep all the information that is important to the execution of a task. RTEMS reserves a TCB for each configured task. This structure contains the task's name, ID, current priority, current and starting states, execution mode, set of notepad locations, TCB user extensions pointer, scheduling control structures and all the data required by a blocked task. The TCB is allocated upon creation of the task and released to the TCB free list upon task deletion.

In RTEMS, a task may exist in the following states:

- Executing – Currently scheduled to the CPU;
- Ready – May be scheduled to the CPU;
- Blocked – Unable to be scheduled to the CPU;
- Dormant – Created task that is not started;
- Non-Existent – Uncreated or deleted task.

The following figure presents a diagram that shows how the states relate themselves.

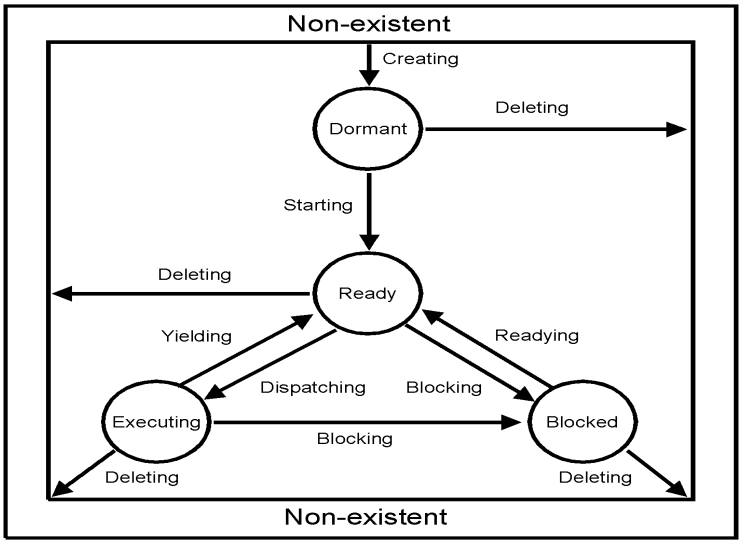


Figure 4. Task state Transitions diagram

Normally, the task scheduling is based on its current state and priority.

A task's priority determines its importance regarding to the remaining tasks executing in the system, more specifically, in the processor. RTEMS supports 255 level of priority, being the highest priority 1 and the lowest 255. Also, tasks in RTEMS can have equal priorities.

Task execution mode is a combination of four components:

- Preemption: preemption allows a task to determine when control of the processor is relinquished. If a higher priority task than the one currently using the processor is made ready, then the processor will be taken away from the currently running task and given to the higher priority task;
- ASR processing: Asynchronous Signal Routine (ASR) is used to determine when signals are to be processed by the corresponding task. When a signal arrives, it will be processed by the responsible task when it is scheduled to execute. If this feature is disabled, the signal is posted for execution when it becomes enabled;
- Timeslicing: timeslicing is used by RTEMS scheduler to determine how the processor is allocated when tasks of equal priority are made ready to execute. In this case, RTEMS will limit the amount of time each task executes, equally. The length of the time slice is application dependent and specified in the Configuration Table;
- Interrupt Level: interrupt level is used to determine which interrupts will be enabled when a task is executing, more specifically, it specifies the interrupt level at which the task will be executed.

Creation of a task which performs floating point operations in RTEMS results in additional memory being allocated for the TCB to store the state of the coprocessor during task switches. If the target processor does not support this feature, an emulation software library should be used for floating point operation, if needed.

In some processors it is possible to dynamically enable and disable the floating point unit. When this occurs, RTEMS will use this feature enabling only the floating point unit when the task currently executing needs it.

RTEMS implements a feature called Per Task Variables, which are used to support global variables whose value may be unique to a task. This feature can be used when a routine is to be spawned repeatedly as several independent tasks.

#### *2.5.1.3 Interrupt Manager*

The interrupt manager allows the application to connect a function to a hardware interrupt vector. RTEMS saves and restores all registers which are not preserved by normal C calling convention for the target processor and invokes the user's Interrupt Service Routine (ISR).

The RTEMS Interrupt Manager guarantees that proper task scheduling and dispatching is performed at the conclusion of an ISR. If proper task scheduling and dispatching is to be performed, the application must use the Interrupt manager for all ISR's which may be interrupted by the highest priority ISR which invokes an RTEMS directive.

RTEMS internally supports 256 interrupt levels which are mapped to the processor's interrupt levels.

During the execution of directive calls, critical sections of code may be executed. In this situation, all maskable interrupts are disabled before the execution of these sections. Non maskable interrupts cannot be disabled. If a directive is invoked during the execution of an ISR, unpredictable results may occur due to the inability of RTEMS to protect its critical sections.

Only some directives can be used in ISR. A relatively detailed list is available in [6], but in a nutshell, the managers that are available for use in ISR are:

- Task Management;
- Clock Management;
- Message, Event and Signal Management;
- Semaphore Management;
- Dual-Ported Memory Management;
- IO Management;
- Fatal Error Management;
- Multiprocessing Management.

#### *2.5.1.4 Clock Manager*

The Clock Manager provides support for time of day related capabilities. This manager requires a real time clock or hardware timer to create the timer interrupts. It operates based upon calendar time. The fields that compose the native format of the data structure that stores time are the following: year; month; day; hour; minute; second and tick[3].

This manager provides the directive needed for Timeslicing scheduling scheme. Basically, this directive decrements the running task's time-remaining counter. If the task's timeslice has expired, then the task will be pre-empted.

#### *2.5.1.5 Timer Manager*

The Timer Manager provides support for timer facilities. A timer is an RTEMS object which allows the application to schedule specific operations to occur at specific times in the future.

[3] Tick is defined to be an integral number of microseconds which is specified by the user in the Configuration Table.

As an example of this RTEMS feature, a timer can be used to implement software Watchdog routines, which only fires to denote that an application error has occurred.

#### *2.5.1.6 Semaphore Manager*

The Semaphore Manager provides support for synchronisation and mutual exclusion capabilities.

RTEMS allows that a task holding the binary semaphore to obtain the same binary multiple times in a nested manner. Simple binary semaphores do not allow nested access and so can be used for task synchronization.

RTEMS supports the priority inheritance and priority ceiling protocols for local, binary semaphores that use the priority task wait queue blocking policy. The implementation of the priority inheritance and priority ceiling algorithms take into account the scenario in which a task holds more than one binary semaphore.

#### *2.5.1.7 Message Manager*

This manager provides communication and synchronisation facilities using RTEMS message queues.

A message is a variable length buffer where information can be stored to support communication.

A message queue permits the passing of messages among tasks and ISR's. Normally, messages are sent and received from the queue in a FIFO order, but RTEMS provides a directive to pass to LIFO order for urgent messages.

#### *2.5.1.8 Event Manager*

This manager provides a high performance method for intertask communication and synchronisation.

Each task contains thirty-two event flags that inform the task of the occurrence of a significant situation. A collection of one or more event flags is referred to as an event set. This manager provides facilities to manage these event sets. RTEMS provides two algorithms for testing if the condition flagged by the event is satisfied or not. The first algorithm states that an event condition is satisfied when at least a single request event is posted[4]. The second states that an event condition is satisfied when every requested event is posted.

#### *2.5.1.9 Signal Manager*

This manager provides the features required for asynchronous communication.

This manager allows a task to optionally define an asynchronous signal routine. This mechanism works the same way as an ISR. When a signal is sent to a task, that task's execution path will be interrupted by the ASR. A signal is sent to a task to inform that task of the occurrence of a significant situation. ASRs are scheduled by the RTEMS and executed in the context of a task, consequently being able to execute any directive. A ASR has a task mode which can be different from that of the task.

[4] An event set is posted when is directed (or sent) to a task

#### *2.5.1.10 Partition Manager*

This manager provides facilities to dynamically allocate memory in fixed-size units.

A partition is a physically contiguous memory area divided into fixed-size buffers that can be dynamically allocated or deallocated. Partitions are managed and maintained as a list of buffers.

#### *2.5.1.11 Region Manager*

This manager provides facilities to dynamically allocate memory in variable sized units.

A region makes up a physically contiguous memory space with user-defined boundaries from which variable-sized segments are dynamically allocated and deallocated. This manager provides facilities similar to the Partition Manager.

#### *2.5.1.12 Dual Ported Memory Manager*

This manager provides facilities for converting addresses between internal and external representations for multiple dual-ported memory areas (DPMA).

The dual-ported memory area is a contiguous block of RAM memory owned by a particular processor but which can be accessed by other processors in the system. The owner accesses the memory using internal addresses, while other processors must use external addresses. RTEMS defines ports for particular mapping of internal and external addresses.

#### *2.5.1.13 I/O Manager*

This manager provides a well defined mechanism for accessing device drivers and a structured methodology for organising device drivers.

If an application uses the I/O Manager, it must specify the address of the Device Driver Table in its Configuration Table. This table contains the entry points for each specific device driver. Each device driver may contain the following entry points:

- Initialisation;
- Open;
- Close;
- Read;
- Write;
- Control.

Each call to this manager must provide a device's major and minor number as arguments. The major number is the number of the index of the requested driver's entry point in the Device Driver Table, and is used to select a specific device driver. The minor number usage is driver specific, but is commonly used to distinguish between a number of devices controlled by the same driver.

This manager also provides facilities for associating names to particular devices. It provides directives to register names and look up the major/minor number pair for the specific device name.

There are some considerations to be taken when using this manager:

- The device driver routines execute on the context of the invoking task. Thus, if the device driver blocks, the task blocks;

- The device driver is free to change the mode of the invoking task, although the device driver routine should restore the original values when it finishes the execution;
- Device drivers may be invoked from ISRs;
- Only local device drivers are accessible through the I/O manager;
- A device driver routine may invoke all other RTEMS directives, including I/O directives on both local and global objects.

The manager does not make any assumptions regarding the construction or operation of any device driver.

The information passed by the application to RTEMS is then passed to the correct device driver entry point.

RTEMS automatically initialises all device drivers when multitasking is initiated.

#### *2.5.1.14 Fatal Error Manager*

This manager provides facilities to process all fatal and irrecoverable errors.

This manager is called when an irrecoverable error condition is detected, either by the kernel or by the application software. Regarding to fatal errors, these can be detected from three sources:

- The executive (RTEMS);
- User system code;
- User application code.

