

EUKARON

M7M01 (Eukaron) R4T1 Microkernel General-Purpose Real-Time Operating System

Technical Reference Manual

Version R4T1 · September 10, 2024

EDI-Systems · www.rtospro.com

System Features

Mutatus · Mesozoa · Eukaron (R.VII)

M7M01 (Eukaron) R4T1

Microkernel General-Purpose Real-Time Operating System (Rev. 4 Typ. 1)

Technical Reference Manual

Minimal Yet Efficient Kernel

The smallest possible kernel. The scheduler runs at user level.

Broad Architectural Support

Supports symmetric or asymmetric multi-processor architectures. Supports uniform or non-uniform memory access architectures. All common hardware features are covered by kernel function calls.

Mature Capability Management

A capability-based third-generation lockless microkernel. Genuine separation of kernel mode and user mode.

Flexible Memory Management

Memory access is controlled through normal or path-compressed page tables. All memory mappings are managed at user level.

Advanced Thread and Process Model

Threads can be transferred between multiple processors. Threads can migrate between multiple processes. Other operating systems can be hosted with full or para-virtualization.

Advanced Synchronous and Asynchronous Communication Mechanism

Synchronous invocations are used for inter-process communication. Asynchronous signals are used to synchronize threads or handle interrupts.

Contents

System Features ii

List of Tables xi

List of Figures xx

Revision History xxi

1 Overview 1

1.1 Introduction . . . 1

1.1.1 Design Goals . . . 1

1.1.2 Software Copyright and License . . . 2

1.1.3 Glossary of Easily Confused Terms . . . 2

1.1.4 Major Reference Systems . . . 3

1.2 Foreword . . . 3

1.2.1 Fully Customized Systems . . . 4

1.2.2 Ultra-Lightweight Systems . . . 4

1.2.3 Basic Real-Time Systems . . . 5

1.2.4 Complex Application Systems . . . 5

1.2.5 Complex Real-Time Application Systems . . . 6

1.3 Performance Metrics of Real-Time Operating Systems and Their Components . . . 7

1.3.1 Kernel Size . . . 8

1.3.2 Execution Time, Worst-Case Execution Time and Their Jitter . . . 8

1.3.3 Interrupt Response Time, Worst-Case Interrupt Response Time and Their Jitter . . . 12

1.3.4 Realistic Interrupt Response Time, Worst-Case Realistic Interrupt Response Time and Their Jitter . . . 13

1.3.5 Input/Output Performance Metrics . . . 14

1.3.6 Virtualization Performance Metrics . . . 14

1.4 RME System Call Format . . . 15

1.4.1 System Call Basics . . . 15

1.4.2 Parameter Passing and Parameter Position Encoding . . . 15

1.4.3 Special Notes . . . 17

1.5 Manual Overview . . . 17

1.6 RME Architecture Overview . . . 17

1.7 Bibliography of This Chapter . . . 18

2 Capability Table and Capability Management 19

2.1 The Concept of Capability . . . 19

2.2 Capability Table Operations and Capability States . . . 19

2.2.1 Capability Types . . . 20

2.2.2 Capability Delegation Reference Count and Parent Capability . . . 20

2.2.3 Capability States . . . 21

2.2.4 Timestamp Counter and Multi-Core Parallelism Considerations. . . . 22
2.2.5 Capability Table Structure . . . 24
2.3 Capability Table Function List . . . 25
2.3.1 Capability Table Creation . . . 25
2.3.2 Capability Table Deletion . . . 27
2.3.3 Capability Delegation . . . 28
2.3.4 Capability Freezing . . . 30
2.3.5 Capability Removal . . . 31
2.4 Bibliography of This Chapter . . . 32
3 Page Tables and Memory Management 34
3.1 Memory Management Overview . . . 34
3.2 Page Table Operations and Structure . . . 34
3.2.1 Kernel Memory and User Memory . . . 35
3.2.2 Page Directory Attributes. . . . 36
3.2.3 Basic Page Directory Operations . . . 36
3.2.4 Implementation of a Normal Multi-Level Page Table Under the Memory Management Unit . . . 37
3.2.5 Implementation of a Path-Compressed Page Table Under the Memory Protection Unit . . . 38
3.3 Page Table Function List . . . 42
3.3.1 Creating a Page Directory . . . 43
3.3.2 Page Directory Deletion . . . 45
3.3.3 Page Mapping . . . 46
3.3.4 Page Removal . . . 47
3.3.5 Page Directory Construction. . . . 48
3.3.6 Page Directory Destruction . . . 49
3.4 Kernel Memory Function List . . . 51
3.5 Bibliography . . . 53
4 Process and Thread Management 54
4.1 Introduction to Processes and Threads. . . . 54
4.1.1 Process Overview . . . 54
4.1.2 Thread Overview . . . 54
4.2 Process Operations . . . 58
4.2.1 Process Creation and Deletion . . . 58
4.2.2 Changing the Capability Table or Page Table of a Process . . . 58
4.3 Thread Operations and States . . . 59
4.3.1 Overview of Thread Operations . . . 59
4.3.2 Thread Creation and Deletion . . . 60
4.3.3 Binding a Thread to a CPU and Unbinding It . . . 60
4.3.4 Setting the Execution Attributes of a Thread . . . 60
4.3.5 Allocating Timeslices to a Thread, Changing the Priority, and Running . . . 60
4.3.6 Overview of Thread State Transitions . . . 63
4.4 Process Function List. . . . 65
4.4.1 Creating a Process . . . 65
4.4.2 Deleting a Process . . . 67
4.4.3 Changing the Capability Table of a Process . . . 69
4.4.4 Changing the Page Table of a Process . . . 70

4.5 Thread Function List . . . 71
4.5.1 Creating a Thread . . . 72
4.5.2 Deleting a Thread . . . 74
4.5.3 Binding a Thread to a CPU . . . 75
4.5.4 Unbinding a Thread from a CPU . . . 77
4.5.5 Setting Thread Execution Attributes . . . 79
4.5.6 Changing Thread Priority . . . 80
4.5.7 Receiving a Thread's Scheduler Event . . . 81
4.5.8 Transferring Run Time Slice Between Threads . . . 82
4.5.9 Switching to a Thread . . . 84
4.6 Bibliography of This Chapter . . . 85

5 Synchronous and Asynchronous Communication Mechanisms 86
5.1 Overview of Synchronous and Asynchronous Communication . . . 86
5.1.1 Overview of Synchronous Communication . . . 86
5.1.2 Overview of Asynchronous Communication . . . 86
5.2 Synchronous Communication Operations . . . 87
5.3 Asynchronous Communication Operations . . . 88
5.4 List of Synchronous Communication Functions . . . 89
5.4.1 Creating a Synchronous Invocation . . . 90
5.4.2 Deleting a Synchronous Invocation . . . 91
5.4.3 Setting Synchronous Invocation Execution Properties . . . 93
5.4.4 Activating a Synchronous Invocation . . . 94
5.4.5 Returning from a Synchronous Invocation . . . 95
5.5 List of Asynchronous Communication Functions . . . 95
5.5.1 Creating an Asynchronous Signal Endpoint . . . 96
5.5.2 Deleting an Asynchronous Signal Endpoint . . . 97
5.5.3 Sending a Signal to an Asynchronous Signal Endpoint . . . 99
5.5.4 Receiving a Signal from an Asynchronous Signal Endpoint . . . 100
5.6 Bibliography of This Chapter . . . 101

