

AMMONITE

M7M02 (Ammonite) R3T1 Microcontroller Embedded Virtual Machine Monitor (Rev. 3 Typ. 1)

Technical Reference Manual

Version R3T1 · March 20, 2025

EDI-Systems · www.edi-systems.org

System Features

Mutatus · Mesozoa · Ammonite (R.VII)

M7M02 (Ammonite) R3T1

Microcontroller Embedded Virtual Machine Monitor (Rev. 3 Typ. 1)

Technical Manual

High-Performance Real-Time Response

Native processes provide excellent real-time performance. Virtual machine processes provide excellent compatibility and execution efficiency.

Broad Architecture Support

Easily portable across multiple architectures. Supports paravirtualization on microcontrollers.

High Security

Hardware isolation between virtual machines. Supports fault restart.

Comprehensive Ecosystem

Supports virtualization of most operating systems. Original system applications run without modification.

High Usability

Provides a graphical project generator.

Contents

System Features ii

List of Tables viii

List of Figures xi

Revision History xii

1 Overview 1

1.1 Introduction . . . 1

1.1.1 Characteristics of microcontrollers . . . 2

1.1.2 Software copyright and license . . . 3

1.1.3 Main reference systems . . . 3

1.2 Preface . . . 3

1.2.1 Native virtual machine monitor . . . 4

1.2.2 Hosted virtual machine monitor . . . 4

1.2.3 Container-based virtual machine monitor . . . 4

1.2.4 Full virtualization virtual machine monitor . . . 5

1.2.5 Paravirtualization virtual machine monitor . . . 5

1.3 Manual overview . . . 5

1.4 References for this chapter . . . 6

2 Virtualization Architecture 7

2.1 Architecture Overview . . . 7

2.1.1 RME Microkernel . . . 8

2.1.2 RVM Virtual Machine Monitor . . . 8

2.1.3 Protection Domains . . . 10

2.1.4 Project Generator . . . 11

2.2 Virtualization Principles . . . 11

2.2.1 Priority Configuration . . . 12

2.2.2 Memory Management . . . 12

2.2.3 CPU Virtualization . . . 13

2.2.4 Interrupt Virtualization . . . 14

2.2.5 Device Virtualization . . . 17

2.2.6 Hypercall Design . . . 18

2.2.7 Cross-Protection-Domain Communication . . . 18

2.3 References for This Chapter . . . 20

3 Native Process 21

3.1 Introduction . . . 21

3.2 Process Descriptor . . . 21

3.3 Registering a Native Process with the RVM . . . 22

3.3.1 Kernel Object Creation Hook . . . 22
3.3.2 Kernel Object Initialization Hook . . . 22
3.4 Macro Definitions That Should Exist in a Native Process . . . 22
3.5 System Calls Available to Native Processes . . . 23
3.5.1 Capability Table Related System Calls . . . 23
3.5.2 Kernel Function Related System Calls . . . 24
3.5.3 Page Table Related System Calls . . . 24
3.5.4 Process Related System Calls . . . 25
3.5.5 Thread Related System Calls . . . 25
3.5.6 Asynchronous Signal Endpoint Related System Calls . . . 26
3.5.7 Synchronous Invocation Related System Calls . . . 26
3.5.8 Event Source Activation Related System Calls . . . 26
3.6 Other Helper Functions Available to Native Processes . . . 27
3.6.1 Variable Clearing . . . 27
3.6.2 Creating a Doubly Linked Circular List . . . 27
3.6.3 Deleting a Node in a Doubly Linked Circular List . . . 27
3.6.4 Inserting a Node in a Doubly Linked Circular List . . . 28
3.6.5 Printing a Single Character . . . 28
3.6.6 Printing an Integer Number . . . 28
3.6.7 Printing an Unsigned Integer Number . . . 29
3.6.8 Printing a String . . . 29
3.7 References for This Chapter . . . 29

4 Virtual Machine 30
4.1 Introduction . . . 30
4.2 Registering Virtual Machines with RVM . . . 30
4.2.1 Filling in Virtual Machine Information . . . 30
4.2.2 Creating and Initializing Virtual Machines . . . 32
4.2.3 Batch Registration of Virtual Machines . . . 32
4.3 Macros That Should Exist Inside a Virtual Machine . . . 32
4.4 Hypercalls Available to Virtual Machines . . . 33
4.4.1 Printing a Single Character . . . 34
4.4.2 Restarting the Virtual Machine . . . 34
4.4.3 Enabling Virtual Global Interrupts . . . 34
4.4.4 Disabling Virtual Global Interrupts . . . 34
4.4.5 Mapping a Physical Interrupt Source to a Virtual Interrupt Source . . . 35
4.4.6 Mapping an Event Source to a Virtual Interrupt Source . . . 35
4.4.7 Removing the Mapping on a Virtual Interrupt Source . . . 36
4.4.8 Locking the Virtual Interrupt Source Mapping . . . 36
4.4.9 Waiting for a Virtual Interrupt Source to Be Triggered . . . 37
4.4.10 Granting Permission to Send to an Event Source . . . 38
4.4.11 Revoking the Authorization to Send to an Event Source . . . 38
4.4.12 Sending to an Event Source . . . 38
4.4.13 Activating and Feeding the Watchdog . . . 38
4.5 Internal Utility Functions Available to Virtual Machines . . . 39
4.5.1 Virtual Machine Data Structure Initialization . . . 39
4.5.2 Running the Virtual Interrupt Vector Handling Loop . . . 40
4.5.3 Masking Virtual Interrupt Vector Handling . . . 40
4.5.4 Unmasking Virtual Interrupt Vector Handling . . . 40

4.5.5 Registering an Ordinary Virtual Interrupt Vector . . . 41
4.5.6 Registering the Clock Virtual Interrupt Vector . . . 41
4.5.7 Registering the Thread Switch Virtual Interrupt Vector . . . 42
4.5.8 Triggering the Thread Switch Virtual Interrupt Vector . . . 42
4.6 References for This Chapter . . . 42

5 Project Generator 43
5.1 Introduction . . . 43
5.1.1 Input Analysis Module . . . 43
5.1.2 Project Generation Module . . . 43
5.1.3 Project Output Module . . . 44
5.1.4 Graphical Frontend Module . . . 44
5.2 Command-Line Parameter Description . . . 44
5.3 Project Description File Structure<Project> . . . 45
5.3.1 Project Name<Name> . . . 45
5.3.2 Project Version<Version> . . . 45
5.3.3 Enable Assertions<Assert_Enable> . . . 45
5.3.4 Enable Debug Printing<Debug_Log_Enable> . . . 45
5.3.5 Enable Fixed Raw Page Tables<Pgtbl_Raw_Enable> . . . 45
5.3.6 Workspace Build System<Buildsystem> . . . 45
5.3.7 Workspace Output Directory<Workspace_Output> . . . 45
5.3.8 Workspace Output Overwrite<Workspace_Overwrite> . . . 46
5.3.9 Chip Configuration<Chip> . . . 46
5.3.10 Extra Memory<Extmem><En> . . . 46
5.3.11 Shared Memory Block<Shmem><Sn> . . . 47
5.3.12 Kernel Configuration<Kernel> . . . 48
5.3.13 Monitor Configuration<Monitor> . . . 51
5.3.14 Native Process Configuration<Process><Pn> . . . 54
5.3.15 Virtual Machine Configuration<Virtual><Vn> . . . 59
5.4 Platform Description File Structure<Platform> . . . 62
5.4.1 Name<Name> . . . 62
5.4.2 Version<Version> . . . 62
5.4.3 Word Length<Wordlength> . . . 62
5.4.4 Allowed Coprocessors<Coprocessor> . . . 62
5.4.5 Guest List<Guest> . . . 62
5.4.6 Build System List<Buildsystem> . . . 62
5.4.7 Toolchain List<Toolchain> . . . 62
5.4.8 Architecture Options<Config><Cn> . . . 62
5.5 Chip Description File Structure<Chip> . . . 63
5.5.1 Name<Name> . . . 63
5.5.2 Version<Version> . . . 63
5.5.3 Platform Architecture<Platform> . . . 63
5.5.4 Compatible List<Compatible> . . . 63
5.5.5 Vendor<Vendor> . . . 63
5.5.6 Number of Interrupt Vectors<Vector> . . . 64
5.5.7 Number of General Memory Protection Regions<Region> . . . 64
5.5.8 Number of Code Memory Protection Regions<Iregion> . . . 64
5.5.9 Number of Data Memory Protection Regions<Dregion> . . . 64
5.5.10 Present Coprocessors<Coprocessor> . . . 64

5.5.11 Architecture Attribute Group<Attribute> . . . 64
5.5.12 Chip Inherent Memory<Memory><Mn> . . . 64
5.5.13 Chip Options<Config><Cn> . . . 65
5.6 Project Generator Error Messages . . . 66
5.6.1 Command-line argument analysis errors (C0000-C9999) . . . 66
5.6.2 Project XML file analysis errors (P0000-P9999) . . . 66
5.6.3 Chip XML file analysis errors (D0000-D9999) . . . 79
5.6.4 Memory allocation errors (M0000-M0999) . . . 81
5.6.5 Capability table allocation errors (M1000-M1999) . . . 84
5.6.6 Kernel object allocation errors (M2000-M2999) . . . 86
5.6.7 Kernel project generation errors (G0000-G0999) . . . 89
5.6.8 User-mode library project generation errors (G1000-G1999) . . . 89
5.6.9 User process project generation errors (G2000-G2999) . . . 89
5.6.10 Workspace generation errors (G3000-G3999) . . . 90
5.6.11 File system operation errors (F0000-F9999) . . . 90
5.6.12 Architecture-related file generation errors (A0000-A9999) . . . 91
5.7 Graphical Front End of the Project Generator . . . 91
5.7.1 Starting the Graphical Front End . . . 91
5.7.2 Configuration Category Selection Area . . . 91
5.7.3 Configuration Object Selection Area . . . 91
5.7.4 Configuration Group Selection Area . . . 91
5.7.5 Configuration Item Display Area . . . 92
5.7.6 Project Generation Button . . . 92
5.8 References for This Chapter . . . 92
6 Using the RVM in a Project 93
6.1 Overview . . . 93
6.2 Dividing Software Components into Protection Domains . . . 93
6.2.1 Granularity of Component Division . . . 93
6.2.2 Requirements of Components for Compatibility and Real-Time Performance . . . 94
6.2.3 Requirements of Components for Device Access . . . 96
6.3 Dividing Interrupt Vectors into Protection Domains . . . 96
6.3.1 Low-Frequency Interrupt Vectors . . . 97
6.3.2 Medium-Frequency Interrupt Vectors . . . 97
6.3.3 High-Frequency Interrupt Vectors . . . 97
6.3.4 Ultra-High-Frequency Interrupt Vectors . . . 98
6.4 Choosing a Communication Method Between Protection Domains . . . 98
6.4.1 Tightly Coupled Native Processes . . . 98
6.4.2 Loosely Coupled Native Processes . . . 99
6.4.3 Native Processes and Virtual Machines . . . 99
6.4.4 Virtual Machines and Virtual Machines . . . 99
6.5 Pre-Generation Settings . . . 100
6.5.1 General Settings . . . 100
6.5.2 Kernel Settings . . . 101
6.5.3 Virtual Machine Monitor Settings . . . 102
6.5.4 Native Process Settings . . . 103
6.5.5 Virtual Machine Settings . . . 104
7 Adding a New Chip to the RVM 105
7.1 Preparations for Adding a New Chip . . . 105

8 Porting the RVM to a New Architecture 106
8.1 Introduction 106
8.2 Porting the User-Mode Library 106
8.2.1 Type Definitions 106
8.2.2 Macros Related to RME Configuration Items and Filled In Automatically by the Project Generator 107
8.2.3 Macros Related to RME Configuration Items That Do Not Change 108
8.2.4 Macros Related to RVM Configuration Items and Filled In Automatically by the Project Generator 109
8.2.5 Data Structure Definitions 109
8.2.6 Porting the Low-Level Assembly Functions 109
8.2.7 Porting the C Language Functions 113
8.3 Porting the Project Generator 114
8.4 References for This Chapter 114
9 Appendix 115
9.1 Known Issues and Constraints of RVM 115
9.1.1 No Multicore Support 115
9.2 References for This Chapter 115

List of Tables

2-1 Meaning of each field in struct __RME_RVM_Flag . . . 9
2-2 Comparison between native processes and virtual machines . . . 10
2-3 Meaning of each field in struct RVM_State . . . 14
2-4 Meaning of each field in struct RVM_Vctf . . . 15

3-1 Structure of the process descriptor . . . 21
3-2 Kernel object creation hook . . . 22
3-3 Kernel object initialization hook . . . 22
3-4 List of macro definitions that should exist in a native process . . . 22
3-5 Capability table related system calls . . . 23
3-6 Kernel function related system calls . . . 24
3-7 Page table related system calls . . . 24
3-8 Process related system calls . . . 25
3-9 Thread related system calls . . . 25
3-10 Asynchronous signal endpoint related system calls . . . 26
3-11 Synchronous invocation related system calls . . . 26
3-12 Parameters required to activate an event source . . . 27
3-13 List of other helper functions available to native processes . . . 27
3-14 Parameters required for variable clearing . . . 27
3-15 Creating a doubly linked circular list . . . 27
3-16 Deleting a node in a doubly linked circular list . . . 28
3-17 Inserting a node in a doubly linked circular list . . . 28
3-18 Printing a single character . . . 28
3-19 Printing an integer number . . . 29
3-20 Printing an unsigned integer number . . . 29
3-21 Printing a string . . . 29

4-1 The meaning of each field in struct RVM_Virt_Map . . . 31
4-2 Virtual machine batch registration hook . . . 32
4-3 List of macros that should exist in a virtual machine process . . . 33
4-4 List of hypercalls available to virtual machines . . . 33
4-5 Printing a single character . . . 34
4-6 Restarting the virtual machine . . . 34
4-7 Enabling virtual global interrupts . . . 34
4-8 Disabling virtual global interrupts . . . 34
4-9 Mapping a physical interrupt source to a virtual interrupt source . . . 35
4-10 Mapping an event source to a virtual interrupt source . . . 36

4-11 Removing the mapping on a virtual interrupt source . . . 36
4-12 Locking the virtual interrupt source mapping . . . 37
4-13 Waiting for a virtual interrupt source to be triggered . . . 37
4-14 Granting permission to send to an event source . . . 38
4-15 Revoking the authorization to send to an event source . . . 38
4-16 Sending to an event source . . . 38
4-17 Activating and feeding the watchdog . . . 39
4-18 List of internal utility functions available to virtual machines . . . 39
4-19 Virtual machine data structure initialization . . . 39
4-20 Running the virtual interrupt vector handling loop . . . 40
4-21 Masking virtual interrupt vector handling . . . 40
4-22 Unmasking virtual interrupt vector handling . . . 40
4-23 Registering an ordinary virtual interrupt vector . . . 41
4-24 Registering the clock virtual interrupt vector . . . 42
4-25 Registering the thread switch virtual interrupt vector . . . 42
4-26 Triggering the thread switch virtual interrupt vector . . . 42

5-1 Supported output toolchains . . . 66
5-2 Error messages that may occur during the command-line argument analysis stage . . . 66
5-3 Error messages that may occur during the preliminary project configuration analysis stage 67
5-4 Error messages that may occur during the kernel configuration analysis stage . . . 68
5-5 Error messages that may occur during the user-mode library configuration analysis stage 70
5-6 Error messages that may occur during the process configuration analysis stage . . . 71
5-7 Error messages that may occur during the virtual machine configuration analysis stage . 73
5-8 Error messages that may occur during the memory segment configuration analysis stage 75
5-9 Error messages that may occur during the shared memory segment configuration analysis stage . . . 76
5-10 Error messages that may occur during the thread configuration analysis stage . . . 76
5-11 Error messages that may occur during the invocation target configuration analysis stage 77
5-12 Error messages that may occur during the invocation entry configuration analysis stage . 77
5-13 Error messages that may occur during the receive signal endpoint configuration analysis stage . . . 78
5-14 Error messages that may occur during the send signal endpoint configuration analysis stage . . . 78
5-15 Error messages that may occur during the physical vector signal endpoint configuration analysis stage . . . 78
5-16 Error messages that may occur during the kernel function configuration analysis stage . 79
5-17 Error messages that may occur during the preliminary chip information analysis stage . 79
5-18 Error messages that may occur during the chip option analysis stage . . . 81
5-19 Error messages that may occur during the memory section allocation stage . . . 82
5-20 Error messages that may occur during the compliance check stage . . . 84
5-21 Error messages that may occur during the local capability number allocation stage . . . 86
5-22 Error messages that may occur during the capability pairing stage . . . 86
5-23 Error messages that may occur during the kernel object trial allocation and allocation stage 87
5-24 Error messages that may occur during the other kernel memory allocation stage . . . 87
5-25 Error messages that may occur during the user-mode library memory allocation stage . . 88

5-26 Error messages that may occur during the process memory allocation stage 88
5-27 Error messages that may occur when generating the user-mode library project 89
5-28 Error messages that may occur when generating the user process project 90
5-29 Error messages that may occur during file system operations 90

6-1 Examples of Component Division 94
6-2 Classification of Interrupts by Occurrence Frequency 96

8-1 RVM type definitions 107
8-2 Other commonly used type definitions 107
8-3 Macros related to RME configuration items and filled in automatically by the project generator 108
8-4 Macros related to RME configuration items that do not change 109
8-5 Macros related to the RVM configuration 109
8-6 Assembly functions involved in RVM porting 110
8-7 Implementation of __RVM_Entry 110
8-8 Implementation of __RVM_Stub 111
8-9 Implementation of RVM_MSB_GET 111
8-10 Implementation of RVM_Inv_Act 112
8-11 Implementation of RVM_Inv_Ret 112
8-12 Implementation of RVM_Svc 112
8-13 C language functions involved in RVM porting 113
8-14 Implementation of RVM_Putchar 113
8-15 Implementation of RVM_Stack_Init 113
8-16 Implementation of RVM_Thd_Print_Exc 114
8-17 Implementation of RVM_Thd_Print_Reg 114

List of Figures

2-1 Simplified diagram of the RVM architecture 8
2-2 Interrupt priorities in the virtualization architecture 12

Revision History

Version	Date (YYYY-MM-DD)	Description
R1T1	2018-03-09	Initial release.
R2T1	2018-09-18	Added porting instructions.
R3T1	2025-03-20	Published the R3T1 Technical Reference Manual.

1. Overview

1.1 Introduction

In modern microcontroller (MCU) applications, the requirements for flexibility, safety and reliability are increasing day by day. More and more microcontroller applications require applications from multiple mutually untrusted sources to run together, and may also simultaneously require real-time reliable parts[1], performance-first parts[2] and certain applications with information security requirements[3] to run together, and they must not interfere with each other. When some applications crash, they are required to be able to restart independently, and in this process it is also required to ensure that those applications that directly control physical electromechanical outputs are not disturbed.

Since the kernel and applications of a traditional real-time operating system (Real-Time Operating System, RTOS) are usually statically linked together, and there is no memory protection isolation between them, they cannot meet the requirement of information isolation between applications; once one of the applications crashes and corrupts kernel data, all other applications will inevitably crash at the same time. Some RTOSes claim to provide memory protection, but their specific implementation mechanisms show that their memory protection can only provide a certain degree of reliability between applications and cannot provide security between applications, and is not suitable for the information security requirements of modern microcontroller applications. In addition, the kernel services[4] of traditional RTOSes are shared between high-priority applications and low-priority applications; once a low-priority application calls a large number of kernel services, it will inevitably affect the response of kernel services to high-priority applications, thereby causing time interference problems[1].

RVM (Real-time Virtual machine Monitor) is a user-mode library based on the RME microkernel, proposed to address the above problems and oriented toward high-performance MCUs. It is a hosted virtual machine monitor running on the RME operating system, which makes it possible to run multiple virtual machines on a microcontroller platform, while also satisfying users' need to run native RME applications. In this way, the requirement of running real-time and non-real-time applications together does not produce time interference between real-time applications and non-real-time applications. It also enables applications from multiple sources to run on the same chip without information security problems.

Since RVM forcibly isolates the secure part and the insecure part of the system, these two parts can be developed and certified separately using different standards. This facilitates debugging and saves development and certification costs, especially when using those relatively complex, less certified and less real-time software packages such as graphical user interface (Graphical User Interface, GUI) and high-level language virtual machines (Python, Java virtual machines), etc. In addition, microcontroller applications already certified on legacy platforms can run directly on RVM as a virtual machine without re-certification; this further saves costs.

This manual provides a functional description of RVM from the user's perspective. For the specific use of each architecture, please refer to the corresponding manual for each architecture. In this manual, we first briefly review several concepts about virtualization, and then introduce the features and API of

[1] such as motor control, etc.
[2] such as network protocol stacks, high-level language virtual machines, etc.
[3] such as encryption and decryption algorithms, etc.
[4] such as kernel timers, etc.

RVM chapter by chapter.

1.1.1 Characteristics of microcontrollers

In practical applications, MCU systems occupy a considerable portion of real-time systems. They often have the following characteristics:

1.1.1.1 Deeply embedded

Microcontroller applications are often integrated with the device itself, and users usually cannot directly perceive the existence of such systems. Once a microcontroller application program[5] is written, it often does not change during the entire life cycle, or changes very rarely. Even if there is an upgrade requirement, the upgrade is often performed by the manufacturer.

1.1.1.2 Highly reliable

Microcontroller applications have an almost paranoid pursuit of reliability, and any system failure will cause relatively serious consequences. From small electric pressure cookers to large cranes, these systems do not allow serious errors, otherwise they will cause major property losses or personal injuries. In addition, there are also very high requirements for the real-time performance of applications; they usually either require the entire system to be a hard real-time system, or require the part of the system that directly controls the physical system to be a hard real-time system.

1.1.1.3 Limited resources

Since microcontrollers themselves have few resources, it is very important to use them efficiently. The software used by microcontrollers is generally optimized for space complexity both in writing and in compilation, and only a small part of the programs that are crucial to performance and functionality use time complexity optimization.

1.1.1.4 Static allocation and linking

Unlike microprocessors, due to deep embedding, the vast majority of resources in microcontroller applications, including memory and devices, are created and allocated when the system is powered on. Generally speaking, it is unwise to use too many dynamic features in microcontrollers, because dynamic features not only put pressure on the real-time response of the system, but also increase the uncertainty of successful execution of functions. If some crucial function and an ordinary function both depend on dynamic memory allocation, then when the ordinary function exhausts the memory, it may cause the memory allocation of the important function to fail. For the same or similar reasons, the vast majority of code in microcontrollers is statically linked, and dynamic linking is generally not used.

1.1.1.5 Physical address space only

Unlike microprocessors, microcontrollers usually do not have a memory management unit (Memory Management Unit, MMU), so they cannot realize the conversion from physical addresses to virtual addresses. However, RVM also supports the use of a memory protection unit (Memory Protection Unit, MPU), and a large number of mid-to-high-end microcontrollers are equipped with an MPU. Unlike the MMU, the MPU often only supports protecting one or several memory ranges, and some also have very complex address alignment restrictions, so memory management functions on microcontrollers are often very limited.

1.1.1.6 Single-core processor

Nowadays, the vast majority of microprocessors are multi-core designs. However, unlike microprocessors, microcontrollers are usually single-core designs, so there is no need to consider many race hazard

[5] usually called “firmware”

problems. This can further simplify the design of the microcontroller user-mode library. Considering that a small number of microcontrollers indeed adopt multi-core designs or even asymmetric multi-core designs, RVM also supports multi-core operating environments. Of course, this first requires that the underlying RME is correctly configured to support multi-core.

Considering the above points, when designing RVM, all kernel objects are completely created at system startup, and at this time, the upper limit of available kernel objects in the system is also determined. Each virtual machine is solidified in the on-chip non-volatile memory of the microcontroller.

1.1.2 Software copyright and license

Taking into comprehensive account the different requirements of microcontroller applications, deeply embedded applications and traditional applications for open-source systems, the license adopted by the RVM microcontroller library is Unlicensed, but special provisions apply to some special cases[6]. These special provisions are decided on a case-by-case basis, and the specific terms will be different for each possible case.

1.1.3 Main reference systems

The implementation of RTOS common services mainly refers to the following systems:

RMProkaron (@EDI)

FreeRTOS (@Real-Time Engineering LTD)

Contiki (@The Contiki Community)

The design of the Contiki virtual machine monitor refers to the implementation of the following systems:

Nova (@TU Dresden)

XEN (@Cambridge University)

KVM (@The Linux Foundation)

The references and reference materials for the other chapters are listed in that chapter.

1.2 Preface

As a virtual machine monitor, RVM relies on the kernel services provided by the underlying RME operating system to maintain operation. RVM adopts paravirtualization technology to minimize virtualization overhead, typically adding only about 1% of CPU overhead.

RVM fully supports low-power technology, and can enable each partition of the virtualized system to operate without ticks, minimizing power requirements.

For RME native process applications, the response time is comparable to that of a small RTOS on the level of FreeRTOS, and can fully satisfy those applications with the most stringent real-time requirements.

For RVM hosted virtual machines[7], the interrupt response time and thread switching time will increase to about 4 times those of the original small RTOS. It is worth noting that RVM itself does not provide any POSIX support; POSIX support depends on the ported operating system itself. Fortunately, many RTOSes have POSIX support, so virtualizing them directly allows their POSIX functionality to be used.

[6] such as security equipment, military systems, aerospace equipment, power systems and medical equipment, etc.
[7] see the classification of virtual machines below

In a full-featured configuration, RME and RVM together occupy a minimum of 16kB of memory[8], plus a fixed occupation of 48kB of program memory, and are suitable for mid-to-high-performance microcontrollers. In addition, if multiple RVM virtual machines use components such as network protocol stacks at the same time, using this technology can even save memory, because shared components such as network protocol stacks can be created as only one copy in the system, and the underlying switching code and management logic of multiple operating systems are completely abstracted out, so there is only one copy in the system.

Here we review the types of virtualization. In this manual, we classify virtual machines into three categories according to their running platform, called native (type I), hosted (type II) and container-based (type III) respectively. At the same time, according to the different ways in which virtual machines support virtualization, they can be further divided into full virtualization (Full Virtualization, FV) and paravirtualization (Para-Virtualization, PV). These two classification methods are independent of each other.

1.2.1 Native virtual machine monitor

A native virtual machine monitor runs directly on the hardware as underlying software, creating multiple instances of physical machines; it is also called a type I virtual machine monitor. Multiple virtual machines run on these instances of physical machines. The advantage of this type of virtual machine monitor lies in high performance and low overhead, but the disadvantage is that it does not have virtual machine management functions itself and requires domain 0[9] to manage other virtual machines, and it cannot run lightweight ordinary processes like a conventional operating system. Such virtual machine monitors include KVM[10], Hyper-V, VMware ESXi Server, Xen, XtratuM, etc.

1.2.2 Hosted virtual machine monitor

A hosted virtual machine monitor runs on the operating system as an application, creating multiple instances of physical machines; it is also called a type II virtual machine monitor. Multiple virtual machines run on these instances of physical machines. The advantage of this type of virtual machine lies in being system-independent, having certain management functions itself, and allowing their virtual machine processes to be scheduled together with other ordinary host processes, so it has been most widely applied, but its efficiency is generally considered to be somewhat lower than that of the native type. Such virtual machine monitors include VMware Workstation, Virtual Box, QEMU[11], etc. RVM is an application developed on RME, and its virtual machines also run on RME as processes, so it also belongs to the hosted virtual machine monitor category.

1.2.3 Container-based virtual machine monitor

The container-based virtual machine monitor is a function provided by the operating system itself, creating multiple isolated independent instances of operating system resource allocation spaces; strictly speaking it does not belong to virtual machine monitors, but because it also has a certain resource partitioning function, it is called a type III virtual machine monitor on some occasions. Multiple applications run directly on these resource allocation space instances. Generally speaking, this type of virtual machine monitor has the highest performance and the lowest memory and time overhead for virtualization, but it requires that applications must be written for the interface it provides. Generally speaking, this interface is consistent with the operating system's own system calls, while other interfaces are not compatible, so its universality is very poor. Such virtual machine monitors include Docker, Pouch, RKT, LXC, etc. When the performance overhead caused by RVM is considered inappropriate, since the RME operating system itself also natively has the capability of a container-based virtual machine monitor, it is also possible not

[8] in this case 4 virtual machines can be supported; if 128kB of memory is allocated, 32 virtual machines can be supported
[9] that is, the first-started virtual machine with management privileges
[10] KVM blurs the boundary between native and hosted, and strictly speaking does not belong to either category
[11] excluding its KVM version

to use RVM and instead use the user-mode library it provides to develop native container-based virtual machine applications. In fact, if no virtual machine is declared in the RVM configuration file, RVM's virtualization function will be automatically removed from the system at compile time to adapt to extremely resource-constrained environments[12].

1.2.4 Full virtualization virtual machine monitor

A full virtualization virtual machine monitor is a virtual machine monitor capable of creating hardware instances completely identical to the original bare metal. Generally speaking, this requires the intervention of an MMU for address translation, and privileged instructions must be handled by binary translation, trap redirection or dedicated hardware[13], and I/O also requires special hardware functions[14] for redirection. Such virtual machines are difficult to implement in an MCU environment, because microcontrollers usually do not provide an MMU, nor do they provide corresponding virtualization hardware for redirection functions. Such virtual machine monitors include VMware Workstation, Virtual Box, QEMU, KVM, Hyper-V, ESX Server, etc.

1.2.5 Paravirtualization virtual machine monitor

A paravirtualization virtual machine monitor is a virtual machine monitor that creates hardware instances somewhat different from the original bare metal. It uses hypercalls (Hypercall) to handle privileged instructions and special I/O, thereby avoiding those complex and unnecessary details that full virtualization must provide. Such virtual machines can be implemented with relatively high performance in the MCU environment. Such virtual machine monitors include Xen, etc. RVM is also a paravirtualization virtual machine monitor, handling system functions through hypercalls.

1.3 Manual overview

This manual explains the basic composition and operating principles of RVM from the user's perspective, and assumes that the reader has a general understanding of the basic concepts of virtualization. In this manual, the "rvm_" prefix before all typedef types is omitted.

The first part of this manual is Chapter 2. In this chapter, the architecture of RVM will be outlined; detailed reading is recommended.

The second part of this manual is Chapters 3 to 5. In these chapters, the technical details of RVM will be described in detail respectively; consultation as needed is recommended. Among them, Chapter 3 introduces detailed information about native processes, Chapter 4 introduces detailed information about virtual machines, and Chapter 5 introduces detailed information about the project generator.

The third part of this manual is Chapter 6. In this chapter, the usage of RVM and its project generator will be described in detail from the user's perspective; detailed reading is recommended.

The last part of this manual is Chapter 8. In this chapter, how to port RVM to a new architecture will be described in detail. If the user does not need to perform porting, there is no need to read this chapter.

[12] such as environments like Cortex-M0+ with RAM around 20KiB
[13] such as Intel VT-x, AMD AMD-V, etc.
[14] such as Intel VT-d, etc.

1.4 References for this chapter

[1] P. Patel, M. Vanga, and B. B. Brandenburg, "TimerShield: protecting high-priority tasks from low-priority timer interference," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 3–12.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, 2004, pp. 455–462.

2. Virtualization Architecture

2.1 Architecture Overview

RVM is a virtual machine monitor built specifically for resource-constrained microcontrollers. It can decompose a single microcontroller into multiple virtual microcontrollers that do not affect one another in time or space. Even if one of the virtual microcontrollers suffers a fatal failure, the other virtual microcontrollers can still operate normally and are not adversely affected, thereby effectively preventing faults or security vulnerabilities from spreading throughout the entire system.

Unlike common products such as KVM, Virtual Box, and VMWare, RVM statically allocates the microcontroller's resources to each virtual microcontroller, and the resource allocation information can be determined at system compile time. Three important factors led to this design: (1) the vast majority of microcontrollers rely on Flash memory to store programs and lack the ability to load dynamic runtime libraries, (2) microcontrollers have very few resources and must use them extremely efficiently, so they can hardly bear the extra overhead brought by dynamic data structures, and (3) introducing too much dynamism into critical systems may make system behavior difficult to predict. Of course, within each virtual microcontroller, features such as dynamic memory allocation and dynamic thread creation can still be used; even if these dynamic behaviors cause a virtual microcontroller to fail, they will not affect the normal operation of the other virtual microcontrollers.

Another difference between RVM and conventional virtualization solutions is that its guests run real-time operating systems (Real-Time Operating System, RTOS). These operating systems have two characteristics: (1) they require system latency to be deterministic, and (2) they require the virtualization overhead to be very low. Conventional virtualization solutions are often based on operation[1] translation, and their underlying primitives often deviate somewhat from the interface semantics of the guest operating system; a simple guest request may be translated into a large number of hypercalls (Hypercall), causing latency amplification of hundreds of times. To solve this problem, RVM (1) designs all code inside it according to real-time system standards, with deterministic execution latency, and (2) designs its internal operation primitives and hypercall semantics to be RTOS-friendly, so that an RTOS operation is translated into only one operation primitive or hypercall.

The last difference between RVM and conventional virtualization solutions is that it often faces (extremely) stringent memory requirements. These requirements are quite extreme[2], but they are not incomprehensible: any CPU and memory overhead used for resource isolation can be saved in designs that do not value product quality; yet microcontrollers are very inexpensive and used in very high volumes, so any increase in cost is greatly reflected in the final selling price of the product; if the isolation overhead is so high that even a slightly more expensive microcontroller is required, the loss outweighs the gain. For this reason, RVM is designed to be extremely memory-frugal. It needs only a minimum of 5kB RAM and 20kB ROM to run, and may therefore be the smallest virtualization environment in the world. This allows it to be widely used on those mainstream microcontroller families with 128kB ROM and 64kB RAM, as long as they have a memory protection unit (Memory Protection Unit, MPU).