RTEMS automatically invokes this manager upon detection of an error it considers to be fatal. Although the precise behaviour of the default fatal error handler is processor specific, in general, it will disable all maskable interrupts, place the error in a known processor dependent place (generally, either on the stack or in a register) and halt the processor. In this particular case, the SPARC/ERC32, the default error handler disables processor interrupts to level 15, places the code in the g1 register and goes into infinite loop to simulate a halt processor instruction.

#### *2.5.1.15 Rate Monotonic Manager*

This manager provides facilities to manage the execution of periodic tasks. It was designed to support application designers who use the Rate Monotonic Scheduling Algorithm to insure that their periodic tasks will meet their deadlines even under transient overload conditions.

The rate monotonic manager makes the following assumptions:

- A periodic task is one which must be executed in a regular interval;
- The intervals between successive iterations of the task is referred to as its period;
- Periodic tasks can be characterised by the length of their period and execution time;
- The period and execution time of a task can be used to determine the processor utilization for that task;
- Processor utilization is the percentage of processor time used and can be calculated on a per-task or system-wide basis.
- An aperiodic task executes at irregular intervals and has only a soft deadline;
- A sporadic task is an aperiodic task with a hard dealine and minimum interarrival time;

- The minimum interarrival time is the minimum period of time which exists between successive iterations of the task.

#### *2.5.1.16 User Extensions Manager*

This manager provides the applications developer facilities to augment the executive by allowing them to supply extension routines which are invoked at critical system events. These events can be the following system events:

- Task creation, initiation, reinitiation, deletion, context switch, begin and end;
- Post task context switch;
- Fatal error detection.

An extension set is defined as a set of routines which are invoked at each of the previously mentioned events, and at which user extensions routines are invoked.

RTEMS provides a pointer to a user-defined data area for each extension set to be linked to each task's control block. This set of pointers is an extension of the TCB and can be used to store additional data required by the user's extension functions.

#### *2.5.1.17 Multiprocessing Manager*

This manager provides facilities for implementing multiprocessor environments.

A major goal design of the executive was to transcend the physical boundaries of the target hardware configuration. This is achieved by presenting the application software with a logical view of the target system where the boundaries between processor nodes are transparent.

This executive supports heterogeneous and homogeneous environments, regarding processors. By having this kind of support, RTEMS allows systems designers to select the most efficient processor for each subsystem of the application. Moreover, configuring a heterogeneous environment is no more difficult than configuring a homogeneous environment.

With these features, the entire system, both hardware and software, can be viewed logically as a single system.

RTEMS implements proxies which are data structures that reside on every node and represent a remote task. The goal of this is to block remote tasks as though they were blocking on a message queue or semaphore. It also implements global objects and two types of tables; a Local Object Table, which contains the information concerning all objects created in that specific node; and a Global Object Table, which contains information regarding all global objects in the system and, consequently, is the same on every node

It also implements a Multiprocessor Communications Interface Layer. This is a set of user-provided procedures which enable the nodes in a multiprocessor system to communicate with one another. This layer is responsible for managing a pool of buffers called packets and for sending these packets between the nodes.

## 2.6 Product Deployment

The RTEMS version 4.5.0 is distributed via anonymous ftp. This release can be found in ftp://ftp.oarcorp.com/pub/rtems/releases/4.5.0 . The complete source code and documentation can be found in www.rtems.com . Almost all components of this RTEMS release are compressed files and have the .tar.gz or .tgz extension. The GNU Zip package is required to uncompress these files. The cross-development environment is based on GNU tools and can also be obtained from the web.

The following sections describe the directory structure of this release. The RTEMS directory structure is designed to meet the following requirements:

- Encourage the development of modular components;
- Isolate processor and target dependent code;
- Allow multiple RTEMS users to perform simultaneous compilation of RTEMS.

The documentation that describes the directory structure of RTEMS is not updated, so the following description is based on direct observation of contents.

### 2.6.1 Top Level Directory Structure

The top level directory for this release has the following contents:

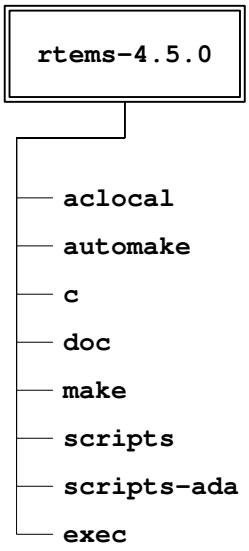


Figure 5. Top Level Directory structure

**aclocal**: This directory contains the macros used by the GNU Autoconf tool for configure RTEMS compilation environment.

**automake**: This directory contains information regarding the GNU Automake tool.

**c**: This directory contains the source code for the C implementation of RTEMS as well as the test suites, sample applications, Board Support Packages, Device Drivers, and support libraries. (This directory will be described in more detail in the next sections).

**doc**: This directory contains the language independent documentation. It is not currently supported (not used).

**make**: This directory contains the make files for RTEMS building. It contains three subdirectories: **template**, **custom** and **compilers** that support the building operations.

**scripts**: This directory contains the RedHat Package Manager (RPM) support for Board Support Packages (BSP). It also contains four subdirectories to support the building.

**scripts-ada**: This directory contains the scripts used to build RPMs for GNAT/RTEMS. It contains three subdirectories to support the building.

**tools**: This directory contains three subdirectories: **update**, **cpu** and **build**. The **update** directory contains tools which were used to aid the upgrade from RTEMS 3.1.0 to RTEMS 3.2.0. The **cpu** directory contains some tools developed by different persons for particular target systems, but with no significant practical use. The **build** directory contains miscellaneous support tools for RTEMS workspaces.

### 2.6.2 Source Code Directory - c

This directory contains two subdirectories: the **src** directory, which contains the source code for this version of the executive, and the **make** subdirectory, which contains files to aid the GNU Make tool.

The **c** directory also contains some files that are also used by the GNU Make tool and some text files containing information regarding the status of the current implementation.

The **src** subdirectory has the following structure:

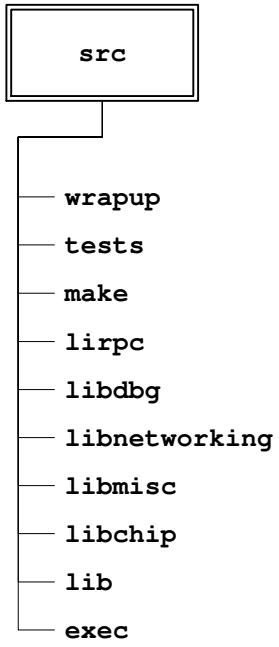


Figure 6. Src subdirectory

The **lib**, **exec** and **tests** subdirectories are important directories and are presented in some detail on the following subsections. The rest of the subdirectories are described next.

**wrapup:** This directory is practically empty. It only contains some files to aid the GNU Make tool;

**make:** This directory contains the make files for RTEMS building. It contains two subdirectories: **custom** and **compilers** that support the building operations;

**librpc:** This directory contains the necessary source files for using Remote Procedure Call / External Data Representation (RPC/XDR) routines on RTEMS;

**libdbg:** This directory contains source code for some remote debugging features to be used with RTEMS for the i386 and PowerPc processors;

**libnetworking:** This directory contains several subdirectories with source code of servers like FTP and TFTP and some others. All these servers are still under development.

It also contains some source code to implement virtual machines for RTEMS, an implementation of PPP (Point to Point Protocol) for RTEMS, as well as some other web based applications for RTEMS;

**libmisc:** This directory contains some monitoring tools, like a Task Stack Overflow Checker, Workspace Consistency Checker, Task Execution Time Monitor, Period Statistics Monitor or a Debug Monitor. It also contains a file compression utility called Untar.

**libchip:** This directory contains some specific drivers for serial connections and network connections.

#### *2.6.2.1 Support Library Source directory – `lib`*

This directory contains the support libraries and BSPs. The following figure shows its structure:

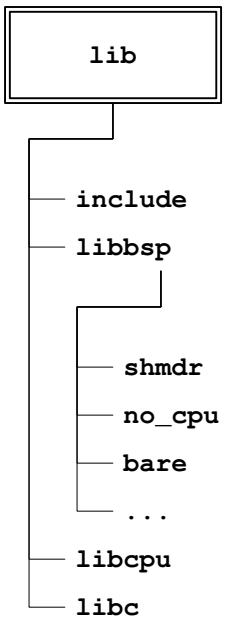


Figure 7. Lib and Libbsp subdirectories

The **lib** contains four subdirectories:

**include**: This directory contains several specification for generic drivers, common to all target systems.

**libbsp**: contains a subdirectory for each CPU family supported by RTEMS. In this subdirectory is another directory for each BSP for that processor family. This directory also contains three important subdirectories worth to mentioning:

**shmdr**: This directory provides an implementation of a shared memory driver. This driver is only required to execute the multiprocessor test suite;

**no_cpu**: This directory provides a template BSP which can be used to develop a specific BSP for an unsupported target board;

**bare**: This directory provides some definition to build RTEMS without using any specific BSP for any CPU type;

**libcpu**: contains libraries which are CPU dependent but not target board dependent;

**libc**: contains the support for the Cygnus NEWLIB C Library that was specifically designed for real-time embedded systems;

As stated before, together with these directories, there are CPU dependent subdirectories. Each BSP provides the modules which comprises a RTEMS BSP. These modules are separated into several directories. The number of directories changes with the target processor.

#### *2.6.2.2 Test Suite Source directory – `test`*

This directory contains the following subdirectories:

**samples:** This directory contains a set of simple sample applications which can be used either to test a board support package or as the starting point for a custom application. This directory has the following structure:

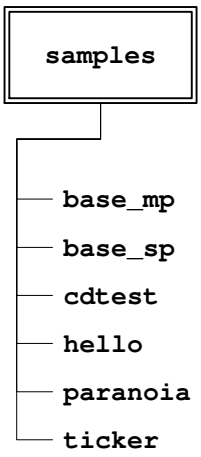


Figure 8. RTEMS Samples directory

The **base_mp** and **base_sp** provide sample implementation of multiprocessor and single processor environment respectively, the **cdtest** provides a simple C++ application using constructors and destructors, the **hello** provides the well know "Hello World" test application, the **paranoia** provides public domain floating point and math library toolset test and finally the **ticker** provides a test for the verification of clock chip device drivers of BSPs.