6 Kernel Function Call Mechanism and Kernel Asynchronous Signal 102
6.1 Overview of the Kernel Call Mechanism . . . 102
6.2 Overview of Kernel Asynchronous Signal . . . 102
6.3 Kernel Call Mechanism Function List . . . 102
6.3.1 Initial Creation of the Kernel Call Mechanism . . . 103
6.3.2 Kernel Call Activation . . . 103
6.4 Kernel Asynchronous Signal Function List . . . 104
6.4.1 Initial Creation of Kernel Signal Endpoint . . . 104
6.4.2 Sending a Signal to a Kernel Signal Endpoint . . . 104
6.4.3 Receiving a Signal from a Kernel Signal Endpoint . . . 104
6.5 Bibliography of This Chapter . . . 104

7 Porting RME to New Architectures 105
7.1 Porting Overview . . . 105
7.2 Pre-Porting Checks . . . 105
7.2.1 Processor . . . 105
7.2.2 Compiler . . . 105
7.2.3 Assembler . . . 106

7.2.4 Linker 106
7.2.5 Debugger 106
7.3 Introduction to the Architecture-Dependent Parts of RME 106
7.3.1 Type Definitions 107
7.3.2 General Macro Definitions 107
7.3.3 Special Macro Definitions 113
7.3.4 Debug Switch Macro Definitions 114
7.3.5 Architecture-Dependent Structures 115
7.3.6 Low-Level Assembly Functions 115
7.3.7 System Interrupt Vectors 116
7.3.8 Other Low-Level Functions 116
7.4 Porting of Type Definitions and Low-Level Assembly Functions 118
7.4.1 Implementation of the Kernel Entry Point 119
7.4.2 Implementation of __RME_Int_Disable 119
7.4.3 Implementation of __RME_Int_Enable 120
7.4.4 Implementation of __RME_User_Enter 120
7.5 Porting of System Interrupt Vectors 120
7.5.1 Interrupt Push and Pop 120
7.5.2 Timer Interrupt Vector 121
7.5.3 System Call Interrupt Vector 122
7.5.4 Exception Handling Interrupt Vector 122
7.6 Writing Device Interrupt Vectors 123
7.6.1 Entry and Exit of Interrupt Vectors 124
7.6.2 Specific Kernel Functions Callable in Interrupt Vectors 124
7.7 Porting of the Kernel Debug Print Function 126
7.8 Porting of Processor Special Function Macro Definitions 126
7.8.1 Read-Acquire Operation 127
7.8.2 Write-Release Operation 127
7.8.3 Compare-and-Swap Atomic Operation 129
7.8.4 Fetch-and-Add Atomic Operation 129
7.8.5 Logical AND Atomic Operation 131
7.8.6 Getting the Most Significant Bit Position of a Word 132
7.9 Porting of Initialization and Boot Functions 132
7.9.1 Implementation of __RME_Lowlvl_Init 132
7.9.2 Implementation of __RME_Boot 133
7.10 Porting of Register Set Related Functions 151
7.10.1 Implementation of __RME_Svc_Param_Get 151
7.10.2 Implementation of __RME_Svc_Retval_Set 152
7.10.3 Implementation of __RME_Thd_Reg_Init 152
7.10.4 Implementation of __RME_Thd_Reg_Copy 153
7.10.5 Implementation of __RME_Thd_Reg_Print 153
7.10.6 Implementation of __RME_Inv_Reg_Save 154
7.10.7 Implementation of __RME_Inv_Reg_Restore 154
7.10.8 Implementation of __RME_Set_Inv_Retval 154
7.11 Porting of Coprocessor Register Set Related Functions 155
7.11.1 Implementation of __RME_Thd_Cop_Check 155
7.11.2 Implementation of __RME_Thd_Cop_SIze 155
7.11.3 Implementation of __RME_Thd_Cop_Init 156
7.11.4 Implementation of __RME_Thd_Cop_Swap 156

7.12 Porting of Kernel Function Call Functions. 158
7.12.1 Implementation of the High-Precision Timer System 158
7.12.2 Implementation of Inter-Processor Interrupts 158
7.12.3 Implementation of Cache Operations. 159
7.13 Porting of Page Table Related Functions 159
7.13.1 Implementation of __RME_Pgt_Set 159
7.13.2 Implementation of __RME_Pgt_Kom_Init 160
7.13.3 Implementation of __RME_Pgt_Check 160
7.13.4 Implementation of __RME_Pgt_Init 161
7.13.5 Implementation of __RME_Pgt_Del_Check 161
7.13.6 Implementation of __RME_Pgt_Page_Map 162
7.13.7 Implementation of __RME_Pgt_Page_Unmap 162
7.13.8 Implementation of __RME_Pgt_Pgdir_Map. 163
7.13.9 Implementation of __RME_Pgt_Pgdir_Unmap. 164
7.13.10Implementation of __RME_Pgt_Lookup 165
7.13.11 Implementation of __RME_Pgt_Walk 165
7.14 Other Function Descriptions 167
7.14.1 Variable Clearing 167
7.14.2 Comparing Two Memory Blocks 167
7.14.3 Copying a Memory Block 167
7.14.4 Printing a Signed Integer. 168
7.14.5 Printing an Unsigned Integer 168
7.14.6 Printing a String. 169
7.15 Bibliography of This Chapter 169

8 Kernel Call Implementation Specifications 170
8.1 Overview of Kernel Call Implementation Specifications 170
8.2 Page Table Kernel Call Implementation Specifications (“P” Standard Extension) . . 171
8.2.1 Four Recommended Forms of Page Tables 172
8.2.2 Clearing All Cache of a Page Table on the Current CPU 174
8.2.3 Clearing a Certain Line of the Cache of a Page Table on the Current CPU 175
8.2.4 Setting the ASID of a Page Table 176
8.2.5 Locking a Certain Page in the TLB 177
8.2.6 Querying or Modifying the Attributes of a Certain Page 177
8.3 Interrupt Controller Kernel Call Implementation Specifications (“I” Standard Extension) 179
8.3.1 Getting or Setting the State of a Certain Local Interrupt Source 179
8.3.2 Getting or Setting the State of a Global Interrupt Source 180
8.3.3 Triggering the Local Interrupt Source of a CPU 180
8.3.4 Triggering the Local Event Source of a CPU 181
8.4 Cache and Prefetching Kernel Call Implementation Specifications (“C” Standard Extension). 182
8.4.1 Getting or Setting the Cache State 183
8.4.2 Getting or Setting the Cache Configuration. 184
8.4.3 Cache Maintenance 184
8.4.4 Locking the Cache 185
8.4.5 Getting or Setting the Prefetch State 186