To meet the above challenges, RVM requires a large amount of address information to be determined statically at system compile time and creates a series of kernel objects with specific parameters. The cor-

[1] or binary

[2] in some cases even tens or just a few bytes must be fought for

responding data structures are also specially designed, which makes filling them in by hand very difficult. For this reason, RVM also provides a project generator, which can generate from the description of the system the initialization code, linker scripts, and even project files corresponding to the relevant build environment that the project requires.

Thus, the RME microkernel, the RVM monitor, the native processes and various RTOS virtual machines running on it, and the project generator used to generate the entire project constitute the RME microcontroller operating system ecosystem. The overall virtualization architecture is shown in the figure below: solid lines in the figure represent hardware isolation mechanisms, and dashed lines represent the boundaries of software modules.

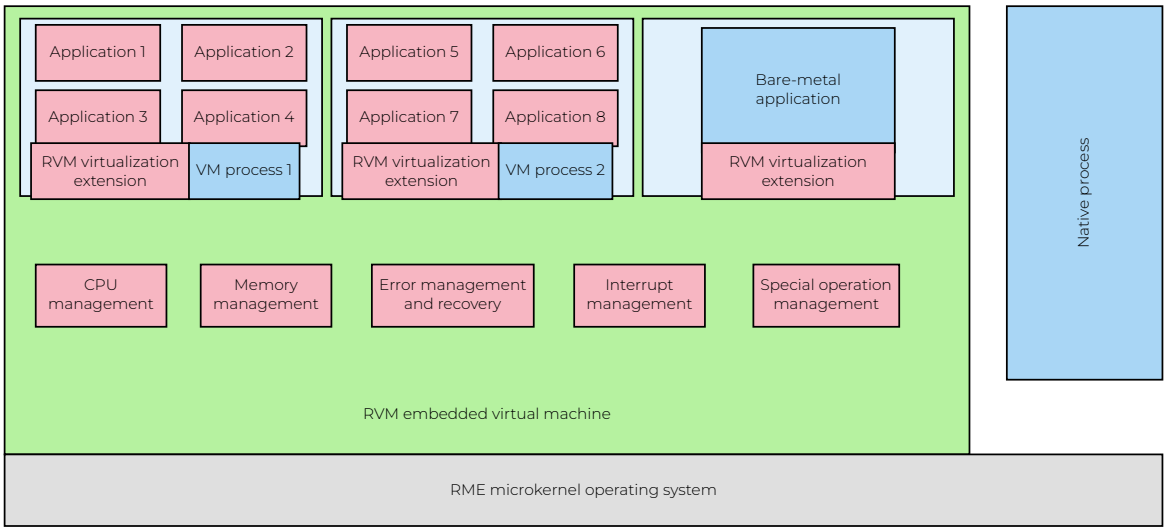


Figure 2-1 Simplified diagram of the RVM architecture

RVM virtual machines run on a virtual machine monitor implemented on the basis of paravirtualization technology. It implements all the basic functions of virtualization: memory management, CPU management, interrupt vector virtualization, and virtual machine error handling. For specific information about these implementations, see the following sections.

2.1.1 RME Microkernel

The RME microkernel exists as the underlying support for RVM and is responsible for providing basic operating system services to RVM. All other components running on the microcontroller exist as RME processes. For information about the RME microkernel, see the relevant kernel documentation.

2.1.2 RVM Virtual Machine Monitor

The RVM virtual machine monitor is responsible for the concrete virtual machine management work and runs on RME as a Type II virtual machine monitor. Specifically, it delivers physical interrupts and event source signals to each virtual machine and responds to hypercall requests from each virtual machine. When a virtual machine encounters an exception, RVM is also responsible for restarting the corresponding virtual machine. The virtual machine monitor includes five daemon threads: the initialization daemon (Initialization Daemon, INIT), the safety daemon coroutine (Safety Daemon, SFTD), the timer daemon coroutine (Timer Daemon, TIMD), the vector daemon coroutine (Vector Daemon, VCTD), and the hypercall daemon thread (Hypercall Daemon, HYPD); the latter three coroutines all run together on top of the virtual machine monitor daemon (Virtual Machine Monitor Daemon, VMMD). The following introduces these five daemon threads one by one.

2.1.2.1 Startup Daemon Thread INIT

INIT is created by RME at startup; it is responsible for initializing the native threads and virtual machines in the system and for starting the remaining four daemon threads. After this work is complete, INIT goes to sleep and is scheduled only when there is no runnable thread in the system. Once it is scheduled, it repeatedly executes the processor sleep instruction, putting the processor into a low-power state to wait for interrupts.

2.1.2.2 Safety Daemon Coroutine SFTD

SFTD runs at the highest priority in the system[3] and is responsible for monitoring the operation of native processes and virtual machines. It normally blocks on the error handling endpoint RVM_Sftd_Sig_Cap; once any exception occurs in a native process or virtual machine, SFTD unblocks and starts running. If the exception occurred in a native process, SFTD restarts the entire system; if the exception occurred in a virtual machine, SFTD restarts only that virtual machine.

2.1.2.3 Vector Daemon Coroutine VCTD

VCTD runs at a priority lower than native processes but higher than virtual machines, and is responsible for forwarding physical interrupts and event activations in the system to the interrupt sources of each virtual machine. It normally blocks on the interrupt vector endpoint RVM_BOOT_INIT_VCT; once this endpoint receives any signal, VCTD unblocks and starts running. VCTD extracts interrupt vector and event source activation information from the two RVM_Flag data structures located at RVM_PHYS_VCT_BASE and RVM_VIRT_EVT_BASE respectively, and forwards it to the virtual machines.

Each RVM_Flag data structure is formed by combining two identical struct __RME_RVM_Flag structures. This is because VCTD requires exclusive access to the struct __RME_RVM_Flag when accessing it, and at that time, if the kernel needs to pass activation information to RVM, it can only choose to fill the other copy of the structure. Each structure contains the following three fields:

Table 2-1 Meaning of each field in struct __RME_RVM_Flag

Field name	Purpose
ptr_t Lock	Determines whether this structure is currently being used by VCTD. If VCTD is using the structure, the kernel cannot operate on it; otherwise a race condition may occur. The kernel will then choose to fill the other structure.
ptr_t Fast	Indicates activation of a fast signal source. If it is 1, the fast interrupt source is activated. Currently, this field is used only in the structure located at RVM_PHYS_VCT_BASE, to convey the clock interrupt signal.
ptr_t Group	Indicates the group of an activated slow signal source; it is a bitmap. If a bit is set, it means that a signal in that group in the Flag field has been activated: if bit 2 is set, it means that a bit of Flag[2] is nonzero. The purpose of setting this field is to quickly find the group in which the activated signal is located.
ptr_t Flag[]	A bitmap indicating the specific activated signal sources. Its length is determined by the number of signals: if the word length of the processor is 32 bits and there are 96 interrupt sources, the length of this field is 96/32=3 machine words.

For example, on a 32-bit processor, if physical interrupt source 34 is activated, the kernel first searches the two struct __RME_RVM_Flag structures located at RVM_PHYS_VCT_BASE and selects the copy whose Lock field is 0. The kernel then sets bit [2] of Flag[1] and sets bit [1] of Group. Finally, the kernel sends a signal to RVM_BOOT_INIT_VCT to activate VCTD. After VCTD is activated, it alternately

[3] Kern_Prio-1

searches the two struct __RME_RVM_Flag structures for set bits. When searching, it first sets the Lock of the first struct __RME_RVM_Flag to 1, then queries which bits in Group are set, and uses that information to query the corresponding Flag[], obtaining the specific interrupt source number. If interrupt source 34 is the one activated, it should find bit [1] of Group set and then query Flag[1]; in Flag[1] it finds its bit [2] set. From this, VCTD can deduce that the activated interrupt source is number 34 and activate those virtual machines that have registered physical interrupt source 34. For information about activating virtual machines, see 2.2.4.

The activation process for event sources is similar to the activation process for physical interrupt sources. For information about event sources, see 2.2.7.

2.1.2.4 Timer Daemon Coroutine TIMD

The timer daemon thread TIMD runs at the same priority as VCTD and is responsible for delivering RVM clock ticks to each virtual machine. It normally blocks on the clock interrupt endpoint RVM_BOOT_INIT_VCT, and once the endpoint receives a timer interrupt, it delivers that interrupt to all virtual machines. Each signal TIMD receives on the clock interrupt endpoint RVM_BOOT_INIT_VCT counts as one RVM clock tick. Each virtual machine can specify a frequency division factor Period[4], and only after Period RVM clock ticks have been received is a virtual clock interrupt sent to the virtual machine.

The timer daemon thread also allows a watchdog timeout Watchdog[5] to be specified for each virtual machine. A virtual machine can use the hypercall RVM_Hyp_Wdg_Clr to feed the watchdog; if the virtual machine has still not fed the watchdog after Watchdog ticks, it is determined that the virtual machine is stuck in an infinite loop, and TIMD will restart it.

2.1.2.5 Hypercall Daemon Coroutine HYPD

The hypercall daemon thread HYPD runs at the same priority as VCTD and is responsible for receiving and handling hypercalls sent by the virtual machines. It normally blocks on the hypercall handling endpoint RVM_BOOT_INIT_VCT, and once the endpoint receives a signal, it handles the hypercall initiated by the current virtual machine. For the specific handling method of hypercalls, see 2.2.6.

2.1.3 Protection Domains

RVM provides two kinds of protection domains, namely native processes and virtual machines. A brief comparison of the two is as follows:

Table 2-2 Comparison between native processes and virtual machines

Protection domain	Confidentiality	Integrity	Availability	Auditability	Real-time	Compactness	Reliability	Usability
Native process	Strong	Weak	Weak	Weak	Strong	Strong	Weak	Weak
Virtual machine	Strong	Strong	Strong	Strong	Weak	Weak	Strong	Strong

[4] see 4.2
[5] see 4.2

2.1.3.1 Native Process

RME native processes run directly on the RME microkernel. They are not constrained by the RVM virtual machine monitor, and their resource consumption is lower than that of virtual machines. They are mainly used to receive signals from interrupt vectors and, after preliminary processing, forward these signals to virtual machines. The timeslices of threads in native processes are all unlimited, and they have the highest priority in the entire system. Even if RVM and all the virtual machines it manages have crashed, the threads in native processes can still run normally, thereby maintaining the safety of the controlled equipment.

Even though native processes are not virtual machines, the project generator still provides complete support for them, and the RVM user-mode library also provides a series of wrappers around system calls to facilitate their use in C programs. The project generator can even generate a project containing no virtual machines; in this case, application functionality can only be implemented using native processes and system calls.

RVM does not provide fault tolerance for native processes; if an exception occurs in any native process, RVM's SFTD considers that a serious problem has occurred in the system and restarts the entire system. Therefore, native processes possess only confidentiality among the three elements of security; once compromised, the integrity and availability of the system are completely lost.

2.1.3.2 Virtual Machine

Each RVM virtual machine is a separate process, inside which the hardware abstraction layer on which the RTOS depends at runtime can be emulated. It has an interrupt vector thread Vct and a user program thread Usr to emulate processor interrupts and the normal program execution flow respectively: to facilitate the RTOS's software interrupt thread switching, the Usr thread can directly trigger the running of the Vct thread, and the Vct thread can directly modify the register set of the Usr thread.

In RVM-based systems, it is recommended to implement any moderately complex functionality in virtual machines, especially language virtual machines, graphical user interfaces, and the like. This is not only because the RTOS in a virtual machine provides a series of libraries, possible POSIX support, and certified software, but also, more importantly, because a virtual machine automatically restarts only itself rather than the entire system after a fault occurs. This is crucial for fault isolation in functional safety.

A virtual machine can access a series of special virtualization-related features through hypercalls, including interrupt management, event source management and communication, etc. These hypercalls are intercepted and responded to by RVM's HYPD.

2.1.4 Project Generator

The project generator provided by RVM can generate a compilable project from a project description file. It can automatically calculate and allocate the addresses of each kernel object and generate the corresponding configuration files, without the user needing to make any manual adjustments.

2.2 Virtualization Principles

After RVM starts, the first to execute is the startup daemon thread INIT. It loads each virtual machine and is responsible for starting the safety daemon thread, the vector daemon thread, the hypercall daemon thread, and the timer daemon thread. Among them, the safety daemon thread restarts a virtual machine when an error occurs in it[6]; the vector daemon thread redirects physical interrupts from devices or event source activations to the virtual interrupt vectors of the corresponding virtual machine; the hypercall

[6] RVM does not natively support user-defined error handling

daemon thread is responsible for handling hypercalls and virtual machine scheduling; and the timer daemon thread is responsible for sending clock interrupts to each virtual machine.

Inside each virtual machine, two threads run: one is responsible for handling interrupts inside the virtual machine, and the other is responsible for running user code. The specific virtualization principles are introduced in the following subsections.

2.2.1 Priority Configuration

In RVM, the safety daemon thread SFTD always runs at the highest priority in the system. Immediately following it are the threads in native processes, which are responsible for quickly handling some interrupts in the system that cannot be sent to virtual machines in time. Next are the three daemon threads VCTD, TIMD, and HYPD, which run at a priority lower than native processes but higher than virtual machines. Then come the interrupt handling thread Vct and the user thread Usr of running virtual machines, with Vct having a higher priority than Usr. After that is the system's own INIT process, which is responsible for calling the idle hook in an infinite loop, putting the system to sleep when no virtual machine is ready. Finally come the interrupt handling threads and user program threads of virtual machines that are not running. The interrupt priorities of the threads in the entire virtualization architecture are as follows:

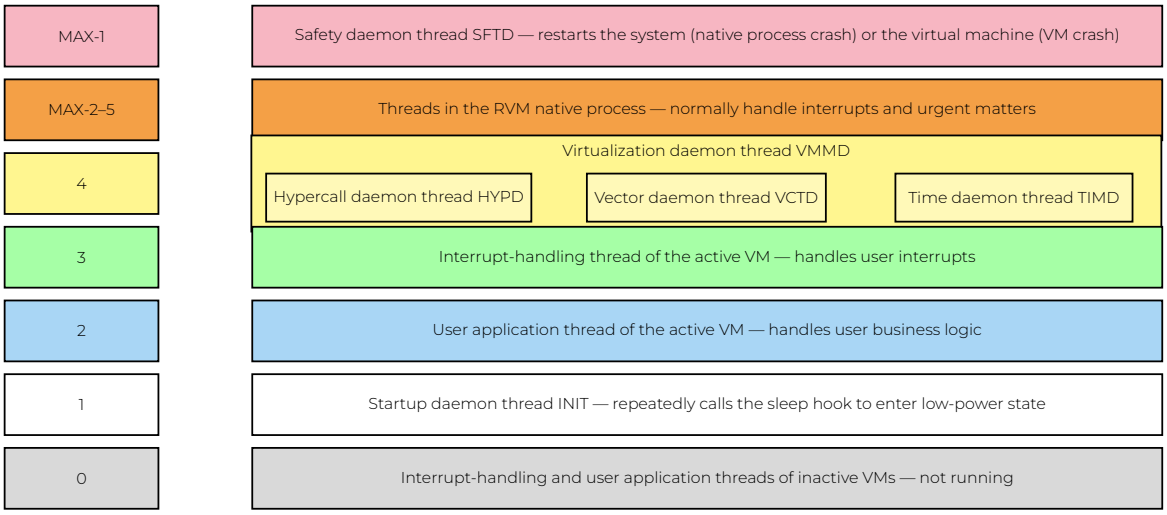


Figure 2-2 Interrupt priorities in the virtualization architecture

The virtual machines are scheduled using a fixed-priority timeslice round-robin method. The priorities among them are determined by the scheduler daemon thread. For system safety, the timeslices and priorities of virtual machines are statically determined at system compile time, and RVM does not provide hypercalls that can modify these two. This fundamentally guarantees time isolation between virtual machines, so that no virtual machine can compromise the real-time performance of the entire system.

2.2.2 Memory Management

In virtual machines, one of the most important matters is memory management. Considering the general characteristics of microcontroller applications, each virtual machine's private memory and shared memory are statically specified at compile time. After the system starts running, modification of the memory mapping is no longer allowed. For each type of processor, the format of page table information is different. Please refer to the corresponding processor's user manual to determine the page table format of that processor.

Constructing and filling in page tables by hand is very tedious; to avoid unnecessary trouble, the project generator provides functions for automatically constructing and filling in page tables.

2.2.3 CPU Virtualization

In different paravirtualization platforms, CPU virtualization mainly has two mainstream design strategies, as follows:

[1] Run each virtual machine as a single thread. This is the most common solution on dedicated paravirtualization platforms; Xen adopts this strategy. In each guest, the thread normally executes application code; once a device interrupt or exception occurs[7], the virtual machine monitor forcibly modifies the thread's execution address, forcing it to execute a piece of interrupt handling code, and after handling is finished, the thread returns by itself to execute in the application. From the virtual machine monitor's point of view, this method is the most convenient: on the one hand, each guest has only one execution context; on the other hand, activating an interrupt requires only modifying that context's instruction pointer. However, from the guest's point of view, the question of whether the end of interrupt (End-Of-Interrupt, EOI) needs to specifically notify the virtual machine monitor becomes difficult to handle. There are mainly two cases: the relatively simple case (1) in which, once the virtual machine monitor has forced a guest to execute a virtual interrupt vector, the virtual machine monitor must not modify the instruction pointer again to force the virtual machine to execute an interrupt vector until the guest uses a dedicated hypercall to notify the virtual machine monitor that the current interrupt has ended; and the more complex case (2) in which the virtual machine monitor always modifies the instruction pointer to force the virtual machine to execute an interrupt vector, regardless of whether the guest is executing a virtual interrupt vector. In case (1), the guest side needs to invoke a hypercall to notify the virtual machine, which undoubtedly adds one hypercall. For a native virtual machine monitor such as Xen, which runs directly in privileged mode, one hypercall is one context switch, and the overhead is still tolerable; for a hosted virtual machine monitor such as RVM, which runs in user mode, one hypercall means two context switches, and the introduced latency cannot be ignored. If the solution in case (2) is adopted, it is even worse: this means that the interrupt vector entry must be reentrant, and therefore requires a very special piece of assembly code. This code must be responsible for recognizing reentrancy before the reentrant execution instance reaches the real interrupt vector and then voluntarily exiting; the exit process must also involve patching the stack frame so that the execution flow returns to the interrupt vector being executed with the correct stack pointer. On the one hand, this code is very difficult to write correctly[8]; on the other hand, its beginning is always a large block of pushes onto the stack. If a series of closely consecutive interrupts occurs, the push code at the beginning will reenter multiple times before stack frame patching can be performed, potentially causing a stack overflow in an embedded environment. Therefore, even though (2) is very efficient in Xen, we do not adopt this approach.

[2] Run every thread in each guest as an independent virtual machine thread. This is a common solution for microkernel-based virtualization; L4Linux adopts this strategy[2]. On microcontrollers, the problems with this strategy are: (1) the original guest operating system's thread interface must somehow be connected to the microkernel's system calls, which requires a full understanding of the original operating system's design and may even involve modifications to architecture-independent parts such as thread control blocks; (2) mapping every guest thread to a microkernel thread creates a large number of microkernel thread objects, which is not a problem in heavyweight operating systems such as Linux whose thread structures are already large, but in RTOSes whose threads are generally extremely lightweight it amounts to multiplying memory consumption; (3) a microkernel thread switch requires a series of operation capability checks, time budget checks, and protection domain switch checks, while RTOS threads that do not even have protection domains do not need these checks at all, so calling the microkernel system call for every thread switch amounts to multiplying execution time consumption, and this is even more so on those microkernels that treat time

[7] On x86, Xen uses the hardware-specific Ring1 to accelerate system call and exception handling without going through Xen, but device interrupts still need to be distributed through Xen

[8] this is even more so for the wide variety of non-standard embedded architectures

resources as independently allocatable resources[9].

To resolve the above dilemma, RVM uses a dual-thread approach for guest CPU virtualization. Each guest has two threads, Vct and Usr; the Vct thread is responsible for interrupt handling, and the Usr thread is responsible for handling user programs. Vct normally does not run and runs only when it receives an interrupt trigger signal from RVM. After interrupt handling is complete, Vct blocks again and Usr starts running. The switch from Vct to Usr uses the microkernel's efficient asynchronous signal mechanism and does not involve RVM, so it is much faster than a hypercall.

When Usr is switched out, its register set is saved at the address specified by Reg_Base[10]. Therefore, to modify Usr's register set in Vct, it is sufficient to modify the register data at that address. Using this method, without needing to understand the original RTOS design, one can replace its hardware abstraction layer, especially its thread switching assembly, line by line, thereby easily emulating the thread context switching that is very common in RTOSes.

2.2.4 Interrupt Virtualization

For virtual machines, interrupt virtualization is indispensable. Each RVM guest has an interrupt handling thread Vct, whose priority is higher than that of the Usr thread that handles user programs. This thread normally blocks on the virtual interrupt vector endpoint dedicated to each virtual machine[11], and is woken up to run only when a virtual interrupt signal from RVM is received.

Interrupts in the system can roughly be divided into three categories: the first is clock interrupts, which are responsible for providing ticks to the operating system; the second is thread switching interrupts, which are responsible for RTOS thread switching; and the third is other system interrupts, which are responsible for responding to transactions of other peripherals. The characteristics of these three categories also differ: clock interrupts occur periodically and frequently; thread switching interrupts are essentially software interrupts sent by a thread to itself; and other system interrupts are irregular interrupts that occur from time to time. Therefore, RVM adopts different virtualization strategies for these three kinds of interrupts.

Each virtual machine maintains an RVM_State data structure, whose address is specified by State_Base[12]. Its internal fields are as follows:

Table 2-3 Meaning of each field in struct RVM_State

Field name	Purpose
ptr_t Vct_Act	Indicates the thread currently running in the virtual machine. If set to 1, the Vct thread is running; otherwise, the Usr thread is running.
struct RVM_Param Vct	Hypercall parameter area dedicated to the Vct thread.
struct RVM_Param Usr	Hypercall parameter area dedicated to the Usr thread.
struct RVM_Vctf Flag	Indicates the virtual interrupt vectors activated in the current virtual machine.

In this data structure, the data structure related to interrupt virtualization is struct RVM_Vctf Flag, whose fields are shown in the following table. For the other fields, see 2.2.6.

[9] such as Composite and some L4 variants
[10] see 4.2
[11] this endpoint is located at capability number 1 inside each virtual machine
[12] see 4.2

Table 2-4 Meaning of each field in struct RVM_Vctf

Field name	Purpose
ptr_t Tim	Indicates clock interrupt activation.
ptr_t Ctx	Indicates thread switching interrupt activation.
u8_t Vct[]	Indicates activation of other virtual interrupt vectors. Each byte corresponds to one virtual interrupt vector.

In this data structure, activation of the clock interrupt and the thread switching interrupt is indicated by two separate machine words, while each of the other virtual interrupt vectors corresponds to one byte. When the corresponding interrupt is activated, the corresponding field is set to 1. Here, the method of one binary bit per interrupt vector is not adopted, so that no atomic logical AND instruction is needed[13]. Since the number of virtual interrupt vectors of each virtual machine is limited, the resulting waste of storage space is tolerable.

For example, if VCTD determines that interrupt vector 4 of a virtual machine is activated, it sets Vct[4] to 1 and sends a signal to the dedicated endpoint of that virtual machine's Vct thread. After the Vct thread receiving the signal is activated, it first scans the entire structure and determines that virtual interrupt vector 4 is activated from Vct[4] being set to 1. After that, the Vct thread first clears Vct[4] to zero and then invokes virtual interrupt vector 4 registered by the user using RVM_Virt_Vct_Reg[14]. In this way, even if a new interrupt 4 arrives while the user-registered interrupt vector is executing, no interrupt loss occurs, because Vct[4] will be set again: after the next loop, the Vct thread will detect that interrupt 4 is pending and enter it again.

A virtual machine's interrupt enable/disable, as well as the correspondence between the virtual machine's virtual interrupt vectors and the corresponding physical interrupt vectors or event sources, can be set through hypercalls. For information about these hypercalls, see 4.4. A standard virtual machine interrupt initialization flow is as follows:

[1] Call RVM_Virt_Tim_Reg to register the clock interrupt,

[2] Call RVM_Virt_Ctx_Reg to register the thread switching interrupt,

[3] Call RVM_Virt_Vct_Reg to register other virtual interrupts,

[4] Call RVM_Hyp_Vct_Phys to map the required physical interrupt sources to virtual interrupt sources,

[5] Call RVM_Hyp_Vct_Evt to map the required event sources to virtual interrupt sources,

[6] Call RVM_Hyp_Vct_Lck to lock the event source mappings to prevent subsequent tampering,

[7] Call RVM_Hyp_Int_Ena at an appropriate place to enable the virtual machine's global interrupts. If the virtual machine also uses a watchdog, RVM_Hyp_Wdg_Clr needs to be called to start the watchdog. Once started, the watchdog cannot be disabled.

[8] The RTOS inside the virtual machine should repeatedly call RVM_Hyp_Vct_Wait in its idle thread so that the virtual machine enters sleep when idle and waits for the next interrupt. If the virtual machine also uses a watchdog, RVM_Hyp_Wdg_Clr needs to be called at an appropriate place to feed the watchdog.

2.2.4.1 Virtualization of Timer Interrupts

RTOSes running on RVM can be divided into two categories according to the way they use timers: periodic kernels and tickless kernels. The former requires the system to have an auto-reload timer that periodically

[13] When VCTD sets a vector, the Vct thread may be resetting one of the vectors. If one binary bit corresponded to one interrupt vector, the Vct thread would have to use an atomic logical AND operation to reset the flag bit. Some architectures, such as ARMv6-M, do not have such an atomic operation, so one byte per flag offers better compatibility.

[14] see 4.5.5

sends interrupts, while the latter requires programming, as needed on each context switch, a one-shot timer that sends an interrupt after a countdown ends. On bare metal, the latter is superior in terms of low-power characteristics, because no clock interrupt is generated when the processor is sleeping or only one task is scheduled. For this reason, we often enable this feature on RTOSes that support tickless kernels.

However, in a virtualization environment the situation is exactly the opposite. A tickless kernel can no longer program the timer directly; it must ask RVM for help through a hypercall. This results in an extra hypercall on every context switch, causing intolerable performance and real-time overhead. In addition, not all RTOSes support tickless kernel mode: even if only one periodic kernel runs on RVM, RVM still has to be woken up periodically to send timer interrupts to it, so reducing power consumption is impossible; this makes the tickless feature of other kernels add only overhead. Even setting that aside, even if all virtual machines running on RVM are tickless, the total number of times they program the timer would still be very large, and RVM must set the hardware timer's timeout to the minimum of all these times. This causes the timer to always be triggered after a short period of time, and is also unlikely to achieve a low-power effect.

For the above reasons, RVM uses a periodic design. The periodic clock signal it needs is generated by the RME underlying timer and sent to RVM's clock handling daemon thread through the RVM_BOOT_INIT_TIM endpoint. This daemon thread calculates the number of time periods currently elapsed and sends clock interrupts to each virtual machine according to the frequency division factor Period[15] configured for each virtual machine.

Specifically, the Vct thread calls the thread switching hook Tim registered by the user using RVM_Virt_Tim_Reg[16]. This hook function can directly use the timer interrupt vector of the original bare-metal RTOS, and when needed can use RVM_Virt_Yield[17] to trigger a context switch[18].

2.2.4.2 Virtualization of Thread Switching Interrupts

A thread switching interrupt is essentially a software interrupt, originating from inside the virtual machine itself. The response efficiency of this interrupt directly determines the thread switching time of the virtual machine, so ensuring its efficiency is very important. For this reason, RVM is specially designed so that the user thread Usr can directly trigger a context switch led by the Vct thread using RVM_Virt_Yield[19] without intervention of the virtual machine monitor, and in the Vct thread the Usr register set, which is mapped in advance to Reg_Base[20], can be modified directly without using any system call. This implementation is as close as possible to the thread switching semantics of common RTOSes, greatly reducing the thread switching overhead inside the virtual machine.

Specifically, the Vct thread calls the thread switching hook Ctx registered by the user using RVM_Virt_Ctx_Reg[21]. This hook implements the saving and restoring of Usr's various registers, thereby completing the RTOS thread switch. A common thread switching method consists of two steps: (1) push all registers onto the old thread's stack, and (2) restore all registers from the new thread's stack; this process is usually implemented in assembly language in a bare-metal RTOS. In RVM, the Ctx hook that implements this flow can be written in C[22].

Unlike a bare-metal RTOS, the saving and restoring of registers must include the hypercall parameter buffer used by the Usr thread, which is located in the struct RVM_State structure's struct RVM_Param Usr field. This is because multiple RTOS threads corresponding to the Usr thread may access it concurrently[23]. For detailed information about this field, see 2.2.6.

[15] see 4.2
[16] see 4.5.7
[17] see 4.5.7
[18] just as a context switching interrupt such as PendSV is triggered in the original RTOS
[19] see 4.5.7
[20] see 4.2
[21] see 4.5.7
[22] for a concrete example, see the RVM port of RMP
[23] for example, they perform hypercalls at the same time

2.2.4.3 Virtualization of Other Interrupts

Because different virtual machines need to receive different peripheral interrupts, other peripheral interrupts must be delivered to each virtual machine on demand. Therefore, the system provides a call interface that allows each virtual machine to set the physical interrupt source corresponding to each of its own virtual interrupt vectors. Each virtual machine can and can only set the physical interrupt source corresponding to its own virtual interrupt vectors. Once that physical interrupt is triggered, the corresponding virtual interrupt vectors of all virtual machines registered on this physical interrupt will receive this interrupt. The smaller the number of a virtual interrupt source, the higher its priority, but no preemption or nesting occurs between virtual interrupts.

The interrupt handling vector corresponding to each virtual interrupt can be registered as an unmaskable "transparent interrupt" or a maskable "perceivable interrupt"[24]. The former is still responded to when interrupts are masked, while the latter is not responded to when interrupts are masked. Timer interrupts and thread switching interrupts are always maskable. For information about these hypercalls, see 4.4.

2.2.5 Device Virtualization

In different virtualization platforms, the virtualization of devices and their drivers mainly has four mainstream design strategies, as follows:

[1] Put the device drivers into a separate privileged virtual machine, Dom0, and have the remaining virtual machines access devices by sending requests to that virtual machine. This is the strategy commonly adopted by native virtual machine monitors such as Xen[1]. The motivation for this strategy is easy to understand: (1) devices handed directly to virtual machines cannot be shared safely; to isolate them from virtual machines, they must be placed in a separate protection domain, and the virtual machine monitor provides virtual machines as the isolation means, (2) guest operating systems already have ready-made drivers, and preparing another set of drivers specifically for the virtual machine monitor would be tantamount to reinventing the wheel. However, this strategy makes any device I/O involve expensive inter-virtual-machine context switches, with enormous performance overhead. In addition, once the virtual machine monitor schedules or distributes Dom0's interrupts improperly, the latency will spread throughout the entire system, causing devastating real-time problems[3]. Therefore, RVM cannot adopt this strategy.

[2] Put the device drivers in the kernel mode in which the virtual machine monitor resides. This is another strategy adopted by native virtual machine monitors such as VMware ESXi. This approach can completely avoid Dom0's performance problems, but it places the virtual machine monitor——that is, the trusted computing base of the entire system——under the threat of malicious drivers. Obviously, this can be done only when the virtual machine monitor vendor is very confident in its drivers, and third-party device vendors must also cooperate with this certification. None of these conditions exist in microcontrollers, so RVM cannot adopt this strategy.

[3] Make the virtual machine monitor and the operating system kernel into one, so that guests naturally enjoy the device drivers that come with the operating system. This is the strategy usually adopted by virtual machine monitors such as KVM. It is actually more or less the same as the previous strategy, except that driver developers generally believe that the development work is done for Linux, and are therefore relatively more motivated.

[4] Place each driver directly in each guest. This is the solution in many industrial and functional safety scenarios, because each virtual machine in an industrial scenario often has its own dedicated role[25],

[24] this emulates the PRIMASK feature of ARMv7-M and the like, allowing the guest to respond to interrupts unrelated to the RTOS while the scheduler is locked, improving real-time performance

[25] one could say a "vertical cut" along functional interfaces, rather than a "horizontal cut" along abstraction-layer interfaces as in most systems

and the functional overlap between guests is very small, as is the degree of hardware sharing. Exploiting this characteristic of industrial embedded scenarios, this approach is often very efficient when devices are partitioned properly, because the virtual machine monitor no longer intervenes in any device operation other than interrupt distribution. However, this approach requires that (1) errors or even malicious operations of any device must not threaten system safety and real-time performance, including the often-neglected problems caused by side channels[26], (2) any device can be exclusively owned by only one virtual machine, with no device sharing between virtual machines possible, and (3) interrupts still have to be delivered into the virtual machine before they can be handled, so real-time performance is insufficient. Therefore, this approach cannot always be used.

Considering the actual situation of microcontrollers, RVM encourages the use of approach 4 whenever possible. If real-time performance is insufficient, or devices need to be shared, one may consider placing the simple, trustworthy upper half of the driver[27] in a native process with extremely fast interrupt response, and placing its lower half and protocol stack in the guest. Native processes and guests can both share a portion of memory for passing data, and when sharing, it must be ensured that the guest does not corrupt the data necessary for thread execution in the native process.

2.2.6 Hypercall Design

A virtual machine can request services from the virtual machine monitor through hypercalls. For information about these hypercalls, see 4.4. To initiate a hypercall, the virtual machine first writes the various parameters required by the hypercall into the parameter buffer agreed upon with RVM, and then sends a signal to HYPD's signal endpoint RVM_Hypd_Sig_Cap[28]. This parameter buffer is located in the RVM_State data structure and exists in two copies: struct RVM_Param Vct is dedicated to the Vct thread, and struct RVM_Param Usr is dedicated to the Usr thread, so that race conditions between the Vct thread and the Usr thread on the parameter buffer can be prevented.