**mptests**: This directory contains the RTEMS Multiprocessor Test Suite.

**psxtests**: This directory contains the RTEMS POSIX API Test Suite.

**sptests**: This directory contains the RTEMS Single Processor Test Suite.

**tmtests**: This directory contains the RTEMS Timing Test Suite.

**libtests**: This directory contains tests for some of the items in the lib directories.

**tmitrontests**: This directory contains timing tests for the ITRON 3.0 implementation

**itrontests**: This directory contains functionality tests for the ITRON 3.0 API implementation.

#### *2.6.2.3 Executive Source directory – `exec`*

This directory contains the source code files of this RTEMS release. Its structure is presented next:

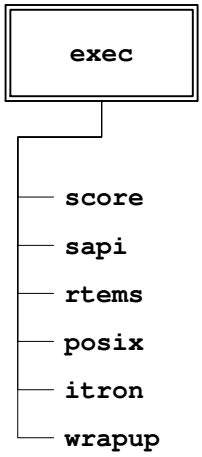


Figure 9. C Executive Directory

At this point the API specific and core source code are separated into different directory trees. The **`rtems`** , **`posix`** and **`itron`** subdirectories contain the C language source files for each module comprising the respective API. The **`rtems`** directory contains the source code for the Classic API, the **`posix`** directory contains a part of the implementation of the POSIX compliant API and the **`itron`** directory contains the implementation of the ITRON 3.0 compliant API. The other subdirectories **`score`** and **`sapi`** contain the super core modules. The **`sapi`** directory contains the files required to support initialisation and the **`score`** directory contains the CPU dependent modules.

# 3. Scope Definition

For efficiency and effectiveness, the scope of the robustness and stress testing activities was limited but yet covering a very significant part of all implemented functionalities in RTEMS. The rationale was to focus on the more critical components of the system.

This section presents the parts of RTEMS that were selected (from the architectural view) for robustness and stress testing. These parts are:

- Executive Core;
- Classic API (including Interface with the Low-Level Device Drivers);
- POSIX API;

These parts were chosen since they include at least one of the following:

- Directives that, in case of failure, can compromise the system integrity;
- Directives that, if misused, can compromise the system integrity;
- Directives that are frequently used to process important information to the system (synchronisation, scheduling, etc), and as consequence, can compromise the system integrity;

## 3.1 RTEMS Executive Core

This section presents the features of the executive core selected to be tested. Since all features present in the core are used by the APIs, all will be tested. The following table shows these features:

| Features Subject of Test |
|---|
| Message Handler; |
| Mutex Handler; |
| Semaphore Handler; |
| Time of Day Handler; |
| Heap Handler; |
| Internal Error Handler; |
| ISR Handler; |
| Object Handler; |
| Thread Handler; |
| Thread Queue Handler; |
| Workspace Handler; |
| Automatic Generation of System Configuration (AGSC), more specifically, the confdefs.h file and |

| |
|---|
| the macros that use this file for automatic generation of system tables; |
| Configurations tables generated by the previous system (AGSC); |
| RTEMS API Configuration Table; |
| CPU Dependent Information Table; |
| Initialisation Task Table; |
| Driver Address Table; |
| User Extensions Table;. |

Table 1. Selected Core Features

## 3.2 RTEMS Classic API

This subsection presents the resource managers chosen from the 17 managers provided in the RTEMS Classic API that were considered to be a critical part of the executive. Dual ported memory and Multiprocessing resources managers were excluded from the scope of the tests since the SPARC/ERC32 flavour of RTEMS does not implement the latter and there was no simulation/hardware facility to test the former.

The chosen resource managers and corresponding directives are shown in Table 2. All directives of the candidate resource managers were included with a few exceptions: directives without parameters and directives associated with handling '*per task global variables*' since they are only relevant for multi-processing systems. The same rationale also applied to the POSIX API.

| Features Subject of Test | |
|---|---|
| Manager | Directives |
| Task Manager | rtems_task_create<br>rtems_task_start<br>rtems_task_restart<br>rtems_task_delete<br>rtems_task_resume<br>rtems_task_set_priority<br>rtems_task_mode |
| Interrrupt Manager | rtems_interrupt_catch |
| Clock Manager[5] | rtems_clock_set<br>rtems_clock_tick |
| Timer Manager[5] | rtems_timer_create<br>rtems_timer_delete<br>rtems_timer_fire_after<br>rtems_timer_fire_when<br>rtems_timer_cancel<br>rtems_timer_reset |
| Semaphore Manager | rtems_semaphore_create<br>rtems_semaphore_delete<br>rtems_semaphore_obtain<br>rtems_semaphore_release<br>rtems_semaphore_flush |

[5] This manager must be tested because it is used in critical parts of other managers

| | |
|---|---|
| Message Manager | rtems_message_queue_create<br>rtems_message_queue_delete<br>rtems_message_queue_send<br>rtems_message_queue_urgent<br>rtems_message_queue_broadcast<br>rtems_message_queue_receive<br>rtems_message_queue_flush<br>rtems_message_queue_get_number_pending |
| Event Manager | rtems_event_send<br>rtems_event_receive |
| Signal Manager | rtems_signal_catch<br>rtems_event_send |
| Partition Manager | rtems_partition_create<br>rtems_partition_delete<br>rtems_partition_get_buffer<br>rtems_partition_return_buffer |
| Region Manager | rtems_region_create<br>rtems_region_delete<br>rtems_region_extend<br>rtems_region_get_segment_size<br>rtems_region_get_segment<br>rtems_region_return_segment |
| I/O Manager | rtems_io_initisalize<br>rtems_io_register_name<br>rtems_io_open<br>rtems_io_close<br>rtems_io_read<br>rtems_io_write<br>rtems_io_control |
| Fatal Error Manager | rtems_fatal_error_occured |
| Rate Monotonic Manager | rtems_rate_monotonic_create<br>rtems_rate_monotonic_delete<br>rtems_rate_monotonic_cancel<br>rtems_rate_monotonic_period |
| User Extensions Manager | rtems_extension_create<br>rtems_extension_delete |

Table 2. Selected Managers and Directives from the RTEMS Classic API

## 3.3 POSIX API

This subsection presents the managers chosen from the 7 managers provided in the RTEMS POSIX API that were considered to be a critical part of the executive.

The chosen managers and corresponding directives are the following:

| Features Subject of Test | |
|---|---|
| Manager | Directives |
| Signal Manager | Sigaddset<br>sigdelset<br>sigfillset<br>sigemptyset<br>sigaction<br>pthread_kill<br>kill<br>sigprocmask<br>sigsuspend |

| | |
|---|---|
| | sigwaitinfo<br>sigtimedwait |
| Mutex | pthread_mutexattr_init<br>pthread_mutexattr_destroy<br>pthread_mutexattr_setprotocol<br>pthread_mutexattr_setprioceiling<br>pthread_mutexattr_setpshared<br>pthread_mutex_init<br>pthread_mutex_destroy<br>pthread_mutex_lock<br>pthread_mutex_trylock<br>pthread_mutex_timedlock<br>pthread_mutex_unlock<br>pthread_mutex_setprioceiling |
| Clock Manager | clock_settime<br>sleep<br>nanosleep<br>clock_gettime |
| Timer Manager | timer_create<br>timer_delete<br>timer_settime |
| Message Manager | mq_open<br>mq_close<br>mq_unlink<br>mq_send<br>mq_receive<br>mq_notify<br>mq_setattr |

Table 3. Selected Managers and Directives from the RTEMS POSIX API

## 3.4 Robustness Testing Scope

The previously listed managers and its directives, both for the Classic and POSIX RTEMS APIs, were used during the robustness testing. They were all subject of test cases definitions according to the fault model defined for robustness testing.

## 3.5 Stress Testing Scope

For the stress testing, the scope defined was reduced: only the Classic API was tested. The selected managers were:

- Task Manager;
- Semaphore Manager;
- Event Manager;
- Interrupt Manager;
- Signal Manager;
- Message Manager;
- Partition Manager.

# 4. Fault Model and Test Methodology

## 4.1 Introduction

This chapter encloses the main findings of the work package WP-RAMS02-220 – "Fault Model Definition". It describes the test methodology and associated fault model to be used in the robustness and stress testing of the RTEMS operating system.

## 4.2 Robustness testing

### 4.2.1 Test methodology

The methodology used in this robustness testing of the RTEMS real-time kernel, consists in testing the RTEMS API calls using out-of-bound parameters.

This methodology is composed by several phases (see Figure 10):

- **Preparation:** Includes all the tasks needed to define the test cases.
- **Test Execution:** Execution of the defined test cases.
- **Log Analysis:** Analysis of the results of the test cases and identification of the RTEMS faults.

Preparation phase comprises the following tasks:

- **Product Analysis and Scope Definition:** Analysis of the product under evaluation (i.e. the RTEMS 4.5.0) and selection of the API calls that will be subjected to the evaluation.
- **Fault Model Definition:** Definition of the in-bound and out-of-bound values that will be used for each of the RTEMS data types.
- **Construction of the Workloads:** Definition and implementation of the applications that will exercise the RTEMS APIs.
- **Definition of the Test Campaigns and Test Suites:** definition of the test suites that will be used to automatically generate the test cases. Test suites are grouped logically in test campaigns.

The Test Execution phase follows the Preparation. During this phase test cases are executed and the results are collected in a database. This task is performed in unattended mode the by Xception.

The final phase of the robustness testing is the Log Analysis. In this phase detailed analysis of the log of each test case is performed comparing the obtained results against the expected values. This phase can be time consuming. For this reason it is important to have a concise workload output that enables the analyst to quickly find out if the result of the test case is consistent with the input parameters or not.