8.5 Hardware Hotplug Kernel Call Implementation Specifications ("H" Standard Extension) . . . 187
8.5.1 Getting or Setting the State of a Physical Processor Package . . . 187
8.5.2 Getting or Setting the State of a Logical Processor . . . 188
8.5.3 Getting or Setting the State of a Physical Memory Package . . . 188
8.6 Clock and Voltage Kernel Call Implementation Specifications ("F" Standard Extension) . . . 190
8.6.1 Putting the Processor to Sleep . . . 190
8.6.2 Restarting the Entire System . . . 191
8.6.3 Powering Off the Entire System . . . 191
8.6.4 Getting or Setting Voltage Parameters . . . 192
8.6.5 Getting or Setting Frequency Parameters . . . 192
8.6.6 Getting or Setting Power States . . . 193
8.6.7 Getting or Setting Safety Protection States . . . 194
8.7 System Monitoring Kernel Call Implementation Specifications ("M" Standard Extension) . . . 195
8.7.1 Getting CPU Functionality Support Statuses . . . 195
8.7.2 Getting or Setting Performance Monitor Configurations . . . 196
8.7.3 Getting or Setting the Value of a Counting Performance Counter . . . 196
8.7.4 Getting or Setting the Value of a Cycle Performance Counter . . . 197
8.7.5 Getting or Setting the Value of a Data Performance Counter . . . 199
8.7.6 Getting or Setting the Value of a Physical Performance Counter . . . 199
8.7.7 Getting or Setting the Value of a Cumulative Performance Counter . . . 200
8.8 Full Virtualization Kernel Call Implementation Specifications ("V" Standard Extension) . . . 202
8.8.1 Creating a Virtual Machine . . . 202
8.8.2 Deleting a Virtual Machine . . . 204
8.8.3 Setting the Page Table Used by a Virtual Machine . . . 205
8.8.4 Getting or Setting the Attributes of a Virtual Machine . . . 206
8.8.5 Creating a Virtual CPU . . . 207
8.8.6 Binding a Virtual CPU to a Virtual Machine . . . 209
8.8.7 Unbinding a Virtual CPU from a Virtual Machine . . . 209
8.8.8 Deleting a Virtual CPU . . . 210
8.8.9 Getting or Setting the Registers of a Virtual CPU . . . 211
8.8.10 Running a Virtual CPU . . . 212
8.9 Security Monitor Kernel Call Implementation Specifications ("S" Standard Extension) . . . 213
8.9.1 Two Typical Security Monitor Implementations . . . 214
8.9.2 Creating a Secure Zone . . . 215
8.9.3 Deleting a Secure Zone . . . 216
8.9.4 Getting or Setting the Attributes of a Secure Zone . . . 217
8.9.5 Creating a Secure Zone Thread . . . 218
8.9.6 Deleting a Secure Zone Thread . . . 219
8.9.7 Getting or Setting the Attributes of a Secure Zone Thread . . . 220
8.9.8 Running a Secure Zone Thread . . . 221
8.10 Debug and Trace Kernel Call Implementation Specifications ("D" Standard Extension) . . . 222
8.10.1 Debug Printing . . . 222

8.10.2 Getting or Setting the Register Set of a Thread . . . 223
8.10.3 Getting or Setting the Invocation Register Set of a Thread . . . 224
8.10.4 Getting the Exception Register Set of a Thread . . . 225
8.10.5 Get or Set Debug Mode . . . 225
8.10.6 Get or Set Instruction Debug Breakpoint State . . . 226
8.10.7 Get or Set Data Debug Breakpoint State . . . 227
8.11 Chapter Bibliography . . . 228

9 Appendix 229
9.1 Implementation of Special Kernel Features . . . 229
9.1.1 Implementation of CPU Hotplug . . . 229
9.1.2 Implementation of Memory Hotplug . . . 229
9.1.3 Implementation of the Separation Kernel . . . 229
9.1.4 Implementation of a Tickless Kernel . . . 230
9.1.5 Implementation of Processor Sleep . . . 232
9.1.6 Saving and Restoring the Coprocessor Context . . . 233
9.1.7 Implementation of Direct User-Mode Page Table Management . . . 238
9.2 Recommended Handling Scheme for the Memory Protection Unit . . . 239
9.2.1 Influence of the Memory Protection Unit . . . 239
9.2.2 Region Organization of the Memory Protection Unit . . . 240
9.2.3 Region Format of the Memory Protection Unit . . . 240
9.3 Factors Limiting Multicore Scalability in RME . . . 241
9.3.1 Update of RME_TIMESTAMP . . . 241
9.3.2 Multiple Cores Simultaneously Creating Kernel Objects in the Same Kernel Memory Region . . . 241
9.3.3 Multiple Cores Simultaneously Sending Signals to the Same Signal Endpoint . . . 242
9.3.4 Multiple Cores Simultaneously Performing Capability Transfer or Removal on the Same Root Capability . . . 242
9.4 Factors Limiting RME on 32-Bit Systems . . . 242
9.4.1 A Thread's Thread ID . . . 242
9.4.2 Calculation of the Quiescence Period . . . 242
9.5 Known Potential Covert Channels in RME . . . 243
9.5.1 Covert Storage Channels . . . 243
9.5.2 Covert Timing Channels . . . 243
9.6 Memory Consistency Model . . . 243
9.7 Known Factors Affecting Real-Time Performance in RME . . . 244
9.7.1 Overly Long Capability Table . . . 244
9.7.2 Page Faults . . . 244
9.7.3 Livelock Caused by Linked Load/Store Conditional . . . 245
9.8 RME Deviations from the C Language Standard and the MISRA Standard . . . 245
9.8.1 C Undefined Behavior: Type-Punning Pointer Aliasing May Exist and Does Not Follow the Strict Aliasing Rule . . . 245
9.8.2 C Undefined Behavior: The Length of a Byte is Always 8 Bits, and the Size of a char Type is Always One Byte . . . 246
9.8.3 C Undefined Behavior: The Actual Representation of a Pointer is Always a Machine-Recognizable Address . . . 246
9.8.4 C Undefined Behavior: The Length of a Pointer is Always Less Than or Equal to One Machine Word . . . 247
9.8.5 C Undefined Behavior: Coercing a Pointer to an Unsigned Integer Value of One Machine Word Always Leaves Its Actual Representation Unchanged . . . 247

9.8.6 C Undefined Behavior: The Length of External Identifiers is Extended to 31 Characters 247
9.8.7 C Undefined Behavior: Modifications to Non-volatile Variables are Always Preserved .247
9.8.8 Description of MISRA-C Deviations 248
9.9 Chinese-English Terminology Quick Reference Table 249
9.10 Bibliography for This Chapter. 250

List of Tables

1-1 The First Case of Intra-Process Thread Switching . . . 9
1-2 The Second Case of Intra-Process Thread Switching . . . 9
1-3 The First Case of Inter-Process Thread Switching . . . 9
1-4 The Second Case of Inter-Process Thread Switching . . . 10
1-5 Intra-Process Inter-Thread Synchronous Communication . . . 10
1-6 Inter-Process Synchronous Communication of Traditional Operating Systems . . . 10
1-7 Inter-Process Synchronous Communication with Thread Migration Techniques . . . 11
1-8 Intra-Thread Asynchronous Communication . . . 11
1-9 Intra-Process Inter-Thread Asynchronous Communication . . . 11
1-10 Inter-Process Asynchronous Communication . . . 12
1-11 Page table operations . . . 12
1-12 Interrupt response time . . . 13
1-13 Realistic interrupt response time . . . 14
1-14 System call parameter position encoding . . . 15
1-15 List of RME system calls . . . 16

2-1 Types of Capabilities . . . 20
2-2 Encoding of Capability Numbers . . . 24
2-3 Capability Table Related Kernel Functions . . . 25
2-4 Capability Table Operation Flags . . . 25
2-5 Parameters Required for Capability Table Creation . . . 26
2-6 Possible Return Values for Capability Table Creation . . . 27
2-7 Parameters Required for Capability Table Deletion . . . 27
2-8 Possible Return Values for Capability Table Deletion . . . 28
2-9 Parameters Required for Capability Delegation . . . 28
2-10 Possible Return Values for Capability Delegation . . . 29
2-11 Parameters Needed for Capability Freezing . . . 31
2-12 Possible Return Values for Capability Freezing . . . 31
2-13 Parameters Needed for Capability Removal . . . 32
2-14 Possible Return Values for Capability Removal . . . 32