After HYPD receives a hypercall, it determines the currently running thread through Vct_Act in the RVM_State data structure. If Vct_Act is 1, the currently running thread is the Vct thread, and the hypercall parameters should be extracted from the Vct thread's parameter buffer; otherwise, the currently running thread is the Usr thread, and the hypercall parameters should be extracted from the Usr thread's parameter buffer. After executing the hypercall, HYPD writes the return value back into the corresponding parameter buffer; after the virtual machine regains the right to run, it retrieves that value as the hypercall's return value.

Because multiple RTOS threads corresponding to one Usr thread may also concurrently access the Usr thread's parameter buffer, during RTOS thread switching the parameter buffer used by the Usr thread is pushed onto and popped from the stack together with the register set, which ensures that no race condition occurs.

2.2.7 Cross-Protection-Domain Communication

For security, by default native processes cannot send signals to one another, native processes and virtual machines cannot send signals to one another, and virtual machines cannot send signals to one another. To allow them to communicate with one another, some configuration is needed to place appropriate signal primitives in each protection domain.

2.2.7.1 Sending Signals from a Native Process to a Native Process

Native processes can communicate through RME's asynchronous signal mechanism. After a thread in one process blocks on a signal endpoint, a thread in another process can unblock it simply by sending a

[26] the most typical is the DMA controller; allowing an untrusted virtual machine to operate it arbitrarily can easily bypass all security mechanisms

[27] usually including only simple register reads and writes and the sending of message queues

[28] this endpoint is located at capability number 0 inside each virtual machine

signal to that endpoint. This requires one process to have the receive capability for the endpoint and the other process to have the send capability for the endpoint. For related information, see 3.5.6.

The project generator can correctly generate this capability layout from the user's description of the native processes. It is only necessary to declare the receive endpoint in the receiving process and then send to that endpoint in the sending process.

2.2.7.2 Sending Signals from a Virtual Machine to a Native Process

To send from a virtual machine to a native process, simply declare a receive signal endpoint in the native process and declare a send signal endpoint in the virtual machine, and the virtual machine can directly initiate signal sending to the native process. For related information, see 3.5.6.

The project generator allows a virtual machine to reference any receive endpoint declared in any native process and can correctly grant the corresponding endpoint send permission to the virtual machine at system startup; the virtual machine only needs to send to that endpoint.

2.2.7.3 Sending Signals from a Native Process or Virtual Machine to a Virtual Machine

Because the Vct and Usr threads of a virtual machine are not allowed to block on signal endpoints, and no kernel object other than the signal endpoint send capability can be defined inside a virtual machine, communication from other protection domains[29] to a virtual machine can only use the method of activating a virtual interrupt. For this purpose, RVM introduces the concept of an "event source"[30]: each virtual machine can freely register its virtual interrupt vectors with an event source through RVM_Hyp_Vct_Evt, and once that event source is activated, all virtual machines that have registered that interrupt source will receive that virtual interrupt.

The priority of native processes is higher than that of virtual machines, so restricting their sending to event sources is meaningless[31]. This is not the case between virtual machines: after a virtual machine starts, it must first request permission in advance to send to event sources through the RVM_Hyp_Evt_Add function, and after the request is completed, use RVM_Hyp_Vct_Lck to lock all event source send permissions[32], after which no further modification of the permissions is allowed. This design is based on the assumption that thereafter, sending to event sources can be done through RVM_Hyp_Evt_Snd. For related information, see 4.4.

To send from a virtual machine to a native process, simply declare a receive signal endpoint in the native process and declare a send signal endpoint in the virtual machine, and the virtual machine can directly initiate signal sending to the native process. For these functions, see 3.5.6.

The total number of event sources and the total number of mappings from event sources/physical interrupt sources to virtual interrupts can be configured in the project generator, and are statically determined before project compilation; the specific mappings are established dynamically at runtime.

[29] regardless of whether the protection domain is a native process or a virtual machine

[30] like a conventional physical interrupt source, an event source is also a global source of stimuli

[31] they only need to keep preempting the processor to prevent the virtual machine monitor and the virtual machines from running, so restricting them is meaningless

[32] as well as the mappings from event sources/interrupt sources to virtual interrupts

2.3 References for This Chapter

[1] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Open Source Software Development Series. Upper Saddle River, NJ, USA: Prentice Hall, 2008.

[2] A. Lackorzynski, "L4Linux porting optimizations," Master's thesis.

[3] S. Yoo, K.-H. Kwak, J.-H. Jo, and C. Yoo, "Toward under-millisecond I/O latency in Xen-ARM," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, Shanghai, China: ACM, 2011.

3. Native Process

3.1 Introduction

Native processes run directly on top of the RME microkernel, so they run the fastest and with the smallest overhead. The RVM virtual machine monitor is only responsible for starting them with INIT, and cannot provide them with any services. Therefore, a native process can only use the system calls of RME itself, and cannot use RVM hypercalls. To make it convenient for users to use RME system calls, the user-mode library provided by RVM wraps them in C language interfaces. These interfaces are all thread-safe.

The project generator can generate a project that contains native processes. A project can even contain only native processes; in this case the functionality of the application must be implemented directly with native processes.

3.2 Process Descriptor

Every native process (and virtual machine) has a process descriptor at its head, and this header records every possible entry point in the process. These entry points are recognized by the virtual machine monitor so that it can initialize its internal data structures. The composition of the process header is shown in the following table.

Table 3-1 Structure of the process descriptor

Address (word)	Content
0	Magic number; it is 0x49535953 for a native process and 0x56495254 for a virtual machine.
1	The number N of entry table entries in the process header.
2	Entry table entry 0.
3	Entry table entry 1.
...	...
N+1	Entry table entry N-2.
N+2	Entry N-1, which is fixed to hold the trampoline. On conventional architectures this entry is useless; on those architectures that restore the PC from the stack[1], a trampoline is required, and this trampoline is responsible for jumping the program to the true entry point for execution. For a description of this trampoline, please refer to8.2.6.2 .

The project generator can generate this process descriptor, and of course it can also be filled in manually.

[1] e.g., ARMv7-M

3.3 Registering a Native Process with the RVM

To register a native process with the RVM, the user must manually use RME system calls to create its capability table, page table, and all kernel objects that may be used inside it, and transfer the capabilities of all kernel objects usable inside the process into that process's capability table. When the permission relationships are very complex, this is very tedious and error-prone. The project generator can generate highly optimized data structures to complete the above flow automatically, greatly reducing the configuration burden. If this procedure needs to be done manually, the following two hook functions in rvm_init.c can be rewritten, and each of the kernel objects can be created and initialized inside them respectively.

3.3.1 Kernel Object Creation Hook

The user should create all kernel objects inside this hook.

Table 3-2 Kernel object creation hook

Function prototype	void RVM_Boot_Kobj_Crt(void)
Return value	None.
Parameters	None.

3.3.2 Kernel Object Initialization Hook

The user should initialize all kernel objects inside this hook.

Table 3-3 Kernel object initialization hook

Function prototype	void void RVM_Boot_Kobj_Init(void)
Return value	None.
Parameters	None.

3.4 Macro Definitions That Should Exist in a Native Process

The following configuration macro definitions need to exist inside a native process; they are referenced by other files inside the native process. These macro definitions can be filled in by the project generator, or filled in manually by the user. The list of these macro definitions is as follows.

Table 3-4 List of macro definitions that should exist in a native process

Macro name	Purpose
RVM_VIRT_LIB_ENABLE	The enable macro of the paravirtualization library. Inside a native process it should be defined as 1. Example: #define RVM_VIRT_LIB_ENABLE (0U)
RVM_PREEMPT_PRIO_NUM	The number of priorities of the kernel. It should remain consistent with the configuration macro of the same name in RME. Example: #define RVM_PREEMPT_PRIO_NUM (4U)
RVM_PREEMPT_VPRIO_NUM	The number of virtual priorities of the virtual machine monitor. It should be consistent with the value in the virtual machine monitor configuration. Example: #define RVM_PREEMPT_VPRIO_NUM (32U)

Macro name	Purpose
RVM_KOM_SLOT_ORDER	The granularity order of kernel memory allocation. It should remain consistent with the configuration macro of the same name in RME. Example: #define RVM_KOM_SLOT_ORDER (4U)
RVM_ASSERT_ENABLE	Controls whether assertions pass by default. If it is defined as 0, assertions are no longer checked. Example: #define RVM_ASSERT_ENABLE (1U)
RVM_DBGLOG_ENABLE	The switch that controls debug printing. If it is defined as 0, debug printing is no longer performed. Example: #define RVM_DBGLOG_ENABLE (1U)

3.5 System Calls Available to Native Processes

All system calls provided by RME can be used directly in native processes. The generic parts of the implementation of these system calls are located in rmv_guest.c and rvm_guest.h under the Guest folder, while the architecture-related parts are located in rvm_guest_xxx.c, rvm_guest_xxx.h, and rvm_guest_xxx_conf.h in the subfolder. The following only gives a brief description of these system calls; for specific information about these system calls, please refer to the RME manual.

3.5.1 Capability Table Related System Calls

These interfaces provide C language interfaces for these capability table related calls. The specific function list is as follows:

Table 3-5 Capability table related system calls

Name	Interface function prototype
Create capability table	ret_t RVM_Cpt_Crt(cid_t Cap_Cpt_Crt, cid_t Cap_Kom, cid_t Cap_Cpt, ptr_t Raddr, ptr_t Entry_Num)
Delete capability table	ret_t RVM_Cpt_Del(cid_t Cap_Cpt_Del, cid_t Cap_Del)
Transfer ordinary capability	ret_t RVM_Cpt_Add(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Flag) Note: this function can only be used to transfer capabilities other than page directory capabilities, kernel function call capabilities, and kernel memory capabilities.
Transfer page table capability	ret_t RVM_Cpt_Pgt(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End, ptr_t Flag) Note: this function cannot be used to transfer other types of capabilities. The two parameters Begin and End mark the start slot and the end slot of the new page table capability, that is, its operation range. This range includes End.

Name	Interface function prototype
Transfer kernel function call capability	ret_t RVM_Cpt_Kern(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End) Note: this function cannot be used to transfer other types of capabilities. The two parameters Begin and End mark the start slot and the end slot of the new kernel function call capability, that is, its operation range. This range includes End.
Transfer kernel memory capability	ret_t RVM_Cpt_Kom(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End, ptr_t Flag) Note: this function cannot be used to transfer other types of capabilities. The two parameters Begin and End mark the start address and the end address of the new kernel memory capability, and they are forcibly aligned to 64Byte . This range does not include End.
Freeze capability	ret_t RVM_Cpt_Frz(cid_t Cap_Cpt_Frz, cid_t Cap_Frz)
Remove capability	ret_t RVM_Cpt_Rem(cid_t Cap_Cpt_Rem, cid_t Cap_Rem)

3.5.2 Kernel Function Related System Calls

These interfaces provide C language interfaces for these kernel function related calls. The specific function list is as follows:

Table 3-6 Kernel function related system calls

Name	Interface function prototype
Kernel call activation	ret_t RVM_Kfn_Act(cid_t Cap_Kfn, ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)

3.5.3 Page Table Related System Calls

These interfaces provide C language interfaces for these page table related calls. The specific function list is as follows:

Table 3-7 Page table related system calls

Name	Interface function prototype
Page directory creation	ret_t RVM_Pgt_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Pgt, ptr_t Raddr, ptr_t Start_Addr, ptr_t Is_Top, ptr_t Size_Order, ptr_t Num_Order)
Delete page directory	ret_t RVM_Pgt_Del(cid_t Cap_Cpt, cid_t Cap_Pgt)
Map memory page	ret_t RVM_Pgt_Add(cid_t Cap_Pgt_Dst, ptr_t Pos_Dst, ptr_t Flag_Dst, cid_t Cap_Pgt_Src, ptr_t Pos_Src, ptr_t Index)
Remove memory page	ret_t RVM_Pgt_Rem(cid_t Cap_Pgt, ptr_t Pos)
Construct page directory	ret_t RVM_Pgt_Con(cid_t Cap_Pgt_Parent, ptr_t Pos, cid_t Cap_Pgt_Child, ptr_t Flags_Childs)
Destruct page directory	ret_t RVM_Pgt_Des(cid_t Cap_Pgt_Parent, ptr_t Pos, cid_t Cap_Pgt_Child)

3.5.4 Process Related System Calls

These interfaces provide C language interfaces for these process related calls. The specific function list is as follows:

Table 3-8 Process related system calls

Name	Interface function prototype
Create process	ret_t RVM_Prc_Crt(cid_t Cap_Cpt_Crt, cid_t Cap_Prc, cid_t Cap_Cpt, cid_t Cap_Pgt)
Delete process	ret_t RVM_Prc_Del(cid_t Cap_Cpt, cid_t Cap_Prc)
Change capability table	ret_t RVM_Prc_Cpt(cid_t Cap_Prc, cid_t Cap_Cpt)
Change page table	ret_t RVM_Prc_Pgt(cid_t Cap_Prc, cid_t Cap_Pgt)

3.5.5 Thread Related System Calls

These interfaces provide C language interfaces for these thread related calls. The specific function list is as follows:

Table 3-9 Thread related system calls

Name	Interface function prototype
Create thread	ret_t RVM_Thd_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Max_Prio, ptr_t Raddr, ptr_t Attr)
Create virtual machine dedicated thread[2]	ret_t RVM_Hyp_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Max_Prio, ptr_t Raddr, ptr_t Attr)
Delete thread	ret_t RVM_Thd_Del(cid_t Cap_Cpt, cid_t Cap_Thd)
Set execution attributes	ret_t RVM_Thd_Exec_Set(cid_t Cap_Thd, void* Entry, void* Stack, void* Param)
Thread binding	ret_t RVM_Thd_Sched_Bind(cid_t Cap_Thd, cid_t Cap_Thd_Sched, cid_t Cap_Sig, tid_t TID, ptr_t Prio)
Virtual machine dedicated thread binding[3]	ret_t RVM_Hyp_Sched_Bind(cid_t Cap_Thd, cid_t Cap_Thd_Sched, cid_t Cap_Sig, tid_t TID, ptr_t Prio, ptr_t Haddr)
Change priority	ret_t RVM_Thd_Sched_Prio(cid_t Cap_Thd, ptr_t Prio)
Change priorities (2)	ret_t RVM_Thd_Sched_Prio2(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1) Note: the priorities of two threads can be modified at one time.
Change priorities (3)	ret_t RVM_Thd_Sched_Prio3(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1, cid_t Cap_Thd2, ptr_t Prio2) Note: the priorities of three threads can be modified at one time.
Change priorities (4)	ret_t RVM_Thd_Sched_Prio4(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1, cid_t Cap_Thd2, ptr_t Prio2, cid_t Cap_Thd3, ptr_t Prio3) Note: the priorities of four threads can be modified at one time.

[2] Do not call unless you want to implement the virtual machine monitor yourself
[3] Do not call unless you want to implement the virtual machine monitor yourself

Name	Interface function prototype
Receive scheduling event	ret_t RVM_Thd_Sched_Rcv(cid_t Cap_Thd)
Unbind	ret_t RVM_Thd_Sched_Free(cid_t Cap_Thd)
Transfer timeslice	ret_t RVM_Thd_Time_Xfer(cid_t Cap_Thd_Dst, cid_t Cap_Thd_Src, ptr_t Time)
Switch to a thread	ret_t RVM_Thd_Swt(cid_t Cap_Thd, ptr_t Is_Yield)

3.5.6 Asynchronous Signal Endpoint Related System Calls

These interfaces provide C language interfaces for these signal endpoint related calls. The specific function list is as follows:

Table 3-10 Asynchronous signal endpoint related system calls

Name	Interface function prototype
Create signal endpoint	ret_t RVM_Sig_Crt(cid_t Cap_Cpt, cid_t Cap_Sig)
Delete signal endpoint	ret_t RVM_Sig_Del(cid_t Cap_Cpt, cid_t Cap_Sig)
Send to endpoint	ret_t RVM_Sig_Snd(cid_t Cap_Sig, ptr_t Number)
Receive from endpoint	ret_t RVM_Sig_Rcv(cid_t Cap_Sig, ptr_t Option)

3.5.7 Synchronous Invocation Related System Calls

These interfaces provide C language interfaces for these thread invocation related calls. The specific function list is as follows:

Table 3-11 Synchronous invocation related system calls

Name	Interface function prototype
Create synchronous invocation	ret_t RVM_Inv_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Inv, cid_t Cap_Prc, ptr_t Raddr)
Delete synchronous invocation	ret_t RVM_Inv_Del(cid_t Cap_Cpt, cid_t Cap_Inv)
Set execution attributes	ret_t RVM_Inv_Set(cid_t Cap_Inv, ptr_t Entry, ptr_t Stack, ptr_t Is_Exc_Ret)
Activate synchronous invocation	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
Return from synchronous invocation	ret_t RVM_Inv_Ret(ptr_t Retval)

3.5.8 Event Source Activation Related System Calls

To send a message from a native process to a virtual machine, the RVM_Prc_Evt_Snd function can be used to activate any event source, and every virtual machine that has registered that event source will receive a virtual interrupt. Since the priority of a native process itself is higher than that of the entire virtual machine subsystem[4], this function therefore imposes no restriction on the event source target, and a native process can decide by itself to send a signal to any event source. For event sources, please refer to2.2.7 .

[4]this even includes all threads in the virtual machine monitor except SFTD

Table 3-12 Parameters required to activate an event source

Prototype	ret_t RVM_Prc_Evt_Snd(ptr_t Evt_Num)	
Parameter name	Type	Description
Evt_Num	ptr_t	The number of the event source.

3.6 Other Helper Functions Available to Native Processes

To facilitate the writing of common data structures, debug printing, and thread stack initialization, RVM also provides some helper functions. All helper functions listed below can be used in native processes.

Table 3-13 List of other helper functions available to native processes

Function	Meaning
RVM_Clear	Clear a variable.
RVM_List_Crt	Create a doubly linked circular list.
RVM_List_Del	Delete a node in a doubly linked circular list.
RVM_List_Ins	Insert a node in a doubly linked circular list.
RVM_Putchar	Print a single character.
RVM_Int_Print	Print an integer number.
RVM_Hex_Print	Print an unsigned integer number.
RVM_Str_Print	Print a string.

3.6.1 Variable Clearing

This function is used to clear a region in the kernel. This function is essentially equivalent to the special case of the memset function of the C language runtime library when filling with 0.

Table 3-14 Parameters required for variable clearing

Prototype	void RVM_Clear(volatile void* Addr, ptr_t Size)	
Parameter name	Type	Description
Addr	volatile void*	The start address of the region to be cleared.
Size	ptr_t	The number of bytes of the region to be cleared.

3.6.2 Creating a Doubly Linked Circular List

This operation initializes the list head of a doubly linked circular list.

Table 3-15 Creating a doubly linked circular list

Function prototype	void RVM_List_Crt(struct RVM_List* Head)
Return value	None.
Parameters	struct RVM_List* Head Pointer to the list head structure to be initialized.

3.6.3 Deleting a Node in a Doubly Linked Circular List

This operation deletes one node or a series of nodes from a doubly linked list.

Table 3-16 Deleting a node in a doubly linked circular list

Function prototype	void RVM_List_Del(struct RVM_List* Prev, struct RVM_List* Next)
Return value	None.
Parameters	struct RVM_List* Prev Pointer to the predecessor node of the node (group) to be deleted. struct RVM_List* Next Pointer to the successor node of the node (group) to be deleted.

3.6.4 Inserting a Node in a Doubly Linked Circular List

This operation inserts a node into a doubly linked list.

Table 3-17 Inserting a node in a doubly linked circular list

Function prototype	void RVM_List_Ins(struct RVM_List* New, struct RVM_List* Prev, struct RVM_List* Next)
Return value	None.
Parameters	struct RVM_List* New Pointer to the new node to be inserted. struct RVM_List* Prev Pointer to the predecessor node of the position to be inserted at. struct RVM_List* Next Pointer to the successor node of the position to be inserted at.

3.6.5 Printing a Single Character

This operation prints one character to the console. This function needs to be declared and implemented by the user. In the implementation of this function, it is usually only necessary to redirect its output to some peripheral. If this function is not implemented, all the debug printing functions later in this document cannot be used. The project generator will provide the prototype of this function in the generated process file, and the user only needs to fill it in with appropriate content.

Table 3-18 Printing a single character

Function prototype	ptr_t RVM_Putchar(char Char)
Meaning	Output one character to the console.
Return value	ptr_t Always returns 0.
Parameters	char Char The character to be output to the system console.

3.6.6 Printing an Integer Number

This operation prints a machine-word-sized integer number in decimal with sign to the debug console. This function has an equivalent macro RVM_DBG_I controlled by the debug switch. To use this function, RVM_Putchar needs to be declared and implemented in the process.

Table 3-19 Printing an integer number

Function prototype	cnt_t RVM_Int_Print(cnt_t Int)
Return value	cnt_t Returns the number of characters printed.
Parameters	cnt_t Int The integer number to be printed.

3.6.7 Printing an Unsigned Integer Number

This operation prints a machine-word-sized unsigned integer number in hexadecimal without prefix to the debug console. This function has an equivalent macro RVM_DBG_H controlled by the debug switch. To use this function, RVM_Putchar needs to be declared and implemented in the process.

Table 3-20 Printing an unsigned integer number

Function prototype	cnt_t RVM_Hex_Print(ptr_t Uint)
Return value	cnt_t Returns the number of characters printed.
Parameters	ptr_t Uint The unsigned integer number to be printed.

3.6.8 Printing a String

This operation prints a string of at most 127 characters to the debug console. This function has an equivalent macro RVM_DBG_S controlled by the debug switch. To use this function, RVM_Putchar needs to be declared and implemented in the process.

Table 3-21 Printing a string

Function prototype	cnt_t RVM_Str_Print(s8_t* String)
Return value	cnt_t Returns the number of characters printed.
Parameters	s8_t* String The string to be printed.

3.7 References for This Chapter

None

4. Virtual Machine

4.1 Introduction

Any small RTOS[1] can be virtualized inside a virtual machine, and a virtual microcontroller environment is provided for it. The time and space of this environment are isolated from those of other virtual machines: a crash of one virtual machine only causes its own restart and does not cause other virtual machines to crash. In this way, the propagation of faults or security vulnerabilities among different components is limited, so that a single point of failure does not cause the entire system to fail.

A virtual machine is a special kind of RME process, and all of its operations are taken over by RVM. Its header also has the same process descriptor as a native process, describing all the entry points in it. However, there are only three fixed entry points in a virtual machine process, namely (1) the interrupt vector handling thread Vct, (2) the user program thread Usr, and (3) the trampoline, and it cannot possess any capabilities other than kernel function calls and the signal sending endpoint. It is not recommended to call any RME system call in a virtual machine other than for signal sending[2]; in principle all functions are taken over by RVM hypercalls.

4.2 Registering Virtual Machines with RVM

In RVM, the configuration of each virtual machine is represented as a structure of type struct RVM_Vmap_Struct. These structures form a constant structure array RVM_Vmap, which internally stores information about all the virtual machines in the system. This information includes the initial execution address, initial stack address, state area address, the memory regions used, and the special kernel calls used by the virtual machine, etc. This information helps RVM load and configure each virtual machine correctly. In addition, RVM also needs an array RVM_Virt composed of another structure type struct RVM_Virt_Struct to store data related to the running state of those virtual machines. The lengths of these two structure arrays must be the same. The user needs to call the function RVM_Virt_Crt during RVM startup to register these virtual machines into the system in batches. Besides filling in the structure information, some kernel objects also need to be created and initialized.

The project generator can perform this process fully automatically, so we recommend using the project generator to create each virtual machine.

4.2.1 Filling in Virtual Machine Information

The user should statically declare the structure array const struct RVM_Vmap_Struct RVM_Vmap[] and fill in the relevant content of each virtual machine. The meaning of each field of this structure is listed in the following table.

[1] such as RMP, FreeRTOS, uC/OS and RT-Thread, etc.
[2] this is for communicating with native processes

Table 4-1 The meaning of each field in struct RVM_Virt_Map

Field name	Purpose
s8_t* Name	The name of the virtual machine. It is a string. It is actually not used by RVM and only serves to facilitate user debugging.
ptr_t Prio	The priority of the virtual machine. This priority must be less than the number of virtual machine priorities configured for RVM.
ptr_t Slice	The running timeslice of the virtual machine. This value must not be set to 0.
ptr_t Period	The period of the virtual machine's clock interrupt. Setting it to X means that RVM sends a virtual clock interrupt to the virtual machine after receiving X ticks. It is recommended to set this value to a number greater than 10.
ptr_t Watchdog	The timeout of the virtual machine watchdog. Setting it to X means that if the virtual machine has run for more than X ticks without feeding the watchdog, the virtual machine is reset. If this item is set to 0, it indicates that the watchdog is not enabled for this virtual machine.
ptr_t Vct_Num	The number of virtual interrupt vectors of the virtual machine; this number cannot exceed 1920. Generally speaking, it is very difficult for any virtual machine to use such a large number of interrupt sources.
struct RVM_Thd_Reg* Reg_Base	The address of the virtual register buffer of the virtual machine. The memory address it points to must be no smaller than the size of the struct RVM_Thd_Reg structure of this architecture. It should be kept consistent with the RVM_VIRT_REG_BASE configuration item registered inside the virtual machine process.
struct RVM_State* State_Base	The address of the state buffer of the virtual machine. The memory address it points to must be no smaller than the size of the struct RVM_State structure. This structure includes the state information of the virtual machine as well as its interrupt vector flags. It should be kept consistent with the RVM_VIRT_STATE_BASE configuration item registered inside the virtual machine process.
ptr_t State_Size	The size of the state buffer of the virtual machine, in bytes.
ptr_t Desc_Base	The base address of the virtual machine description header. It is located at the beginning of the code segment of the virtual machine process.
cid_t Vct_Sig_Cap	The capability number of the virtual machine's interrupt vector endpoint. This capability number is a capability number in RVM. The same applies to the various capability numbers mentioned below.
cid_t Vct_Thd_Cap	The capability number of the virtual machine's interrupt vector handling thread.
ptr_t Vct_Stack_Base	The stack base address of the virtual machine's interrupt vector handling thread.
ptr_t Vct_Stack_Size	The stack size of the virtual machine's interrupt vector handling thread. This size must exceed the stack size of the virtual interrupt vector function that uses the most stack.
cid_t Usr_Thd_Cap	The capability number of the virtual machine's user program handling thread.
ptr_t Usr_Stack_Base	The stack base address of the virtual machine's user program handling thread.

Field name	Purpose
ptr_t Usr_Stack_Size	The stack size of the virtual machine's user program handling thread. For a single-threaded bare-metal program, this size must exceed the stack size used by the user program logic; since the various threads of an RTOS after startup usually switch to dedicated stacks managed by the RTOS itself, this value only needs to exceed the stack size used by the RTOS startup and initialization sequence. If the RTOS kernel uses this stack during operation, then this stack must be designed to be at least sufficient for the RTOS to use.

The project generator can automatically generate this data structure; of course, manual filling is also an option.

4.2.2 Creating and Initializing Virtual Machines

After completing the filling of virtual machine information, we also need to create a signal endpoint for each virtual machine and transfer it to slot 1 of the capability table inside that virtual machine, so that the interrupt vector handling thread Vct can receive interrupts. The transferred capability should possess all operation attributes for receiving and sending, namely RVM_SIG_FLAG_ALL. In addition, the signal endpoint for virtual machine hypercalls must be transferred to slot 0 of the capability table inside that virtual machine, so that the virtual machine can initiate hypercalls. The transferred capability should only allow the send operation attribute, namely RVM_SIG_FLAG_SND.

4.2.3 Batch Registration of Virtual Machines

The user should first statically declare the structure array struct RVM_Virt_Struct RVM_Virt[] to hold runtime data, and then call the function RVM_Virt_Crt at the end of the kernel object initialization hook RVM_Boot_Kobj_Init to register the virtual machines in batches.

Table 4-2 Virtual machine batch registration hook

Function prototype	void RVM_Virt_Crt(struct RVM_Virt_Struct* Virt, const struct RVM_Vmap_Struct* Vmap, rvm_ptr_t Virt_Num)
Return value	None.
Parameters	struct RVM_Virt_Struct* Virt The first address of the statically declared array of virtual machine runtime data structures. const struct RVM_Vmap_Struct* Vmap The first address of the statically declared array of virtual machine information structures. ptr_t Virt_Num The number of virtual machines. The lengths of both statically declared structure arrays above should equal this value.

4.3 Macros That Should Exist Inside a Virtual Machine

In addition to the macros that must exist inside a native process, the following configuration macros also need to exist inside a virtual machine. These macros can be filled in by the project generator or manually by the user. The list of these macros is as follows.

Table 4-3 List of macros that should exist in a virtual machine process

Macro name	Purpose
RVM_VIRT_LIB_ENABLE	The enable macro of the paravirtualization library. Inside a virtual machine it should be defined as 1. Example: #define RVM_VIRT_LIB_ENABLE (1U)
RVM_VIRT_VCT_NUM	The number of virtual interrupt vectors. It should be kept consistent with the configuration item of the same name registered in the RVM virtual machine structure. Example: #define RVM_VIRT_VCT_NUM (10U)
RVM_VIRT_STATE_BASE	The base address of the virtual machine state area. It should be kept consistent with the configuration item of the same name registered in the RVM virtual machine structure. Example: #define RVM_VIRT_STATE_BASE (0x20005700U)
RVM_VIRT_STATE_SIZE	The size of the virtual machine state area, in bytes. It should be kept consistent with the configuration item of the same name registered in the RVM virtual machine structure. Example: #define RVM_VIRT_STATE_SIZE (0x50U)
RVM_VIRT_REG_BASE	The base address of the user thread virtual register set. It should be kept consistent with the configuration item of the same name registered in RVM. Example: #define RVM_VIRT_REG_BASE (0x20005400U)
RVM_VIRT_REG_SIZE	The size of the user thread virtual register set, in bytes. It should be kept consistent with the configuration item of the same name registered in RVM. Example: #define RVM_VIRT_REG_SIZE (0x70U)

4.4 Hypercalls Available to Virtual Machines

Communication between virtual machines and basic management are all implemented through hypercalls. These hypercalls can be called in the user thread or in virtual interrupt service routines.

Table 4-4 List of hypercalls available to virtual machines

Function	Meaning
RVM_Hyp_Putchar	Print a single character.
RVM_Hyp_Reboot	Restart the virtual machine.
RVM_Hyp_Int_Ena	Enable virtual global interrupts.
RVM_Hyp_Int_Dis	Disable virtual global interrupts.
RVM_Hyp_Vct_Phys	Map a physical interrupt source to a virtual interrupt source.
RVM_Hyp_Vct_Evt	Map an event source to a virtual interrupt source.
RVM_Hyp_Vct_Del	Remove the mapping on a virtual interrupt source.
RVM_Hyp_Vct_Lck	Lock the virtual interrupt source mapping.
RVM_Hyp_Vct_Wait	Wait for a virtual interrupt source to be triggered.
RVM_Hyp_Evt_Add	Authorize sending to an event source.
RVM_Hyp_Evt_Del	Revoke the authorization to send to an event source.

Function	Meaning
RVM_Hyp_Evt_Snd	Send to an event source.
RVM_Hyp_Wdg_Clr	Activate and feed the watchdog.

4.4.1 Printing a Single Character

This hypercall prints a single character in the debug output of the virtual machine monitor.

Table 4-5 Printing a single character

Function prototype	void RVM_Hyp_Putchar(char Char)
Return value	None. This call always succeeds.
Parameters	char Char The character to output to the system console.

4.4.2 Restarting the Virtual Machine

This hypercall unconditionally restarts the virtual machine.

Table 4-6 Restarting the virtual machine

Function prototype	void RVM_Hyp_Reboot(void)
Return value	None. This call always succeeds.
Parameters	None.

4.4.3 Enabling Virtual Global Interrupts

This hypercall enables the virtual machine's virtual interrupt reception. After a virtual machine starts, virtual interrupts are disabled by default, and at this time no more interrupts are sent to the virtual machine. This is to simulate the behavior of a microcontroller: the global interrupts of a microcontroller are generally also disabled when it starts. Therefore, an RTOS in a virtual machine needs to call this function after initialization is complete to enable virtual interrupts. This hypercall has no nesting count, so the user needs to perform nesting counting by themselves.

Table 4-7 Enabling virtual global interrupts

Function prototype	void RVM_Hyp_Int_Ena(void)
Return value	None. This call always succeeds.
Parameters	None.

4.4.4 Disabling Virtual Global Interrupts

This hypercall disables virtual interrupts. Once virtual interrupts are disabled, RVM will no longer send any interrupts to this virtual machine, so even transparent interrupts will go unanswered. This hypercall has no nesting count, so the user needs to perform nesting counting by themselves.

Table 4-8 Disabling virtual global interrupts

Function prototype	void RVM_Hyp_Int_Dis(void)
Return value	None. This call always succeeds.
Parameters	None.

4.4.5 Mapping a Physical Interrupt Source to a Virtual Interrupt Source

This hypercall maps a physical interrupt source to a virtual interrupt source of this virtual machine. After the physical interrupt source is triggered, an interrupt is received on the corresponding virtual interrupt source, and the virtual machine calls the handler function registered for that interrupt. Within the same virtual machine, the same physical interrupt source can only be mapped once, and the same virtual interrupt source can only be mapped by one physical interrupt source or event source; however, the same physical interrupt source can be mapped to different virtual machines respectively, in which case once the physical interrupt source is triggered, all the virtual machines receive the virtual interrupt. Each mapping consumes one mapping block; the number of mapping blocks in the system is determined by RVM_VIRT_MAP_NUM at RVM configuration time.

This hypercall is not real-time. Therefore, we only recommend calling it during system startup.