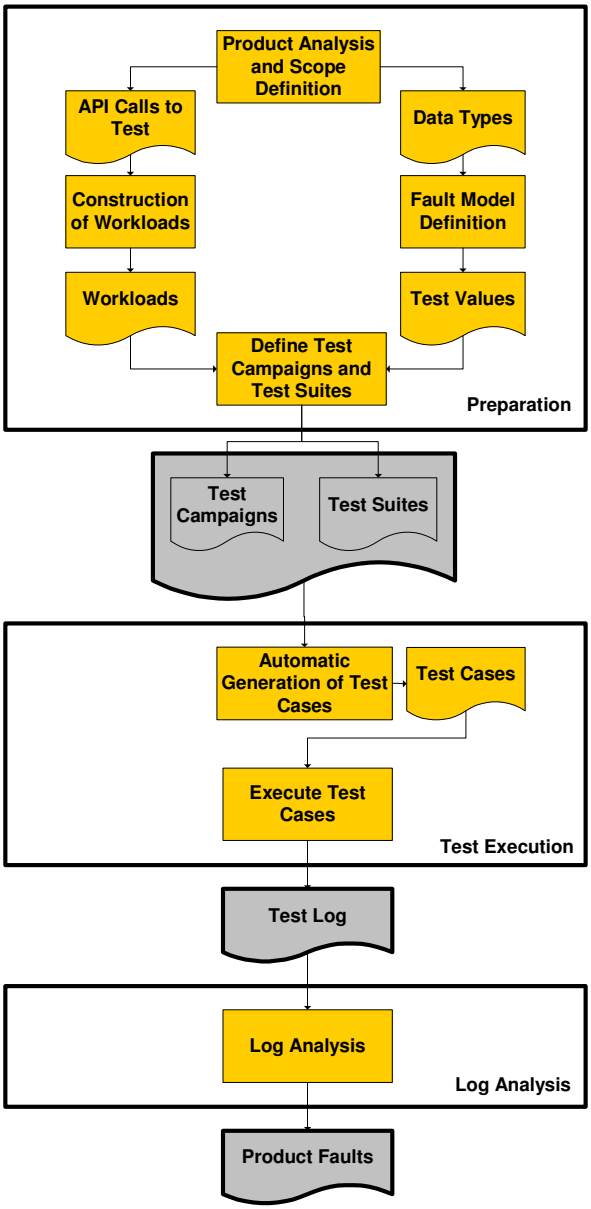


Figure 10 - Robustness Testing Methodology

### 4.2.2 Data types

For each data type, a class of test values was defined in order to build the mutants (a mutant is the source code file that results from the application of a single mutation on a data parameter). These values differ from typical data values fed to functional tests (or unit tests) since they should exercise the error handling/robustness features of the part of the software that they are exercising. These values typically reflect boundary or "magic" values in the data type range, or values that are semantic out-of-bounds in the scope of usage in a function call.

The test values for the basic data types were defined taking into account previous experience from the application of robustness testing techniques at Critical Software SA and elsewhere, namely from Ballista project [11]. The initial candidate of test values was as follows:

- **Integers data types:** 0, 1, -1, MAX_INTEGER, MIN_INTEGER, selected powers of two, powers of two minus one, powers of two plus one;

- **Pointers data types:** NULL, -1 (cast to a pointer), pointers to *free()'ed* memory, pointers to *malloc()'ed* buffers of various powers of two in size.
- **Floats data types:** 0.0, 1.0, -1.0, ±MAX_FLOAT, ±MIN_FLOAT, PI and e;

As the test cases were to be executed over the RTEMS kernel, the floating point data types were discarded. Concerning the other types, in order to prevent an explosion in the count of test cases but on the other hand, keeping good test coverage, only a subset of the specified test values were selected. Still, some rules apply. If the type to be used in a mutation is a pointer to a function, then the only possible value that will be used to create mutants is the NULL pointer. This decision is based on the fact that it is not interesting to pass invalid function pointers as parameters (e.g., to register a device driver with functions that have invalid signature).

These test values are not supposed to represent all the values that may be interesting to test with the given data types – they were chosen to provide a reasonable range of exceptional (non-nominal) input conditions to the software under test.

Table 4 lists the main basic RTEMS data types and the selected set of candidate test values.

Table 4 shows RTEMS structured types which will also be candidate placeholders for mutations on its data members.

| Data Type | Aliases | Basic Type | Test Values |
|---|---|---|---|
| __uint32_t | __ULong | unsigned long int | 0, 1, 18446744073709551615 (i.e. MAX_UINT) |
| Char | unsigned char | | 0, 255 |
| clockid_t | _CLOCKID_T_ | unsigned long int | 0, 1, 18446744073709551615 |
| Int | signed int, test | | 0, 1, -1, 2147483647, -2147483648 |
| Long | | | 0, 1, -1, 9223372036854775807, -9223372036854775808 |
| mode_t | | int | 0, 1, -1, 2147483647, -2147483648 |
| mqd_t | Objects_Id | unsigned int | 0, 1, 4294967295 |
| pid_t | | int | 0, 1, -1, 2147483647, -2147483648 |
| pthread_conbild_t | | unsigned long int | 0, 1, 18446744073709551615 |
| pthread_key_t | | unsigned long int | 0, 1, 18446744073709551615 |
| pthread_mutex_t | | unsigned long int | 0, 1, 18446744073709551615 |
| pthread_t | | unsigned long int | 0, 1, 18446744073709551615 |
| rtems_asr_entry | | void | NULL |
| rtems_attribute | | unsigned int | 0, 1, 4294967295 |
| rtems_device_major_number | | unsigned int | 0, 1, 4294967295 |
| rtems_device_minor_number | | unsigned int | 0, 1, 4294967295 |
| rtems_event_set | | unsigned int | 0, 1, 4294967295 |
| rtems_id | Objects_Id | unsigned int | 0, 1, 4294967295 |
| rtems_interval | Watchdog_Interval | unsigned int | 0, 1, 4294967295 |
| rtems_isr_entry | | void | NULL |
| rtems_mode | Modes_Control | unsigned int | 0, 1, 4294967295 |
| rtems_name | unsigned32 | unsigned int | 0, 1, 4294967295 |
| rtems_option | | unsigned int | 0, 1, 4294967295 |
| rtems_signal_set | | unsigned int | 0, 1, 4294967295 |
| rtems_signed16 | signed16 | signed short int | -32768, 0, 1, 32767 |
| rtems_signed32 | signed32 | signed int | 0, 1, -1, 2147483647, -2147483648 |
| rtems_signed8 | signed8 | signed char | 0, -128, 127 |
| rtems_status_code | | int | 0, 1, -1, 2147483647, -2147483648 |
| rtems_task_entry | | void | NULL |
| rtems_task_priority | Priority_Control | unsigned int | 0, 1, 4294967295 |
| rtems_timer_service_routine_entry | | void | NULL |
| rtems_unsigned16 | unsigned16 | unsigned short int | 0, 1, 65535 |
| rtems_unsigned32 | unsigned32 | unsigned int | 0, 1, 4294967295 |
| rtems_unsigned8 | unsigned8 | unsigned char | 0, 1, 255 |
| rtems_vector_number | ISR_Vector_number | unsigned int | 0, 1, 4294967295 |
| sem_t | | int | 0, 1, -1, 2147483647, -2147483648 |
| signed char | | | 0, -128, 127 |

| Data Type | Aliases | Basic Type | Test Values |
|---|---|---|---|
| signed short int | short, short int | | -32768, 0, 1, 32767 |
| sigset_t | | unsigned long int | 0, 1, 18446744073709551615 |
| sigval | union sigval | void | NULL |
| size_t | | int | 0, 1, -1, 2147483647, -2147483648 |
| ssize_t | | int | 0, 1, -1, 2147483647, -2147483648 |
| time_t | _TIME_T_ | long | 0, 1, -1, 9223372036854775807, -9223372036854775808 |
| timer_t | _TIMER_T_ | unsigned long int | 0, 1, 18446744073709551615 |
| unsigned int | | | 0, 1, 4294967295 |
| unsigned long int | | | 0, 1, 18446744073709551615 |
| unsigned short int | unsigned short | | 0, 1, 65535 |
| User_extensions_fatal_extension | | void | NULL |
| User_extensions_thread_begin_extension | | void | NULL |
| User_extensions_thread_create_extension | | void | NULL |
| User_extensions_thread_delete_extension | | void | NULL |
| User_extensions_thread_exitted_extension | | void | NULL |
| User_extensions_thread_restart_extension | | void | NULL |
| User_extensions_thread_start_extension | | void | NULL |
| User_extensions_thread_switch_extension | | void | NULL |
| Void | | | NULL |

Table 4 – RTEMS Basic Data Types and Associated Test Values

### 4.2.3 RTEMS API functions as fault placeholders

Once eligible data types are selected and a set of associated test values is defined, the next step in the methodology points us to the identification of the candidate locations for data mutation.

The target set of functions (or directives as RTEMS documentation calls them) were already defined and listed in chapter 3. This step applies then for the location of actual function calls in the set of application programs (workload files) applicable for a particular test suite. This step is performed in an automated way with full support from the test tool, which parses the selected files. This parsing will result in a list of functions (with its signatures, as in Figure 12). For details on how theses features are supported in the tool please check chapter 5.

For example, if we had the following two C functions:

```
rtems_status_code rtems_task_create(
  rtems_name           name,
  rtems_task_priority  initial_priority,
  unsigned32           stack_size,
  rtems_mode           initial_modes,
  rtems_attribute      attribute_set,
  Objects_Id          *id
);

rtems_status_code rtems_task_delete(
```

```
  Objects_Id id;
);
```

Figure 11 - Example C functions signatures

The signatures for them are as shown in Figure 12.

```
<?xml version="1.0" encoding="UTF-8"?>
<Message type="RTEMSFunctionsConfig">
  <Function Name="rtems_task_create" ReturnType="rtems_status_code"
        IsPointer="NO">
    <ParametersList>
      <Parameter Name="name" Type="rtems_name" IsPointer="NO"/>
      <Parameter Name="initial_priority" Type="rtems_task_priority"
        IsPointer="NO" />
      <Parameter Name="stack_size" Type="unsigned32" IsPointer="NO" />
      <Parameter Name="initial_modes" Type="rtems_mode" IsPointer="NO" />
      <Parameter Name="attribute_set" Type="rtems_attribute" IsPointer="NO" />
      <Parameter Name="id" Type="Objects_Id" IsPointer="YES" />
    </ParametersList>
  </Function>
  <Function Name="rtems_task_delete" ReturnType="rtems_status_code"
        IsPointer="NO">
    <ParametersList>
      <Parameter Name="id" Type="Objects_Id" IsPointer="NO" />
    </ParametersList>
  </Function>
</Message>
```

Figure 12 - XML Containing Functions Signatures

Note: The “IsPointer” attributes in the “Function” and “Parameter” elements must have one of two values: “NO” or “YES”. If the “IsPointer” is “YES”, then the parameter (or the function return type, in the case of the “Function” element) is a pointer.

Besides setting the ground for the next stage – automated test procedure generation – the output of this step enables comparison of the actual set of functions (covered within a test campaign) with the candidate set.

Automated test cases generation

Once program files (workloads), candidate functions and test values (classed by data types) are defined, the algorithm to automatically generate the test cases is straightforward.

A test case is generated by defining one mutant of the original application file. This mutant differs from the original application on a single source code instruction (in this specific case, a function call) where a single parameter at a time is mutated with a test value, according with its basic type. Files are parsed sequentially and for each parameter of a call, a count of X tests is generated where X is the size of the set of the test values for that data type.