3-1 Page Directory Terminology Definitions . . . 34
3-2 Standard Page Access Attributes . . . 35
3-3 Basic Page Directory Operations . . . 36
3-4 Features of Common MMUs . . . 38
3-5 Features of Common MPUs . . . 39
3-6 Restrictions of Immediately Updating MPU Metadata When Updating the Page Table . . . 40

3-7 Restrictions of Updating MPU Metadata on Page Fault . . . 41
3-8 Page Table Related Kernel Functions . . . 42
3-9 Page Table Operation Flags . . . 42
3-10 Parameters Required for Creating a Page Directory . . . 43
3-11 Possible Return Values for Creating a Page Directory . . . 44
3-12 Parameters Needed for Page Directory Deletion . . . 45
3-13 Possible Return Values for Page Directory Deletion . . . 45
3-14 Parameters Needed for Page Mapping . . . 46
3-15 Possible Return Values for Page Mapping . . . 46
3-16 Parameters Needed for Page Removal . . . 47
3-17 Possible Return Values for Page Removal . . . 48
3-18 Parameters Needed for Page Directory Construction . . . 48
3-19 Possible Return Values for Page Directory Construction . . . 49
3-20 Parameters Needed for Page Directory Destruction . . . 49
3-21 Possible Return Values for Page Directory Destruction . . . 50
3-22 Kernel Memory Operation Flags . . . 51
3-23 Parameters Needed for Kernel Memory Capability Delegation . . . 51
3-24 Specific Meaning of P3 (Flag) in Kernel Memory Capability Delegation . . . 52
3-25 Specific Meaning of N:C (Ext_Flag) in Kernel Memory Capability Delegation . . . 53

4-1 States of Threads . . . 59
4-2 Categories of Threads . . . 61
4-3 Meaning of Timeslice Macro Definitions . . . 61
4-4 Rules of Normal Timeslice Transfers . . . 61
4-5 Rules of Infinite Timeslice Transfers . . . 62
4-6 Rules of Reclaim Timeslice Transfers . . . 62
4-7 Meaning of the Labels in the Thread State Transition Diagram . . . 64
4-8 Process-Related Kernel Functions . . . 65
4-9 Process Operation Flags . . . 65
4-10 Parameters Needed for Process Creation . . . 65
4-11 Possible Return Values for Process Creation . . . 67
4-12 Parameters Needed for Process Deletion . . . 67
4-13 Possible Return Values for Process Deletion . . . 68
4-14 Parameters Needed for Changing the Capability Table of a Process . . . 69
4-15 Return Values Needed for Changing the Capability Table of a Process . . . 69
4-16 Parameters Needed for Changing the Page Table of a Process . . . 70
4-17 Possible Return Values for Changing the Page Table of a Process . . . 70
4-18 Thread Related Kernel Functions . . . 71
4-19 Thread Operation Flags . . . 71
4-20 Parameters Needed for Thread Creation . . . 72
4-21 Possible Return Values for Thread Creation . . . 73
4-22 Parameters Needed for Thread Deletion . . . 74
4-23 Possible Return Values for Thread Deletion . . . 74
4-24 Parameters Needed for Binding a Thread to a CPU . . . 75
4-25 Possible Return Values for Binding a Thread to a CPU . . . 76
4-26 Parameters Needed for Unbinding a Thread from a CPU . . . 77

4-27 Possible Return Values for Unbinding a Thread from a CPU . . . 77
4-28 Parameters Needed for Setting Thread Execution Attributes . . . 79
4-29 Possible Return Values for Setting Thread Execution Attributes . . . 79
4-30 Parameters Needed for Changing Thread Priority . . . 80
4-31 Possible Return Values for Changing Thread Priority . . . 80
4-32 Parameters Needed for Receiving a Thread's Scheduler Event . . . 81
4-33 Possible Return Values for Receiving a Thread's Scheduler Event . . . 82
4-34 Parameters Needed for Transferring Run Time Slice Between Threads . . . 82
4-35 Possible Return Values for Transferring Run Time Slice Between Threads . . . 83
4-36 Parameters Needed for Switching to a Thread . . . 84
4-37 Possible Return Values for Switching to a Thread . . . 85

5-1 Four Receiving Options of Endpoint Signals . . . 89
5-2 Kernel Functions Related to Synchronous Communication . . . 89
5-3 Synchronous Invocation Operation Flags . . . 89
5-4 Parameters Needed for Synchronous Invocation Creation . . . 90
5-5 Possible Return Values for Synchronous Invocation Creation . . . 91
5-6 Parameters Needed for Synchronous Invocation Deletion . . . 91
5-7 Possible Return Values for Synchronous Invocation Deletion . . . 92
5-8 Parameters Needed for Setting Synchronous Invocation Execution Properties . . . 93
5-9 Possible Return Values for Setting Synchronous Invocation Execution Properties . . . 93
5-10 Parameters Needed for Activating a Synchronous Invocation . . . 94
5-11 Possible Return Values for Activating a Synchronous Invocation . . . 94
5-12 Parameters Needed for Returning from a Synchronous Invocation . . . 95
5-13 Possible Return Values for Returning from a Synchronous Invocation . . . 95
5-14 Kernel Functions Related to Asynchronous Communication . . . 95
5-15 Signal Endpoint Operation Flags . . . 95
5-16 Parameters Needed for Signal Endpoint Creation . . . 96
5-17 Possible Return Values for Asynchronous Signal Endpoint Creation . . . 97
5-18 Parameters Needed for Asynchronous Signal Endpoint Deletion . . . 97
5-19 Possible Return Values for Asynchronous Signal Endpoint Deletion . . . 98
5-20 Parameters Needed for Sending a Signal to an Asynchronous Signal Endpoint . . . 99
5-21 Possible Return Values for Sending a Signal to an Asynchronous Signal Endpoint . . . 100
5-22 Parameters Needed for Receiving a Signal from an Asynchronous Signal Endpoint . . . 100
5-23 Possible Return Values for Receiving a Signal from an Asynchronous Signal Endpoint . . . 101

6-1 Kernel Call Related Kernel Functions . . . 102
6-2 Kernel Call Operation Flags . . . 103
6-3 Parameters Needed for Kernel Call Activation . . . 103
6-4 Possible Return Values for Kernel Call Activation . . . 103

7-1 Overview of Type Definitions . . . 107
7-2 Overview of General Macro Definitions . . . 107
7-3 Overview of Special Macro Definitions . . . 113
7-4 Overview of Debug Switch Macro Definitions . . . 114
7-5 Overview of Architecture-Dependent Structures . . . 115
7-6 Overview of Low-Level Assembly Functions . . . 115