Table 4-9 Mapping a physical interrupt source to a virtual interrupt source

Function prototype	ret_t RVM_Hyp_Vct_Phys(ptr_t Phys_Num, ptr_t Vct_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed physical interrupt vector number or virtual interrupt vector number is out of range.
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked.
	RVM_ERR_PHYS	This physical vector has already been mapped to some virtual interrupt vector of this virtual machine.
	RVM_ERR_VIRT	This virtual interrupt vector has already been mapped to a physical interrupt vector or event source.
	RVM_ERR_MAP	The mapping blocks are exhausted, so mapping cannot be performed.
Parameters	ptr_t Phys_Num	The interrupt vector number of the physical interrupt source.
	ptr_t Vct_Num	The interrupt vector number of the virtual interrupt source. The smaller this number, the higher the non-preemptive priority.

4.4.6 Mapping an Event Source to a Virtual Interrupt Source

This hypercall maps an event source to a virtual interrupt source of this virtual machine. After the event source is triggered, an interrupt is received on the corresponding virtual interrupt source, and the virtual machine calls the handler function registered for that interrupt. Within the same virtual machine, the same event source can only be mapped once, and the same virtual interrupt source can only be mapped by one physical interrupt source or event source; however, the same event source can be mapped to different virtual machines respectively, in which case once the event source is triggered, all the virtual machines receive the virtual interrupt. Each mapping consumes one mapping block; the number of mapping blocks in the system is determined by RVM_VIRT_MAP_NUM at RVM configuration time.

This hypercall is not real-time. Therefore, we only recommend calling it during system startup.

Table 4-10 Mapping an event source to a virtual interrupt source

Function prototype	ret_t RVM_Hyp_Vct_Evt(ptr_t Evt_Num, ptr_t Vct_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed event source number or virtual interrupt vector number is out of range.
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked.
	RVM_ERR_PHYS	This event source has already been mapped to some virtual interrupt vector of this virtual machine.
	RVM_ERR_VIRT	This virtual interrupt vector has already been mapped to a physical interrupt vector or event source.
	RVM_ERR_MAP	The mapping blocks are exhausted, so mapping cannot be performed.
Parameters	ptr_t Evt_Num	The event source number of the event source.
	ptr_t Vct_Num	The interrupt vector number of the virtual interrupt source. The smaller this number, the higher the non-preemptive priority.

4.4.7 Removing the Mapping on a Virtual Interrupt Source

This hypercall removes any mapping on a virtual interrupt source, whether it is a physical interrupt source or an event source.

Table 4-11 Removing the mapping on a virtual interrupt source

Function prototype	ret_t RVM_Hyp_Vct_Del(ptr_t Vct_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed virtual interrupt vector number is out of range.
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked.
	RVM_ERR_VIRT	This virtual interrupt vector has not been registered to any physical interrupt vector or event source.
Parameters	ptr_t Vct_Num	The interrupt vector number of the virtual interrupt source.

4.4.8 Locking the Virtual Interrupt Source Mapping

This hypercall locks the virtual interrupt source mapping of the current virtual machine, which includes the mapping from physical interrupt sources to virtual interrupt sources, the mapping from event sources to virtual interrupt sources, and the virtual machine's event source sending authorization. Once the lock takes effect, it is permanent. Unless the current virtual machine crashes and restarts, no further modification can be made to the virtual interrupt source mapping. This is to prevent certain virtual machines, after being compromised, from mapping a large number of physical interrupt sources or event sources, or sending in large quantities to event sources, thereby causing denial-of-service attacks.

We recommend completing all the above initialization at the startup of each virtual machine, and then using this hypercall to lock them. Since on a microcontroller it is impossible to modify the code segments placed in Flash or loaded into RAM at power-on[3], it can be guaranteed that the execution of the current virtual machine before locking is trustworthy. In this way, the information security of the system can be fully guaranteed after locking. Only for those very complex and highly flexible virtual machines do we not perform locking.

Table 4-12 Locking the virtual interrupt source mapping

Function prototype	ret_t RVM_Hyp_Vct_Lck(void)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked; there is no need to lock it again.
Parameters	None.	

4.4.9 Waiting for a Virtual Interrupt Source to Be Triggered

This hypercall puts the virtual machine into a dormant state and waits until any of its virtual interrupt sources is activated next time before continuing execution. To call this hypercall, the virtual machine's virtual global interrupts must be enabled; otherwise RVM rejects this hypercall[4]. Generally speaking, this hypercall is placed in the RTOS idle thread hook and called repeatedly; its equivalent on a microcontroller is those instructions that may cause entry into a dormant state[5].

Table 4-13 Waiting for a virtual interrupt source to be triggered

Function prototype	ret_t RVM_Hyp_Vct_Wait(void)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_STATE	The virtual global interrupts of this virtual machine are disabled, so it cannot enter dormancy.
Parameters	None.	

[3] even the RME kernel does not have permission to modify them; only the loader (Bootloader) has this permission

[4] if this hypercall were not rejected, the virtual machine could not be woken up once it entered dormancy; on a real processor, disabling interrupts first and then going to sleep likewise causes a crash

[5] such as WFE and WFI of ARMv7-M

4.4.10 Granting Permission to Send to an Event Source

This hypercall authorizes the virtual machine to perform a send operation to an event source. Performing a send operation to any event source requires such authorization.

Table 4-14 Granting permission to send to an event source

Function prototype	ret_t RVM_Hyp_Evt_Add(ptr_t Evt_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed event source number is out of range.
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked.
	RVM_ERR_EVT	This virtual machine already has permission to send to this event source; no authorization is needed.
Parameters	ptr_t Evt_Num	The event source number of the event source.

4.4.11 Revoking the Authorization to Send to an Event Source

This hypercall revokes the virtual machine's authorization to send to an event source. After revocation, the virtual machine can no longer send to that event source.

Table 4-15 Revoking the authorization to send to an event source

Function prototype	ret_t RVM_Hyp_Evt_Del(ptr_t Evt_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed event source number is out of range.
	RVM_ERR_STATE	The virtual interrupt source mapping of this virtual machine has already been locked.
	RVM_ERR_EVT	This virtual machine never had permission to send to this event source; no revocation is needed.
Parameters	ptr_t Evt_Num	The event source number of the event source.

4.4.12 Sending to an Event Source

This hypercall causes the virtual machine to send to an event source. This send activates the interrupt vectors of all virtual machines registered on that event source.

Table 4-16 Sending to an event source

Function prototype	ret_t RVM_Hyp_Evt_Snd(ptr_t Evt_Num)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed event source number is out of range.
	RVM_ERR_EVT	This virtual machine does not have permission to send to this event source.
Parameters	ptr_t Evt_Num	The number of the event source to activate.

4.4.13 Activating and Feeding the Watchdog

This hypercall is available when the Watchdog item in the virtual machine information is not 0. When a virtual machine is created, the watchdog is disabled by default; when this hypercall is called for the first

time, the watchdog starts and begins counting down to account for the virtual machine's running time[6], and once the countdown reaches 0 the virtual machine is restarted. This hypercall can also be used to feed the watchdog. Once enabled, the watchdog cannot be stopped, unless the virtual machine restarts due to a fault or a watchdog timeout.

Table 4-17 Activating and feeding the watchdog

Function prototype	ret_t RVM_Hyp_Wdg_Clr(void)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_STATE	The ptr_t Watchdog item is 0, so this virtual machine cannot enable the watchdog.
Parameters	None.	

4.5 Internal Utility Functions Available to Virtual Machines

In addition to hypercalls, virtual machines have some internal utility functions they can use. These internal utility functions are not hypercalls; they achieve their effects merely by adjusting the data structures inside the virtual machine, and are therefore much more efficient than hypercalls.

Table 4-18 List of internal utility functions available to virtual machines

Function	Meaning
RVM_Virt_Init	Virtual machine data structure initialization.
RVM_Vct_Loop	Run the virtual interrupt vector handling loop.
RVM_Virt_Int_Mask	Mask virtual interrupt vector handling.
RVM_Virt_Int_Unmask	Unmask virtual interrupt vector handling.
RVM_Virt_Vct_Reg	Register an ordinary virtual interrupt vector handler function.
RVM_Virt_Tim_Reg	Register a clock virtual interrupt vector handler function
RVM_Virt_Ctx_Reg	Register a thread switch virtual interrupt vector handler function.
RVM_Virt_Yield	Trigger the virtual thread switch interrupt vector.

4.5.1 Virtual Machine Data Structure Initialization

This operation initializes the internal data structures of the virtual machine. This operation should be called first by the interrupt handling thread Vct when the virtual machine starts. In cases where the project generator is used, the project generator calls this function in the generated Vct thread.

Table 4-19 Virtual machine data structure initialization

Function prototype	void RVM_Virt_Init(void)
Return value	None. This operation always succeeds.
Parameters	None.

[6] the watchdog only counts the running time of the virtual machine itself, not the running time of the real system

4.5.2 Running the Virtual Interrupt Vector Handling Loop

This operation processes the interrupt vectors inside the virtual machine in a loop and never returns once called. This operation should be called last by the interrupt handling thread Vct. In cases where the project generator is used, the project generator calls this function in the generated Vct thread.

Table 4-20 Running the virtual interrupt vector handling loop

Function prototype	void RVM_Vct_Loop(void)
Return value	None. This operation never returns.
Parameters	None.

4.5.3 Masking Virtual Interrupt Vector Handling

This operation masks the handling of interrupt vectors inside the virtual machine. The virtual machine still receives interrupts, so the user program thread Usr will still be preempted by the interrupt handling thread Vct, but the received interrupts are all pended and processed together after masking ends. Compared with the RVM_Hyp_Int_Dis hypercall that disables virtual global interrupts, this operation does not need to invoke the virtual machine monitor and is therefore much more efficient, making it very suitable for situations where the RTOS locks its scheduler. This operation has no nesting count, so the guest RTOS needs to perform nesting counting by itself.

Table 4-21 Masking virtual interrupt vector handling

Function prototype	void RVM_Virt_Int_Mask(void)
Return value	None. This operation always succeeds.
Parameters	None.

4.5.4 Unmasking Virtual Interrupt Vector Handling

This operation removes the masking of interrupt vector handling inside the virtual machine and is the inverse operation of RVM_Virt_Int_Mask. After unmasking, if it is found that virtual interrupts were pended during interrupt masking[7], the Vct thread is immediately triggered to run to handle them. This operation has no nesting count, so the guest RTOS needs to perform nesting counting by itself. This operation does not mask transparent interrupts that are set to RME_VCT_TRANSPARENT.

Table 4-22 Unmasking virtual interrupt vector handling

Function prototype	void RVM_Virt_Int_Unmask(void)
Return value	None. This operation always succeeds.
Parameters	None.

[7] triggered but not yet handled

4.5.5 Registering an Ordinary Virtual Interrupt Vector

This operation registers a handler function for an ordinary virtual interrupt vector. This handler function will be called when the virtual interrupt vector is activated. When a virtual machine starts, all interrupt vectors are unregistered by default; if a virtual interrupt with no registered interrupt vector handler function is activated, that interrupt is simply skipped.

Table 4-23 Registering an ordinary virtual interrupt vector

Function prototype	ret_t RVM_Virt_Vct_Reg(ptr_t Vct_Num, void (*Vct)(void), ptr_t Vct_Attr)	
Return value	ret_t	If successful, returns 0. If it fails, the following negative values are returned:
	RVM_ERR_RANGE	The passed interrupt vector number of the virtual interrupt source is out of range.
Parameters		
ptr_t Vct_Num		The interrupt vector number of the virtual interrupt source.
void (*Vct)(void)		A function pointer pointing to the interrupt vector handler function. This function is recommended to be of type void (*)(void); if it is of another type, the values of its parameters are undefined and its return value is discarded directly. If 0[8] is passed for this item, it means removing the registration of this interrupt vector.
ptr_t Vct_Attr		The attribute of the virtual interrupt source. Currently it only includes its masking attribute: when it is RME_VCT_MASKABLE, this interrupt is a perceptible interrupt to the guest operating system, and it is masked by RVM_Virt_Int_Mask, but the interrupt IPC functions of the RTOS guest can be called inside it[9]; when it is RME_VCT_TRANSPARENT, this interrupt is a transparent interrupt to the guest operating system, and it is not masked by RVM_Virt_Int_Mask, but the interrupt IPC functions of the RTOS guest cannot be called inside it. If better interrupt real-time performance is required and interrupt IPC does not need to be called, the latter can be chosen.

4.5.6 Registering the Clock Virtual Interrupt Vector

This operation registers the handler function for the clock virtual interrupt vector. Whenever a clock interrupt occurs, this handler function is called. The guest RTOS should register its clock virtual interrupt vector through this operation in its initialization sequence. The clock virtual interrupt vector is always a maskable perceptible interrupt.

[8] a null pointer, i.e., NULL
[9] such as the FromISR series of FreeRTOS, the _ISR series of RMP, etc.

Table 4-24 Registering the clock virtual interrupt vector

Function prototype	void RVM_Virt_Tim_Reg(void (*Tim)(void))
Return value	None. This operation always succeeds.
Parameters	void (*Tim)(void) A function pointer pointing to the clock interrupt vector handler function. This function is recommended to be of type void (*)(void); if it is of another type, the values of its parameters are undefined and its return value is discarded directly. If 0[10] is passed for this item, it means removing the registration of the clock interrupt vector.

4.5.7 Registering the Thread Switch Virtual Interrupt Vector

This operation registers the handler function for the thread switch virtual interrupt vector. Whenever a thread switch interrupt occurs, this handler function is called. The guest RTOS should register its thread switch virtual interrupt vector through this operation in its initialization sequence. The thread switch virtual interrupt vector is always a maskable perceptible interrupt.

Table 4-25 Registering the thread switch virtual interrupt vector

Function prototype	void void RVM_Virt_Ctx_Reg(void (*Ctx)(void))
Return value	None. This operation always succeeds.
Parameters	void (*Ctx)(void) A function pointer pointing to the thread switch interrupt vector handler function. This function is recommended to be of type void (*)(void); if it is of another type, the values of its parameters are undefined and its return value is discarded directly. If 0[11] is passed for this item, it means removing the registration of the thread switch interrupt vector.

4.5.8 Triggering the Thread Switch Virtual Interrupt Vector

This operation directly triggers the thread switch interrupt vector of the virtual machine. If the virtual machine's global interrupts are disabled or interrupt vector handling is masked, the thread switch interrupt vector will be pended and will execute immediately after the global interrupts are enabled or the masking is removed.

Table 4-26 Triggering the thread switch virtual interrupt vector

Function prototype	void RVM_Virt_Yield(void)
Return value	None. This operation always succeeds.
Parameters	None.

4.6 References for This Chapter

None

[10] a null pointer, i.e., NULL
[11] a null pointer, i.e., NULL

5. Project Generator

5.1 Introduction

The project generator is responsible for generating the required boot files, linker scripts and build system from the project description file, sparing the tedium of manually configuring the system. Internally, it is divided into three parts: the input analysis module, the project generation module and the project output module. It can be invoked through a command-line interface or through a graphical frontend.

The project generator does not support non-ASCII paths[1], and reports an error when such a case is detected. Supporting non-ASCII paths is meaningless, because a considerable number of development tools themselves do not support non-ASCII paths; in that case, even if the project is correctly generated in a non-ASCII path, there is no hope of compiling it successfully.

5.1.1 Input Analysis Module

The input analysis module is responsible for analyzing the various description files and packaging them into an internal data structure passed to the project generation module. It performs the following work:

1. Analyze the project description file (*.rvp),
2. Read the appropriate platform description file (*.rva) according to the project description file,
3. Read the appropriate chip description file (*.rvc) according to the project description file,
4. Check the validity of each description file, and
5. Load the generation tools related to the architecture platform, compilation toolchain, build system and guest operating system corresponding to the project description file.

5.1.2 Project Generation Module

The project generation module is responsible for reading the project information, platform information and chip information provided by the input analysis module, generating the various data required by the project, and handing these data to the project output module to generate the actual project. It performs the following work:

1. Align the memory blocks declared in the project,
2. Allocate actual base addresses for the memory blocks declared in the project,
3. Add shared memory to the memory layout of each process,
4. Allocate page tables for each process,
5. Allocate capability numbers for each kernel object,
6. Determine the capability hosting (authorization) layout of the kernel objects, and
7. Allocate kernel memory for each kernel object.

[1] such as paths containing Chinese, Japanese, French, and so on

5.1.3 Project Output Module

The project output module is responsible for reading the output of the project generation module and generating the actual project files. It performs the following work:

1. Generate the kernel configuration header files (rme_platform_conf.h and rme_platform_xxx.h),
2. Generate the kernel boot header file (rme_boot.h),
3. Generate the kernel boot source file (rme_boot.c),
4. Generate the kernel hook source file (rme_hook.c),
5. Generate the kernel interrupt vector source file (rme_handler_xxx.c),
6. Generate the kernel linker script (rme_platform_xxx.ld etc.),
7. Generate the kernel build project file (makefile etc.),
8. Generate the virtual machine monitor configuration header files (rvm_platform_conf.h and rvm_platform_xxx.h),
9. Generate the virtual machine monitor boot header file (rvm_boot.h),
10. Generate the virtual machine monitor boot source file (rvm_boot.c),
11. Generate the virtual machine monitor hook source file (rvm_hook.c),
12. Generate the virtual machine monitor linker script (rvm_platform_xxx.ld etc.),
13. Generate the virtual machine monitor build project file (makefile etc.),
14. Generate the main header file of each process (prc_xxx.h),
15. Generate the entry point source files of each process (prc_xxx_thd_xxx.c and prc_xxx_inv_xxx.c),
16. Generate the descriptor source file of each process (prc_xxx_desc.c),
17. Generate the main source file of each process (prc_xxx.c),
18. If the process is a virtual machine, generate the virtual machine configuration header file (varies with the guest operating system),
19. If the process is a virtual machine, generate the virtual machine configuration source file (varies with the guest operating system),
20. Generate the linker script of each process (prc_xxx.ld etc.),
21. Generate the build project file of each process (makefile etc.), and
22. Generate the workspace containing the above project files (if applicable).

5.1.4 Graphical Frontend Module

Since the internal structure of the project description file is also rather cumbersome, manual editing easily introduces errors, so the project generator also provides a graphical frontend for editing projects. This graphical frontend makes project editing more intuitive and convenient.

5.2 Command-Line Parameter Description

The command-line parameter of the project generator consists of only one item, -i, which specifies the path of the project description file. The project description file has the suffix *.rvp and internally contains all other information required to generate the project.

For example: -i path/to/input.rvp

5.3 Project Description File Structure<Project>

The project main file records all information about the project, and its suffix is *.rvp; all of its internal tags are located under one large <Project> tag.

All tags, as long as they are described below, must not be omitted regardless of whether they have content.

5.3.1 Project Name<Name>

This configuration item records the name of the project. Generally, it is consistent with the name of the project file.

The project generator does not care about the value of this configuration item.

5.3.2 Project Version<Version>

This configuration item records the version of the project.

The project generator does not care about the value of this configuration item.

5.3.3 Enable Assertions<Assert_Enable>

This configuration item determines whether the assertions in the code are checked, and can be “Yes” or “No”. If it is assumed that all assertions are correct and no check is performed, “No” can be chosen; this slightly improves the running speed and reduces the code size.

5.3.4 Enable Debug Printing<Debug_Log_Enable>

This configuration item determines whether the debug print macros beginning with RVM_DBG_ are enabled, and can be “Yes” or “No”. If debug print output is not needed, “No” can be chosen; this slightly improves the running speed and reduces the code size.

5.3.5 Enable Fixed Raw Page Tables<Pgtbl_Raw_Enable>

This configuration item determines whether the fixed user-mode MPU metadata is enabled, and can be “Yes” or “No”. Once enabled, the MPU metadata of each process, which contains all of its address space information, will be generated all at once in advance and loaded to the kernel when the process is created; the system calls involving page tables and the dynamic page function in the kernel will no longer be compiled, and the page table kernel object will no longer be created at startup. This is very effective in scenarios where space is tight and the additional flexibility provided by dynamic pages is not needed.

5.3.6 Workspace Build System<Buildsystem>

This configuration item records the build system used by the workspace. Some build tools[2] allow multiple projects to be combined into one workspace; here its category is selected. Of course, “None” can also be selected, in which case no workspace is output.

5.3.7 Workspace Output Directory<Workspace_Output>

This configuration item is a relative path that records the output directory of the workspace.

[2] such as Keil uVision, which allows a *.uvmpw workspace to contain multiple *.uvprojx project files

5.3.8 Workspace Output Overwrite<Workspace_Overwrite>

This configuration item specifies whether the workspace file will be overwritten when the project is re-generated, and can be “Yes” or “No”. Sometimes we have already made certain manual modifications to the workspace file and do not want it to be overwritten; in this case “No” can be chosen.

5.3.9 Chip Configuration<Chip>

This configuration group is responsible for the selection of the architecture platform and the chip model. The project compiler will extract information from this configuration item to determine the platform description file (*.rva) and the chip description file (*.rvc) to be read.

5.3.9.1 Target Platform<Platform>

This configuration item records the architecture platform of the project. Each platform is represented by a code, for example A7M corresponds to the ARMv7-M platform. Each platform corresponds to one platform description file, and they are located at Include/Platform/XXX/rvm_platform_xxx.rva. For the structure of these description files, please see 5.4.

5.3.9.2 Chip Class<Class>

This configuration item records the specific chip class. The class only determines the internal computing resource configuration of the chip and does not specify a specific package; for example, STM32F767IG is compatible with STM32F767IGT6 and STM32F767IGK6. Each chip class corresponds to one chip description file, and they are located at Include/Platform/XXX/Chip/rvm_platform_xxx.rvc. For the structure of these description files, please see 5.5.

5.3.9.3 Chip Name<Name>

This configuration item records the exact full name of the chip, such as STM32F767IGT6. The project generator will check whether this full name is among the compatible chips specified by the chip class. If not, the project generator will report an error.

5.3.9.4 Enable Coprocessors<Coprocessor>

This configuration item is a comma-separated list that records the coprocessors that need to be enabled in the project, such as the FPU; only the coprocessors enabled here will be initialized by the kernel. The project generator will check whether the coprocessor exists in the chip. If not, the project generator will report an error.

5.3.9.5 Initialization Configuration<Config>

The initialization configuration group includes the initialization parameters of the architecture and the initialization parameters of the chip; these parameters are all stored in the form <tag> value</tag>. These parameters vary with the architecture and the chip; all the parameters declared in the platform description file and the chip description file must be filled in here, no more and no less.

5.3.10 Extra Memory<Extmem><En>

This configuration group contains the attributes of the extra memory blocks to be added into the system. Extra memory refers to the external memory that the microcontroller can access, such as externally expanded SDRAM, SRAM, and so on. If the project generator is to use them, then their information must be registered inside this group. In this case, the total available memory is the sum of the chip’s built-in memory[3] and the extra memory listed here, and the shared memory blocks[4] and the process-private

[3] please see 5.5.12
[4] please see 5.3.11

memory blocks will be allocated from the total available memory. In addition, appropriate external memory initialization code[5] must be written in the kernel startup hook, to ensure that these memories can be accessed correctly.

5.3.10.1 Extra Segment Base Address<Base>

This configuration item is a hexadecimal value that determines the start address of this external memory segment.

5.3.10.2 Extra Segment Size<Size>

This configuration item is a hexadecimal value that determines the length of this external memory segment.

5.3.10.3 Extra Segment Type<Type>

This configuration item determines the type of the memory segment. There are three types to choose from: code (Code), data (Data), and device (Device). When allocating memory, the project generator will use this type to determine what purpose the segment of memory is used for.

5.3.10.4 Extra Segment Attributes<Attribute>

This configuration item determines the attributes of the memory segment. The memory segment attributes include six kinds: readable (Readable, R), writable (Writable, W), executable (eXecutable, X), bufferable (Bufferable, B), cacheable (Cacheable, C), and static mapping (Static, S), which can be superposed in any combination. When allocating memory, the project generator will only place memory segments whose attributes are a subset of this attribute onto this physical memory segment. For example, if the user specifies that the attributes of an extra memory segment do not include writable[6], then the project generator will never allocate writable content on this memory segment.

Generally, the attributes of a code segment ROM should be set to RXBCS, and the attributes of a data segment RAM should be set to RWBCS. The attributes of a device segment should be set to RWS.

5.3.11 Shared Memory Block<Shmem><Sn>

This configuration group contains the attributes of the shared memory blocks present in the system. A shared memory block refers to a memory region that can be accessed by multiple processes or virtual machines, such as a data buffer, and so on; it is similar to shared memory in systems such as Linux, and can be used for controlled information transfer between multiple processes or virtual machines. The shared memory blocks declared in this configuration group will be allocated from the total available memory[7], and then added into the page table of each process or virtual machine that references it.

5.3.11.1 Shared Segment Name<Name>

This configuration item is a required field, requiring the user to enter the name of this shared memory segment. The name must be a valid identifier and must be unique.

5.3.11.2 Shared Segment Base Address<Base>

This configuration item is an optional hexadecimal value. The user should fill in the start address of this shared memory segment. If “Auto” is filled in, it means that the location of this shared memory segment is automatically allocated by the compiler.

[5] such as the FSMC, FMC, or EMC possessed by microcontrollers such as STM32, GD32, HC32, etc.
[6] such as an external XIP SPI Flash
[7] please see 5.3.10

5.3.11.3 Shared Segment Size<Size>

This configuration item is a required hexadecimal value. The user should fill in the length of this shared memory segment.

5.3.11.4 Shared Segment Alignment<Align>

This configuration item is a required decimal value that determines the alignment of this memory.

When the segment base address is "Auto", if N is filled in, then the segment of memory is aligned to 2^N during automatic allocation; if "Auto" is filled in, it means that the alignment of this shared memory segment is determined by the compiler at its own discretion.

When the segment base address is a fixed value, this configuration item must be filled in as "Auto".

For the same segment of memory, the alignments supported by different architectures may differ.

5.3.11.5 Shared Segment Type<Type>

This configuration item determines the type of the memory segment. There are three types to choose from: code (Code), data (Data), and device (Device). When allocating memory, the compiler will use this type to determine which kind of memory this shared memory segment is allocated to.

5.3.11.6 Shared Segment Attributes<Attribute>

This configuration item determines the attributes of the memory segment. The memory segment attributes include six kinds: readable (Readable, R), writable (Writable, W), executable (eXecutable, X), bufferable (Bufferable, B), cacheable (Cacheable, C), and static mapping (Static, S), which can be superposed in any combination. When checking the references to the shared memory declared in each process or virtual machine, the compiler will only allow a subset of this attribute to appear. For example, if the user specifies that the attributes of this shared memory segment do not include writable, then the compiler will check and ensure that the subsequent references to the shared memory do not include the write attribute.

5.3.12 Kernel Configuration<Kernel>

This configuration group is responsible for the basic configuration of the RME kernel, including the memory used by the kernel, the priorities supported by the kernel, and the storage locations of the various files included in the kernel project, and so on.

5.3.12.1 Code Segment Base Address<Code_Base>

This configuration item determines the base address of the code segment used by the kernel.

5.3.12.2 Code Segment Size<Code_Size>

This configuration item determines the length of the code segment used by the kernel. The code segment of the kernel includes the assembly code of the kernel and its read-only data area, and is usually a fixed value on a given architecture.

5.3.12.3 Data Segment Base Address<Data_Base>

This configuration item determines the base address of the data segment used by the kernel.

5.3.12.4 Data Segment Size<Data_Size>

This configuration item determines the length of the data segment used by the kernel. The data segment of the kernel includes the kernel's read-write data area, the memory area for kernel objects, and the kernel stack area, and usually needs to be adjusted according to the number of kernel objects created. If the generator reports that the kernel data memory is exhausted and kernel objects cannot be allocated, then you can try to modify this value to be somewhat larger.

5.3.12.5 Kernel Stack Size<Stack_Size>

This configuration item determines the length of the kernel stack segment used by the kernel. As a microkernel running on a microcontroller, a 1kB kernel stack is sufficient for most purposes.

5.3.12.6 Extra Kernel Object Memory Size<Extra_Kom>

This configuration item determines, after the system has created all the kernel objects related to the generator-generated code, the length of the extra kernel memory that additionally needs to be reserved for the user to create kernel objects. It only needs to be set to a non-zero number when the user wants to manually add kernel object creation code in the generator-generated RVM startup code. Generally, we do not recommend that the user manually create kernel objects, because this means that the user may compromise the information security of the system.

5.3.12.7 Kernel Memory Allocation Granularity<Kom_Order>

This configuration item determines the kernel memory allocation granularity of the system, which is a power of 2. It is recommended to choose the most suitable one among 8 bytes, 16 bytes, and 32 bytes: fill in 3 for 8 bytes, 4 for 16 bytes, and 5 for 32 bytes. When creating kernel objects, the system will align all kernel memory to this granularity. If the system itself has relatively little memory, it is recommended to use 8-byte aligned allocation to reduce fragmentation; if the system itself has more memory, then 32-byte allocation can be adopted to reduce the size of the memory allocation registry itself. By default, setting it to 16-byte allocation is sufficient.

5.3.12.8 Number of Priorities<Kern_Prio>

This configuration item determines the number of kernel-mode priorities in the system, and should be filled with a decimal value that is an integer multiple of the processor word length. Generally, filling in the processor word length is sufficient; for example, fill in 32 for a 32-bit processor.

5.3.12.9 Build System<Buildsystem>

Determines which build system the kernel project adopts.

5.3.12.10 Build Toolchain<Toolchain>

Determines which build toolchain the kernel project adopts.

5.3.12.11 Optimization Level<Optimization>

Determines the optimization level used by the kernel project. The available optimization levels are no optimization (O0), basic optimization (O1), full optimization (O2), extreme optimization (O3), speed optimization (Of), and size optimization (Os).

5.3.12.12 Kernel Root Directory<Kernel_Root>

Determines the location of the RME kernel source root folder, which is a path relative to the project description file. The generated kernel project will reference the RME kernel files stored here.

5.3.12.13 Project Output Directory<Project_Output>

Determines the output directory of the kernel project, which is a path relative to the project description file.

5.3.12.14 Project Overwrite<Project_Overwrite>

Determines whether to overwrite the previously generated kernel project file[8] when regenerating, and can be “Yes” or “No”. If the user has manually edited the project and wishes to keep these changes, “No” can

[8] such as Keil uVision’s *.uvprojx, etc.

be chosen to keep the original old version.

5.3.12.15 Linker Script Output Directory<Linker_Output>

Determines the output directory of the kernel project linker script rme_platform_xxx.ld[9], which is a path relative to the project description file. The kernel linker script is always overwritten in each generation.

5.3.12.16 Configuration Header File Output Directory<Config_Header_Output>

Determines the output directory of the kernel configuration header files rme_platform_conf.h and rme_platform_xxx.h, which is a path relative to the project description file. The kernel configuration header files are always overwritten in each generation.

5.3.12.17 Boot Header File Output Directory<Boot_Header_Output>

Determines the output directory of the kernel boot header file rme_boot.h, which is a path relative to the project description file. The kernel boot header file is always overwritten in each generation.

5.3.12.18 Boot Source File Output Directory<Boot_Source_Output>

Determines the output directory of the kernel boot source file rme_boot.c, which is a path relative to the project description file. It is always overwritten in each generation. In this source file, the creation of the kernel interrupt vector signal endpoints is completed.

5.3.12.19 Hook Source File Output Directory<Hook_Source_Output>

Determines the output directory of the kernel hook source file rme_hook.c, which is a path relative to the project description file. This file internally contains four hook functions, which can be used to insert some operations into the kernel initialization process. Among them:

1. The RME_Boot_Pre_Init hook is called before the kernel initializes the data structures, and the user can complete in it the software and hardware initialization work that must be completed before the kernel starts.
2. The RME_Boot_Post_Init hook is called before the kernel enters user-mode execution, and the user can complete in it the customized creation of additional kernel objects. The number of extra entries reserved in the boot capability table[10] is determined by <Monitor><Extra_Captbl>, and the reserved extra kernel memory is determined by <Kernel><Extra_Kom>. Among them, the available capability table entries start from RME_RVM_CAP_DONE_FRONT up to RME_RVM_INIT_CPT_SIZE-1, and the available kernel memory starts from RME_RVM_KOM_DONE_FRONT up to RME_KOM_VA_SIZE-1. It should be noted that these extra reserved resources are shared with RVM's startup hooks[11]; if you choose to use these resources in RME's hooks, they can no longer be used again in RVM's hooks. Of course, it is also possible to use part of the resources to create kernel objects at RME startup, and leave the other part of the resources for creating kernel objects at RVM startup; in this case, the division of the two parts of resources needs to be done manually by the user.
3. The RME_Reboot_Failsafe hook is called when a serious failure[12] occurs in the kernel, and the user can complete the corresponding safety handling work in it.
4. The RME_Hook_Kfn_Handler hook is called when the kernel encounters an undefined kernel function call, and the user can customize additional kernel function calls in it.

[9] the suffix here may also differ depending on the selected toolchain
[10] that is, the capability table used by the RVM virtual machine monitor process; it is the first capability table in the system
[11] the values of the corresponding macros are also exactly the same; for these macros please see 8.2.2
[12] such as an assertion check failure

5.3.12.20 Hook Source File Overwrite<Hook_Source_Overwrite>

Determines whether to overwrite the previously generated kernel hook source file when regenerating, and can be "Yes" or "No". If the user has manually edited the hook source file and wishes to keep the changes, "No" can be chosen to keep the original old version.