The tool provides support for the tester to select a subset of the total set of test procedures possible.

### 4.2.4 Remarks

While tuned for the specific test campaigns of RTEMS the methodology herein described is completely generic and can be applied to other products.

The same reasoning applies to the test facility, with the following remarks:

1. The product is coded in the C programming language.

2. The functions to be tested can be any function for which the source code is available or, at least, the signature is known.

## 4.3 Stress testing

### 4.3.1 Test methodology

The approach used in the stress testing of the RTEMS 4.5.0 consisted in the execution of several workloads that makes extremely high usage of system resources. Two types of resources were evaluated:

- Physical resources – CPU time and memory;
- Logical resources – task, semaphores, message queues, etc.

Stressing of these resources was accomplished by:

- Creating a large number of resources (only for logical resources);
- Making intensive use of the created resources using different entities (e.g. a large number of tasks accessing the same message queue).

The usage of the resources was gradually increased until it gets to a value which the system could not stand. This is possible because RTEMS 4.5.0 has no hard-coded limits to the several types of objects that can be created (task, message queues, etc), being this limit defined by the applications at compile time and by the available physical resources (mainly, the memory).

The methodology used consists of three phases (see Figure 13):

- **Preparation:** Includes all the activities needed to define the test cases.
- **Test Execution:** Execution of the defined test cases.
- **Log Analysis:** Analysis of the results of the test cases execution and identification of the RTEMS 4.5.0 faults.

The Preparation phase comprises four tasks:

- **Product Analysis and Scope Definition:** Analysis of the product under evaluation (i.e. the RTEMS 4.5.0) and selection of system resources that will be subjected to stress testing evaluation.
- **Construction of Workloads:** definition and implementation of applications that will exercise the system resources.
- **Stress Model Definition:** Definition of the desired level of usage for the resources under evaluation (e.g. define the number of tasks to be created).
- **Definition of Test Campaigns and Test Suites:** definition of the test campaigns, test suites and test cases. Test cases are logically grouped in test suites which in turn are logically grouped in test campaigns.

During the Test Execution phase the test cases are executed with support of Xception. The results are collected into a database to be analysed in the following phase. This task is performed in a fully automated mode by Xception.

The final phase of this methodology is the Log Analysis. In this phase a detailed analysis of the log of each test case is performed and the results are reported. Typically a list of product faults is compiled at this time. For this phase it is important to

have clear and well structured log outputs from tests, so that the analysis is performed quickly and correctly.

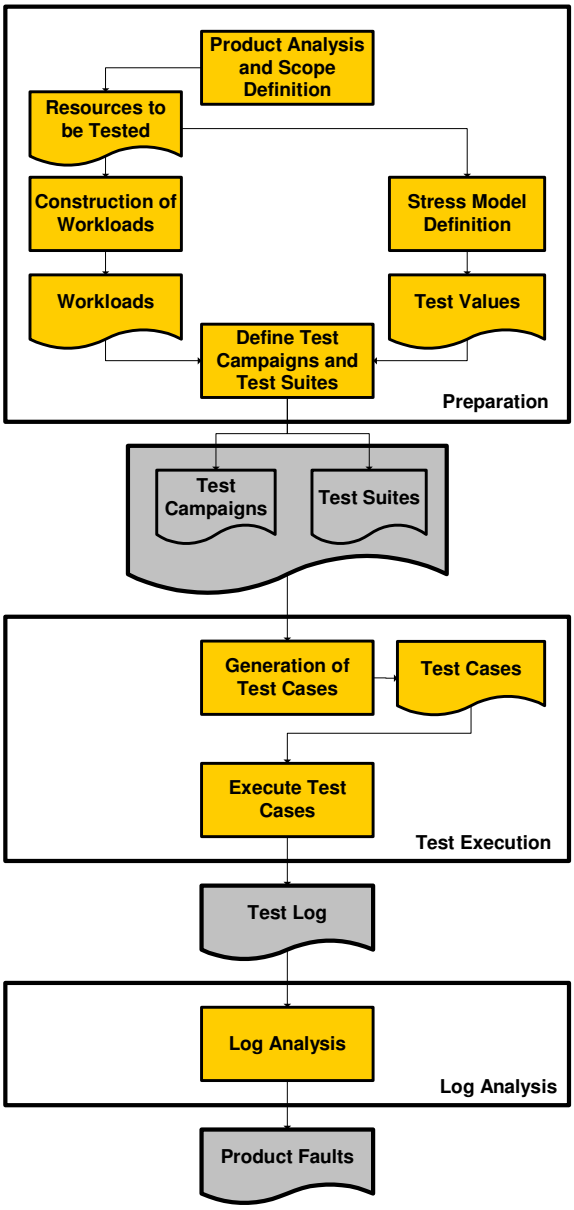


Figure 13 – Stress Testing Methodology

### 4.3.2 Workloads definition

In order to perform the evaluation some applications that exercise the RTEMS directives under test are required. In this evaluation, a set of such applications, called workloads, are used.

Each selected RTEMS resource manager will be stress tested with a different workload. However, every workload follows a similar approach: they all implement a scalable producer/consumer algorithm that uses the selected manager resources.

There are several parameters that can be modified in the workload in order to adjust the degree of load of an RTEMS resource manager. Although there are several workload specific parameters (e.g., the maximum size of a message in a message

queue for the Message Manager workload), some generic parameters are common to all the workloads. These generic parameters are listed in Table 5.

| Data Type | Parameter Name | Description |
|---|---|---|
| rtems_unsigned32 | NUMBER_OF_PRODUCERS | Number of producer tasks |
| rtems_unsigned32 | PRODUCERS_TASK_STACK_SIZE | Producers tasks stack size |
| rtems_task_priority | PRODUCERS_PRIORITY | Producers priority |
| rtems_mode | PRODUCERS_TASK_MODE | Producers tasks mode |
| rtems_attribute | PRODUCERS_TASK_ATTR | Producers tasks attributes |
| rtems_unsigned32 | NUMBER_OF_CONSUMERS | Number of consumers tasks |
| rtems_unsigned32 | CONSUMERS_TASK_STACK_SIZE | Consumers tasks stack size |
| rtems_mode | CONSUMERS_TASK_MODE | Consumers tasks mode |
| rtems_attribute | CONSUMERS_TASK_ATTR | Consumers tasks attributes |
| rtems_task_priority | CONSUMERS_PRIORITY | Consumers priority |
| rtems_unsigned32 | NUMBER_OF_SYSTEMS | Number of systems (producers/consumers systems) |
| rtems_interval | TEST_TIMEOUT | Number of ticks to wait until workload finishes |

Table 5 - Workload Generic Parameters

A very important generic parameter is NUMBER_OF_SYSTEMS. This parameter controls the scalability of the workload. If, for instance, we have only one producer (NUMBER_OF_PRODUCERS with value 1) and two consumers (NUMBER_OF_CONSUMERS with value 2), but the number of systems is set to ten, this means that ten (1 x 10) producers will produce (workload specific) items that will be consumed by twenty (2 x 10) consumers.

All the parameters are defined at workload compilation time through the C pre-processor #define directives in the workload header files.

The source code of the defined workloads can be found in Annex B.

### 4.3.3 Stress Model

A systematic approach was defined to stress the RTEMS resource managers through the definition of set of values to the different workload parameters. These sets of values were defined in the entire range of possible values for each parameter, taking into account their data type.

When more than one parameter is being modified at the same time within a workload, a combination of all the workload parameters values will be used, obtaining a representative sampling of the parameters interconnection.

To generate the values of the workload parameters, the following formula was used:

$$test\_values = \left\{ 2^{\left( m^{\left( \frac{i}{n+1} \right)} \right)}, \forall i \in \{1, n\} \right\}$$

Equation 1 - Test Values Formula

where m is the size of the data type (e.g., 32 for the rtems_unsigned32 type) and n is the desired number of values.

The main goal of this equation was to generate non-linear distribution of test values, in a way that most of the values are small values and only a small percentage of them are large values. The reason for using this type of distribution is that most of the larger values will result in an invalid parameter. For instance, the number of producers tasks is given by an unsigned 32 bits data type. However, with the available resources on the target system (mainly, the RAM memory), it is not possible to create 232 tasks along with the consumers and all the other resources needed by a workload. With this equation, we reduce the number of invalid parameters that would have no meaning in stress tests.

As all the workload parameters are unsigned 32 bits data types, the m was defined with 32. The n variable was set to 4, i.e., 4 test values were defined for each workload parameter. The reason for setting a low value to n was that the number of test cases explodes with the increase of this variable. For instance, if a test suite mutates 3 workload parameters, with 4 test values, 64 test cases would be generated; if the number of test values was 5, then 125 test cases would be generated. The following table presents the test values to the workload parameters as provided by Equation 1.

| n | Test values for 32 bits types |
|---|---|
| 1 | 4 |
| 2 | 16 |
| 3 | 256 |
| 4 | 65536 |

Table 6 - Parameters Test Values

# 5. Test Set-up and Execution Environment Description

## 5.1 Introduction

The Xception toolset was selected to execute the test cases defined in WP-RAMS02-310 – "Define Robustness Test Cases" and WP-RAMS02-410 – "Define Stress Test Cases".

The reason of selecting Xception to execute these test cases is the ability that this product has to generate source code level mutants in a workload application that interfaces with the operating system. Apart from the creation of mutants, Xception gives the user the chance to automatically compile and load them into the target system (simulator or board) and execute them, collecting its outputs.

Xception RTEMS-ERC32-SW provides an environment where a user can quickly generate, in a systematic and easy way, mutated versions of an RTEMS application. These mutations will then be executed in an unattended mode in the target system, performing robustness/stress test on specific function calls, namely on the RTEMS Classic API and on the RTEMS POSIX API.

## 5.2 Test Execution Process Overview

The test execution process is accomplished by the following steps

- Introduction of the defined test campaigns and test suits in the Xception and automatic generation of the test cases.
- Execution of the test cases
- Test Log collection

Figure 14 illustrates the test execution process using the Xception environment when configured for the ERC32/RTEMS target system.