7-7 Overview of System Interrupt Vectors . . . 116
7-8 Overview of Kernel Debug Print Functions . . . 116
7-9 Overview of Processor Special Function Macro Definitions . . . 116
7-10 Overview of Initialization and Boot Functions . . . 116
7-11 Overview of Register Set Related Functions . . . 117
7-12 Overview of Functions Related to the Coprocessor Register Set . . . 117
7-13 Overview of Kernel Function Call Functions . . . 117
7-14 Overview of Page Table Related Functions . . . 118
7-15 Overview of Commonly Used Types . . . 118
7-16 Implementation of __RME_Int_Disable . . . 119
7-17 Implementation of __RME_Int_Enable . . . 120
7-18 Implementation of __RME_User_Enter . . . 120
7-19 Implementation of the Timer Interrupt Vector . . . 121
7-20 Implementation of the System Call Interrupt Vector . . . 122
7-21 Implementation of the Exception Interrupt Vector . . . 122
7-22 Kill the Current Thread . . . 122
7-23 Obtaining the Page Table of the Current Process of a Thread . . . 123
7-24 Prototype of Interrupt Functions Written in C . . . 124
7-25 Parameters Required for Sending a Signal to a Kernel Endpoint . . . 125
7-26 Possible Return Values for Sending a Signal to a Kernel Endpoint . . . 125
7-27 Parameters Required for Performing a Thread Switch on Interrupt Vector Exit . . . 126
7-28 Implementation of __RME_Putchar . . . 126
7-29 Implementation of RME_READ_ACQUIRE . . . 127
7-30 Implementation of RME_WRITE_RELEASE . . . 128
7-31 Implementation of RME_COMP_SWAP . . . 129
7-32 Implementation of RME_FETCH_ADD . . . 130
7-33 Implementation of RME_FETCH_AND . . . 131
7-34 Implementation of RME_MSB_GET . . . 132
7-35 Implementation of __RME_Lowlvl_Init . . . 133
7-36 Implementation of __RME_Boot . . . 133
7-37 Overview of the Functions That __RME_Boot Needs to Call . . . 133
7-38 Parameters Needed for Initializing the Kernel Object Table at Boot Time . . . 135
7-39 Possible Return Values for Initializing the Kernel Object Table at Boot Time . . . 135
7-40 Parameters Needed for Initializing the Local Storage of Each Processor at Boot Time . . . 135
7-41 Parameters Needed for Creating the Initial Capability Table at Boot Time . . . 136
7-42 Possible Return Values for Creating the Initial Capability Table at Boot Time . . . 136
7-43 Parameters Needed for Creating Other Capability Tables at Boot-time . . . 137
7-44 Possible Return Values for Creating Other Capability Tables at Boot-time . . . 137
7-45 Parameters Needed for Creating Page Directories at Boot-time . . . 138
7-46 Possible Return Values for Creating Page Directories at Boot-time . . . 139
7-47 Parameters Needed for Constructing Page Directories at Boot-time . . . 140
7-48 Possible Return Values for Constructing Page Directories at Boot-time . . . 141
7-49 Parameters Needed for Adding Pages to Page Directories at Boot-time . . . 141
7-50 Possible Return Values for Adding Pages to Page Directories at Boot-time . . . 142
7-51 Parameters Needed for Creating the First Process at Boot-time . . . 143
7-52 Possible Return Values for Creating the First Process at Boot-time . . . 144

7-53 Parameters Needed for Creating Kernel Function Capability at Boot-time . . . 145
7-54 Possible Return Values for Creating Kernel Function Capability at Boot-time . . . 145
7-55 Parameters Needed for Creating Kernel Memory Capability at Boot-time . . . 146
7-56 Possible Return Values for Creating Kernel Memory Capability at Boot-time . . . 147
7-57 Parameters Needed for Creating Kernel Signal Endpoints at Boot-time . . . 147
7-58 Possible Return Values for Creating Kernel Signal Endpoints at Boot-time . . . 148
7-59 Parameters Needed for Creating Initial Threads at Boot-time . . . 149
7-60 Possible Return Values for Creating Initial Threads at Boot-time . . . 151
7-61 Implementation of __RME_Svc_Param_Get . . . 152
7-62 Implementation of __RME_Svc_Retval_Set . . . 152
7-63 Implementation of __RME_Thd_Reg_Init . . . 152
7-64 Implementation of __RME_Thd_Reg_Copy . . . 153
7-65 Implementation of __RME_Thd_Reg_Print . . . 153
7-66 Implementation of __RME_Inv_Reg_Save . . . 154
7-67 Implementation of __RME_Inv_Reg_Restore . . . 154
7-68 Implementation of __RME_Set_Inv_Retval . . . 154
7-69 Implementation of __RME_Thd_Cop_Check . . . 155
7-70 Implementation of __RME_Thd_Cop_Size . . . 155
7-71 Implementation of __RME_Thd_Cop_Init . . . 156
7-72 Implementation of __RME_Thd_Cop_Swap . . . 156
7-73 Implementation of Kernel Function Call Functions . . . 158
7-74 Implementation of __RME_Pgt_Set . . . 159
7-75 Implementation of __RME_Pgt_Kom_Init . . . 160
7-76 Implementation of __RME_Pgt_Check . . . 160
7-77 Implementation of __RME_Pgt_Init . . . 161
7-78 Implementation of __RME_Pgt_Del_Check . . . 161
7-79 Implementation of __RME_Pgt_Page_Map . . . 162
7-80 Implementation of __RME_Pgt_Page_Unmap . . . 163
7-81 Implementation of __RME_Pgt_Pgdir_Map . . . 163
7-82 Implementation of __RME_Pgt_Pgdir_Unmap . . . 164
7-83 Implementation of __RME_Pgt_Lookup . . . 165
7-84 Implementation of __RME_Pgt_Walk . . . 166
7-85 Parameters Needed for Variable Clearing . . . 167
7-86 Parameters Needed for Comparing Two Memory Blocks . . . 167
7-87 Parameters Needed for Copying a Memory Block . . . 168
7-88 Parameters Needed for Printing a Signed Integer . . . 168
7-89 Parameters Needed for an Unsigned Integer . . . 168
7-90 Parameters Needed for Printing a String . . . 169

8-1 Overview of Page Table Kernel Calls . . . 171
8-2 Overview of the Four Recommended Forms of Page Tables . . . 172
8-3 Kernel Calls That a Multi-Core Memory Protection Unit Needs to Implement . . . 173
8-4 Kernel Calls That a Hardware-Filled Memory Management Unit Needs to Implement . . . 174
8-5 Parameters Needed to Clear All Cache of a Page Table on the Current CPU . . . 175
8-6 Suggested Return Value Implementation for Clearing All Cache of a Page Table on the Current CPU . . . 175