5.3.12.21 Interrupt Vector Source File Output Directory<Handler_Source_Output>

Determines the output directory of the kernel interrupt vector top-half source files rme_handler_xxx.c, which is a path relative to the project description file. Each physical interrupt vector corresponds to an independent source file; the user can edit the RME_Vct_XXX_Handler function in it to do some work when entering the physical interrupt, such as implementing the top-half processing of the interrupt[13]. This function allows the user to return a return value, which determines how signals are sent after the top half of the interrupt ends: (1) if RME_RVM_VCT_SIG_ALL is returned, signals are sent to both its own dedicated endpoint and the RME_BOOT_INIT_VCT endpoint; (2) if RME_RVM_VCT_SIG_SELF is returned, a signal is sent only to its own dedicated endpoint; (3) if RME_RVM_VCT_SIG_INIT is returned, a signal is sent only to the RME_BOOT_INIT_VCT endpoint; (4) if RME_RVM_VCT_SIG_NONE is returned, no signal is sent to any endpoint.

5.3.12.22 Interrupt Vector Source File Overwrite<Handler_Source_Overwrite>

Determines whether to overwrite the previously generated interrupt vector source files when regenerating. If the user has manually edited the interrupt vector source files and wishes to keep the changes, "No" can be chosen to keep the original old version.

5.3.13 Monitor Configuration<Monitor>

This configuration group is responsible for the basic configuration of the RVM virtual machine monitor, including the memory used by RVM, the number of virtual priorities supported, the number of event sources supported and the number of virtual interrupt mappings, as well as the storage locations of the various files included in the RVM project, and so on.

5.3.13.1 Code Segment Size<Code_Size>

This configuration item determines the length of the code segment used by RVM. The code segment of RVM includes its assembly code and its read-only data area, and is usually a fixed value on a given architecture. It should be noted that the RVM code start address is always located directly after the kernel code segment, so the code memory base address is determined automatically and does not need to be set manually by the user.

5.3.13.2 Data Segment Size<Data_Size>

This configuration item determines the length of the data segment used by RVM. The data segment of RVM includes its read-write data area and stack area, and is usually determined according to whether the virtual machine function is enabled and the number of virtual machines. If a memory shortage is reported when compiling RVM, then you can try to modify this value to be somewhat larger. It should be noted that the data segment of RVM is always located directly after the kernel data segment, so the data segment base address is determined automatically and does not need to be set manually by the user.

5.3.13.3 Startup Daemon Thread Stack Size<Init_Stack_Size>

This configuration item determines the stack length of the startup daemon thread INIT. Generally, 256B is sufficient.

[13] the top-half-bottom-half dichotomy is a very common interrupt handling method: it puts the most critical parts of interrupt handling in the kernel for execution, and passes the less critical parts to user mode for execution.

5.3.13.4 Safety Daemon Thread Stack Size<Sftd_Stack_Size>

This configuration item determines the stack length of the safety daemon thread SFTD. Generally, 256B is sufficient.

5.3.13.5 Virtual Machine Daemon Thread Stack Size<Vmmd_Stack_Size>

This configuration item determines the stack length of the virtual machine daemon thread VMMD. Generally, 512B is sufficient. This thread will also be used for the cooperative operation of the hypercall daemon coroutine HYPD, the vector daemon coroutine VCTD, and the time daemon coroutine TIMD.

5.3.13.6 Extra Capability Table Slot Count<Extra_Captbl>

This configuration item determines, after the system has created all the kernel objects related to the generator-generated code, the number of extra capability table slots that additionally need to be reserved for the user to create kernel objects. It only needs to be set to a non-zero number when the user wants to manually add kernel object creation code in the generator-generated RVM startup code. Generally, we do not recommend that the user manually create kernel objects, because this means that the user may compromise the information security of the system.

5.3.13.7 Enable Idle Sleep<Idle_Sleep_Enable>

This configuration item determines whether to put the processor into the sleep state when there are no native processes and virtual machines available to run, and can be "Yes" or "No". If you do not wish to enter the low-power sleep state in order to minimize the interrupt latency, "No" can be chosen.

5.3.13.8 Number of Virtualization Priorities<Virt_Prio>

This configuration item determines the number of priorities supported by the virtual machine monitor. This priority count is only effective for virtual machines, and must be an integer multiple of the processor word length. Generally, filling in the processor word length is sufficient; for example, filling in 32 for a 32-bit processor is enough. If virtual machines are not used, this can be filled in as 0.

5.3.13.9 Number of Virtual Machine Event Sources<Virt_Event>

This configuration item determines the number of event sources supported by the virtual machine monitor; it must be an integer multiple of the processor word length, and cannot be 0 when virtual machines are used. At the same time, the number of event sources cannot exceed the hard limit of 1024. Event sources are used for communication from native processes to virtual machines and among the virtual machines themselves, and they can be mapped to the virtual interrupt vectors of virtual machines.

5.3.13.10 Number of Virtual Machine Vector Mappings<Virt_Map>

This configuration item determines the total number of mappings from event sources or physical interrupt sources to virtual interrupt vectors; it must be greater than the number of event sources, and cannot be 0 when virtual machines are used. One event source or physical interrupt source can be mapped to multiple virtual vectors, and each mapping from a specific source to a specific virtual vector consumes one mapping count.

5.3.13.11 Build System<Buildsystem>

Determines which build system the virtual machine monitor project adopts.

5.3.13.12 Build Toolchain<Toolchain>

Determines which build toolchain the virtual machine monitor project adopts.

5.3.13.13 Optimization Level<Optimization>

Determines the optimization level used by the virtual machine monitor project. The available optimization levels are no optimization (O0), basic optimization (O1), full optimization (O2), extreme optimization (O3), speed optimization (Of), and size optimization (Os).

5.3.13.14 Virtual Machine Monitor Root Directory<Monitor_Root>

Determines the location of the RVM virtual machine monitor source root folder, which is a path relative to the project description file. The generated virtual machine monitor project will reference the virtual machine monitor source files here by relative path.

5.3.13.15 Project Output Directory<Project_Output>

Determines the output directory of the RVM project, which is a path relative to the project description file.

5.3.13.16 Project Overwrite<Project_Overwrite>

Determines whether to overwrite the previously generated virtual machine monitor project file[14] when regenerating, and can be "Yes" or "No". If the user has manually edited the project and wishes to keep these changes, "No" can be chosen to keep the original old version.

5.3.13.17 Linker Script Output Directory<Linker_Output>

Determines the output directory of the virtual machine monitor project linker script rvm_platform_xxx.ld[15], which is a path relative to the project description file. The linker script is always overwritten in each generation.

5.3.13.18 Configuration Header File Output Directory<Config_Header_Output>

Determines the output directory of the virtual machine monitor configuration header files rvm_platform_conf.h and rvm_platform_xxx.h, which is a path relative to the project description file. The configuration header files are always overwritten in each generation.

5.3.13.19 Boot Header File Output Directory<Boot_Header_Output>

Determines the output directory of the virtual machine monitor boot header file rvm_boot.h, which is a path relative to the project description file. The boot header file is always overwritten in each generation.

5.3.13.20 Boot Source File Output Directory<Boot_Source_Output>

Determines the output directory of the virtual machine monitor boot source file rvm_boot.c, which is a path relative to the project description file. The boot source file is always overwritten in each generation.

5.3.13.21 Hook Source File Output Directory<Hook_Source_Output>

Determines the output directory of the virtual machine monitor hook source file rvm_hook.c, which is a path relative to the project description file. This file is user-editable, and can be used to do some hardware initialization operations before and after the virtual machine monitor starts. Among them:

1. The RVM_Boot_Pre_Init hook is called before the virtual machine monitor initializes the data structures, and the user can complete in it the software and hardware initialization work that must be completed before the virtual machine monitor starts.

[14] such as Keil uVision's *.uvprojx, etc.
[15] the suffix here may also differ depending on the selected toolchain

2. The RVM_Boot_Post_Init hook is called after the initialization of the virtual machine monitor ends, and the user can complete in it the customized creation of additional kernel objects. The number of extra entries reserved in the boot capability table[16] is determined by <Monitor><Extra_Captbl>, and the reserved extra kernel memory is determined by <Kernel><Extra_Kom>. Among them, the available capability table entries start from RVM_CAP_DONE_FRONT up to RVM_BOOT_CPT_SIZE-1, and the available kernel memory starts from RVM_KOM_DONE_FRONT up to RVM_KOM_VA_SIZE-1. It should be noted that these extra reserved resources are shared with RME's startup hooks[17]; if you choose to use these resources in RME's hooks, they can no longer be used again in RVM's hooks. Of course, it is also possible to use part of the resources to create kernel objects at RME startup, and leave the other part of the resources for creating kernel objects at RVM startup; in this case, the division of the two parts of resources needs to be done manually by the user.

5.3.13.22 Hook Source File Overwrite<Hook_Source_Overwrite>

Determines whether to overwrite the previously generated virtual machine monitor hook source file when regenerating, and can be "Yes" or "No". If the user has manually edited the hook source file and wishes to keep the changes, "No" can be chosen to keep the original old version.

5.3.14 Native Process Configuration<Process><Pn>

This configuration group is responsible for the configuration of the native processes in the system. The system can have one or more native processes, so this class can have one or more instances. These instances are all displayed in the second column on the left. Each instance has ten configuration groups, and they each occupy one tab.

5.3.14.1 Process Name<Name>

This configuration item is a required field, and determines the name of the process. The name must be a valid identifier, and must not duplicate the names of other processes and virtual machines in the system.

5.3.14.2 Extra Capability Table Slot Count<Extra_Captbl>

This configuration item is a required integer value. The user should fill in the number of capability table slots of this process that additionally need to be reserved for user use. It only needs to be set to a non-zero number when the user wants to manually add kernel objects to the capability table of this process. Generally, we do not recommend that the user manually create kernel objects, because this means that the user may compromise the information security of the system.

5.3.14.3 Enable Coprocessors<Coprocessor>

This configuration item is a comma-separated list that records the coprocessors that need to be enabled in the process project, such as the FPU; only the coprocessors enabled here will be initialized by the kernel. The project generator will check whether the coprocessor is enabled in the project. If not, the project generator will report an error.

5.3.14.4 Build System<Buildsystem>

Determines which build system the process project adopts.

5.3.14.5 Build Toolchain<Toolchain>

Determines which build toolchain the process project adopts.

[16] that is, the capability table used by the RVM virtual machine monitor process; it is the first capability table in the system
[17] the values of the corresponding macros are also exactly the same; for these macros please see 8.2.2

5.3.14.6 Optimization Level<Optimization>

Determines the optimization level used by the process project. The available optimization levels are no optimization (O0), basic optimization (O1), full optimization (O2), extreme optimization (O3), speed optimization (Of), and size optimization (Os).

5.3.14.7 Project Output Directory<Project_Output>

Determines the output directory of the process project, which is a path relative to the project description file.

5.3.14.8 Project Overwrite<Project_Overwrite>

Determines whether to overwrite the previously generated process project file[18] when regenerating, and can be "Yes" or "No". If the user has manually edited the project and wishes to keep these changes, "No" can be chosen to keep the original old version.

5.3.14.9 Linker Script Output Directory<Linker_Output>

Determines the output directory of the process project linker script prc_xxx_xxx.ld[19], which is a path relative to the project description file. The linker script is always overwritten in each generation.

5.3.14.10 Main Header File Output Directory<Main_Source_Output>

Determines the output directory of the process main header file prc_xxx.h, which is a path relative to the project description file. The main header file is always overwritten in each generation.

5.3.14.11 Main Source File Output Directory<Main_Source_Output>

Determines the output directory of the process main source file prc_xxx.c and the descriptor file prc_xxx_desc.c, which is a path relative to the project description file. The main source file and the descriptor file are always overwritten in each generation; the main source file contains a user-fillable debug print function RVM_Putchar, and if the user fills it in and enables debug printing, then all the debug print macros beginning with RVM_DBG_ can be used.

5.3.14.12 Entry Point Source File Output Directory<Entry_Source_Output>

Determines the output directory of the thread and invocation entry point source files prc_xxx_thd_xxx.c and prc_xxx_inv_xxx.c in the process, which is a path relative to the project description file. These files correspond one-to-one with the threads and invocation entries defined in the process; modifying these files can take over the execution of the related threads or invocations. Among them, native threads should never return, and invocations must call RVM_Inv_Ret at the end to actively return.

This configuration item applies only to native processes, and does not exist for virtual machines.

5.3.14.13 Entry Point Source File Overwrite<Entry_Source_Overwrite>

Determines whether to overwrite the previously generated process entry point source files when regenerating, and can be "Yes" or "No". If the user has manually edited the entry point source files and wishes to keep these changes, "No" can be chosen to keep the original old version.

This configuration item applies only to native processes, and does not exist for virtual machines.

5.3.14.14 Private Memory Block<Memory><Mn>

This configuration group contains the attributes of the private memory blocks present in this process. This group must contain at least one code (Code) memory block and one data (Data) memory block; the

[18] such as Keil uVision's *.uvprojx, etc.
[19] the suffix here may also differ depending on the selected toolchain

first declared code memory block will hold all the code of the process, and the first declared data memory block will hold all the data area and stack area of the process. The remaining code memory blocks and data memory blocks are usually used to hold shared code and data, and can be used in the form of pointers.

5.3.14.14.1 Private Segment Name<Name>

This configuration item is an optional field[20], and can determine the name of this private memory segment. The name must be a valid C identifier, and must be case-insensitively different among the memory segments within a process. The project generator and RVM itself do not use this name, but if the user provides it, macro definitions for the base address and length of each memory segment will be generated in the kernel boot header file rme_boot.h, the virtual machine monitor boot header file rvm_boot.h, and the process main header file prc_xxx.h, to facilitate user programming: the base address and length of a code segment are represented as CODE_XXX_BASE and CODE_XXX_SIZE respectively, the length and base address of a data segment are represented as DATA_XXX_BASE and DATA_XXX_SIZE respectively, and the length and base address of a device segment are represented as DEVICE_XXX_BASE and DEVICE_XXX_SIZE respectively.

If the user does not fill in this field, then this memory segment will exist as an anonymous memory segment, and the above macros will not be defined.

5.3.14.14.2 Private Segment Base Address<Base>

This configuration item is an optional hexadecimal value. The user should fill in the start address of this shared memory segment. If "Auto" is filled in, it means that the location of this shared memory segment is automatically allocated by the compiler. The device segment is an exception: its segment base address cannot be "Auto", and must be specified manually.

5.3.14.14.3 Private Segment Size<Size>

This configuration item is a required hexadecimal value. The user should fill in the length of this shared memory segment.

5.3.14.14.4 Private Segment Alignment<Align>

This configuration item is a required decimal value that determines the alignment of this memory.

When the segment base address is "Auto", if N is filled in, then the segment of memory is aligned to 2^N during automatic allocation; if "Auto" is filled in, it means that the alignment of this segment is determined by the compiler at its own discretion.

When the segment base address is a fixed value, this configuration item must be filled in as "Auto".

For the same segment of memory, the alignments supported by different architectures may differ.

5.3.14.14.5 Private Segment Type<Type>

This configuration item determines the type of the memory segment. There are three types to choose from: code (Code), data (Data), and device (Device). When allocating memory, the compiler will use this type to determine which kind of memory this shared memory segment is allocated to.

5.3.14.14.6 Private Segment Attributes<Attribute>

This configuration item determines the attributes of the memory segment; for the content of the attributes, please see 5.3.11.6.

5.3.14.15 Shared Memory Block Reference<Shmem><Sn>

This configuration group contains the attributes of the shared memory blocks to be referenced by this process.

[20] the <Name> tag of an anonymous memory segment may be absent, or may be present but empty; the project generator accepts both forms

5.3.14.15.1 Shared Segment Name<Name>

This configuration item determines the shared memory segment to be added into this memory. This shared memory segment must be declared in 5.3.11.

5.3.14.15.2 Shared Segment Attributes<Attribute>

This configuration item determines the attributes when the shared memory block is mapped into this process; this attribute must be a subset of the attributes at the time the shared memory segment was declared; this achieves fine-grained controlled sharing. For example, if the attributes of a shared memory block at declaration are RWBCS, but the attributes are RBCS when referenced in this process, then this process can only read but cannot write this shared memory block[21]. For the content of the attributes, please see 5.3.11.6.

5.3.14.16 Thread<Thread><Tn>

This configuration group contains the attributes of the threads running in this process. Each process must have at least one thread.

This configuration group applies only to native processes, and does not exist for virtual machines.

5.3.14.16.1 Thread Name<Name>

This configuration item is a required field, and determines the name of the thread. The name must be a valid C identifier, and should be unique within the same process.

5.3.14.16.2 Thread Stack Size<Stack_Size>

This configuration item is a required hexadecimal value that determines the size of the thread stack. The size can be decided according to the requirements.

5.3.14.16.3 Thread Parameter<Parameter>

This configuration item is an optional hexadecimal value that determines the parameter passed to the thread when it starts. If not filled in, it defaults to 0.

5.3.14.16.4 Thread Priority<Priority>

This configuration item is a required integer value that determines the real-time priority of the thread. The priority must be between Kern_Prio-2 and 5[22]. For example, if the kernel priority count Kern_Prio is 32, then this value can be chosen between 5 and 30. The larger the value, the higher the priority of the thread.

5.3.14.17 Invocation Target<Invocation><In>

This configuration group contains the related information of the invocation targets set in this process. The process may or may not have invocation targets; when a thread in another process activates an invocation target, it jumps to the invocation target for execution, and returns to the original process after execution.

This configuration item applies only to native processes, and does not exist for virtual machines.

5.3.14.17.1 Invocation Name<Name>

This configuration item is a required field, and determines the name of the invocation. The name must be a valid C identifier, and should be unique within the same process.

[21] only other processes that reference this shared memory block and whose attributes include W have write permission

[22] the highest priority Kern_Prio-1 is occupied by the safety daemon thread SFTD, while priority 4 and below are used by RVM and the virtual machines it manages

5.3.14.17.2 Invocation Stack Size<Stack_Size>

This configuration item is a required hexadecimal value that determines the size of the invocation stack. The size can be decided according to the requirements.

5.3.14.18 Invocation Entry<Port><Pn>

This configuration group contains the related information of the invocations that can be called from this process. The process may or may not have invocation entries; once an invocation entry is activated, the thread that activates it will jump to the invocation target for execution, and after completion return to its own process to continue execution.

This configuration item applies only to native processes, and does not exist for virtual machines.

5.3.14.18.1 Invocation Target Name<Name>

This configuration item determines the specific target that the invocation is to enter.

5.3.14.18.2 Name of the Process Containing the Invocation Target<Process>

This configuration item determines the process in which the invocation target is located. A reference to an invocation is uniquely determined by the target name and the name of the process in which the target is located.

5.3.14.19 Signal Receive Endpoint<Receive><Rn>

This configuration group contains all the receive signal endpoints on which the threads in this process can block to receive information. The process may or may not have receive signal endpoints; once the send signal endpoint corresponding to a receive signal endpoint is activated, the thread on the receive signal endpoint will be released from waiting and start running, until it blocks again.

5.3.14.19.1 Receive Endpoint Name<Name>

This configuration item is a required field, and determines the name of the receive endpoint. The name must be a valid C identifier, and should be unique within the same process. In the process main header file prc_xxx.h, macro definitions of the form RCV_XXX for each receive endpoint will be generated, to facilitate the user's reference in system calls such as RME_Sig_Rcv.

5.3.14.20 Signal Send Endpoint<Send><Sn>

This configuration group contains all the send signal endpoints to which the threads in this process can send. When the process has just been created, this item is empty. The process may or may not have send signal endpoints; once a send signal endpoint is activated, the thread on the corresponding receive signal endpoint will be released from waiting and start running, until it blocks again.

This configuration item can also exist in virtual machines, where it serves as an important means for virtual machines to send signals to native processes. For more information, please see 2.2.7.2.

5.3.14.20.1 Signal Receive Endpoint Name<Name>

This configuration item determines the name of the signal receive endpoint corresponding to this signal send endpoint. In the process main header file prc_xxx.h, macro definitions of the form SND_XXX_PRC_XXX for each send endpoint will be generated, to facilitate the user's reference in system calls such as RME_Sig_Snd.

5.3.14.20.2 Name of the Process Containing the Signal Receive Endpoint<Process>

This configuration item determines the process in which the signal receive endpoint is located. A reference to a signal receive endpoint is uniquely determined by the endpoint name and the name of the process in which the endpoint is located.

5.3.14.21 Vector Receive Endpoint<Vector><Vn>

This configuration group contains the vector signal endpoints of all the physical interrupts that the threads in this process can block to receive. The process may or may not have physical vector receive endpoints; once the interrupt vector is activated, the thread on the corresponding physical vector endpoint will be released from waiting and start running, until it blocks again.

This configuration item applies only to native processes, and does not exist for virtual machines.

5.3.14.21.1 Vector Endpoint Name<Name>

This configuration item determines the name of the physical interrupt vector. The name can be decided by the user. In the process main header file prc_xxx.h, macro definitions of the form VCT_XXX for each send endpoint will be generated, to facilitate the user's reference in system calls such as RME_Sig_Rcv.

5.3.14.21.2 Vector Number<Number>

This configuration item determines the vector number of the physical interrupt vector. The vector number must be less than the total number of vectors declared in the chip description file, and each vector number can appear only once in the configuration file.

5.3.14.22 Kernel Function Call<Kfunc><Kn>

This configuration group contains all the kernel function calls that the threads in this process can activate. The process may or may not have kernel function calls; the threads in the process can access some special operations that need to be executed in kernel mode by activating kernel function calls.

This configuration item can also exist in virtual machines, where it can be used to authorize specific virtual machines to perform some kernel-level sensitive operations. It should be noted that if kernel function calls are abused, the security of the system will be severely damaged, so any kernel function authorization behavior needs to be made cautiously after a comprehensive system security assessment.

5.3.14.22.1 Function Call Name<Name>

This configuration item determines the name of this kernel function call. The name must be a valid C identifier, and should be unique within the same process. In the process main header file prc_xxx.h, macro definitions of the form KFN_XXX for each send endpoint will be generated, to facilitate the user's reference in system calls such as RME_Kfn_Act.

5.3.14.22.2 Major Function Number Start Value<Begin>

This configuration item is a hexadecimal value that determines the start point of the major function numbers allowed by this kernel function call.

5.3.14.22.3 Major Function Number End Value<End>

This configuration item is a hexadecimal value that determines the end point of the major function numbers allowed by this kernel function call. This value cannot be lower than the start value. If the start point of the major function numbers of a kernel function call is 0x10 and the end point is 0x1F, then the major function numbers of the kernel functions that can be called are between 0x10 and 0x1F, including 0x10 and 0x1F.

5.3.15 Virtual Machine Configuration<Virtual><Vn>

This configuration group is responsible for the configuration of the virtual machines in the system. The system can have one or more virtual machines, so this class can have one or more instances. Virtual machines are actually implemented based on native processes; they have most of the configuration items of native processes, but are also somewhat different. Compared with native processes, virtual machines cannot have threads<Thread>, invocation targets<Invocation>, invocation entries<Port>, signal receive endpoints<Receive>, or vector receive endpoints<Vector>, but can have private memory blocks<Memory>,

shared memory block references<Shmem>, and signal send endpoints<Send>. In addition, compared with native processes, virtual machines also have some additional configuration items; these virtual-machine-specific configuration items will be explained one by one next.

5.3.15.1 Interrupt Handling Thread Stack Size<Vector_Stack_Size>

This configuration item is a hexadecimal value that determines the stack size of the virtual machine interrupt handling thread Vct. This size can be compared with the size of the dedicated stack for interrupt vectors in a bare-metal RTOS; the specific value depends on the stack depth used by the registered virtual interrupt vector handling functions. Generally, 512B is a very good reference value.

5.3.15.2 User Program Thread Stack Size<User_Stack_Size>

This configuration item is a hexadecimal value that determines the stack size of the virtual machine user program thread Usr. This size can be compared with the startup stack size in a bare-metal RTOS; it generally does not need to be very large, as long as it can complete the initialization of the RTOS itself; after the RTOS starts, this stack is generally no longer used, and instead a dedicated stack is allocated for each RTOS thread. Generally, 256B is a very good reference value.

5.3.15.3 Priority<Priority>

This configuration item is an integer value that determines the virtualization priority of the virtual machine. The priority must be between the virtual machine priority count-1 and 0. For example, if the virtualization priority count is 32, then this value can be chosen between 0 and 31. The larger the value, the higher the priority of the virtual machine.

It should be noted that the virtual machine priority and the thread priority in native processes are not the same thing; the priority of a virtual machine is lower than that of any native process, and it uses the virtualization priority. Only when all the threads in all native processes are blocked does a high-priority virtual machine run; only when the high-priority virtual machines are completely blocked and not running does it come to the turn of the low-priority virtual machines to run.

5.3.15.4 Timeslice<Timeslice>

This configuration item is an integer value greater than 0 that determines the number of timeslices of the virtual machine, in units of RVM clock ticks[23]. In the system, virtual machines of different priorities are in a fully preemptive relationship, while virtual machines of the same priority are in a timeslice round-robin scheduling relationship. It is generally recommended to set this value relatively large, such as 50 or 100, so as to avoid frequent switching between virtual machines and improve the performance of the system.

5.3.15.5 Clock Period<Period>

This configuration item is an integer value greater than 0 that determines the frequency division coefficient of the virtual clock interrupt of the virtual machine, in units of RVM clock ticks: only after every Period RVM clock ticks is a timer interrupt Tim[24] sent to this virtual machine. It is generally recommended to set this value relatively large, such as 5 or 10, to prevent the clock ticks from consuming a large amount of system resources.

5.3.15.6 Watchdog Timeout<Watchdog>

This configuration item is an integer value that determines the watchdog timeout of the virtual machine, in units of RVM clock ticks. Once a virtual machine loses response and the watchdog times out, RVM will restart that timed-out virtual machine. Generally, the watchdog timeout is recommended to be set

[23] please see 2.1.2.4, the same below
[24] please see 2.2.4.1

to about 1 second, which is roughly 100 or 1000 RVM clock ticks. If this configuration item is set to 0, it means that the watchdog is not enabled for this virtual machine.

The watchdog timeout is calculated according to the running time of the virtual machine, rather than the running time of the real system. Assuming the watchdog timeout is set to 1 second, it means that the net running time of the virtual machine needs to exceed 1 second without calling RVM_Hyp_Wdg_Clr to feed the watchdog before a watchdog timeout occurs.

5.3.15.7 Number of Virtual Interrupt Vectors<Vect_Num>

This configuration item determines the number of virtual interrupt sources of the virtual machine. It is recommended to set this value as small as possible on the premise of being sufficient, so as to save system resources.

5.3.15.8 Guest Operating System Type<Guest_Type>

This configuration item determines the type of the guest operating system being run.

5.3.15.9 Guest Operating System Root Directory<Guest_Root>

This configuration item determines the root directory of the guest operating system, which is a path relative to the project description file. The generated virtual machine project will reference the files of the guest operating system from this directory.

5.3.15.10 Guest Header File Output Directory<Virtual_Header_Output>

This configuration item determines the output directory of the guest RTOS header files, which is a path relative to the project description file. Which configuration header files are generated specifically, and what their contents are, is determined by the specific guest operating system selected.

5.3.15.11 Guest Header File Overwrite<Virtual_Header_Overwrite>

Determines whether to overwrite the previously generated guest RTOS header files when regenerating, and can be "Yes" or "No". If the user has manually edited the guest header files and wishes to keep these changes, "No" can be chosen to keep the original old version. It is worth noting that even if No is chosen, which header files are retained and which header files are overwritten is still determined by the specific guest generation tool, but the generation tool will try as much as possible to keep those user-editable header files from being overwritten.

5.3.15.12 Guest Header File Output Directory<Virtual_Source_Output>

This configuration item determines the output directory of the guest RTOS source files, which is a path relative to the project description file. Which configuration source files are generated specifically, and what their contents are, is determined by the specific guest operating system selected.

5.3.15.13 Guest Header File Overwrite<Virtual_Header_Overwrite>

Determines whether to overwrite the previously generated guest RTOS source files when regenerating, and can be "Yes" or "No". If the user has manually edited the guest RTOS source files and wishes to keep these changes, "No" can be chosen to keep the original old version. It is worth noting that even if No is chosen, which source files are retained and which source files are overwritten is still determined by the specific guest generation tool, but the generation tool will try as much as possible to keep those user-editable source files from being overwritten.

5.4 Platform Description File Structure<Platform>

This file records the basic information of the platform. The project generator uses this information to determine the specific model and compatibility of the platform.

5.4.1 Name<Name>

The name of the architecture. It must be consistent with the standard name[25] of the platform. The project generator relies on this name to locate the platform description file, and cross-checks that the architecture abbreviations in the project file, the platform description file, and the chip description file are consistent.

5.4.2 Version<Version>

This configuration item records the version of the platform description file.

The project generator does not care about the value of this configuration item.

5.4.3 Word Length<Wordlength>

This configuration item records the processor word length of the platform, in bits. Typical values are 32 or 64.

5.4.4 Allowed Coprocessors<Coprocessor>

This configuration item is a comma-separated list that records the coprocessors allowed in the architecture, such as the model of the FPU, etc. The project generator uses this to check the legality of a specific chip configuration file. If an architecture has no coprocessors, it can be left blank.

5.4.5 Guest List<Guest>

This configuration item is a comma-separated list that records all guest operating systems allowed by the architecture.

5.4.6 Build System List<Buildsystem>

This configuration item is a comma-separated list that records all build systems allowed by the architecture.

5.4.7 Toolchain List<Toolchain>

This configuration item is a comma-separated list that records all compilation toolchains allowed by the architecture.

5.4.8 Architecture Options<Config><Cn>

This configuration group records the inherent options that the architecture itself possesses. The project compiler attempts to query the configuration of these inherent options from the project.

5.4.8.1 Name<Name>

This configuration item determines the internal name of the option. The project generator uses this name to retrieve the corresponding option from the project file.

[25] The abbreviated folder name of the platform

5.4.8.2 Category<Type>

This configuration item is a field that determines the nature of the option. There are only two categories: one is Choice (selection) and the other is Range (range). The former means that one of the values listed by the <Range> tag can be selected as its value; the latter means that any value between the two values listed by the <Range> tag is acceptable.

5.4.8.3 Macro Definition<Macro>

This configuration item is a field that determines the macro name corresponding to the option. The project generator generates the configuration file based on this macro name.

5.4.8.4 Value Range<Range>

This configuration item is a comma-separator field. When the option is of Choice type, this separator field lists all possible values that the option can take, and these values may be numbers or arbitrary strings; when the option is of Range type, this separator field may contain only two non-negative integer values, and these two values determine the value range.

5.5 Chip Description File Structure<Chip>

This file records the basic information of the chip. This basic information will be used by the generator to determine the specific model and compatibility of the chip.

5.5.1 Name<Name>

This configuration item is a field that determines the name of the chip family. All chips belonging to this category are described using this chip XML. Generally, the various chips placed in one category differ only slightly in pin count and peripheral count, and their memory layouts should all be completely identical; the subsequent configuration items of the chip description file define these in detail.

5.5.2 Version<Version>

This configuration item records the version of the chip description file.

The project generator does not care about the value of this configuration item.

5.5.3 Platform Architecture<Platform>

This configuration item determines the platform architecture of the chip. This name must be consistent with the short name of the platform folder in which the chip resides. The project generator will cross-check that the architecture abbreviations in the project file, the platform description file, and the chip description file are consistent.

5.5.4 Compatible List<Compatible>

This configuration item is a comma-separated list that determines the full names of the specific chips compatible with this description file. Only chips listed in this field can be compatible; the specific chip model given in the project description file must be in this list.

5.5.5 Vendor<Vendor>

This configuration item determines the manufacturer of the chip. It is used when generating projects for certain toolchains.

5.5.6 Number of Interrupt Vectors<Vector>

This configuration item determines the number of external device interrupt vectors of the chip.

5.5.7 Number of General Memory Protection Regions<Region>

This configuration item determines the number of general MPU regions of the chip. General MPU regions can be used for code segments as well as for data segments and device segments.

5.5.8 Number of Code Memory Protection Regions<Iregion>

This configuration item determines the number of code-dedicated MPU regions of the chip. These MPU regions differ from general MPU regions in that they can only be used for code segments.

The project generator does not read this option for the time being.

5.5.9 Number of Data Memory Protection Regions<Dregion>

This configuration item is a decimal number that determines the number of data-dedicated MPU regions of the chip. These MPU regions differ from general MPU regions in that they can only be used for data segments and device segments.

The project generator does not read this option for the time being.

5.5.10 Present Coprocessors<Coprocessor>

This configuration item is a comma-separated list that records the coprocessors present in a specific chip, such as the model of the FPU, etc. The project generator checks its legality against the architecture description file, and will also check the legality of a specific project description file on this basis. If a specific chip has no coprocessors, it can be left blank.

5.5.11 Architecture Attribute Group<Attribute>

This configuration group determines the architecture-specific attributes. This information may differ for each architecture, such as the specific type of the CPU, the presence and model of the FPU, and so on. These attributes cannot be set by the user; they are configuration items inherent to the chip itself.

5.5.12 Chip Inherent Memory<Memory><Mn>

This configuration group records the on-chip memory information of the chip. All on-chip memory present on the chip must be registered here. The specific registration method is the same as that for extra memory[26].

5.5.12.1 Memory Segment Base Address<Base>

This configuration item is a hexadecimal value that determines the start address of this memory segment.

5.5.12.2 Memory Segment Length<Size>

This configuration item is a hexadecimal value that determines the length of this memory segment.

5.5.12.3 Memory Segment Category<Type>

This configuration item is a field that determines the category of the memory segment. There are three possible categories: code (Code), data (Data) and device (Device).

[26] Please refer to 5.3.10

5.5.12.4 Memory Segment Attributes<Attribute>

This configuration item is a field that determines the attributes of the memory segment. There are six memory segment attributes: readable (Readable, R), writable (Writable, W), executable (eXecutable, X), bufferable (Bufferable, B), cacheable (Cacheable, C) and static mapping (Static, S), and they can be superimposed in any combination.

Generally, the attributes of a code segment ROM should be set to RXBCS, and the attributes of a data segment RAM should be set to RWBCS. The attributes of a device segment should be set to RWS.