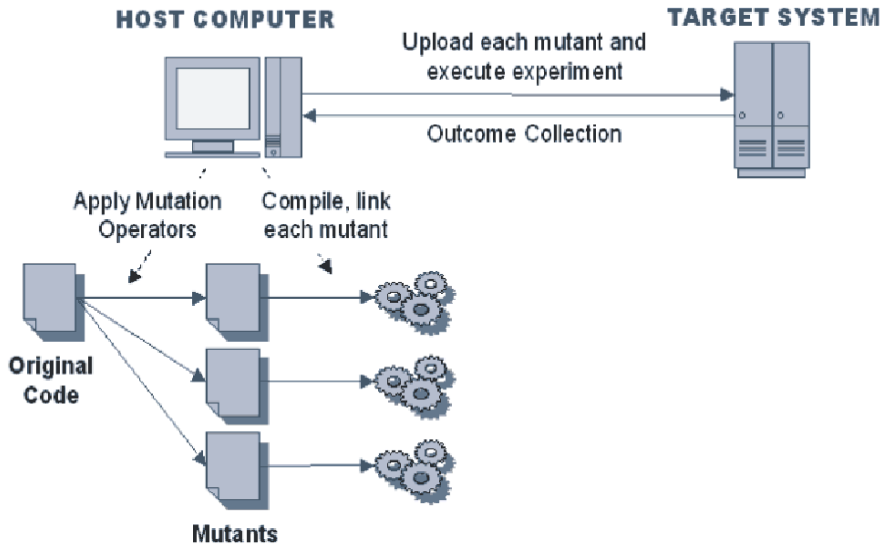


Figure 14 - Software Fault Injection Process

All operations carried out by the user during the test execution process are performed using the Xception Experiment Management Environment (EME). The EME will run on a workstation in which a cross-compilation system (CCS) shall be available. In the Xception terminology, this workstation is called the *Host*.

The introduction of the test campaigns and test suites related data is performed using a wizard. After this step test cases are automatically generated accordingly the test values defined for each of the data types and accordingly the set of selected API function defined as fault placeholders.

The test values are defined in an external XML file that describes all the known basic types, its aliases (typedef'ed data types) and their test values. This file may be edited by the user to add some more data types or test values. Data structures are also defined in this XML file. The user may also add some data structures to the file. However, the mutations on data structures are performed taking into account it's basic types.

Listing in Figure 15 shows an example of the XML file containing the type's definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<Message type="RTEMSTypesConfig">
  <DataType Name="rtems_unsigned8">
    <Aliases>
      <Alias>unsigned8</Alias>
    </Aliases>
    <BasicType>unsigned char</BasicType>
    <TestValues>
      <!-- UCHAR_MIN -->
      <Value>0</Value>
      <Value>1</Value>
      <!-- UCHAR_MAX -->
      <Value>255</Value>
    </TestValues>
  </DataType>
  <DataStructure Name="siginfo_t">
    <Aliases>
      <Alias>struct siginfo_t</Alias>
    </Aliases>
    <AttributesList>
      <Attribute Name="si_signo" Type="int" />
      <Attribute Name="si_code" Type="int" />
      <Attribute Name="si_value" Type="sigval" />
    </AttributesList>
  </DataStructure>
</Message>
```

Figure 15: XML Containing Data Types Definition

The Xception RTEMS-ERC32-SW already has a pre-defined set of functions definitions from RTEMS APIs that will be searched in the source code during the parsing (those listed in Table 2 and Table 3). However, the user can add (or remove) new functions to this set by either using the fault definition wizard or by manually editing the XML configuration file in the EME configuration directory.

During the execution of the test cases, Xception creates the mutants following the test case definition. Then, the executable binary is built using the CCS. The faulty application (*mutant*) is finally uploaded to the target system and executed. Data related to the execution can be logged in order to ease the result analysis process.

All the experiments will be executed on a target system simulator, as it is much more efficient than using the actual board with the ERC32 processor. The simulator that will be used is the GNU GDB embedded SiS.

All data collected during the experiment is transferred to the host and stored on the Xception database.

## 5.3 Test Log Analysis

A convention was defined for the output of the robustness testing workload applications. The goal was to get a homogeneous output for every different workload in order to easy the result analysis. It was defined that each workload shall send to the standard output every return value of the functions being tested and, eventually, every error code. These values shall be printed in a single line and with the following format:

- `<Function Name>(): <Return Value>;`

where `<Function Name>` is the name of the function called and `<Return Value>` is the value that was returned by the function (or an error code). For each tested function, an assertion shall also be printed, indicating whether the function call did what it was supposed to do or not.

By using this very simple convention, results analysis can be simplified by the use of scripts to parse the output.

The result analysis is supported by a relational database system where test cases definition and test log are stored.

# 6. Results Summary

## 6.1 Robustness Testing Results

Two different APIs of the RTEMS 4.5.0 were subject to robustness testing:

- RTEMS Classic API and
- RTEMS POSIX API.

### 6.1.1 Classic API

During the robustness testing of the Classic API 527 test cases were defined. The execution of these test cases raised 34 issues that shall be analysed as potential RTEMS faults. Table 7 shows the distribution of the test cases and faults among the several RTEMS managers.

| Manager | Test Cases | Raised Issues |
|---|---|---|
| Clock | 68 | 0 |
| Event | 18 | 0 |
| Fatal Error | 3 | 0 |
| Interrupt | 5 | 0 |
| IO | 50 | 6 |
| Message | 83 | 8 |
| Partition | 27 | 2 |
| Rate Monotonic | 24 | 1 |
| Region | 67 | 7 |
| Semaphore | 33 | 1 |
| Signal | 10 | 1 |
| Task | 55 | 4 |
| Timer | 67 | 3 |
| User Extensions | 17 | 1 |
| **Total** | **527** | **34** |

Table 7 – Classic API Robustness Testing: Test Cases/Raised Issues per Manager

Table 8 shows the criticality of the raised issues per RTEMS resource manager.

| Manager | Critical | Minor | Total |
|---|---|---|---|
| Clock | 0 | 0 | **0** |
| Event | 0 | 0 | **0** |
| Fatal Error | 0 | 0 | **0** |
| Interrupt | 0 | 0 | **0** |
| IO | 5 | 1 | **6** |
| Message | 2 | 6 | **8** |
| Partition | 0 | 2 | **2** |
| Rate Monotonic | 0 | 1 | **1** |
| Region | 4 | 3 | **7** |
| Semaphore | 0 | 1 | **1** |
| Signal | 0 | 1 | **1** |
| Task | 2 | 2 | **4** |
| Timer | 2 | 1 | **3** |
| User Extensions | 0 | 1 | **1** |
| **Total** | **15** | **19** | **34** |

Table 8 – Classic API Robustness Testing: Raised Issues Criticality per Manager

### 6.1.2 POSIX API

During the robustness testing of the POSIX API 528 test cases were defined. The execution of these test cases raised 15 issues that shall be analysed as potential RTEMS faults. Table 9 shows the distribution of the test cases and faults among the several RTEMS managers.

| Manager | Test Cases | Raised Issues |
|---|---|---|
| Clock | 32 | 0 |
| Message | 122 | 3 |
| Mutex | 223 | 4 |
| Signal | 122 | 5 |
| Timer | 29 | 3 |
| **Total** | **528** | **15** |

Table 9 – POSIX API Robustness Testing: Test Cases/Raised Issues per Manager

Table 10 shows the criticality of the raised issues per RTEMS resource manager.

| Manager | Critical | Minor | Total |
|---|---|---|---|
| Clock | 0 | 0 | 0 |
| Message | 2 | 1 | 3 |
| Mutex | 1 | 3 | 4 |
| Signal | 1 | 4 | 5 |
| Timer | 0 | 3 | 3 |
| **Total** | **4** | **11** | **15** |

Table 10 – POSIX API Robustness Testing: Potential Faults Criticality per Manager

### 6.1.3 Overall Results

The overall number of test cases and raised issues during the robustness testing is shown in Table 11.

| API | Test Cases | Raised Issues |
|---|---|---|
| Classic | 527 | 34 |
| POSIX | 528 | 15 |
| **Total** | **1055** | **49** |

Table 11 - Overall Results

## 6.2 Stress Testing Results

### 6.2.1 Classic API

During the stress testing of the RTEMS 4.5.0 Classic API, 452 test cases were defined. The execution of these test cases resulted in 74 pass and 378 fails. Table 12 shows the distribution of the test cases and faults among the several RTEMS managers.

| RTEMS Manager | Number of Test Cases | Number of Test Cases failed |
|---|---|---|
| Task Manager | 20 | 16 |
| Semaphore Manager | 68 | 51 |
| Message Manager | 80 | 74 |
| Signal Manager | 68 | 56 |
| Interrupt Manager | 20 | 14 |
| Event Manager | 68 | 56 |
| Partition Manager | 128 | 111 |
| **Total** | **452** | **378** |

Table 12 - Number of Passed/Failed Test Cases

During the test cases execution, three different situations lead to a test case failure:

- Application linkage failure: when the linker fails to create a binary image to execute. The ld command issues an error message saying that the region ram is full for a specified binary section (the .bss section).

- RTEMS initialisation failure: RTEMS fails to initialise. During the test cases execution, RTEMS failed the initialisation in two different ways. In the first one, it failed after detecting that there was not enough RAM memory to initialise the application. This failure is detected and the initialisation is aborted. In the second, RTEMS fails after trying to initialise the application accessing an invalid (inexistent) memory address. In this case, the error is not detected.

- RTEMS objects initialisation failure: the application starts running but fails on the creation of the specified resources due to lack of memory. These resources, e.g. message queues or new tasks, are dynamically created by the workload according to the workload parameters.

The following table provides a synthesis of the distribution of these situations by the different managers.

| RTEMS Manager | Application Linkage | RTEMS Initialisation | RTEMS Objects Initialisation | Total |
|---|---|---|---|---|
| Task Manager | 0 | 16 | 0 | 16 |
| Semaphore Manager | 19 | 32 | 0 | 51 |
| Message Manager | 19 | 37 | 18 | 74 |
| Signal Manager | 31 | 25 | 0 | 56 |
| Interrupt Manager | 3 | 11 | 0 | 14 |
| Event Manager | 31 | 25 | 0 | 56 |
| Partition Manager | 19 | 40 | 52 | 111 |
| **Total** | **122** | **186** | **74** | **378** |

Table 13 - Test Cases Failure Distribution

As stated above, the RTEMS initialisation failures may be separated in two categories: detected and not detected. Detected are those where the RTEMS initialisation process aborts and exits with an error; not detected lead to an invalid memory access. Table 14 presents the distribution of these two kinds of initialisation failures.

| RTEMS Manager | Detected | Not Detected | Total |
|---|---|---|---|
| Task Manager | 8 | 8 | 12 |
| Semaphore Manager | 12 | 20 | 32 |
| Message Manager | 12 | 25 | 37 |
| Signal Manager | 12 | 13 | 25 |
| Interrupt Manager | 4 | 7 | 11 |
| Event Manager | 12 | 13 | 25 |
| Partition Manager | 12 | 28 | 40 |
| **Total** | **72** | **114** | **182** |

Table 14 - RTEMS Initialisation Failures

All the not detected failures were considered to be critical and although the high number of these failures, they seem to be all related with the same single problem: an invalid memory access during the RTEMS initialisation.

Another issue is related with an incorrect error code that is returned by a partition manager directive was found. This issue was detected by 16 of the test cases. It was classified as minor

Table 15 presents the identified issues in RTEMS according to its criticality.

| RTEMS Manager | Critical | Minor | Total |
|---|---|---|---|
| All | 1 | 0 | 1 |
| Partition Manager | 0 | 1 | 1 |
| **Total** | **1** | **1** | **2** |

Table 15- Identified issues in RTEMS 4.5.0 by Criticality

## 6.3 Problems Found

This section highlights some of the potential problems RTEMS found during the robustness and stress testing. For the complete list of raised issues please refer to [1] and [2].

### 6.3.1 Classic API

#### *6.3.1.1 Unexpected Change of the Control Flow*

When the task identifier is set to 0 on the rtems_task_start call an unexpected change of the control flow of the application occurs: the control flow came back to the rtems_task_create directive call (see test case result RTEMS-TCR-TSKSTR-021 in annex A of [1]).

#### *6.3.1.2 Data Access / Memory not Aligned / Illegal Instruction Exceptions*

In several situations the test cases end up with unhandled traps. The criticality assigned to the potential fault by these test cases was C*ritical* in some cases and *Low* on the other.

In the following situations a *Data Access* or *Memory Not Aligned* exception was generated and criticality *Critical* was assign to the potential fault:

- When a segment of size 0 is requested using the rtems_region_get_segment directive
- Every time an invalid segment is provided either to rtems_region_set_segment_size or to rtems_region_return_segment
- Every time a RTEMS IO directive is called with the device major number set to 0.

Every time a variable addressed is requested by the RTEMS directive to return some value but instead of that the NULL pointer was provided a *Data Access* or *Memory Not Aligned* exception was generated. Criticality minor was assigned to these faults.

There are some RTEMS directives that request a pointer to a user function to be called later by the kernel in case of a specific event occurs. There were two test cases where the NULL pointer was provided to this functions resulting in an *Illegal Instruction* exception.

#### *6.3.1.3 No Error Code*

In some test cases, although providing an invalid parameter to the RTEMS directive under test, no error code was returned. These test cases sometimes succeed and other failed. For instance, when attempting to create a task with stack size 0, 1 or 4Gb the rtems_task_create returned RTEMS_ SUCCESSFUL instead of RTEMS_INVALID_SIZE (for 0 and 1 stack size) and RTEMS_UNSATISFIED (for 4Gb stack size).

The same behaviour can be observed when attempting to send the signal 0 or trying to register a device driver with a major number set to MAX_UNSIGNED32. On the first case the rtems_signal_send directive returns RTEMS_ SUCCESSFUL but no signal is ever received. On the second case the rtems_io_register_name returns RTEMS_ SUCCESSFUL instead of RTEMS_INVALID_NUMBER.

#### *6.3.1.4 Wrong Error Code*

When attempting to create a message queue for MAX_UNSIGNED32 messages, rtems_message_queue_create directive returned RTEMS_INVALID_NUMBER instead of RTEMS_TOO_MANY as stated in the documentation.

### 6.3.2 POSIX API

#### *6.3.2.1 POSIX Compliance*

During the evaluation of the POSIX API of the RTEMS 4.5.0 several compliance issues were raised. Some of these are described below.

When creating a timer using the POSIX timer_create call, the identifier of the clock to be used by the timer must me provided as a parameter. POSIX specification defines only the CLOCK_REALTIME clock identifier, leaving the possibility to the existence of other clocks in a specific system. RTEMS does not define any other clock. For this reason, it should return -1 when timer_create is called with clockid parameter set to a value different from CLOCK_REALTIME and set the errno to EINVAL. What RTEMS is doing is to return an error only when this parameter is 0 (zero), accepting any positive integer value as a clockid (see test case result RTEMS-TCR-PX-TMRCRT-003 in annex A of [1] for further details).

No error is returned by the mq_open function when called to create a message queue with parameter attr.mq_maxmsg (maximum number of messages) set to 0. According to the POSIX specification mq_open should return EINVAL in case mq_attr structure has the mq_maxmsg attribute set with a value less than or equal to 0. As a result of this, the application will block whenever mq_send is called to send a message to a message queue. The same behaviour was observed when attr.mq_maxsize (maximum size of a message) is set to 0.

#### *6.3.2.2 Kernel Crash*

When creating a mutex with the attr.process_shared parameter set to a value different from PTHREAD_PROCESS_PRIVATE, the application ends its execution and the following message is outputted by the kernel:

```
assertion "the_attr->process_shared == PTHREAD_PROCESS_PRIVATE"
failed: file "../../../../../../../rtems-
4.5.0/c/src/exec/posix/src/mutexinit.c", line 96
```

# 7. Metrics

## 7.1 Methodology

The metrics collected during the RTEMS test cases execution were based on the code coverage. However, and due to the fact that the GNU gcov tool has not been ported to the RTEMS/ERC32 architecture, there was no way to automate this task.

The solution found for this problem was to use the GNU gdb debugger with the embedded simulator. By executing the workloads in step-by-step mode, it was possible to count the how many times a line of code was executed. After the collection of these values, the RTEMS executed code can be manually analysed. From this activity, several outputs were gathered to obtain some interesting metrics, e.g. the percentage of error handling code lines that were executed during the RTEMS tests.

This manual approach is, however, very time consuming and, for this reason, only one manager was selected for collection of metrics. The test cases corresponding to the RTEMS Classic API Message Manager, were executed and coverage values were collected and processed with the help of Perl scripts.

The files that were considered in the RTEMS code analysis for this manager are listed in Table 16. These files were selected by building the call graph of the following RTEMS directives:

- *rtems_message_queue_create;*
- *rtems_message_queue_delete;*
- *rtems_message_queue_send;*
- *rtems_message_queue_receive;*
- *rtems_message_queue_get_number_pending;*
- *rtems_message_queue_ident;*
- *rtems_message_queue_urgent;*
- *rtems_message_queue_flush;*
- *rtems_message_queue_broadcast.*

| Filenames | | |
|---|---|---|
| address.inl | msgqdelete.c | sysstates.inl |
| attr.inl | msgqflush.c | thread.inl |
| chain.c | msgqgetnumberpending.c | threadclearstate.c |
| chain.inl | msgqident.c | threaddispatch.c |
| coremsg.c | msgqreceive.c | threadq.c |
| coremsg.inl | msgqsend.c | threadqdequeue.c |
| coremsgbroadcast.c | msgqsubmit.c | threadqdequeuefifo.c |
| coremsgclose.c | msgqtranslatereturncode.c | threadqdequeuepriority.c |
| coremsgflush.c | msgqurgent.c | threadqenqueue.c |
| coremsgflushsupp.c | object.inl | threadqenqueuefifo.c |
| coremsginsert.c | objectallocate.c | threadqflush.c |
| coremsgseize.c | objectclearname.c | threadsetstate.c |
| coremsgsubmit.c | objectcomparenameraw.c | tqdata.inl |
| heap.inl | objectcopynameraw.c | userext.c |
| heapallocate.c | objectfree.c | userext.inl |
| heapfree.c | objectget.c | watchdog.inl |
| message.inl | objectnametoid.c | watchdoginsert.c |
| msgqallocate.c | options.inl | wkspace.inl |
| msgqbroadcast.c | priority.inl | |
| msgqcreate.c | states.inl | |

Table 16- RTEMS Message Manager Files

## 7.2 Results

After collecting the files from GNU gdb containing the number of times each code line has been executed (these files can be found in Annex B), and after an analysis of the RTEMS code, Table 17 and Table 18 were built.

The RTEMS source code analysis activity was aimed to distinguish between:

- Error handling code from non error handling code;
- Unreachable/Dead code from executable code.

The unreachable/dead code that was found was mainly related with code related with the multiprocessing. As the tested version did not include this functionality, this code was considered to be unreachable/dead.

The tables have the following information:

- **File Name:** name of the file.
- **LOC:** number of lines of code in the file. These values were collected by using the SLOCCount tool [12].
- **Unreachable/Dead Lines:** Number of lines in the file that are never executed or that are not even compiled. An example of code that would not be compiled is the code for multiprocessing support: it is inside of an *ifdef* pre-processor directive that always return *false*. The values in this column include lines for error handling.
- **Total Number of Error Handling Lines:** number of lines inside a file whose aim is the error handling. Note that the values in this column include the unreachable/dead code lines that are for error handling. This means that this

column is the sum of the next three columns (unreachable/dead error handling lines, executed error handling lines and unexecuted error handling lines).

- **Unreachable/Dead Error Handling Lines:** number of error handling lines of code that are unreachable.
- **Executed Error Handling Lines:** number of error handling lines of code that were executed at least once.
- **Unexecuted Error Handling Lines:** number of error handling lines of code that were never executed. Note that the values in this column do not include the unreachable/dead error handling lines.

| File Name | LOC | Unreachable/Dead Lines | Total number of error handling lines | Unreachable/Dead Error Handling Lines | Executed Error Handling Lines | Unexecuted Error Handling Lines |
|---|---|---|---|---|---|---|
| coremsg.c | 55 | 3 | 2 | 0 | 2 | 0 |
| coremsg.inl | 129 | 8 | 0 | 0 | 0 | 0 |
| coremsgbroadcast.c | 49 | 7 | 0 | 0 | 0 | 0 |
| coremsgclose.c | 26 | 3 | 0 | 0 | 0 | 0 |
| coremsgflush.c | 20 | 3 | 0 | 0 | 0 | 0 |
| coremsgflushsupp.c | 35 | 3 | 0 | 0 | 0 | 0 |
| coremsginsert.c | 49 | 3 | 0 | 0 | 0 | 0 |
| coremsgseize.c | 66 | 3 | 5 | 0 | 5 | 0 |
| coremsgsubmit.c | 87 | 7 | 14 | 0 | 9 | 5 |
| message.inl | 24 | 0 | 0 | 0 | 0 | 0 |
| msgqallocate.c | 25 | 3 | 0 | 0 | 0 | 0 |
| msgqbroadcast.c | 63 | 20 | 3 | 1 | 2 | 0 |
| msgqcreate.c | 106 | 43 | 46 | 25 | 17 | 4 |
| msgqdelete.c | 64 | 26 | 6 | 4 | 2 | 0 |
| msgqflush.c | 48 | 15 | 3 | 1 | 2 | 0 |
| msgqgetnumberpending.c | 47 | 14 | 3 | 1 | 2 | 0 |
| msgqident.c | 32 | 3 | 0 | 0 | 0 | 0 |
| msgqreceive.c | 63 | 14 | 3 | 1 | 2 | 0 |
| msgqsend.c | 25 | 3 | 0 | 0 | 0 | 0 |
| msgqsubmit.c | 97 | 35 | 5 | 3 | 2 | 0 |
| msgqtranslatereturncode.c | 46 | 3 | 20 | 8 | 4 | 8 |
| msgqurgent.c | 25 | 3 | 0 | 0 | 0 | 0 |
| **total** | **1181** | **222** | **110** | **44** | **49** | **17** |

Table 17 - Message Manager Coverage

Table 17 contains only the code coverage for the source files that are directly related with the Message Manager. The code contained in these files is the core of the message queue manager.

As this table shows, the total number of error handling code lines executed during the test cases execution is 49. This value represents 75% of the total executable error handling code (total number of error handling code minus unreachable/dead error handling code).

| File Name | LOC | Unreachable/Dead Lines | Total number of error handling lines | Unreachable/Dead Error Handling Lines | Executed Error Handling Lines | Unexecuted Error Handling Lines |
|---|---|---|---|---|---|---|
| address.inl | 38 | 0 | 0 | 0 | 0 | 0 |
| attr.inl | 74 | 8 | 0 | 0 | 0 | 0 |
| chain.c | 84 | 14 | 0 | 0 | 0 | 0 |
| chain.inl | 150 | 0 | 2 | 0 | 0 | 2 |
| coremsg.c | 55 | 3 | 2 | 0 | 2 | 0 |
| coremsg.inl | 129 | 8 | 0 | 0 | 0 | 0 |
| coremsgbroadcast.c | 49 | 7 | 0 | 0 | 0 | 0 |
| coremsgclose.c | 26 | 3 | 0 | 0 | 0 | 0 |
| coremsgflush.c | 20 | 3 | 0 | 0 | 0 | 0 |
| coremsgflushsupp.c | 35 | 3 | 0 | 0 | 0 | 0 |