8-7 Parameters Needed to Clear a Certain Line of the Cache of a Page Table on the Current CPU 175
8-8 Suggested Return Value Implementation for Clearing a Certain Line of the Cache of a Page Table on the Current CPU . . . 176
8-9 Parameters Needed to Set the ASID of a Page Table . . . 176
8-10 Suggested Return Value Implementation for Setting the ASID of a Page Table . . . 177
8-11 Parameters Needed to Lock a Certain Page in the TLB . . . 177
8-12 Suggested Return Value Implementation for Locking a Certain Page in the TLB . . . 177
8-13 Parameters Needed to Query or Modify the Attributes of a Certain Page . . . 178
8-14 Suggested Return Value Implementation for Querying or Modifying the Attributes of a Certain Page . . . 178
8-15 Overview of Interrupt Controller Kernel Calls . . . 179
8-16 Parameters Needed to Get or Set the State of a Certain Local Interrupt Source . . . 179
8-17 Recommended Return Value Implementation for Getting or Setting the State of a Local Interrupt Source . . . 180
8-18 Parameters Needed for Getting or Setting the State of a Global Interrupt Source . . . 180
8-19 Recommended Return Value Implementation for Getting or Setting the State of a Global Interrupt Source . . . 180
8-20 Parameters Needed for Triggering the Local Interrupt Source of a CPU . . . 181
8-21 Recommended Return Value Implementation for Triggering the Local Interrupt Source of a CPU . . . 181
8-22 Parameters Needed for Triggering the Local Event Source of a CPU . . . 181
8-23 Recommended Return Value Implementation for Triggering the Local Event Source of a CPU . . . 182
8-24 Overview of Cache and Prefetching Kernel Calls . . . 183
8-25 Parameters Needed for Getting or Setting the Cache State . . . 183
8-26 Recommended Return Value Implementation for Getting or Setting the Cache State . . . 183
8-27 Parameters Needed for Getting or Setting the Cache Configuration . . . 184
8-28 Recommended Return Value Implementation for Getting or Setting the Cache Configuration 184
8-29 Parameters Needed for Cache Maintenance . . . 184
8-30 Recommended Return Value Implementation for Cache Maintenance . . . 185
8-31 Parameters Needed for Locking the Cache . . . 185
8-32 Recommended Return Value Implementation for Locking the Cache . . . 185
8-33 Parameters Needed for Getting or Setting the Prefetch State . . . 186
8-34 Recommended Return Value Implementation for Getting or Setting the Prefetch State . . . 186
8-35 Overview of Hardware Hotplug Kernel Calls . . . 187
8-36 Parameters Needed for Getting or Setting the State of a Physical Processor Package . . . 187
8-37 Recommended Return Value Implementation for Getting or Setting the State of a Physical Processor Package . . . 188
8-38 Parameters Needed for Getting or Setting the State of a Logical Processor . . . 188
8-39 Recommended Return Value Implementation for Getting or Setting the State of a Logical Processor . . . 188
8-40 Parameters Needed for Getting or Setting the State of a Physical Memory Package . . . 189
8-41 Recommended Return Value Implementation for Getting or Setting the State of a Physical Memory Package . . . 189
8-42 Overview of Clock and Voltage Kernel Calls . . . 190
8-43 Putting the Processor to Sleep . . . 190
8-44 Recommended Return Value Implementation for Putting the Processor to Sleep . . . 191

8-45 Restarting the Entire System . . . 191
8-46 Recommended Return Value Implementation for Restarting the Entire System . . . 191
8-47 Powering Off the Entire System . . . 191
8-48 Recommended Return Value Implementation for Powering Off the Entire System . . . 192
8-49 Parameters Needed for Getting or Setting Voltage Parameters . . . 192
8-50 Recommended Return Value Implementation for Getting or Setting Voltage Parameters . . . 192
8-51 Parameters Needed for Getting or Setting Frequency Parameters . . . 192
8-52 Recommended Return Value Implementation for Getting or Setting Frequency Parameters 193
8-53 Parameters Needed for Getting or Setting Power States . . . 193
8-54 Recommended Return Value Implementation for Getting or Setting Power States . . . 193
8-55 Parameters Needed for Getting or Setting Safety Protection States . . . 194
8-56 Recommended Return Value Implementation for Getting or Setting Safety Protection States 194
8-57 Overview of System Monitoring Kernel Calls . . . 195
8-58 Parameters Needed for Getting CPU Functionality Support Statuses . . . 195
8-59 Recommended Return Value Implementation for Getting CPU Functionality Support Statuses . . . 196
8-60 Parameters Needed for Getting or Setting Performance Monitor Configurations . . . 196
8-61 Recommended Return Value Implementation for Getting or Setting Performance Monitor Configurations . . . 196
8-62 Parameters Needed for Getting or Setting the Value of a Counting Performance Counter . 197
8-63 Recommended Return Value Implementation for Getting or Setting the Value of a Counting Performance Counter . . . 197
8-64 Parameters Needed for Getting or Setting the Value of a Cycle Performance Counter . . . 198
8-65 Recommended Return Value Implementation for Getting or Setting the Value of a Cycle Performance Counter . . . 198
8-66 Parameters Needed for Getting or Setting the Value of a Data Performance Counter . . . 199
8-67 Recommended Return Value Implementation for Getting or Setting the Value of a Data Performance Counter . . . 199
8-68 Parameters Needed for Getting or Setting the Value of a Physical Performance Counter . . 200
8-69 Recommended Return Value Implementation for Getting or Setting the Value of a Physical Performance Counter . . . 200
8-70 Parameters Needed for Getting or Setting the Value of a Cumulative Performance Counter 200
8-71 Recommended Return Value Implementation for Getting or Setting the Value of a Cumulative Performance Counter . . . 201
8-72 Overview of Full Virtualization Kernel Calls . . . 202
8-73 Parameters Needed for Creating a Virtual Machine . . . 203
8-74 Recommended Return Value Implementation for Creating a Virtual Machine . . . 203
8-75 Parameters Needed for Deleting a Virtual Machine . . . 204
8-76 Recommended Return Value Implementation for Deleting a Virtual Machine . . . 205
8-77 Parameters Needed for Setting the Page Table Used by a Virtual Machine . . . 205
8-78 Recommended Return Value Implementation for Setting the Page Table Used by a Virtual Machine . . . 206
8-79 Parameters Needed for Getting or Setting the Attributes of a Virtual Machine . . . 206
8-80 Recommended Return Value Implementation for Getting or Setting the Attributes of a Virtual Machine . . . 207
8-81 Parameters Needed for Creating a Virtual CPU . . . 207
8-82 Recommended Return Value Implementation for Creating a Virtual CPU . . . 208

8-83 Parameters Needed for Binding a Virtual CPU to a Virtual Machine . . . 209
8-84 Recommended Return Value Implementation for Binding a Virtual CPU to a Virtual Machine209
8-85 Parameters Needed for Unbinding a Virtual CPU from a Virtual Machine . . . 210
8-86 Recommended Return Value Implementation for Unbinding a Virtual CPU from a Virtual Machine . . . 210
8-87 Parameters Needed for Deleting a Virtual CPU . . . 210
8-88 Recommended Return Value Implementation for Deleting a Virtual CPU . . . 211
8-89 Parameters Needed for Getting or Setting the Registers of a Virtual CPU . . . 211
8-90 Recommended Return Value Implementation for Getting or Setting the Registers of a Virtual CPU . . . 212
8-91 Parameters Needed for Running a Virtual CPU . . . 212
8-92 Recommended Return Value Implementation for Running a Virtual CPU . . . 213
8-93 Overview of Security Monitor Kernel Calls . . . 214
8-94 Parameters Needed for Creating a Secure Zone . . . 215
8-95 Recommended Return Value Implementation for Creating a Secure Zone . . . 216
8-96 Parameters Needed for Deleting a Secure Zone . . . 216
8-97 Recommended Return Value Implementation for Deleting a Secure Zone . . . 217
8-98 Parameters Needed for Getting or Setting the Attributes of a Secure Zone . . . 217
8-99 Recommended Return Value Implementation for Getting or Setting the Attributes of a Secure Zone . . . 218
8-100 Parameters Needed for Creating a Secure Zone Thread . . . 218
8-101 Recommended Return Value Implementation for Creating a Secure Zone Thread . . . 219
8-102 Parameters Needed for Deleting a Secure Zone Thread . . . 219
8-103 Recommended Return Value Implementation for Deleting a Secure Zone Thread . . . 220
8-104 Parameters Needed for Getting or Setting the Attributes of a Secure Zone Thread . . . 220
8-105 Recommended Return Value Implementation for Getting or Setting the Attributes of a Secure Zone Thread . . . 221
8-106 Parameters Needed for Running a Secure Zone Thread . . . 221
8-107 Recommended Return Value Implementation for Running a Secure Zone Thread . . . 222
8-108 Overview of Debug and Trace Kernel Calls . . . 222
8-109 Parameters Needed for Debug Printing . . . 223
8-110 Recommended Return Value Implementation for Debug Printing . . . 223
8-111 Parameters Needed for Getting or Setting the Register Set of a Thread . . . 223
8-112 Recommended Return Value Implementation for Getting or Setting the Register Set of a Thread . . . 224
8-113 Parameters Needed for Getting or Setting the Invocation Register Set of a Thread . . . 224
8-114 Recommended Return Value Implementation for Getting or Setting the Invocation Register Set of a Thread . . . 224
8-115 Parameters Needed for Getting the Exception Register Set of a Thread . . . 225
8-116 Recommended Return Value Implementation for Getting the Exception Register Set of a Thread . . . 225
8-117 Parameters needed for getting or setting the debug mode . . . 226
8-118 Suggested return value implementation for getting or setting the debug mode . . . 226
8-119 Parameters needed for getting or setting the instruction debug breakpoint state . . . 226
8-120 Suggested return value implementation for getting or setting the instruction debug breakpoint state . . . 227
8-121 Parameters needed for getting or setting the data debug breakpoint state . . . 227