5.5.13 Chip Options<Config><Cn>

This class records the inherent options that the chip itself possesses[27]. The project compiler attempts to query the configuration of these inherent options from the project. These options are organized in the same way as the architecture inherent options.

5.5.13.1 Name<Name>

This configuration item determines the internal name of the option. The project generator uses this name to retrieve the corresponding option from the project file.

5.5.13.2 Category<Type>

This configuration item is a field that determines the nature of the option. There are only two categories: one is Choice (selection) and the other is Range (range). The former means that one of the values listed by the <Range> tag can be selected as its value; the latter means that any value between the two values listed by the <Range> tag is acceptable.

5.5.13.3 Macro Definition<Macro>

This configuration item is a field that determines the macro name corresponding to the option. The project generator generates the configuration file based on this macro name.

5.5.13.4 Value Range<Range>

This configuration item is a comma-separator field. When the option is of Choice type, this separator field lists all possible values that the option can take, and these values may be numbers or arbitrary strings; when the option is of Range type, this separator field may contain only two non-negative integer values, and these two values determine the value range.

[27] Excluding the inherent options of the architecture; those options are listed in the platform description file rather than here; please refer to 5.4.8

5.6 Project Generator Error Messages

5.6.1 Command-line argument analysis errors (C0000-C9999)

All output formats supported by this tool are as follows:

Table 5-1 Supported output toolchains

Output toolchain option	Project organization	Compiler
makefile	Makefile	GCC
keil	Keil uVision	ARMCC 5.0

All possible error messages that may occur in this step are as follows (C0000 -- C9999):

Table 5-2 Error messages that may occur during the command-line argument analysis stage

Error number	Error message	Cause
C0000	Too many or too few input parameters.	Too few or too many command-line arguments were entered.
C0001	More than one input file.	More than one input file was specified.
C0002	More than one output path.	More than one output path was specified.
C0003	More than one output format.	More than one output format was specified.
C0004	Unrecognized command line argument.	A command-line argument that cannot be recognized.
C0005	No input file specified.	No input file was specified.
C0006	No output path specified.	No output path was specified.
C0007	No output project format specified.	No output project format was specified.

5.6.2 Project XML file analysis errors (P0000-P9999)

This module analyses the project description XML file, that is, the XML file generated by the GUI module. It first reads the XML file according to the input path and generates its DOM, then analyses each component in the DOM step by step, and finally generates internal data structures that are convenient for reference. The specific content of each sub-step is explained section by section below.

5.6.2.1 Project configuration analysis (P0000-P0099)

In this step, the extra memory information, the shared memory information and some basic project information in the system are analysed and then directly filled into the internal data structures. All possible error messages that may occur are as follows (P0000 -- P0099):

Table 5-3 Error messages that may occur during the preliminary project configuration analysis stage

Error number	Error message	Cause
P0000	Project XML parsing failed.	The project XML file contains a syntax error and cannot be parsed.
P0001	Project XML is malformed.	The <Project> tag does not exist in the project XML.
P0002	Name section is missing.	It does not exist.
P0003	Name section is empty.	It is empty.
P0004	Platform section is missing.	The target platform <Platform> does not exist.
P0005	Platform section is empty.	The target platform <Platform> is empty.
P0006	Chip class section is missing.	The chip name <Name> does not exist.
P0007	Chip class section is empty.	The chip name <Name> is empty.
P0008	Chip fullname section is missing.	The chip class <Class> does not exist.
P0009	Chip fullname section is empty.	The chip class <Class> is empty.
P0010	RME section is missing.	The <RME> tag does not exist in the project XML.
P0011	RVM section is missing.	The <RVM> tag does not exist in the project XML.
P0012	Extra memory section is missing.	The <Extmem> tag does not exist in the project XML.
P0013	Extra memory section parsing internal error.	An unknown error was encountered while parsing the <Extmem> tag and its contents. Please check the integrity of this tag.
P0014	Extra memory type section is missing.	The extra segment type <Type> does not exist.
P0015	Extra memory type section is empty.	The extra segment type <Type> is empty.
P0016	Extra memory type is malformed.	The extra segment type <Type> must be one of Code, Data or Device. The current value cannot be recognized.
P0017	Shared memory section is missing.	The <Shmem> tag does not exist in the project XML.
P0018	Shared memory section parsing internal error.	An unknown error was encountered while parsing the <Shmem> tag and its contents. Please check the integrity of this tag.
P0019	Shared memory type section is missing.	The shared segment type <Type> does not exist.
P0020	Shared memory type section is empty.	The shared segment type <Type> is empty.
P0021	Shared memory name cannot be omitted.	The shared segment name <Name> is mandatory and cannot be omitted.
P0022	Shared memory type is malformed.	The extra segment type <Type> must be one of Code, Data or Device. The current value cannot be recognized.

Error number	Error message	Cause
P0023	Process section parsing internal error.	An unknown error was encountered while parsing the <Process> tag and its contents. Please check the integrity of this tag.
P0024	Virtual machine section parsing internal error.	An unknown error was encountered while parsing the <Virtual> tag and its contents. Please check the integrity of this tag.
P0025	Project XML and chip XML platform does not match.	The platform type in the project XML and that in the chip XML do not correspond. Please check whether the project XML specifies a wrong platform.
P0026	Project XML and chip XML chip class does not match.	The chip class in the project XML and that in the chip XML do not correspond. Please check whether the project XML specifies a wrong chip class.
P0027	The specific chip designated in project XML not found in chip XML.	The chip specified in the project XML is not compatible in the chip XML. Please check whether a wrong chip model is specified.
P0028	The specific platform is currently not supported.	The platform specified in the project XML is currently not supported. Please check whether a wrong platform is specified.

5.6.2.2 Kernel configuration analysis (P0100-P0199)

This stage analyses the basic configuration information of the kernel; all of it is located under the <RME> tag. All possible errors that may occur are shown in the table below (P0100 -- P0199):

Table 5-4 Error messages that may occur during the kernel configuration analysis stage

Error number	Error message	Cause
P0100	Code base section is missing.	The code segment base address <Code_Base> does not exist.
P0101	Code base is not a valid hex integer.	The code segment base address <Code_Base> is not a hexadecimal number.
P0102	Code size section is missing.	The code segment size <Code_Size> does not exist.
P0103	Code size is not a valid hex integer.	The code segment size <Code_Size> is not a hexadecimal number.
P0104	Data base section is missing.	The data segment base address <Data_Base> does not exist.
P0105	Data base is not a valid hex integer.	The data segment base address <Data_Base> is not a hexadecimal number.
P0106	Data size section is missing.	The data segment size <Data_Size> is not a hexadecimal number.
P0107	Data size is not a valid hex integer.	The data segment size <Data_Size> is not a hexadecimal number.
P0108	Stack size section is missing.	The kernel stack size <Stack_Size> does not exist.
P0109	Stack size is not a valid hex integer.	The kernel stack size <Stack_Size> is not a hexadecimal number.

Error number	Error message	Cause
P0110	Extra kernel memory section is missing.	The extra kernel object memory size <Extra_Kom> does not exist.
P0111	Extra kernel memory is not a valid hex integer.	The extra kernel object memory size <Extra_Kom> is not a hexadecimal number.
P0112	Kernel memory order section is missing.	The kernel memory allocation granularity <Kom_Order> does not exist.
P0113	Kernel memory order is not a valid unsigned integer.	The kernel memory allocation granularity <Kom_Order> is not a decimal number.
P0114	Priority number section is missing.	The number of priorities <Kern_Prio> does not exist.
P0115	Priority number is not a valid unsigned integer.	The number of priorities <Kern_Prio> is not a decimal number.
P0116	Compiler option section is missing.	The <Compiler> tag does not exist.
P0117	Platform option section is missing.	The <Platform> tag does not exist.
P0118	Platform option section parsing internal error.	An unknown error was encountered while parsing the <Platform> tag and its contents. Please check the integrity of this tag.
P0119	Chip option section is missing.	The <Chip> tag does not exist.
P0120	Chip option section parsing internal error.	An unknown error was encountered while parsing the <Chip> tag and its contents. Please check the integrity of this tag.
P0121	Chip option mismatch in project option list.	An option provided under the <Chip> tag cannot be found in the chip XML, or not all options in the chip XML are specified. Please check whether a wrong or redundant tag is specified, or some tags are missing.
P0122	Value conversion failure.	Conversion of the value set for a Range-type option failed. Please check whether this option is set to an unsigned integer in the project XML.
P0123	Value overflow.	The value set for a Range-type option exceeds the upper bound of the allowed range. Please check and set it again.
P0124	Value underflow.	The value set for a Range-type option exceeds the lower bound of the allowed range. Please check and set it again.
P0125	Value not found.	The value set for a Select-type option is not in the list of allowed options. Please check and set it again.

5.6.2.3 User-mode library configuration analysis (P0200-P0299)

This stage analyses the basic configuration information of the user-mode library; all of it is located under the <RVM> tag. All possible errors that may occur are shown in the table below (P0200 -- P0299):

Table 5-5 Error messages that may occur during the user-mode library configuration analysis stage

Error number	Error message	Cause
P0200	Code size section is missing.	The code segment size <Code_Size> does not exist.
P0201	Code size is not a valid hex integer.	The code segment size <Code_Size> is not a hexadecimal number.
P0202	Data size section is missing.	The data segment size <Data_Size> does not exist.
P0203	Data size is not a valid hex integer.	The data segment size <Data_Size> is not a hexadecimal number.
P0204	Stack size section is missing.	The startup daemon thread stack size <Init_Stack_Size> does not exist.
P0205	Stack size is not a valid hex integer.	The startup daemon thread stack size <Init_Stack_Size> is not a hexadecimal number.
P0206	Extra capability table size section is missing.	The number of extra capability table slots <Extra_Captbl> does not exist.
P0207	Extra capability table size is not a valid unsigned integer.	The number of extra capability table slots <Extra_Captbl> is not a decimal number.
P0208	Compiler section is missing.	The <Compiler> tag does not exist.
P0209	Virtual machine priorities section is missing.	The number of virtualization priorities <Virt_Prio> does not exist.
P0210	Virtual machine priorities is not a valid unsigned integer.	The number of virtualization priorities <Virt_Prio> is not a decimal number.
P0211	Virtual machine event number section is missing.	The number of virtual machine event sources <Virt_Event> does not exist.
P0212	Virtual machine event number is not a valid unsigned integer.	The number of virtual machine event sources <Virt_Event> is not a decimal number.
P0213	Virtual machine mapping total number section is missing.	The number of virtual machine vector mappings <Virt_Map> does not exist.
P0214	Virtual machine mapping total number is not a valid unsigned integer.	The number of virtual machine vector mappings <Virt_Map> is not a decimal number.

5.6.2.4 Process configuration analysis (P0300-P0399)

This stage analyses the basic configuration information of each process; all of it is located under the <Process> tag. All possible errors that may occur are shown in the table below (P0300 -- P0399):

Table 5-6 Error messages that may occur during the process configuration analysis stage

Error number	Error message	Cause
P0300	Name section is missing.	The process name <Name> does not exist.
P0301	Name section is empty.	The process name <Name> is empty.
P0302	Extra capability table size section is missing.	The number of extra capability table slots <Extra_Captbl> does not exist.
P0303	Extra capability table size is not a valid unsigned integer.	The number of extra capability table slots <Extra_Captbl> is not a decimal number.
P0304	Compiler section is missing.	The <Compiler> tag does not exist.
P0305	Memory section is missing.	The <Memory> tag does not exist.
P0306	Memory section parsing internal error.	An unknown error was encountered while parsing the <Memory> tag and its contents. Please check the integrity of this tag.
P0307	Memory section is empty.	The <Memory> tag is empty, which means there is no private memory segment.
P0308	Memory type section is missing.	This tag does not exist.
P0309	Memory type section is empty.	This tag is empty.
P0310	Memory type is malformed.	This tag must be one of Code, Data or Device. The current value cannot be recognized.
P0311	Shared memory section is missing.	The <Shmem> tag does not exist.
P0312	Shared memory section parsing internal error.	An unknown error was encountered while parsing the <Shmem> tag and its contents. Please check the integrity of this tag.
P0313	Shared memory type section is missing.	This tag does not exist.
P0314	Shared memory type section is empty.	This tag is empty.
P0315	Shared memory type is malformed.	This tag must be one of Code, Data or Device. The current value cannot be recognized.
P0316	Thread section is missing.	The <Thread> tag does not exist.
P0317	Thread section parsing internal error.	An unknown error was encountered while parsing the <Thread> tag and its contents. Please check the integrity of this tag.
P0318	Thread section is empty.	The <Thread> tag is empty, which means there is no thread.
P0319	Invocation section is missing.	The <Invocation> tag does not exist.

Error number	Error message	Cause
P0320	Invocation section parsing internal error.	An unknown error was encountered while parsing the <Invocation> tag and its contents. Please check the integrity of this tag.
P0321	Port section is missing.	The <Port> tag does not exist.
P0322	Port section parsing internal error.	An unknown error was encountered while parsing the <Port> tag and its contents. Please check the integrity of this tag.
P0323	Receive section is missing.	The <Receive> tag does not exist.
P0324	Receive section parsing internal error.	An unknown error was encountered while parsing the <Receive> tag and its contents. Please check the integrity of this tag.
P0325	Send section is missing.	The <Send> tag does not exist.
P0326	Send section parsing internal error.	An unknown error was encountered while parsing the <Send> tag and its contents. Please check the integrity of this tag.
P0327	Vector section missing.	The <Vector> tag does not exist.
P0328	Vector section parsing internal error.	An unknown error was encountered while parsing the <Vector> tag and its contents. Please check the integrity of this tag.
P0329	Kernel function section missing.	The <Kernel> tag does not exist.
P0330	Kernel function section parsing internal error.	An unknown error was encountered while parsing the <Kernel> tag and its contents. Please check the integrity of this tag.
P0331	No primary code section exists.	There must be at least one private Code memory segment in this process as the primary code memory segment, which is used to store code.
P0332	No primary data section exists.	There must be at least one private Data memory segment in this process as the primary data memory segment, which is used to store data.
P0333	Primary code section does not have RXS attribute.	The primary code segment of the process does not possess the complete RXS attributes (readable and executable, and statically mapped). It must possess these attributes to serve as the primary code segment.
P0334	Primary data section does not have RWS attribute.	The primary data segment of the process does not possess the complete RWS attributes (readable, writable and statically mapped). It must possess these attributes to serve as the primary data segment.
P0335	No thread exists.	There must be at least one thread in the process, but there is currently none.

5.6.2.5 Virtual machine configuration analysis (P0400-P0499)

This stage analyses the basic configuration information of each virtual machine; all of it is located under the <Virtual> tag. All possible errors that may occur are shown in the table below (P0400 -- P0499):

Table 5-7 Error messages that may occur during the virtual machine configuration analysis stage

Error number	Error message	Cause
P0400	Name section is missing.	This tag does not exist.
P0401	Name section is empty.	This tag is empty.
P0402	Guest operating system selection section is missing.	This tag does not exist.
P0403	Guest operating system selection section is empty.	This tag is empty.
P0404	Extra capability table size section is missing.	This tag does not exist.
P0405	Extra capability table size is not a valid unsigned integer.	This tag is not a decimal number.
P0406	Stack size section is missing.	This tag does not exist.
P0407	Stack size is not a valid hex integer.	This tag is not a hexadecimal number.
P0408	Priority section is missing.	This tag does not exist.
P0409	Priority is not a valid unsigned integer.	This tag is not a hexadecimal number.
P0410	Timeslices section is missing.	This tag does not exist.
P0411	Timeslices is not a valid unsigned integer.	This tag is not a decimal number.
P0412	Timeslices cannot be zero.	This tag cannot be 0.
P0413	Timer interrupt period section is missing.	This tag does not exist.
P0414	Timer interrupt period is not a valid unsigned integer.	This tag is not a decimal number.
P0415	Timer interrupt period cannot be zero.	This tag cannot be 0.
P0416	Watchdog timeout section is missing.	This tag does not exist.
P0417	Watchdog timeout is not a valid unsigned integer.	This tag is not a decimal number.
P0418	Virtual vector number section is missing.	This tag does not exist.
P0419	Virtual vector number is not a valid unsigned integer.	This tag is not a decimal number.

Error number	Error message	Cause
P0420	Virtual vector number too large.	This tag is greater than 768, which is not supported.
P0421	Virtual vector number cannot be zero.	This tag cannot be 0.
P0422	Compiler section is missing.	The <Compiler> tag does not exist.
P0423	Memory section is missing.	The <Memory> tag does not exist.
P0424	Memory section parsing internal error.	An unknown error was encountered while parsing the <Memory> tag and its contents. Please check the integrity of this tag.
P0425	Memory section is empty.	The <Memory> tag is empty, which means there is no private memory segment.
P0426	Memory type section is missing.	This tag does not exist.
P0427	Memory type section is empty.	This tag is empty.
P0428	Memory type is malformed.	This tag must be one of Code, Data or Device. The current value cannot be recognized.
P0429	Shared memory section is missing.	The <Shmem> tag does not exist.
P0430	Shared memory section parsing internal error.	An unknown error was encountered while parsing the <Shmem> tag and its contents. Please check the integrity of this tag.
P0431	Shared memory type section is missing.	This tag does not exist.
P0432	Shared memory type section is empty.	This tag is empty.
P0433	Shared memory type is malformed.	This tag must be one of Code, Data or Device. The current value cannot be recognized.
P0434	Send section is missing.	The <Send> tag does not exist.
P0435	Send section parsing internal error.	An unknown error was encountered while parsing the <Send> tag and its contents. Please check the integrity of this tag.
P0436	Kernel function section missing.	The <Kernel> tag does not exist.
P0437	Kernel function section parsing internal error.	An unknown error was encountered while parsing the <Kernel> tag and its contents. Please check the integrity of this tag.
P0438	No primary code section exists.	There must be at least one private Code memory segment in this virtual machine as the primary code memory segment, which is used to store code.
P0439	No primary data section exists.	There must be at least one private Data memory segment in this virtual machine as the primary data memory segment, which is used to store data.

Error number	Error message	Cause
P0440	Primary code section does not have RXS attribute.	The primary code segment of the virtual machine does not possess the complete RXS attributes (readable and executable, and statically mapped). It must possess these attributes to serve as the primary code segment.
P0441	Primary data section does not have RWS attribute.	The primary data segment of the virtual machine does not possess the complete RWS attributes (readable, writable and statically mapped). It must possess these attributes to serve as the primary data segment.

5.6.2.6 Memory segment configuration analysis (P0500-P0599)

Memory segment configuration analysis is called when analysing the extra memory (<Extmem>), the system shared memory (<Project><Shmem>) and the memory (<Memory>) of each process (<Process>) or virtual machine (<Virtual>). It is also called when analysing the built-in memory of the chip, so these errors also apply to the analysis of the built-in chip memory (<Chip><Memory>, see later). All possible errors that may occur are shown in the table below (P0500 -- P0599):

Table 5-8 Error messages that may occur during the memory segment configuration analysis stage

Error number	Error message	Cause
P0500	Base section is missing.	The extra segment base address <Base>, the shared segment base address <Base>, this tag, this tag or the storage segment base address <Base> does not exist.
P0501	Base is neither automatically allocated nor a valid hex integer.	The extra segment base address <Base>, the shared segment base address <Base>, this tag, this tag or the storage segment base address <Base> is not a hexadecimal number, and is not marked as automatically allocated.
P0502	Size section is missing.	The extra segment size <Size>, the shared segment size <Size>, this tag, this tag or the storage segment size <Size> does not exist.
P0503	Size is not a valid hex integer.	The extra segment size <Size>, the shared segment size <Size>, this tag, this tag or the storage segment size <Size> is not a hexadecimal number.
P0504	Size cannot be zero.	The extra segment size <Size>, the shared segment size <Size>, this tag, this tag or the storage segment size <Size> cannot be 0.
P0505	Size is out of bound.	The extra segment size <Size>, the shared segment size <Size>, this tag, this tag or the storage segment size <Size> is larger than 4GB.
P0506	Type section is missing.	The extra segment type <Type>, the shared segment type <Type>, this tag, this tag or the storage segment type <Type> does not exist.
P0507	Device-type memory cannot be automatically allocated.	Memory of the Device category cannot be set as automatically allocated; its start address must be specified manually.

Error number	Error message	Cause
P0508	Attribute section is missing.	The extra segment attribute <Attribute>, the shared segment attribute <Attribute>, this tag, this tag or the storage segment attribute <Attribute> does not exist.
P0509	Attribute section is empty.	The extra segment attribute <Attribute>, the shared segment attribute <Attribute>, this tag, this tag or the storage segment attribute <Attribute> is empty.
P0510	Attribute does not allow any access and is malformed.	The extra segment attribute <Attribute>, the shared segment attribute <Attribute>, this tag, this tag or the storage segment attribute <Attribute> does not allow any form of access (not readable, not writable and not executable).

5.6.2.7 Shared memory segment reference configuration analysis (P0600-P0699)

The shared memory segment references described in this section refer to the references to previously declared shared memory (<Shmem>) that are located in each process (<Process>) or virtual machine (<Virtual>), rather than the declarations of the shared memory segments themselves. All possible errors that may occur are shown in the table below (P0600 -- P0699):

Table 5-9 Error messages that may occur during the shared memory segment configuration analysis stage

Error number	Error message	Cause
P0600	Name section is missing.	The shared segment name <Name> does not exist.
P0601	Name section is empty.	The shared segment name <Name> is empty.
P0602	Attribute section is missing.	The shared segment attribute <Attribute> does not exist.
P0603	Attribute section is empty.	The shared segment attribute <Attribute> is empty.
P0604	Attribute does not allow any access and is malformed.	The shared segment attribute <Attribute> does not allow any form of access (not readable, not writable and not executable), which is not allowed.

5.6.2.8 Thread configuration analysis (P0700-P0799)

Thread configuration analysis is used when analysing the threads (<Thread>) of each process (<Process>). All possible errors that may occur are shown in the table below (P0700 -- P0799):

Table 5-10 Error messages that may occur during the thread configuration analysis stage

Error number	Error message	Cause
P0700	Name section is missing.	The thread name <Name> does not exist.
P0701	Name section is empty.	The thread name <Name> is empty.
P0702	Stack size section is missing.	The thread stack size <Stack_Size> does not exist.
P0703	Stack size is not a valid hex integer.	The thread stack size <Stack_Size> is not a hexadecimal number.

Error number	Error message	Cause
P0704	Stack size cannot be zero.	The thread stack size <Stack_Size> cannot be 0.
P0705	Parameter section is missing.	The thread parameter <Parameter> does not exist.
P0706	Parameter is not a valid hex integer.	The thread parameter <Parameter> is not a hexadecimal number.
P0707	Priority section is missing.	The thread priority <Priority> does not exist.
P0708	Priority is not a valid unsigned integer.	The thread priority <Priority> is not a decimal number.

5.6.2.9 Invocation target configuration analysis (P0800-P0899)

Invocation target configuration analysis is used when analysing the invocation targets (<Invocation>) of each process (<Process>). All possible errors that may occur are shown in the table below (P0800 -- P0899):

Table 5-11 Error messages that may occur during the invocation target configuration analysis stage

Error number	Error message	Cause
P0800	Name section is missing.	The invocation name <Name> does not exist.
P0801	Name section is empty.	The invocation name <Name> is empty.
P0802	Stack size section is missing.	The invocation stack size <Stack_Size> does not exist.
P0803	Stack size is not a valid hex integer.	The invocation stack size <Stack_Size> is not a hexadecimal number.
P0804	Stack size cannot be zero.	The invocation stack size <Stack_Size> cannot be 0.

5.6.2.10 Invocation entry configuration analysis (P0900-P0999)

Invocation entry configuration analysis is used when analysing the invocation entries (<Port>) of each process (<Process>). All possible errors that may occur are shown in the table below (P0900 -- P0999):

Table 5-12 Error messages that may occur during the invocation entry configuration analysis stage

Error number	Error message	Cause
P0900	Name section is missing.	The invocation target name <Name> does not exist.
P0901	Name section is empty.	The invocation target name <Name> is empty.
P0902	Process name section is missing.	The name of the process in which the invocation target resides <Process> does not exist.
P0903	Process name section is empty.	The name of the process in which the invocation target resides <Process> is empty.

5.6.2.11 Receive signal endpoint configuration analysis (P1000-P1099)

Receive signal endpoint configuration analysis is used when analysing the receive signal endpoints (<Receive>) of each process (<Process>). All possible errors that may occur are shown in the table below (P1000 -- P1099):

Table 5-13 Error messages that may occur during the receive signal endpoint configuration analysis stage

Error number	Error message	Cause
P1000	Name section is missing.	The receive endpoint name <Name> does not exist.
P1001	Name section is empty.	The receive endpoint name <Name> is empty.

5.6.2.12 Send signal endpoint configuration analysis (P1100-P1199)

Send signal endpoint configuration analysis is used when analysing the send signal endpoints (<Send>) of each process (<Process>). All possible errors that may occur are shown in the table below (P1100 -- P1199):

Table 5-14 Error messages that may occur during the send signal endpoint configuration analysis stage

Error number	Error message	Cause
P1100	Name section is missing.	The signal send endpoint name <Name> does not exist.
P1101	Name section is empty.	The signal send endpoint name <Name> is empty.
P1102	Process name section is missing.	The name of the process in which the signal send endpoint resides <Process> does not exist.
P1103	Process name section is empty.	The name of the process in which the signal send endpoint resides <Process> is empty.

5.6.2.13 Physical vector signal endpoint configuration analysis (P1200-P1299)

Physical vector signal endpoint configuration analysis is used when analysing the physical vector signal endpoints (<Vector>) of each process (<Process>). It is also used when analysing the physical signal endpoints of the chip (<Chip><Vector>, see later). All possible errors that may occur are shown in the table below (P1200 -- P1299):

Table 5-15 Error messages that may occur during the physical vector signal endpoint configuration analysis stage

Error number	Error message	Cause
P1200	Name section is missing.	The vector endpoint name <Name> does not exist.
P1201	Name section is empty.	The vector endpoint name <Name> is empty.
P1202	Number section is missing.	The vector endpoint name <Name> does not exist.
P1203	Number is not a valid unsigned integer.	The vector endpoint name <Name> is not a decimal number.

5.6.2.14 Kernel function configuration analysis (P1300-P1399)

Kernel function configuration analysis is used when analysing the kernel functions (<Kernel>) of each process (<Process>) or virtual machine (<Virtual>). All possible errors that may occur are shown in the table below (P1300 -- P1399):

Table 5-16 Error messages that may occur during the kernel function configuration analysis stage

Error number	Error message	Cause
P1300	Name section is missing.	The function call name <Name> does not exist.
P1301	Name section is empty.	The function call name <Name> is empty.
P1302	Starting kernel function number section is missing.	The starting value of the primary function number <Begin> does not exist.
P1303	Starting kernel function number is not a valid hex integer.	The starting value of the primary function number <Begin> is not a hexadecimal number.
P1304	Ending kernel function number section is missing.	The ending value of the primary function number <End> does not exist.
P1305	Ending kernel function number is not a valid hex integer.	The ending value of the primary function number <End> is not a hexadecimal number.
P1306	Ending kernel function number is smaller than the starting kernel function number.	The ending value is larger than the starting value, which is not allowed. The ending value should be at least equal to the starting value.

5.6.3 Chip XML file analysis errors (D0000-D9999)

This module analyses the chip description XML files. These files are bundled with the compiler. The specific content of each sub-step is explained section by section below.

5.6.3.1 Preliminary chip information analysis (D0000-D0099)

In this step, the extra memory information, the shared memory information and some basic project information in the system are analysed and then directly filled into the internal data structures. All possible error messages are as follows (D0000 -- D0099):

Table 5-17 Error messages that may occur during the preliminary chip information analysis stage

Error number	Error message	Cause
D0000	Chip XML parsing failed.	The chip XML file contains a syntax error and cannot be parsed.
D0001	Chip XML is malformed.	The <Chip> tag does not exist in the chip XML.
D0002	Class section is missing.	The name <Name> does not exist.
D0003	Class section is empty.	The name <Name> is empty.
D0004	Compatible variant section is missing.	The compatible list <Compatible> does not exist.
D0005	Compatible variant section is empty.	The compatible list <Compatible> is empty.
D0006	Vendor section is missing.	The vendor <Vendor> does not exist.
D0007	Vendor section is empty.	The vendor <Vendor> is empty.
D0008	Platform section is missing.	The platform architecture <Platform> does not exist.
D0009	Platform section is empty.	The platform architecture <Platform> is empty.

Error number	Error message	Cause
D0010	Cores section is missing.	This tag does not exist.
D0011	Cores is not an unsigned integer.	This tag is empty.
D0012	Regions section is missing.	The number of general memory protection regions <Region> does not exist.
D0013	Regions is not an unsigned integer.	The number of general memory protection regions <Region> is empty.
D0014	Attribute section missing.	The architecture attribute group <Attribute> does not exist.
D0015	Attribute section parsing internal error.	An unknown error occurred while parsing the architecture attribute group <Attribute> and its contents. Please check the <Attribute> tag and its contents.
D0016	Memory section is missing.	The <Memory> tag does not exist in the chip XML.
D0017	Memory section parsing internal error.	An unknown error was encountered while parsing the <Memory> tag and its contents. Please check the integrity of this tag.
D0018	Memory section is empty.	The inside of the <Memory> tag is empty.
D0019	Memory type section is missing.	The memory section type <Type> does not exist.
D0020	Memory type section is empty.	The memory section type <Type> is empty.
D0021	Memory type is malformed.	The memory section type <Type> must be one of Code, Data or Device. The current value cannot be recognized.
D0022	No code section exists.	There is no memory available for code sections in the chip.
D0023	No data section exists.	There is no memory available for data sections in the chip.
D0024	No device section exists.	There is no memory available for device sections in the chip.
D0025	Option section is missing.	The <Option> tag does not exist in the chip XML.
D0026	Option section parsing internal error.	An unknown error was encountered while parsing the <Option> tag and its contents. Please check the integrity of this tag.
D0027	Vector section missing.	The <Vector> tag does not exist in the chip XML.
D0028	Vector section parsing internal error.	An unknown error was encountered while parsing the <Vector> tag and its contents. Please check the integrity of this tag.

5.6.3.2 Chip memory analysis (P0500-P0599)

This step analyses the memory storage of the chip itself. The errors that may occur in this part (P0500 - P0599) have already been introduced earlier, so we omit them here.

5.6.3.3 Chip option analysis (D0100-D0199)

This step analyses the options of the chip itself. All possible error messages are as follows (D0100 - D0199):

Table 5-18 Error messages that may occur during the chip option analysis stage

Error number	Error message	Cause
D0100	Name section is missing.	This tag does not exist.
D0101	Name section is empty.	This tag is empty.
D0102	Type section is missing.	This tag does not exist.
D0103	Type section is empty.	This tag is empty.
D0104	Type is malformed.	This tag must be either Select or Range. The current value cannot be recognized.
D0105	Macro section is missing.	This tag does not exist.
D0106	Macro section is empty.	This tag is empty.
D0107	Range section is missing.	This tag does not exist.
D0108	Range section is empty.	This tag is empty.
D0109	Range section have an empty value.	An empty value appears in this tag. An empty value means that there is no content between two commas, or no content before the first comma, or no content after the last comma.
D0110	Range typed option cannot have more or less than two ends specified.	This tag of a Range typed option must have exactly two values marking its starting and ending points. More or fewer are not allowed.
D0111	Starting point or ending point conversion failure.	Some value in this tag of a Range typed option is not an integer. Please check that its starting and ending points are indeed unsigned integers.
D0112	Range typed option starting point must be smaller than the corresponding ending point.	The starting point value in this tag of a Range typed option must be strictly smaller than the ending point value, otherwise the option will have nothing to select.
D0113	Select typed option's possible values must all be valid C identifiers or numbers.	All possible values in this tag of a Select typed option must all be valid C identifiers, or a number.

5.6.3.4 Chip vector analysis (P1200-P1299)

This step analyses the interrupt vectors of the chip itself. The errors that may occur in this part (P1200 - P1299) have already been introduced earlier, so we omit them here.

5.6.4 Memory allocation errors (M0000-M0999)

This step determines the start address of each automatically allocated memory section, and decides the specific construction method of the page tables. It can in turn be divided into three sub-steps: memory alignment, memory section allocation and page table generation.

5.6.4.1 Memory alignment (A0000-A9999)

This step mainly checks and aligns each memory block. The memory reserved for the RME and the RVM must be aligned to a granularity determined by the target architecture. The stacks of the individual threads must also be aligned to this granularity. The start address and size of each memory block are also checked, and if any case does not conform to this alignment granularity, an error will be reported. These errors differ from architecture to architecture; for the specific errors produced, refer to the corresponding documentation of that architecture.

5.6.4.2 Memory section allocation (M0000-M0099)

This step assigns fixed memory addresses to those memory sections that do not have a specified start address. In this step, we first add the extra memory sections into the chip's memory layout and check whether there is any overlap; if there is none, we then determine the start addresses of the unallocated code and data memory sections.

The allocator uses a highly intelligent algorithm to determine the position of each memory section. After allocation finishes, we check the memory layout of each process to determine that there is no overlap, and to determine that the primary code sections of the individual processes are mutually independent[28]. Finally, we also check that all those memory sections for which the user specified a start address and length fall within the chip memory layout.