| coremsginsert.c | 49 | 3 | 0 | 0 | 0 | 0 |
| coremsgseize.c | 66 | 3 | 5 | 0 | 5 | 0 |
| coremsgsubmit.c | 87 | 7 | 14 | 0 | 9 | 5 |
| heap.inl | 100 | 0 | 0 | 0 | 0 | 0 |
| heapallocate.c | 61 | 8 | 2 | 0 | 0 | 2 |
| heapfree.c | 63 | 0 | 11 | 0 | 0 | 11 |
| message.inl | 24 | 0 | 0 | 0 | 0 | 0 |
| msgqallocate.c | 25 | 3 | 0 | 0 | 0 | 0 |
| msgqbroadcast.c | 63 | 20 | 3 | 1 | 2 | 0 |
| msgqcreate.c | 106 | 43 | 46 | 25 | 17 | 4 |
| msgqdelete.c | 64 | 26 | 6 | 4 | 2 | 0 |
| msgqflush.c | 48 | 15 | 3 | 1 | 2 | 0 |
| msgqgetnumberpending.c | 47 | 14 | 3 | 1 | 2 | 0 |
| msgqident.c | 32 | 3 | 0 | 0 | 0 | 0 |
| msgqreceive.c | 63 | 14 | 3 | 1 | 2 | 0 |
| msgqsend.c | 25 | 3 | 0 | 0 | 0 | 0 |
| msgqsubmit.c | 97 | 35 | 5 | 3 | 2 | 0 |
| msgqtranslatereturncode.c | 46 | 3 | 20 | 8 | 4 | 8 |
| msgqurgent.c | 25 | 3 | 0 | 0 | 0 | 0 |
| object.inl | 116 | 0 | 5 | 0 | 1 | 4 |
| objectallocate.c | 33 | 3 | 0 | 0 | 0 | 0 |
| objectclearname.c | 22 | 3 | 0 | 0 | 0 | 0 |
| objectcomparenameraw.c | 25 | 3 | 0 | 0 | 0 | 0 |
| objectcopynameraw.c | 23 | 3 | 0 | 0 | 0 | 0 |
| objectfree.c | 30 | 3 | 0 | 0 | 0 | 0 |
| objectget.c | 43 | 13 | 5 | 0 | 2 | 3 |
| objectnametoid.c | 52 | 7 | 5 | 0 | 3 | 2 |
| options.inl | 15 | 0 | 0 | 0 | 0 | 0 |
| priority.inl | 92 | 0 | 0 | 0 | 0 | 0 |
| states.inl | 132 | 0 | 0 | 0 | 0 | 0 |
| sysstates.inl | 50 | 0 | 0 | 0 | 0 | 0 |
| thread.inl | 130 | 3 | 8 | 0 | 0 | 8 |
| threadclearstate.c | 39 | 0 | 0 | 0 | 0 | 0 |
| threaddispatch.c | 67 | 23 | 0 | 0 | 0 | 0 |
| threadq.c | 35 | 0 | 0 | 0 | 0 | 0 |
| threadqdequeue.c | 26 | 3 | 3 | 3 | 0 | 0 |
| threadqdequeuefifo.c | 48 | 5 | 5 | 1 | 4 | 0 |

| File Name | LOC | Unreachable/Dead Lines | Total number of error handling lines | Unreachable/Dead Error Handling Lines | Executed Error Handling Lines | Unexecuted Error Handling Lines |
|---|---|---|---|---|---|---|
| threadqdequeuepriority.c | 146 | 4 | 4 | 0 | 4 | 0 |
| threadqenqueue.c | 39 | 5 | 0 | 0 | 0 | 0 |
| threadqenqueuefifo.c | 48 | 6 | 0 | 0 | 0 | 0 |
| threadqflush.c | 24 | 5 | 0 | 0 | 0 | 0 |
| threadsetstate.c | 41 | 0 | 0 | 0 | 0 | 0 |
| tqdata.inl | 21 | 0 | 0 | 0 | 0 | 0 |
| userext.c | 119 | 0 | 2 | 0 | 0 | 2 |
| userext.inl | 56 | 0 | 0 | 0 | 0 | 0 |
| watchdog.inl | 102 | 0 | 0 | 0 | 0 | 0 |
| watchdoginsert.c | 47 | 0 | 0 | 0 | 0 | 0 |
| wkspace.inl | 15 | 0 | 0 | 0 | 0 | 0 |
| **Total** | **3387** | **341** | **162** | **48** | **63** | **51** |

Table 18 - Total Code Coverage

Table 18 presents the total coverage code for the referred RTEMS directives. This code includes several functions not directly related with the RTEMS manager under test, e.g. watchdog timer functions.

With these vales, the percentage of error handling code lines executed during the test cases execution is now 56%.

# 8. Methodology Feedback

## 8.1 Robustness Testing

One of the most evident advantages of the robustness testing methodology used is that it allows the definition and execution of a large number of test cases with a reduced effort. The effectiveness of the method is also promising. A considerable number of potential robustness problems were identified given the effort spent. The following table shows a comparison of the number of test cases defined and executed, number of raised issues and effort required concerning the robustness testing of each of the RTEMS APIs.

| API | Test Cases | Raised Issues | Effort (hours) | Effort per Test Case (hours) | Effort per Raised Issue (hours) |
|---|---|---|---|---|---|
| Classic | 527 | 34 | 160 | 0,30 | 4,7 |
| POSIX | 528 | 15 | 80 | 0,15 | 5 |
| **Total** | **1055** | **49** | **240** | **0,23** | **4,8** |

Table 19 - Some metrics on the robustness testing methodology used

Another interesting point worth mention is the coverage achieved with this methodology. According to the results obtained for the message queue RTEMS resource manager (see Table 17), 75% of the total executable error handling code at API level was exercised by the test cases.

The application of this methodology is very straightforward concerning the robustness testing of system calls or libraries APIs.

### 8.1.1 Possible Improvements

Although there are several indicators that the applied methodology is valuable there are also some points that might be improved.

One of the major drawbacks of the automation of the robustness testing used is that the results analysis could not be performed in an automated way. This is due to the fact that in opposition to the invalid parameters generation, the results analysis requires information on the semantics of the directives called and on the internal state of the Kernel. For instance, it is not possible to tell the result of an rtems_task_start without knowing if the task is created or not. This could be achieved using some logical model of the system. This logical model could be constructed in a formal language, having as an input the requirements of the product under evaluation. It could be used to compute the specified behaviour of the product and in this way automate the result analysis. Of course this requires an extra effort in the preparation phase of the evaluation if this logical model of the system does not exists. If available this logical model could also be used to generate sets of system calls input parameters in a more clever way.

It is known that the inputs of a given system call are not only its parameters but also the state in which the system is in. This leads to another possible improvement of the methodology. This improvement would be achieved by not only providing invalid parameters to the system calls but also performing the system calls in the several different kernel states.

## 8.2 Stress Testing

Like the methodology used in the robustness testing, the stress testing methodology used allowed the definition and execution a considerable number of test cases with a reduced effort.

The number of resources created was gradually increased until it reached a value that the system could not stand, typically due to lack of memory. The number of tasks accessing each resource was also pushed to the limit. However the results achieved by the stress testing were not as promising as the results of the robustness testing at least considering the number of potential faults uncovered by each of the methodologies. This might be due to one or both of the following reasons:

- Characteristics of the product under evaluation. It may stand better the stressing of its resources than tolerating invalid parameters on the APIs.
- The methodology defined for the stress testing was not the most appropriated.

### 8.2.1 Possible Improvements

In future to analyse better the effectiveness of this stress methodology, monitoring of the level of stress could also be performed. For instance, the number of accesses per time unit to a given resource could be measured. The execution time of the test cases can also be increased in order to uncover problems related with cumulative faults.

Stress testing would also benefit from the knowledge of internal RTEMS architecture. This knowledge could be used to identify sensitive resources of the kernel and to figure out ways of stressing them.