8-122 Suggested return value implementation for getting or setting the data debug breakpoint state228

9-1 Obtaining the Number of Timeslices Until the Nearest Timeout 231
9-2 Recording the Elapsed Time 231
9-3 “Off-Initial-Dirty” Type Switch Strategy 1 235
9-4 “Off-Initial-Dirty” Type Switch Strategy 2 236
9-5 “Off-Initial-Clean-Dirty” Type Switch Strategy 237
9-6 RME Deviations from the Required Section of MISRA-C:2012 248
9-7 RME Deviations from the Advisory Section of MISRA-C:2012 249
9-8 Chinese-English Terminology Quick Reference Table 249

List of Figures

1-1 Monolithic Kernel Architecture 6
1-2 Microkernel Architecture 7
1-3 Brief architecture overview of RME 18

2-1 Life Cycle of Capabilities 23
2-2 Timeline of Capability Freezing 23
2-3 Capability Table Structure Diagram 25

3-1 An Example of a Normal Multi-Level Page Table 37
3-2 An Example of a Path-Compressed Page Table 39
3-3 Example of Kernel Memory Capability Delegation 53

4-1 Thread State Transition Diagram 63

5-1 Example of Synchronous Thread Migration 87
5-2 Example of Asynchronous Signal Sending and Receiving 87

9-1 Typical Implementation of a Heterogeneous Multi-core Separation Kernel 230
9-2 Typical Implementation Scheme of a Separation Kernel Based on NUMA 230

Revision History

Revision	Date (Year-Month-Day)	Description
R1T1	2017-07-12	Initial release.
R2T1	2017-12-02	Added porting directions.
R3T1	2018-06-04	Added the appendix.
R3T2	2018-08-17	Added the kernel function call section and updated the manual to reflect kernel changes.
R3T3	2019-09-17	Fixed several places and slightly modified the handling of thread scheduler events.
R4T1	2024-09-10	Released the R4T1 technical reference manual.

1. Overview

1.1 Introduction

In modern embedded systems (Embedded System), as the demands on computation grow, the prevalence of multi-core systems is increasing rapidly, and at the same time the trend of heterogeneous computing is becoming ever more pronounced. At the same time, the functional complexity and market share of microcontroller systems with markedly different architectures are rising rapidly, and the requirements for resource management are also growing. However, for multi-core systems, while computing capability grows, their real-time responsiveness is often impaired by resource access contention, so it is necessary to write a fully parallel kernel that is lock-free and has little shared state; for microcontroller (Microcontroller Unit, MCU) systems, their memory management approaches differ, and the traditional abstraction methods often cannot meet the requirements, so it is impossible to use the same system-level programming paradigm as medium and large computers, and it is therefore necessary to write a new system to unify the memory management interface.

At the same time, in modern high-performance systems, the importance of lightweight virtualization functionality is gradually increasing. In large servers, virtualization can make resource management more flexible, and can further enable applications such as hot migration of virtual machines, facilitating hardware maintenance; in high-performance embedded systems, virtualization (Virtualization) can allow microcontrollers or digital signal processors (Digital Signal Processor, DSP) to securely run third-party binary code, or to run multiple high-level language virtual machines, without causing security and resource control problems. In both scenarios, virtual machines are required to respond in real time, to be highly efficient, and to be scalable and extensible.

RME (RTOS-Mutate-Eukaron) real-time operating system (Real-Time Operating System, RTOS) is an extremely scalable (Scalable) and expandable (Expandable) fully preemptive microkernel real-time operating system. It provides all the features provided by third-generation microkernels: a flexible user-mode scheduler, complete low-level storage management, efficient communication mechanisms, and special optimization capabilities for specific hardware. The RME operating system can run on microcontrollers with only 64kB of read-only memory (Read-Only Memory, ROM) and 16kB of random-access memory (Random-Access Memory, RAM), and can also run on high-performance servers with hundreds of CPU sockets and hundreds of TB of memory, while possessing equally superior operating efficiency.

1.1.1 Design Goals

As a third-generation microkernel, the design goals of RME mainly include the following four points: flexibility, security, reliability and real-time responsiveness.

Flexibility (Flexibility) means that the system can adapt to a variety of application scenarios, and fully exploit the potential of various architectures when running on them, while using as many common parts as possible. To this end, RME was designed with a range of possible platforms, from microcontrollers to supercomputers, in mind, and follows strict coding compatibility specifications to accommodate all possible compilation environments.

Security (Security) means the controlled confidentiality (Confidentiality), integrity (Integrity) and availability (Availability) of the various resources in the system. To achieve the above three requirements,

RME takes capabilities (Capability) as the foundation of the system. This means that, in the system, everything is controlled and managed by capabilities; to operate on a kernel object, the corresponding capability must be passed in through a system call. At user level, different kernel object managers manage different categories of capabilities, so their coupling is very low, and consequently the degree of isolation between them is relatively high.

Reliability (Reliability) means that there are few or no functional errors in the system, and that when such errors occur, a micro-reboot (Micro-reboot) of the subsystem can be performed. As a microkernel, RME is inherently more reliable than monolithic kernels; to minimize the occurrence of errors, the architecture-independent part of the RME code has all been subjected to white-box testing with full branch coverage.

Real-time responsiveness (Real-time Responsiveness) means that all execution paths in the system are bounded by definite time limits, and these limits are short enough not to cause problems. RME was designed with the predictability of all execution paths in mind, and can maintain its own hard real-time responsiveness even in multi-core environments.

1.1.2 Software Copyright and License

Taking comprehensive account of the different requirements of microcontroller applications, deeply embedded applications and traditional applications on open-source systems, the license adopted by the RME kernel itself is Unlicensed, but special provisions apply to some special cases[1] These special provisions are case-specific, and the specific terms will differ for each possible situation.

1.1.3 Glossary of Easily Confused Terms

In this book, the basic terms that are easily confused are defined as follows:

1.1.3.1 Operating System

Refers to the basic software running on a device that performs the lowest-level management of hardware resources such as the processor and memory.

1.1.3.2 Process

A process refers to the smallest independent protection space that possesses certain resources. These resources can be certain kernel objects, a certain range of memory, certain devices, etc. Generally speaking, this protection space will correspond to an instance of a program that is currently executing.