All possible error messages are as follows (M0000 -- M0099):

Table 5-19 Error messages that may occur during the memory section allocation stage

Error number	Error message	Cause
M0000	Chip internal and/or extra memory section overlapped.	The chip's memory layout has overlaps. For example, the added extra memory and the memory provided by the chip itself overlap, which is physically impossible.
M0001	RME code section is invalid, either wrong range or wrong attribute.	The location of the RME's code section is abnormal, because in the chip's memory layout this location does not allow code to be placed, or is not readable, not executable, or cannot be statically mapped.
M0002	RVM code section is invalid, either wrong range or wrong attribute.	The location of the RVM's code section is abnormal, because in the chip's memory layout this location does not allow code to be placed, or is not readable, not executable, or cannot be statically mapped.
M0003	Code section is invalid, either wrong range or wrong attribute.	The location of a process's code section or shared code section is abnormal, because in the chip's memory layout this location does not allow code to be placed, or cannot satisfy the access permissions declared for that code section.
M0004	Code memory fitter failed.	Code section allocation failed. This may be because too much code storage is used, or because too many code sections are declared and the code memory thus becomes fragmented.

[28] There is no such requirement for the primary data section

Error number	Error message	Cause
M0005	RME data section is invalid, either wrong range or wrong attribute.	The location of the RME's data section is abnormal, because in the chip's memory layout this location does not allow code to be placed, or is not readable, not writable, or cannot be statically mapped.
M0006	RVM data section is invalid, either wrong range or wrong attribute.	The location of the RVM's code section is abnormal, because in the chip's memory layout this location does not allow data to be placed, or is not readable, not writable, or cannot be statically mapped.
M0007	Data section is invalid, either wrong range or wrong attribute.	The location of a process's data section or shared data section is abnormal, because in the chip's memory layout this location does not allow code to be placed, or cannot satisfy the access permissions declared for that data section.
M0008	Data memory fitter failed.	Data section allocation failed. This may be because too much data storage is used, or because too many data sections are declared and the data memory thus becomes fragmented.
M0009	Name is duplicate or invalid.	Shared memory of the same type has a duplicate name, or references to the same shared memory within the same process are duplicated.
M0010	Entry not found in global database.	The reference to the shared memory within the process cannot be found in the shared memory declarations.
M0011	Attributes not satisfied by global entry.	The reference to the shared memory within the process
M0012	Private and/or shared memory section overlapped.	An address space overlap occurred within the same process. This may be because two private memory blocks with specified start addresses and lengths overlap, or because shared memory and private memory overlap.
M0013	Process primary code section overlapped.	The private primary code spaces of different processes overlap in address. This is not allowed, because the primary code space is exclusive to each process.
M0014	Device memory have wrong attributes.	The device sections in the chip's memory layout cannot satisfy the access permissions declared for that device memory.
M0015	Device memory segment is out of bound.	The declared device memory cannot be found in the chip's memory layout. This may be because this location of the chip does not allow device memory to be placed.

5.6.4.3 Page table generation (A0000-A9999)

This step generates the page tables of each process and each virtual machine in the project. This part is entirely determined by the specific architecture, so the specific error messages produced need to be looked up in the manual corresponding to that architecture.

5.6.5 Capability table allocation errors (M1000-M1999)

This step automatically allocates the capability table capability numbers of each kernel object. There are two capability numbers: one is the global capability number used when creating the kernel object in the RVM, and the other is the local capability number used when referencing these capabilities in each process. It can in turn be divided into five steps: compliance check, local capability number allocation, global capability number allocation, capability macro name allocation and capability pairing.

5.6.5.1 Compliance check (M1000-M1099)

This step mainly performs the more complex parameter checks on kernel objects that cannot be determined when parsing the XML. All possible error messages are as follows (M1000 -- M1099):

Table 5-20 Error messages that may occur during the compliance check stage

Error number	Error message	Cause
M1000	Total number of kernel priorities must be a multiple of word size.	The total number of kernel-mode priorities must be an integer multiple of the machine word size. Values that are not an integer multiple are not allowed.
M1001	Kernel memory allocation granularity order must be no less than 2^3=8 bytes.	The kernel memory allocation granularity order must be greater than or equal to 3.
M1002	Kernel memory allocation granularity order must be no more than 2^5=32 bytes.	The kernel memory allocation granularity order must be less than or equal to 5.
M1003	There are neither virtual machines nor processes in the system.	There are neither virtual machines nor native processes in the system. This is not allowed.
M1004	Virtual machine exists but the total number of virtual machine priorities is set to 0.	A virtual machine exists in the system, but the number of virtual machine priorities is set to 0. This is not allowed.
M1005	Total number of virtual machine priorities must be a multiple of word size.	The total number of virtual machine priorities must be an integer multiple of the machine word size. Values that are not an integer multiple are not allowed.
M1006	Virtual machine exists but the total number of virtual event sources is set to 0.	A virtual machine exists in the system, but the number of virtual event sources is set to 0. This is not allowed.
M1007	Total number of virtual event sources must be a multiple of word size.	The total number of virtual event sources must be an integer multiple of the machine word size. Values that are not an integer multiple are not allowed.
M1008	Total number of virtual event sources cannot exceed 1024.	The total number of virtual event sources cannot exceed 1024.

Error number	Error message	Cause
M1009	Total number of virtual event to interrupt mappings cannot be smaller than the virtual event source number.	The allowed upper limit of the total mappings from event sources or physical interrupts to virtual machine virtual vectors must be greater than the total number of virtual event sources.
M1010	Priority must be bigger than service daemons' priority (4).	The priority of threads in user processes must be greater than the priority of each service daemon thread, that is, must be greater than 4.
M1011	Priority must be smaller than safety daemon priority (Kern_Prios-1).	The priority of threads in user processes must be smaller than the priority of the safety daemon thread, that is, must be smaller than the total number of kernel priorities minus 1.
M1012	Priority must be smaller than total number of virtual machine priorities.	The priority of a virtual machine must be smaller than the total number of virtual machine priorities.
M1013	Extra capacity cannot be larger than the architectural limits.	The number of entries in the extra capability table cannot exceed the number allowed by the architecture. On 32-bit machines this limit is 128; on 64-bit machines this limit is 32768.
M1014	Kernel function number range exceeded architectural limits.	The function number of the kernel function exceeds the range allowed by the architecture. On 32-bit machines its maximum value is 2^{16}, and on 64-bit machines its maximum value is 2^{32}.
M1015	Name is duplicate or invalid.	Duplicate kernel object names exist in the system. This may be a duplicate process name, or a duplicate name of the same kind of kernel object within a process.
M1016	Name/process name is duplicate or invalid.	The targets declared by an invocation entry or a signal sending endpoint within the same process have the same name and the same owning process. This is not allowed.
M1017	Name/number is duplicate or invalid.	The name or interrupt number of the interrupt vector is duplicated. This is not allowed.

5.6.5.2 Local capability number allocation (M1100-M1199)

This step mainly allocates the capability numbers in the local capability table of each process. All possible error messages are as follows (M1100 -- M1199):

Table 5-21 Error messages that may occur during the local capability number allocation stage

Error number	Error message	Cause
M1100	Capability table size exceeded architectural limits.	The size of the local capability table exceeds the value allowed by the architecture. This is not allowed.

5.6.5.3 Global capability number allocation (M1200-M1299)

This step mainly allocates the global capability numbers used when the RVM creates each kernel resource. This stage produces no error messages.

5.6.5.4 Capability macro name allocation (M1300-M1399)

This step mainly allocates the capability macro names used for referencing capabilities. This stage produces no error messages.

5.6.5.5 Capability pairing (M1400-M1499)

This step pairs the signal sending endpoints and the invocation entries with the signal receiving endpoints and the invocation targets respectively. If the compiler finds a signal sending endpoint or an invocation entry that cannot be paired, it reports an error. This step also checks whether all declared physical vector names can be found in the chip XML, and reports an error if they cannot. All possible error messages are as follows (M1400 -- M1499):

Table 5-22 Error messages that may occur during the capability pairing stage

Error number	Error message	Cause
M1400	Invalid process name.	The process referenced by the signal sending endpoint or the invocation entry does not exist.
M1401	Invalid invocation name.	The invocation target referenced by the invocation entry does not exist.
M1402	Invalid receive endpoint name.	The signal receiving endpoint referenced by the signal sending endpoint does not exist.
M1403	Invalid vector number.	The vector number of the physical vector endpoint is inconsistent with that recorded in the chip XML.
M1404	Invalid vector name.	The vector name referenced by the physical vector endpoint is not recorded in the chip XML.

5.6.6 Kernel object allocation errors (M2000-M2999)

This step allocates the kernel memory addresses of each kernel object, as well as the memory layout of the threads and invocation targets within each process. It can in turn be divided into five steps: kernel object memory trial allocation, kernel object memory allocation, other kernel memory allocation, user-mode library memory allocation and process memory allocation.

5.6.6.1 Kernel object memory trial allocation and allocation (M2000-M2099)

Because the size of the primary capability table of the user-mode library is unknown when kernel object allocation begins, a trial allocation stage is needed. The trial allocation stage allocates the memory addresses of all kernel objects, but does not allocate the size occupied by the primary capability table. After the number of kernel objects is determined, the size of the primary capability table is determined immediately, and then a round of allocation is performed to obtain the real addresses of each kernel object. All possible error messages are as follows (M2000 -- M2099):

Table 5-23 Error messages that may occur during the kernel object trial allocation and allocation stage

Error number	Error message	Cause
M2000	RVM capability table size exceeded the architectural limit.	The size of the RVM's capability table exceeds the length allowed by the architecture. Please consider reducing the number of extra capability table slots of the RVM, or reducing the complexity of the system.

5.6.6.2 Other kernel memory allocation (M2100-M2199)

This step allocates the other memory that exists in the kernel, such as kernel stacks and so on. Some memory sections are required for the kernel to run, and other memory sections are required for the kernel to communicate with the user-mode library. All possible error messages are as follows (M2100 -- M2199):

Table 5-24 Error messages that may occur during the other kernel memory allocation stage

Error number	Error message	Cause
M2100	RME data section is not big enough, unable to allocate vector flags.	The RME's data section is not large enough to allocate the vector flag memory. Please consider making the RME's data section larger.
M2101	RME data section is not big enough, unable to allocate event flags.	The RME's data section is not large enough to allocate the vector flag memory. Please consider making the RME's data section larger.
M2102	RME data section is not big enough, unable to allocate kernel stacks.	The RME's data section is not large enough to allocate the vector flag memory. Please consider making the RME's data section larger.
M2103	RME data section is not big enough, unable to allocate kernel object memory.	The RME's data section is not large enough to allocate the vector flag memory. Please consider making the RME's data section larger.

5.6.6.3 User-mode library memory allocation (M2200-M2299)

This step allocates user-mode library memory, such as the stacks used by each daemon thread. All possible error messages are as follows (M2200 -- M2299):

Table 5-25 Error messages that may occur during the user-mode library memory allocation stage

Error number	Error message	Cause
M2200	RVM data section is not big enough, unable to allocate init thread stack.	The RVM's data section is not large enough to allocate the startup daemon thread stack. Please consider making the RVM's data section larger.
M2201	RVM data section is not big enough, unable to allocate safety daemon thread stack.	The RVM's data section is not large enough to allocate the safety daemon thread stack. Please consider making the RVM's data section larger.
M2202	RVM data section is not big enough, unable to allocate virtual machine monitor daemon thread stack.	The RVM's data section is not large enough to allocate the hypercall daemon thread stack. Please consider making the RVM's data section larger.
M2203	RVM data section is not big enough, unable to allocate vector handling daemon stack.	The RVM's data section is not large enough to allocate the vector daemon thread stack. Please consider making the RVM's data section larger.
M2204	RVM data section is not big enough, unable to allocate timer handling thread stack.	The RVM's data section is not large enough to allocate the time daemon thread stack. Please consider making the RVM's data section larger.

5.6.6.4 Process memory allocation (M2300-M2399)

This step allocates the memory of the process itself, such as the stacks of each thread and each invocation target. If the process is actually a virtual machine, memory related to the virtual machine monitor must also be allocated. All possible error messages are as follows (M2300 -- M2399):

Table 5-26 Error messages that may occur during the process memory allocation stage

Error number	Error message	Cause
M2300	Data section size is not big enough, unable to allocate stack.	The process's data section is not large enough to allocate the stacks of threads or invocation targets. Please consider making the process's data section larger.
M2301	Data section size is not big enough, unable to allocate virtual machine interrupt flags.	The virtual machine's data section is not large enough to allocate the virtual vector flag memory area. Please consider making the virtual machine's data section larger.
M2302	Data section size is not big enough, unable to allocate virtual machine parameters.	The virtual machine's data section is not large enough to allocate the hypercall parameter area. Please consider making the virtual machine's data section larger.
M2303	Data section size is not big enough, unable to allocate virtual machine registers.	The virtual machine's data section is not large enough to allocate the virtualization register area. Please consider making the virtual machine's data section larger.

5.6.7 Kernel project generation errors (G0000-G0999)

This step generates the RME kernel project. It can in turn be divided into five steps: folder structure generation, configuration header file generation, startup sequence source file generation, user-editable source file generation and architecture-related file generation.

In the folder structure generation step, the various folders required by the kernel are created, and the kernel source code is also filled into the folders.

In the configuration header file generation step, the configuration header file used for selecting the architecture is generated.

In the startup sequence source file generation step, the header files and source files used to create kernel objects at startup are created.

In the user-editable source file generation step, the user-editable source files are created. The user-editable source files are mainly responsible for filling in the kernel-mode interrupt vectors.

In the architecture-related file generation stage, the project of the relevant toolchain or IDE is created, thereby organizing the generated source files into a compilable project.

Overall, this step and its five sub-steps have no independent error messages.

5.6.8 User-mode library project generation errors (G1000-G1999)

This step is responsible for generating the RVM user-mode library project. It can in turn be divided into four steps: folder structure generation, configuration header file generation, startup sequence source file generation and architecture-related file generation.

In the folder structure generation step, the various folders required by the user-mode library project are created, and the user-mode library source code is also filled into the folders.

In the configuration header file generation step, the configuration header file used for selecting the architecture is generated.

In the startup sequence source file generation step, the header files and source files used to create kernel objects at startup are created.

In the user-editable source file generation step, the user-editable source files are created. The user-editable source files are mainly responsible for further initialization as well as writing some customizable handling logic.

In the architecture-related file generation stage, the project of the relevant toolchain or IDE is created, thereby organizing the generated source files into a compilable project.

Overall, the error messages that may occur in this step are as follows (G1000 -- G1999):

Table 5-27 Error messages that may occur when generating the user-mode library project

Error number	Error message	Cause
G1000	Internal capability table computation failure.	Internal capability table size computation error; the kernel object creation code cannot be generated. The occurrence of this error indicates that the compiler has encountered an internal failure; please report this failure to us.

5.6.9 User process project generation errors (G2000-G2999)

This step is responsible for generating the project of each user process. It can in turn be divided into three steps: folder structure generation, process source file generation and architecture-related file generation.

In the folder structure generation step, the various folders required by the process project are created, and the library source code needed by the process to invoke system services is also filled into the folders.

In the process source file generation step, the macro definitions of the various kernel objects that are declared in the process and can be referenced locally are provided, so that users can reference them conveniently; in addition, empty functions for the threads and invocation targets declared by the user are also created, waiting for the user to fill in.

In the architecture-related file generation step, the project of the relevant toolchain or IDE is created, thereby organizing the generated source files into a compilable project.

Overall, the error messages that may occur in this step are as follows (G2000 -- G2999):

Table 5-28 Error messages that may occur when generating the user process project

Error number	Error message	Cause
G2000	Virtual machine should not have threads other than Vect and User.	Only two threads are allowed to exist in a virtual machine: one is the User thread responsible for running the user program, and the other is the Vect thread responsible for receiving interrupts. The occurrence of this error indicates that the compiler has encountered an internal failure; please report this failure to us.

5.6.10 Workspace generation errors (G3000-G3999)

This step is responsible for generating the workspace. Each generated sub-project is placed under the workspace, facilitating unified management and compilation. It can in turn be divided into two steps: folder structure generation and architecture-related file generation.

In the folder structure generation step, the folders of the workspace are created.

In the architecture-related file generation step, we directly generate the architecture-related workspace files.

Overall, this step and its two sub-steps have no independent error messages.

5.6.11 File system operation errors (F0000-F9999)

In the project generation steps mentioned above, file system read and write operations are used extensively. The error messages that may occur with these file read and write operations are as follows (F1000 -- F1999):

Table 5-29 Error messages that may occur during file system operations

Error number	Error message	Cause
F0000	Folder creation failed.	A problem was encountered while creating a folder in the output folder. Check whether the output folder allows read and write.
F0001	Cannot open source file.	Cannot open the source file used as input. Check whether the output folder allows read and write.
F0002	Cannot read file.	Cannot open the file for reading. Check whether the output folder allows read and write.
F0003	Windows/Linux stat failed.	The stat system call used to determine the file size failed. Check whether the file system related configuration is correct.
F0004	Cannot open destination file.	Cannot open the destination file used as output. Check whether the output folder allows read and write.

5.6.12 Architecture-related file generation errors (A0000-A9999)

Among all the generation steps mentioned in this chapter, only the generation of architecture-related files is unique. The generation of these files is architecture-dependent, so their errors need to be determined by consulting the architecture-related manual.

5.7 Graphical Front End of the Project Generator

Since generating a system by inputting XML is still quite troublesome and error-prone, this compiler also provides a graphical user interface. By means of the description of the graphical interface module in this chapter, we also introduce the functions of the compiler.

5.7.1 Starting the Graphical Front End

Open the executable file of the compiler and wait for it to finish loading. After loading is complete, the following interface pops up, and the user needs to select the location where the project is to be generated, the architecture of the microcontroller to be used, and the model of the microcontroller itself.

After the user has finished selecting, the compiler officially starts and its working interface pops up. The workspace can be divided into three parts, namely the configuration group selection area, the configuration item selection area, and the configuration content area. We will introduce the purpose and usage of each area separately in the following sections.

In the following sections, each heading will show the name of the relevant field in the XML file. The meaning of the identifier before the field name is as follows: S stands for string, D stands for drop-down selection box, C stands for multiple-choice box, H stands for hexadecimal number, and N stands for integer. If the identifier is surrounded by round brackets "()", then the item may or may not be filled in; if the identifier is surrounded by square brackets "[]", then the item must be filled in.

5.7.2 Configuration Category Selection Area

This area is responsible for selecting the configuration category. There are seven configuration categories in total, namely project configuration, hardware configuration, storage configuration, RME kernel configuration, RVM user-mode library configuration, native process configuration and virtual machine configuration. Among them, each of the first five configuration categories has only one item, while the number of items in the latter two categories depends on the number of created processes and virtual machines. These configuration categories are all displayed in the first column on the left, where the eighth button at the very bottom is responsible for generating the project when clicked. The figure below shows the state of a typical configuration category selection area.

5.7.3 Configuration Object Selection Area

For the native process and virtual machine configuration categories, each native process and virtual machine object has a series of configuration groups. In this case, the second column on the left is responsible for displaying these native process or virtual machine objects. When the corresponding object is selected, the configuration groups of that object are displayed in the configuration group selection area. The figure below shows the state of a typical configuration object selection area.

5.7.4 Configuration Group Selection Area

The configuration group selection area is responsible for the individual configuration groups under a configuration category. For each configuration category, the content of its configuration groups is different; please refer to the specific chapters. The figure below shows the state of the configuration group selection area of a typical process configuration category.

5.7.5 Configuration Item Display Area

The configuration item display area is responsible for the display and configuration of specific configuration items. Similarly, for each configuration group, the content of its configuration items is different, and reference should be made to the specific chapters. The figure below shows the state of the configuration items of a typical memory block.

5.7.6 Project Generation Button

The project generation button is at the very bottom of the left sidebar of the interface. After clicking this button, an interface pops up allowing the user to select the IDE and address corresponding to the project to be generated. Each architecture may support different IDEs; once both have been selected, click the generate project button to generate the final project. On the mobile APP, the project generation function is not provided; instead, an XML file that can be read by the desktop version is generated. The project generation interface is as follows.

5.8 References for This Chapter

None

6. Using the RVM in a Project

6.1 Overview

This chapter describes the general workflow for using the RVM and its project generator in a project. The workflow first divides functionality into protection domains, then writes the project description file to reflect this division, and finally makes certain adjustments to the generated project to add behavior details.

Due to space limitations, this chapter describes only the settings of some important configuration items; for other configuration items, please refer to the preceding chapters.

6.2 Dividing Software Components into Protection Domains

Unlike bare-metal development, in a project that uses the RVM, each software component is assigned to a different protection domain, and the permissions of each protection domain are restricted to constrain the behavior of the software within it. Once one protection domain crashes due to a coding fault, the remaining protection domains keep running and are not disturbed; even if a protection domain is hijacked by a hacker, the hacker can obtain only the permissions of that protection domain and cannot control the entire system.

Deciding how to divide the components of the software into appropriate protection domains is crucial. At least the following three aspects must be considered: (1) the granularity of component division, (2) the requirements of components for compatibility and real-time performance, and (3) the requirements of components for device access. To better control system performance, it is recommended to first run various benchmarks on the target microcontroller, and after the tests are completed, decide the component division granularity according to the cost of each operation.

6.2.1 Granularity of Component Division

The granularity of component division refers to the size of an individual component. In theory, any software is infinitely divisible, down to the level of a single instruction, and the finer the division granularity, the better the protection effect[1]. However, the finer the division granularity, the more cross-protection-domain communication there is, and cross-protection-domain communication has a cost[2]. Therefore, it is recommended to divide components into an appropriate size to avoid this overhead as much as possible. A reasonable component division must satisfy the principle of high cohesion and low coupling: the internal connections within a component should be tight, while the connections between components should be sparse. In other words, a component should be responsible for one function that cannot be or is difficult to split, all states and logic of that function should be located within the component, and the interfaces that the component provides to other components should be concise and easy to use.

[1] A protection domain divided to the extreme is roughly equivalent to data-flow security

[2] These costs can be seen in benchmarks

At the start of the division, each independent function of the software can be divided into components as finely as possible, and then one can observe whether there is tight coupling of control flow and data flow in the division. Tightly coupled components have one of the following characteristics:

1. The frequency of mutual invocation between the two components is far higher than the average level of other components, for example, repeatedly calling another component within a computation loop; once the two components are placed in separate protection domains, their communication overhead becomes intolerable.

2. The two components share a data structure, and both may initiate write operations on that data structure that are difficult to isolate and difficult to make atomic, such as graph transformation operations, data sorting, etc.; placing the two components in separate protection domains is meaningless, because a crash of one component inevitably leaves the data structure incomplete, thereby causing the other component to crash.

3. The two components can be regarded as a split of a natural function that is better described as a whole, and after merging, the description of the component becomes more concise instead; if this is the case, assigning the two components to separate protection domains may be of limited significance, because they are not a single logical function and do not need independent recovery capability after a crash.

If such tight coupling is found in the existing division, the two components can be merged into one component; when all tightly coupled components have been merged, the result is a reasonable component partition. If performance still falls short of expectations after the initial consolidation, components can continue to be merged until a good trade-off between security and performance is obtained.

6.2.2 Requirements of Components for Compatibility and Real-Time Performance

After the division of components into protection domains is completed, the form of the protection domains must be further decided. The RVM provides two kinds of protection domains, namely native processes and virtual machines, each with its own advantages and disadvantages.

In terms of compatibility, native processes must be programmed using RME native system calls, and their compatibility with other runtime environments is poor, so manual porting is required if such an environment is to be used; virtual machines, by contrast, can flexibly use any guest RTOS and are directly compatible with the entire ecosystem of that RTOS. In terms of real-time performance, native processes do not need the virtual machine monitor as an intermediary, so cross-protection-domain communication performs better and interrupt latency is lower, allowing performance comparable to a bare-metal RTOS; virtual machines are the opposite: cross-protection-domain communication requires mediation by the virtual machine monitor, and the interrupt latency is approximately 3-4 times that of bare metal.

Several common examples are described in the table below.

Table 6-1 Examples of Component Division

Component Name	Component Division	Reason	Representative Software
Keyed remote-control code decoding	Native process	Requires high real-time performance, the code is simple, and accurate timer ticks are needed.	PT2262/PT2272 decoding 38kHz infrared decoding

Component Name	Component Division	Reason	Representative Software
Electric drive driver	Native process	Requires high real-time performance, the code is simple, and does not depend on an RTOS.	Stepper/servo motor driver Inverter switching transistor control driver
Single-wire bus driver	Native process	Requires high real-time performance, the code is simple, and does not depend on an RTOS.	DS18B20 ATECC608A
Audio bus driver	Native process	Requires high real-time performance, the code is simple, and the audio buffer must be refreshed in time.	Various I2S sound card drivers Various SPDIF output drivers
Ladder diagram executor	Native process	Although the code is complex, it is usually strictly reviewed and needs to be executed periodically, so it must be ported to a native process.	OpenPLC
Network protocol stack	Virtual machine	The real-time requirement is not high, but the internal logic is relatively complex and requires RTOS support.	LwIP uIP
Graphical user interface	Virtual machine	The real-time requirement is not high, and it may depend on RTOS support.	uC/GUI LVGL
File system	Virtual machine	The real-time requirement is not high, and it may depend on RTOS support.	ElmChan-FatFS Yaffs
Script interpreter	Virtual machine	The real-time requirement is not high, and it may depend on RTOS support.	Micropython Lua
Serial memory driver	Virtual machine	Although this driver is simple, its real-time requirement is not high, so it can be placed in a virtual machine.	AT24C02 driver W25Q64 driver

6.2.3 Requirements of Components for Device Access

Some driver components may need to access devices, so the devices must be exposed to the protection domain in which the component resides. The most common and simplest approach is (1) to make only the required devices visible to the protection domain, which can be achieved by adding the device's memory mapping to the memory description of that protection domain. If (2) multiple protection domains all need to use the same device, the device driver can be encapsulated in a separate protection domain, and other protection domains can delegate transfers to it.

For some devices with strong autonomy, using approach (1) to expose them directly to untrusted protection domains would lead to major information security risks. Although code inside a protection domain is incapable of breaking through the space-time protection domain, it can configure these autonomous devices and use them as a springboard to access arbitrary memory addresses, or generate high-frequency interrupts to launch a denial-of-service attack. The most common devices of this kind are direct memory access (Direct Memory Access, DMA) and timers; malicious code can manipulate the former to modify kernel space, or manipulate the latter to generate high-frequency interrupts.

For hardware such as DMA that autonomously accesses memory, the recommended approach is to hand it over to a separate trusted protection domain; when other protection domains need to use DMA, they must delegate it to the trusted protection domain. Although this increases DMA latency, it guarantees that DMA cannot be exploited to break the isolation boundary. Of course, if the microcontroller has an input/output memory protection unit (I/O Memory Protection Unit, IOMPU), this can also be used to limit the access range of DMA, in which case a separate trusted protection domain is not needed.

For hardware such as timers that autonomously initiate high-frequency interrupts, in addition to handing it over to a trusted protection domain, one can also throttle (Throttle) the number of interrupts in its interrupt vector, shutting it off once the interrupt frequency is found to be too high, until the user enables the interrupt again. Of course, if the microcontroller has hardware throttling capability, this can also be used to limit the interrupt frequency.

6.3 Dividing Interrupt Vectors into Protection Domains

The RVM's interrupt distribution chain starts at the kernel and goes all the way down into the virtual machine. On this chain, the closer a part is to the kernel, the better its real-time performance and the higher the interrupt frequency it supports, but the lower its security and the higher the reliability requirements on the interrupt vector. Therefore, for an interrupt vector of a particular frequency, an appropriate position must be chosen for it. This subsection classifies interrupts into four categories according to occurrence frequency: low frequency, medium frequency, high frequency, and ultra-high frequency, as shown in the table below[3].

Table 6-2 Classification of Interrupts by Occurrence Frequency

Category	Occurrence Frequency	Interval Time	Representative Interrupt Sources	Recommended Division Method
Low frequency	Within 1kHz	More than 1 millisecond; more than 100k cycles	Touch screen interrupt	Virtual machine interrupt/thread

[3] Although some interrupt sources have a low frequency, they require very low latency; in this case, their frequency should be calculated based on latency; for example, if an interrupt occurs once every millisecond but must be responded to within 1 microsecond after it occurs, its interrupt frequency should be treated as 1MHz

Category	Occurrence Frequency	Interval Time	Representative Interrupt Sources	Recommended Division Method
Medium frequency	1-10kHz	1 millisecond-100 microseconds; more than 10k cycles	IIC communication interrupt	Within a native process
High frequency	10-100kHz	10 microseconds-100 microseconds; more than 1k cycles	SPI communication interrupt	Kernel-perceptible interrupt
Ultra-high frequency	Above 1MHz	Less than 1 microsecond; within 100 cycles	Ultra-high baud rate[4] serial port interrupt	Kernel-transparent interrupt

6.3.1 Low-Frequency Interrupt Vectors

This kind of interrupt vector has very low performance requirements, so it can remain inside the virtual machine. For ported code, the overall structure needs no changes; it is only necessary to move the interrupt acknowledgement handling into the kernel, while the bulk of the interrupt can remain inside the virtual machine; when handled this way, the interrupt latency is generally several thousand machine cycles, that is, around a dozen microseconds. These interrupts are generally generated by human-machine interaction devices or communication devices with buffers, and good results can be achieved without very aggressive response times.

In terms of security, the execution of this kind of interrupt vector is fully governed by the virtual machine monitor, and no matter what error occurs, it will not cause any impact outside the virtual machine.

6.3.2 Medium-Frequency Interrupt Vectors

This kind of interrupt vector has certain performance requirements, and leaving it inside the virtual machine may cause interrupts to be lost. Therefore, moving it into a native process is necessary; when handled this way, the interrupt latency is generally several hundred machine cycles, that is, around a few microseconds. Specifically, the vector can be declared in a native process; the project generator will generate the handling stub corresponding to this vector in the kernel and generate the signal endpoint corresponding to this vector in the native process. Interrupt acknowledgement must be performed in the handling stub, and when the stub returns, indicate `RME_RVM_VCT_SIG_SELF` to send a signal only to its dedicated endpoint. The endpoint in the native process then receives this signal and unblocks the interrupt vector.

In terms of security, the execution of this kind of interrupt vector is still restricted by the protection domain provided by the kernel, but is no longer fully governed by the virtual machine monitor; it has only confidentiality and does not have integrity or availability, so its code must be reviewed to ensure that no security vulnerabilities are introduced.

6.3.3 High-Frequency Interrupt Vectors

This kind of interrupt vector imposes quite high performance requirements, and leaving it in any protection domain is impossible, because the interrupt latency to a protection domain requires at least several microseconds. Therefore, the most critical part of such interrupt vectors can only be placed in the kernel,

[4] 1M and above

and the processing of the non-critical part is deferred by sending a signal to a native process or sending an event to a virtual machine; when handled this way, the interrupt latency is generally several dozen machine cycles, that is, around a few hundred nanoseconds, but the entire interrupt vector may run for several hundred machine cycles in order to complete sending to the signal endpoint; if another interrupt vector is executing when this interrupt occurs, the interrupt must wait until the previous interrupt vector finishes executing, causing a delay of several hundred machine cycles.

In terms of security, this kind of interrupt vector is not subject to any restriction. If the code is not strictly reviewed, it will lead to serious functional and information security problems.

6.3.4 Ultra-High-Frequency Interrupt Vectors

This kind of interrupt vector often imposes extreme requirements on both response time and occurrence frequency, so much so that any signal endpoint send from within it is impossible; otherwise, on the one hand the latency would exceed the limit, and on the other hand CPU occupancy would be too high, and user mode might even be unable to execute, losing the meaning of delivering signals to user mode. Sometimes these interrupts must also be able to preempt any execution in the system, including kernel execution, to ensure a fixed latency. In this case, the only option is to make these interrupt vectors transparent interrupts and give them a preemption priority higher than the kernel[5]: on entry they push only those registers that are caller-saved in the calling convention, then immediately begin executing C code, and after completion they immediately pop and exit without notifying the kernel; when handled this way, the interrupt latency is generally several dozen machine cycles, that is, around a few hundred nanoseconds, and the running time of the entire interrupt vector can also be made less than one hundred machine cycles, achieving bare-metal-level responsiveness.

To enable ultra-high-frequency interrupts to somehow notify user mode when they occur, a separate perceptible interrupt vector can be set up, triggered through a software interrupt[6] when this interrupt vector exits, and it can notify the kernel on behalf of this vector. Of course, considering the ultra-high frequency of the interrupts, the frequency of triggering the perceptible interrupt cannot be too high; otherwise the perceptible interrupt would consume the entire CPU; in this case, cumulative counting or rate limiting can be applied to the interrupts, so that multiple ultra-high-frequency interrupts correspond to only one perceptible interrupt trigger.

In terms of security, this kind of interrupt vector is not subject to any restriction. If the code is not strictly reviewed, it will lead to serious functional and information security problems.

6.4 Choosing a Communication Method Between Protection Domains

The RVM provides three communication methods between protection domains, namely invocation, signal endpoints, and event sources. Among them, invocation and signal endpoints can only be used between native processes, while event sources are suitable for communication between native processes and virtual machines and between virtual machines and virtual machines. This subsection introduces the recommended communication method case by case.

6.4.1 Tightly Coupled Native Processes

For tightly coupled, performance-sensitive communication between native processes, invocation is recommended; it allows a thread in one process to migrate into another process to run, and both on entry (RVM_Inv_Act) and on return (RVM_Inv_Ret) a small amount of data can be carried in the register set, without the need to transfer it through shared memory. It is the fastest, but it may cause concurrent

[5] The preemption priority of a transparent interrupt is
[6] Such as the STIR register trigger method on ARMv7-M

access within the target process: after an old thread migrates into the target process, its access to data structures may be interrupted by a new thread selected by the scheduler; at this time the new thread can also migrate into the target process and initiate a synchronous access before the old thread's access ends.

Therefore, if invocation is to be used, the possible concurrency must be handled by oneself: add locks, or use reasonable lock-free data structures.

6.4.2 Loosely Coupled Native Processes

For loosely coupled, performance-insensitive communication between native processes, signal endpoints and shared memory are recommended. A signal endpoint itself has serialization capability; even if the sending side (RVM_Sig_Snd) has multiple threads, the receiving side (RVM_Sig_Rcv) can still be a single thread, which can greatly simplify communication programming.

It is worth noting that a signal endpoint has only receiving-side single-thread blocking capability, but does not have sending-side blocking capability, nor receiving-side multi-thread blocking capability. Fortunately, these missing capabilities can be implemented using native signal endpoints: if sending-side blocking is required, a receive endpoint can be prepared for each sending thread, and after the receiving side completes the service and returns the result, it returns a signal to the corresponding sender endpoint, causing the sender to unblock; if receiving-side multi-thread blocking capability is required, multiple receive endpoints can be used, and the sending side can determine whether a receiving thread is busy before sending in order to decide which receiving thread to wake. Of course, this can also be achieved by cleverly arranging the priorities of the server and the clients: if the server's priority is higher than that of the clients[7], then once the server is woken, the clients naturally block immediately, and the above two problems are readily solved; moreover, a client can indicate its own identity by the number of signals it sends.

6.4.3 Native Processes and Virtual Machines

Communication from a native process to a virtual machine and from a virtual machine to a native process must be implemented through signal endpoints. Sending from a virtual machine to a native process is no different from sending between native processes: a sending endpoint exists in the virtual machine and a receiving endpoint exists in the native process; once a send is initiated on that sending endpoint in the virtual machine, the thread on the receiving endpoint in the native process is unblocked. However, sending from a native process to a virtual machine is different: the receiving virtual machine must register its virtual interrupt with an event source (RVM_Hyp_Vct_Evt) in its startup code, then lock the virtual machine's permission settings (RVM_Hyp_Vct_Lck) to prevent subsequent tampering[8], and finally bind the corresponding interrupt vector entry to that virtual interrupt (RVM_Virt_Vct_Reg)[9]; a native process, by contrast, can send to any event source (RVM_Prc_Evt_Snd)[10]. Once the sending native process sends a signal, the virtual interrupt of the receiving virtual machine is triggered.

6.4.4 Virtual Machines and Virtual Machines

Communication between virtual machines is also carried out through event sources. The sending virtual machine can first grant itself the permission to send to an event source (RVM_Hyp_Evt_Add), and then lock the virtual machine's permission settings. The receiving side registers its virtual interrupt with the event source in its startup code, then likewise locks the virtual machine's permission settings, and finally binds the corresponding interrupt vector entry to that virtual interrupt.

[7] and they are all located on the same CPU, which is naturally satisfied for the RVM

[8] Since the virtual machine is forced to run a segment of user-customized trusted startup code on every restart to perform authorization and locking, self-authorization is safe

[9] the internal interrupt vector entry

[10] This does not break the availability of the virtual machine, because all native processes inherently have higher priority than virtual machines

6.5 Pre-Generation Settings

Before generating a project, sufficient valid information must be provided to the project generator. For the use of the project generator, please refer to Chapter 5 ; this subsection only introduces the points to note when filling in the various settings.

6.5.1 General Settings

This subsection introduces the recommended method for the project's general settings.

6.5.1.1 Setting Basic Options

"Basic Options" includes three items: "Project Name", "Enable Assertions", and "Enable Debug Printing", of which "Project Name" can be decided as needed. "Enable Assertions" enables the kernel's internal stricter boundary condition checks, and "Enable Debug Printing" outputs information on the debug printing port. The debug printing port is generally a serial port, and this serial port is specified in the chip's RME startup header file.

6.5.1.2 Setting the Workspace

"Workspace" includes "Workspace Build System" and "Workspace Output Overwrite". These two items control whether a unified workspace is generated and whether that workspace needs to be overwritten. The location of the "Workspace" is specified on the generator's command line or in the settings of its graphical interface.

Many build systems, such as Keil and Makefile, can generate a top-level workspace; once a build is ordered in the top-level workspace, the builds of the individual sub-workspaces start simultaneously. To use this feature, a build system must be selected in "Workspace Build System", and the build systems of the kernel, virtual machine monitor, native processes, and virtual machines in the subsequent configuration files must all be the same as this one; otherwise, only "None" can be selected for this item, that is, no unified high-level workspace build system is generated.

"Workspace Output Overwrite" refers to whether, when the entire project is regenerated, the old versions of the workspace files should be overwritten with the newly generated workspace files. If the old versions of the files already contain some modifications that the user wishes to keep, the user can choose not to overwrite; in fact, this is also the default option provided by the graphical interface configurator. The "Output Overwrite" option also applies to some other options in this chapter; at those options we will point out that they also have an "Output Overwrite" option, without further explaining the intent of that option.

6.5.1.3 Setting Chip Information

"Chip Information" includes the chip's "Architecture", "Category", and "Name", as well as the configuration items of the specific chip. Taking STM32F767IGT6 as an example, its architecture is ARMv7-M, so the architecture should be "A7M"; the type is "STM32F767", and the specific full chip name is "STM32F767IGT6". The reason for separating "Category" and "Name" into two setting items is that some chips are very similar in hardware configuration and differ only in package[11]; there is no need to separate them into different categories, because each category corresponds to an independent chip description file. Chip information

After the chip is selected, the coprocessors to be enabled, as well as some other parameters, must also be decided. "Coprocessor" generally refers to a floating-point processor; for example, the Cortex-M7 core of the STM32F767 has three coprocessor instruction sets, FPV4_SP, FPV5_SP, and FPV5_DP. The selected coprocessors will be enabled in the project; coprocessors not selected here are not allowed to be used in any process or virtual machine.

[11] Package differences are generally reflected in the suffix

"Other Parameters" follow no fixed rule, but generally include at least the timer timeout value setting. "Timer Timeout Value" refers to the number of counting periods of the RME kernel system tick timer; once this number of timer periods has elapsed, a clock interrupt is generated. In general, it is recommended to set this value so that interrupts occur less often than once per millisecond, so that the interrupt frequency is not too high and does not consume a large amount of system performance. Taking STM32F767 as an example, its main frequency is 216MHz, so the timer timeout value should be set to at least 216000. The maximum timer period can be relaxed all the way to the minimum time granularity required by the system: assuming that all delay times in the system are in units of 10 milliseconds, the timer timeout value can be set to 10 milliseconds to minimize system overhead.

6.5.1.4 Setting Off-Chip Memory

"Off-Chip Memory" refers to the external memory of the microcontroller. Some microcontrollers have an off-chip memory interface, and such off-chip memory can be accessed directly by memory access instructions just like on-chip memory. Since off-chip memory is not in the chip description file, if the project generator is to know of its existence, it must be declared separately here.

Some chips have no on-chip ROM/RAM; in this case all ROM/RAM is off-chip memory, which needs to be added separately at this location when creating the project so that it can be used in subsequent configuration.

6.5.1.5 Setting Shared Memory

"Shared Memory" refers to device memory that multiple protection domains have permission to access. Their base addresses and lengths, as well as the maximum possible permissions, must be declared in the shared memory list before they can be referenced in the shared memory lists of the various protection domains, so that each relevant protection domain is able to access that region of shared memory. The permissions at the time of reference must be a subset of the maximum permissions; this means that even for the same region of shared memory, different protection domains may have different access permissions to it.

The name of each shared memory region must be specified for later reference; except that the base address of device-segment memory must be specified manually and its alignment must be "Auto", the base addresses and alignments of the other categories of memory can all be decided by the project generator itself.

6.5.1.6 Setting the Page Table Mode

If memory permissions are not modified throughout the process[12], and dynamically mapped pages are not used[13], the "Enable Fixed Bare Page Table" switch can be turned on to remove the page table logic from the kernel; in this case, the page table is replaced by a fixed MPU data table in ROM, saving the space occupied by the page table data structures and management code.

6.5.2 Kernel Settings

This subsection introduces the recommended settings for the RME kernel.

6.5.2.1 Adding Memory

The kernel needs one code-segment memory and one data-segment memory to hold itself. The start address of the code segment is recommended to be placed at the beginning of the on-chip ROM address space, where the interrupt vector table is generally also stored. The size of the code segment only needs to be enough to hold the kernel and pass linking. The start address of the data segment is likewise recom-

[12] This is the case in the vast majority of situations where the project generator is used
[13] The memory declared by each protection domain

mended to be placed at the beginning of the on-chip RAM address space[14], and its size only needs to be sufficient to pass compilation. The data segment contains, in addition to the kernel's own fixed data structures, kernel objects, kernel stacks, and so on, but these structures have already been taken into account by the generated linker script; as long as linking passes, the amount of RAM allocated is appropriate.

To simplify the user's mental model and improve system reliability, in all official standard ports, RME always disables the memory protection unit during execution and thus uses the default memory mapping attributes[15]. The user must ensure that the code segment and data segment are usable under the default mapping and have appropriate permissions and cache attributes; this holds for the on-chip memory of all current microcontrollers[16].

6.5.2.2 Kernel Parameter Settings

"Kernel Stack Size" is the size of the stack used by the kernel. Setting this value to 1kB is generally sufficient; if the custom interrupt top half occupies too large a stack, it can be changed to 4kB.

"Extra Kernel Object Memory Size" is generally configured as 0, unless kernel objects need to be created manually; this is only needed for full customization.

"Kernel Memory Allocation Granularity" is the power of 2 used as the granularity when kernel objects are created. The larger the granularity, the coarser the allocation, but the smaller the space occupied by the kernel memory registry table that records whether a block of memory is allocated. Common kernel objects are mostly within 16 bytes.

For small systems, setting it to 3 is recommended, in which case kernel memory is allocated at a granularity of 8 bytes; for more complex situations that consume more kernel memory, it can be set to 4, in which case kernel memory is allocated at a granularity of 16 bytes to reduce the size of the kernel memory registry table; if the system is extremely complex, setting it to 5 or even 6 is also acceptable, in which case the granularity is 32 or 64.

6.5.2.3 Toolchain Settings

In "Toolchain Settings", "Toolchain", "Build System", and "Optimization Level" can be set. "Toolchain" and "Build System" are linked; not all combinations of toolchain and build system can be selected. Once a toolchain is selected, only build systems compatible with it will appear in the build system options, and once the project is generated, it cannot be changed later. "Optimization Level" can be freely set to an appropriate level.

6.5.2.4 File Path Settings

"File Path Settings" sets the relative paths of various files. These relative paths are relative to the subproject folders, which are created together when the project is created. If there are no special requirements, these paths can use the default values given in the project generator's graphical interface, and no changes are needed.

6.5.3 Virtual Machine Monitor Settings

This subsection introduces the recommended settings for the RVM virtual machine monitor.

6.5.3.1 Adding Memory

The virtual machine monitor also needs one code-segment memory and one data-segment memory to hold itself. However, unlike the kernel, the start addresses of its code segment and data segment are

[14] If the chip has tightly-coupled memory (Tightly-Coupled Memory, TCM), it is recommended to place it there to maximize performance

[15] That is, the mapping attributes used by classic bare-metal programs; for example, in ARMv7-M, on-chip memory starting at 0x2000000 is RWBXCS by default

[16] To facilitate bare-metal programming, manufacturers usually design the default memory mapping attributes this way

placed directly after the kernel, so only the sizes of its code segment and data segment can be configured; these sizes also only need to be sufficient to pass compilation, and as long as linking passes, the memory allocation is appropriate.

To adapt to various microcontroller architectures, the virtual machine monitor itself always maps the entire address space with cacheable and bufferable attributes, including those potential device regions. This is not a problem, because the virtual machine monitor does not access any device[17]. However, the virtual machine monitor needs to access a region of kernel memory that holds the flag bits of interrupt sources and event sources, and this kernel memory uses background region attributes in the kernel. Therefore, the user must ensure that the kernel memory is also cacheable and bufferable under the background region attributes.

6.5.3.2 Virtualization Parameter Settings

Unlike the kernel, the virtual machine monitor has three stack sizes to configure, namely "Startup Daemon Thread Stack Size", "Safety Daemon Thread Stack Size", and "Virtual Machine Daemon Thread Stack Size", which is because the virtual machine monitor consists of three threads. The "startup daemon thread" and the "safety daemon thread" have relatively simple responsibilities, and a 256B stack is generally sufficient; the virtual machine daemon thread, however, requires a stack of at least 512B. Where memory space is abundant, the stack budget can be doubled.

"Number of Extra Capability Table Slots" is generally configured as 0, unless kernel objects need to be created manually; this is only needed for full customization.

"Enable Idle Sleep" indicates that the virtual machine monitor's "startup daemon thread" puts the processor into sleep mode when idle. This item is generally set to Yes; if one does not wish to enter the low-power sleep state in order to minimize interrupt latency, "No" can be selected.

"Number of Virtualization Priorities" is generally configured as one machine word, because the processors supported by the RVM are at least 32-bit, and 32 priorities are sufficient for a typical virtualization system. If the virtualization feature is not used, this parameter can be configured as 0.

"Number of Virtual Machine Event Sources" and "Number of Virtual Machine Vector Mappings" can be configured as needed, however many are used. The number of event sources must be an integer multiple of the processor word size and cannot exceed 1024; the number of mappings has no such restriction. If the virtualization feature is not used, both can be configured as 0.

6.5.3.3 Toolchain Settings

This setting is no different from the kernel setting; please refer to the relevant section on kernel settings.

6.5.3.4 File Path Settings

This setting is not much different from the kernel setting; it is recommended to keep the default options unchanged.

6.5.4 Native Process Settings

This subsection introduces the recommended settings for native processes.

6.5.4.1 Adding Private Memory

A native process's private memory includes at least one code-segment memory, at least one data-segment memory, and any number of device-segment memories. At compile time, its code segment and data segment are always placed on the first code-segment memory and the first data-segment memory in the list, respectively. Except that the base address of device-segment memory must be specified manually and

[17] If the user actively inserts code that accesses devices into the virtual machine monitor's hooks, then memory access attributes must be noted; to reduce the user's mental burden, it is not recommended to access devices in the virtual machine monitor's hooks (such as filters for physical interrupt sources and event sources)

its alignment must be “Auto”, the base addresses and alignments of the other categories of memory can all be decided by the project generator itself. If no name is specified, the memory region is an anonymous memory region, and the macros for its base address and length are unavailable.

6.5.4.2 Adding Shared Memory

A native process may or may not contain shared memory. To reference shared memory, it is sufficient to fill in their names and their permissions in this process. For the settings of the base address, length, alignment, and maximum permissions of shared memory, see6.5.1.5 .

6.5.5 Virtual Machine Settings

This subsection introduces the recommended settings for virtual machines.

6.5.5.1 Adding Private Memory

Similar to a native process, a virtual machine’s private memory also includes at least one code-segment memory, at least one data-segment memory, and any number of device-segment memories. At compile time, its code segment and data segment are likewise always placed on the first code-segment memory and the first data-segment memory in the list, respectively. However, because the register set memory of the Usr thread inside the virtual machine is shared with the kernel, the access attributes of the virtual machine’s first data segment must be consistent with the default memory mapping attributes.

6.5.5.2 Adding Shared Memory

The method for adding shared memory is the same as for native processes.

7. Adding a New Chip to the RVM

7.1 Preparations for Adding a New Chip

Adding a new chip requires completing three parts of work: the RME header file, the RVM header file, and the chip description file. The original draft only gives the headings for these three aspects and does not provide specific steps; the content of this section is to be supplemented in a later version.

8. Porting the RVM to a New Architecture

8.1 Introduction

The process of porting the RVM to another architecture is called the porting of the RVM itself. Before porting, you must first confirm that the RME has already been ported to that architecture. RVM porting consists of two parts: the first part is the porting of the RVM itself, and the second part is the writing of the RVM guest library.

The intended readers of this chapter are developers who need to port the RVM to a new architecture.

8.2 Porting the User-Mode Library

Porting the RVM itself includes its type definitions, macro definitions, data structures and function bodies. The source files of the architecture-dependent parts of the RVM code are all placed under the corresponding architecture name in the Platform folder. For example, the folder name for the ARMv7-M architecture is Platform/A7M. Its header files are in Include/Platform/A7M, and so on for other architectures.

Each architecture contains one or more source files and one or more header files. When the user-mode library includes architecture-dependent header files, it always includes rvm_platform.h, which is a header file that includes the top-level header file of the corresponding architecture. When changing the compilation target platform of the RVM, the inclusion of the corresponding header files for different targets is achieved by modifying this header file. For example, to compile for the ARMv7-M architecture, this header file should include all the header files of the low-level function implementations for ARMv7-M.

Before porting, you can first browse the existing ports and look for the port of an architecture whose logical organization is closest to that of the target architecture. Then you can copy that port and use it as a template for modification.

8.2.1 Type Definitions

For each architecture/compiler, the first part that needs to be ported is the RVM type definitions. When defining the type definitions, you need to confirm one by one the number of processor word lengths corresponding to each primitive integer type of the compiler. The RVM has the following five type definitions in total:

Table 8-1 RVM type definitions

Type	Purpose
tid_t	The type of a thread ID. This type should be typedef'd to a signed integer equal in size to the processor word length. Example: typedef tid_t long;
ptr_t	The type of a pointer-sized integer. This type should be typedef'd to an unsigned integer equal in size to the processor word length. Example: typedef ptr_t unsigned long;
cnt_t	The type of a count variable. This type should be typedef'd to a signed integer equal in size to the processor word length. Example: typedef cnt_t long;
cid_t	The type of a capability ID. This type should be typedef'd to a signed integer equal in size to the processor word length. Example: typedef cid_t long;
ret_t	The type of a function return value. This type should be typedef'd to a signed integer equal in size to the processor word length. Example: typedef ret_t long;

To make writing low-level functions more convenient, it is recommended to use the following typedefs to define the fixed-width integer types that are frequently used. When defining these integer types, you also need to determine exactly how many machine words long the compiler's char, short, int, long, etc. are. Some compilers do not provide a 64-bit integer, in which case this type can be omitted.

Table 8-2 Other commonly used type definitions

Type	Meaning
s8_t	A signed 8-bit integer. For example: typedef char s8_t;
s16_t	A signed 16-bit integer. For example: typedef short s16_t;
s32_t	A signed 32-bit integer. For example: typedef int s32_t;
s64_t	A signed 64-bit integer. For example: typedef long s64_t;
u8_t	An unsigned 8-bit integer. For example: typedef unsigned char u8_t;
u16_t	An unsigned 16-bit integer. For example: typedef unsigned short u16_t;
u32_t	An unsigned 32-bit integer. For example: typedef unsigned int u32_t;
u64_t	A signed 64-bit integer. For example: typedef unsigned long u64_t;

8.2.2 Macros Related to RME Configuration Items and Filled In Automatically by the Project Generator

The macros listed below are related to the RME's kernel configuration. They need to remain consistent with the RME's compile-time options. The specific meaning of these macros is not introduced here; please refer to the corresponding macros in the RME manual. The project generator automatically maintains

the consistency of these macros, so in the header file you only need to list these macros and fill in a typical value for each option.

Table 8-3 Macros related to RME configuration items and filled in automatically by the project generator

Macro name	Corresponding RME macro
RVM_WORD_ORDER	RME_WORD_ORDER
RVM_REGION_FIXED	RME_PGT_RAW_USER
RVM_KOM_VA_START	RME_KOM_VA_START
RVM_KOM_VA_SIZE	RME_KOM_VA_SIZE
RVM_KOM_SLOT_ORDER	RME_KOM_SLOT_ORDER
RVM_PREEMPT_PRIO_NUM	RME_PREEMPT_PRIO_NUM
RVM_PHYS_VCT_NUM	RME_RVM_PHYS_VCT_NUM
RVM_PHYS_VCTF_BASE	RME_RVM_PHYS_VCTF_BASE
RVM_PHYS_VCTF_SIZE	RME_RVM_PHYS_VCTF_SIZE
RVM_VIRT_EVT_NUM	RME_RVM_VIRT_EVT_NUM
RVM_VIRT_EVTF_BASE	RME_RVM_VIRT_EVTF_BASE
RVM_VIRT_EVTF_SIZE	RME_RVM_VIRT_EVTF_SIZE
RVM_INIT_CPT_SIZE	RME_RVM_INIT_CPT_SIZE
RVM_CAP_BOOT_FRONT	RME_RVM_CAP_BOOT_FRONT
RVM_KOM_BOOT_FRONT	RME_RVM_KOM_BOOT_FRONT
RVM_CAP_DONE_FRONT	RME_RVM_CAP_DONE_FRONT
RVM_KOM_DONE_FRONT	RME_RVM_KOM_DONE_FRONT

In addition, there are some architecture-related configuration macros. These configuration macros also need to remain consistent with the relevant macros determined at compile time in the kernel, and the project generator will also fill them in automatically.

8.2.3 Macros Related to RME Configuration Items That Do Not Change

The macros listed below are related to the RME's kernel configuration. They need to remain consistent with the RME's compile-time options. The specific meaning of these macros is not introduced here; please refer to the corresponding macros in the RME manual.

Table 8-4 Macros related to RME configuration items that do not change

Macro name	Corresponding RME macro
EXTERN	EXTERN
RVM_THD_WORD_SIZE	This item should be filled with the fixed size of the thread kernel object, in units of word length
RVM_PGTBL_WORD_SIZE_NOM(X)	RME_PGTBL_WORD_SIZE_NOM(X)
RVM_PGTBL_WORD_SIZE_TOP(X)	RME_PGTBL_WORD_SIZE_TOP(X)

In addition, there are some architecture-related configuration macros. These configuration macros also need to remain consistent with the relevant macros determined at compile time in the kernel.

8.2.4 Macros Related to RVM Configuration Items and Filled In Automatically by the Project Generator

The macros listed below are related to the RVM's configuration and can be chosen flexibly according to the RVM's configuration state. They are filled in automatically by the project generator, so in the header file you only need to list these macros and fill in a typical value for each option.

Table 8-5 Macros related to the RVM configuration

Macro name	Purpose
RVM_PREEMPT_VPRIO_NUM	The number of virtual machine priorities of the RVM. Must be an integer multiple of the processor word length. Example: #define RVM_PREEMPT_VPRIO_NUM (32U)
RVM_VIRT_MAP_NUM	The maximum number of mappings from event sources or physical interrupt sources to virtual machines that the virtual machine monitor allows. This value must not be smaller than the number of event sources. Example: #define RVM_VIRT_MAP_NUM (32U)

In addition, there are some architecture-related configuration macros. The project generator fills them in automatically.

8.2.5 Data Structure Definitions

The only RVM data structure that needs to be ported is RVM_Regs. It is a structure for the processor's register set, used in the RVM to read and modify the register set of VM user threads. This structure should be exactly the same as the one in the RME.

8.2.6 Porting the Low-Level Assembly Functions

The RVM has to implement the following 6 assembly functions in total. These 6 assembly functions are mainly responsible for the startup of the RVM itself and system calls, etc. The list of these functions is as follows:

Table 8-6 Assembly functions involved in RVM porting

Function	Meaning
__RVM_Entry	The entry point of the RVM.
__RVM_Stub	A trampoline for thread creation and thread invocation.
RVM_MSB_GET	Obtain the position of the most significant bit of a word. It is actually a macro.
RVM_Inv_Act	Perform a thread invocation.
RVM_Inv_Ret	Perform a thread invocation return.
RVM_Svc	Call the RME's system call.

The specific implementation details and caveats of these functions are introduced one by one in the following paragraphs.

8.2.6.1 Implementation of __RVM_Entry

This function is the entry point of the RVM; it is responsible for initializing the RVM's C runtime library and jumping to the RVM's main function. This function must be linked at the head of the RVM image, and its link address must be the same as the entry of the Init thread determined when the RME kernel is compiled.

Table 8-7 Implementation of __RVM_Entry

Function prototype	void __RVM_Entry(void)
Meaning	The entry point of the RVM.
Return value	None.
Parameters	None.

8.2.6.2 Implementation of __RVM_Stub

This function is used to jump to the real entry when a thread runs for the first time or when a thread invocation runs. For typical architectures this function is not necessary; you only need to implement it as a direct return, and this function is not used either.

The usage scenarios of this function roughly include the following two types:

1. For architectures such as ARMv7-M where the PC is saved on the user stack when an interrupt exits, we cannot establish the return address by setting the thread's PC when creating a new thread. We can only simulate a return stack in memory and use the PC value in the return stack to set the address to which execution will return. Since this return stack may be used repeatedly[1], we do not want to modify the PC value contained in it frequently. Therefore, we set the PC value in the return stack to __RVM_Stub. In it, we assign to the PC the values placed in other registers when the thread was created, correctly pass its parameters to the thread or invocation, and formally jump to the thread's entry for execution.
2. For architectures such as RISC-V that require registers such as gp to be set in advance[2], if the thread's pc is set directly to the thread entry when creating a new thread, the gp register cannot be set in advance. Therefore, we set all entry points within a process to __RVM_Stub and adopt a strategy similar to ARMv7-M, placing the real entry point on the stack. When execution reaches __RVM_Stub, gp is set first, then the real entry point is taken from the stack, and then execution jumps to that entry point.

   In either case above, once __RVM_Stub is used, the corresponding stack initialization flow needs to be implemented in RVM_Stack_Init.

[1] For example, we create thread A, then destroy thread A, and then try to create thread B on the same stack

[2] A simple link-time optimization (Link-Time Optimization, LTO) enabled by default, "linker relaxation (Linker Relaxation)", requires this register to be set

Table 8-8 Implementation of __RVM_Stub

Function prototype	void __RVM_Stub(ptr_t Entry, ptr_t Param)
Meaning	Assist in jumping to the correct thread or invocation entry.
Return value	None.
Parameters	ptr_t Entry The entry of the thread. ptr_t Param The parameter of the thread.

8.2.6.3 Implementation of RVM_MSB_GET

This function returns the position of the most significant bit of the word. The most significant bit is defined as the position where the first "1" appears, counted starting from the LSB[3]. For example, if the number is the 32-bit 0x12345678, then the first "1" appears at bit 28, and this function returns 28.

Table 8-9 Implementation of RVM_MSB_GET

Function prototype	ptr_t _RVM_MSB_GET(ptr_t Val)
Meaning	Obtain the position of the most significant bit of an unsigned number equal in size to the processor word length, that is, the position of the first digit "1" counted from left to right in its binary representation.
Return value	ptr_t Returns the position of the first "1".
Parameters	ptr_t Val The number whose most significant bit position is to be computed.

Since this function needs to be implemented efficiently, its implementation differs greatly across processors. For architectures that provide a most-significant-bit instruction, simply implement this function in assembly and use that instruction. For architectures that provide a count-leading-zeros instruction (CLZ)[4], you can also use an assembly function to first compute the number of leading zeros, and then subtract this value from the processor word length minus 1 (in bits). For example, 0x12345678 has 3 leading zeros in total, and subtracting 3 from 31 gives 28.

For architectures that do not implement a specific instruction, the binary search method is recommended. First determine whether the high halfword of a word is 0; if it is not 0, perform a binary search within this high halfword; if it is 0, perform a binary search in the low halfword, until the position of the first "1" is determined. When binary search narrows down to 16 or 8 bits, a lookup table can be used to map directly to the relative position of the first "1" within those 16 or 8 bits, so that further binary search is unnecessary; then the position of the first "1" can be computed by combining the results of each binary search step. If you do not want to write this assembly function by hand, the RVM also provides a generic version, _RVM_MSB_Generic, and this macro can be defined directly as that function name.

8.2.6.4 Implementation of RVM_Inv_Act

This function performs a standard thread invocation. When implementing it, take care to place the parameters into the correct registers in order according to the kernel calling convention.

[3] The LSB counts as bit 0
[4] such as ARM

Table 8-10 Implementation of RVM_Inv_Act

Function prototype	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
Meaning	Perform a standard thread invocation.
Return value	ret_t The return value of the thread invocation system call itself.
Parameters	cid_t Cap_Inv The capability ID of the thread invocation capability. ptr_t Param The parameter of the thread invocation. ptr_t* Retval This parameter is used for output; it is a pointer to where the return value of the thread invocation is stored. If 0 (NULL) is passed in, it means that the return value returned by RVM_Inv_Ret is not to be received.

Since the RME kernel does not take responsibility for saving any registers other than the control-flow-related registers during a thread invocation, user mode must push them onto the stack. If user mode does not use some of these registers, those unused registers need not be pushed onto the stack. This function should be implemented as a standard invocation, that is, push all general-purpose register sets but do not push coprocessor register sets. The function should also determine whether the passed-in Retval pointer is 0 (NULL); if it is 0, the return value of the thread invocation is not output to the address pointed to by Retval.

8.2.6.5 Implementation of RVM_Inv_Ret

This function is used to return from a thread invocation. When implementing it, take care to place the parameters into the correct registers in order according to the kernel calling convention.

Table 8-11 Implementation of RVM_Inv_Ret

Function prototype	ret_t RVM_Inv_Ret(ptr_t Retval)
Meaning	Return from a thread invocation.
Return value	ret_t If successful, the function does not return; if it fails, it returns the return value of the thread invocation return system call.
Parameters	ptr_t Retval The return value of the thread invocation. If the caller's ptr_t* Retval is not 0 (NULL), then this parameter will be assigned the caller's *Retval.

8.2.6.6 Implementation of RVM_Svc

This function is used to perform the RME's system calls. When implementing it, take care to place the parameters into the correct registers in order according to the kernel calling convention.

Table 8-12 Implementation of RVM_Svc

Function prototype	ret_t RVM_Svc(ptr_t Op_Capid, ptr_t Arg1, ptr_t Arg2, ptr_t Arg3)
Meaning	Perform an RME system call.
Return value	ret_t The return value of the RME system call.

Function prototype	ret_t RVM_Svc(ptr_t Op_Capid, ptr_t Arg1, ptr_t Arg2, ptr_t Arg3)
Parameters	ptr_t Op_Capid Parameter P0 composed from the system call number N and the capability table capability ID C. ptr_t Arg1 The first parameter P1 of the system call. ptr_t Arg2 The second parameter P2 of the system call. ptr_t Arg3 The third parameter P3 of the system call.

8.2.7 Porting the C Language Functions

The RVM has five C language functions in total, covering low-level debug printing, thread stack initialization, entering a low-power state, page table setup, and so on. The list of these functions is as follows:

Table 8-13 C language functions involved in RVM porting

Function	Meaning
RVM_Putchar	Low-level printing function; prints one character to the console.
RVM_Stack_Init	Initialize a thread's thread stack.
RVM_Idle	Put the system into a sleep state.
RVM_Thd_Print_Exc	Print a string to the console according to the thread error reason.
RVM_Thd_Print_Reg	Print a thread's register set.

8.2.7.1 Implementation of RVM_Putchar

This operation prints one character to the console. This function is somewhat special; it does not need to be written by the user.

Table 8-14 Implementation of RVM_Putchar

Function prototype	void RVM_Putchar(char Char)
Meaning	Output one character to the console.
Return value	None.
Parameters	char Char The character to be output to the system console.

8.2.7.2 Implementation of RVM_Stack_Init

This operation initializes a thread's thread stack. Some architectures, such as ARMv7-M, require a thread's stack to be initialized before the thread runs. This function is used to initialize the contents of the stack. If the architecture does not require the stack to be initialized before the thread runs, this function can be implemented to simply return a suitable stack address.

Table 8-15 Implementation of RVM_Stack_Init

Function prototype	ptr_t RVM_Stack_Init(ptr_t Stack, ptr_t Size, ptr_t* Entry, ptr_t Stub)
Meaning	Initialize a thread's thread stack.

Function prototype	ptr_t RVM_Stack_Init(ptr_t Stack, ptr_t Size, ptr_t* Entry, ptr_t Stub)
Return value	ptr_t The stack pointer address used to initialize the thread's register set (that is, the one to be passed to RVM_Thd_Exec_Set).
Parameters	ptr_t Stack The start address of the stack area. ptr_t Size The size of the stack area. ptr_t* Entry The entry of the thread. Considering the scenarios where the entry point needs to be changed to __RVM_Stub, this parameter is also used for output. ptr_t Stub The entry location of __RVM_Stub. This parameter may not necessarily be used.

8.2.7.3 Implementation of RVM_Thd_Print_Exc

This operation prints the exact reason for the thread error, so as to facilitate debugging.

Table 8-16 Implementation of RVM_Thd_Print_Exc

Function prototype	void RVM_Thd_Print_Exc(cid_t Cap_Thd)
Meaning	Print the exact reason for the thread error.
Return value	None.
Parameters	cid_t Cap_Thd The capability ID of the thread. It can be a first-level encoding or a second-level encoding.

8.2.7.4 Implementation of RVM_Thd_Print_Reg

This operation prints all register sets of a thread. The specific register values need to be obtained using the standard kernel function call RVM_KERN_DEBUG_REG_MOD defined in the RME extension standard.

Table 8-17 Implementation of RVM_Thd_Print_Reg

Function prototype	ret_t RVM_Thd_Print_Reg(cid_t Cap_Thd)
Meaning	Print a thread's register set.
Return value	ret_t Returns 0 on success and a negative value on failure.
Parameters	cid_t Cap_Thd The capability ID of the thread. It can be a first-level encoding or a second-level encoding.

8.3 Porting the Project Generator

The content of this section has not yet been provided in the original manuscript and will be supplemented in a later version.

8.4 References for This Chapter

None

9. Appendix

9.1 Known Issues and Constraints of RVM

RVM has some known limitations and constraints. These limitations and constraints are described one by one below.

9.1.1 No Multicore Support

RVM does not currently support multicore microcontrollers. This is because the product logic of microcontrollers and microprocessors is completely different. The greatest significance of multicore microcontrollers lies in real-time performance and reliability: the functions of the different cores are statically allocated during system design, and once any hardware or software problem occurs in one core—even a permanent deadlock or core damage caused by a hardware fault—the other cores can continue to run unaffected. For this reason, the differences between the cores of multicore microcontrollers are often very large; some CPUs support an FPU while others do not; some CPUs support an MPU while others do not; and even the instruction sets of the cores are completely incompatible. Porting RME and RVM to such platforms is possible, but their ease of use would be greatly reduced. If you are interested in such an application, please contact us.

9.2 References for This Chapter

None