1.1.3.3 Thread

A thread refers to a schedulable entity in an operating system that possesses an independent execution stack and an independent instruction stream. A process can contain multiple threads, and they share one process address space.

1.1.3.4 Coroutine

A coroutine refers to a schedulable entity in an operating system that possesses only an independent instruction stream but no independent stack. A thread can contain multiple coroutines, and these coroutines share one thread stack.

1.1.3.5 Static Allocation

Refers to an allocation method in which the resource allocation scheme is decided at system compile time.

[1] Such as security equipment and medical equipment, etc.

1.1.3.6 Semi-Static Allocation

Refers to an allocation method in which the resource allocation scheme is decided during system startup and is not changed afterwards during runtime.

1.1.3.7 Dynamic Allocation

Refers to an allocation method in which the resource allocation can be changed during system runtime.

1.1.3.8 Soft Real-Time

Refers to a real-time guarantee in which operations should be completed within their deadlines in the vast majority of cases, but a small portion of operations is also allowed to occasionally complete beyond their deadlines.

1.1.3.9 Hard Real-Time

Refers to a real-time guarantee in which all operations must be completed within their deadlines.

1.1.3.10 Constant Real-Time

Refers to a real-time guarantee in which all operations are O(1) with respect to user input and system configuration, and all executions can be completed within some practically meaningful constant time limit. This is the strongest of all real-time guarantees.

1.1.3.11 Constant Real-Time to a Certain Value

Refers to a real-time guarantee in which all operations and responses are O(1) while a certain value that can be determined by the system configuration remains unchanged, and all executions can be completed within some practically meaningful constant time limit.

1.1.4 Major Reference Systems

The capability table, signal endpoint and synchronous invocation take Composite (@GWU) as a reference.

Part of the implementation of the page table takes Composite (@GWU) as a reference.

The dynamic page functionality of the page table takes uCLinux (@Emcraft) as a reference.

The concept of kernel memory capabilities takes Fiasco.OC (@TU Dresden) as a reference.

The operation flags of capabilities and the implementation of the maximum priority of threads take seL4 (@Data61/CSIRO) as a reference.

The implementation of the lightweight scheduling queues in the kernel takes RMProkaron (@EDI) as a reference.

The implementation of the system call interface takes Linux (@The Linux Foundation/Linus Torvalds) as a reference.

The implementation of the separation kernel takes Barrelfish (@Microsoft/ETH Zurich) as a reference.

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

1.2 Foreword

An operating system is a kind of basic software running on a device that performs the lowest-level management of hardware resources such as the processor and memory. For a real-time operating system, every operation of the system must be correct and timely, and its execution time must be predictable. Generally speaking, there are two types of real-time operating systems: the first is soft real-time systems, and the second is hard real-time systems. For a soft real-time system, it suffices that the responses of the program are within their deadlines in most cases; for a hard real-time system, the responses of the system

must be within their deadlines at all times. In reality, few real-time applications or operating systems are entirely soft real-time or entirely hard real-time; they are usually an organic combination of a soft real-time part and a hard real-time part. For example, the LCD display and human-machine interface part is usually soft real-time, while the motor control part is usually hard real-time.

Generally speaking, the vast majority of real-time systems are embedded systems. An embedded system refers to a dedicated system that is highly customized in both software and hardware, and it imposes very stringent requirements on the system's performance, power consumption, size and operating environment. Such systems include highly customized systems such as industrial automation controllers, flight controllers and rocket controllers, as well as relatively general-purpose systems such as industrial control computers.

Traditionally, owing to the limitations of hardware capabilities, real-time systems were generally rather simple, and it was feasible to cope with them using a simple operating system or no operating system at all. At present, with the introduction of new-generation microcontrollers and microprocessors, the complexity of application software has greatly increased, which makes the use of new-generation real-time operating systems inevitable.

The new generation of real-time operating systems is more powerful than existing real-time systems in aspects such as reliability, portability and flexibility, especially with regard to the utilization of multi-core processors and to the design of and consideration for parallelization. Taking these design requirements into account, a microkernel is an inevitable choice. A microkernel implements a minimal set of primitives, which can be used to place the vast majority of the functionality that traditional operating systems keep in kernel mode into user-mode servers. If one of the drivers or servers fails, we can often confine the fault to a small scope so that it does not spread to the entire operating system. Therefore, the various user-mode servers can be restarted individually and can also be redundantly backed up, greatly improving the security and reliability of the system.

Moreover, a microkernel generally makes the writing of multi-core parallel-related code easier, because the kernel is simple and the number of synchronization mechanisms that need to be added is smaller. This enables us to make extensive use of read-modify-write (Read-Copy-Update, RCU) techniques, making it possible to write a lock-free kernel. Furthermore, because the capabilities of the microkernel are all placed in tables and aligned to a cache line, we can always avoid the false-sharing (False-sharing) problem caused by cache coherence, thereby greatly improving the kernel's performance under multi-processor or multi-node non-uniform memory access architectures (Non-Uniform Memory Access, NUMA).

Here we review the categories of operating systems. In this manual, we divide common operating systems into five categories, respectively called fully customized systems, ultra-lightweight systems, basic real-time systems, complex application systems and complex real-time application systems. Below we will introduce the concepts of these operating systems respectively.

1.2.1 Fully Customized Systems

Such systems are operating systems fully customized for a certain application. They do not possess system services and software abstraction layers in the sense of general operating systems, nor do they possess a distinction between kernel space and user space; applications run directly on the bare metal. Generally speaking, unikernels (Unikernel) all belong to this category. The vast majority of microcontroller foreground/background bare-metal processing systems also belong to this category.

Typical such operating systems include Rump (unikernel) and Mirage OS (unikernel).

1.2.2 Ultra-Lightweight Systems

An ultra-lightweight system is the minimal system that can be called an operating system. Generally speaking, it usually runs on 8-bit or even 4-bit machines, does not require a system timer, has no distinction between kernel mode and user mode, and can run on multiple architectures without porting. It

usually consists of only a small piece of code responsible for task switching or even just a few macros, and can be compiled and run without a customized linker script.

It generally consists of a while() main loop, and in that loop it calls the task functions one by one. The task functions are simple state machines, and each time one is entered it selects a state to run. The various tasks use the same stack. The task code is compiled together with the kernel. The tasks behave as cooperative coroutines that cannot preempt one another, and they do not need to be reentrant.

Its priority system is implemented using the priority configuration of hardware interrupts, and the hardware interrupt functions directly process everything rather than deferring processing to tasks. Interrupts are completely transparent to the operating system, and the operating system does not need to know whether an interrupt has arrived.

Typical such operating systems include EDI's RMS (ultra-lightweight coroutine library) and FAU's Sloth (enhanced lightweight coroutine library).

1.2.3 Basic Real-Time Systems

A basic real-time system is the minimal system that preliminarily exhibits the basic characteristics of real-time systems. It generally runs on high-end 8/16-bit machines and low-end 32-bit machines, and requires a system timer. It has no distinction between user mode and kernel mode, but the memory management unit (Memory Management Unit, MMU) or memory protection unit (Memory Protection Unit, MPU) can be configured to protect a certain range of memory. It requires simple architecture-dependent assembly code to perform context stack switching. To make it runnable on multiple architectures, modifying this piece of assembly code is mandatory. Porting also often involves the system timer, stack switching, interrupt management and coprocessor management. Such systems may or may not use a customized linker script; generally speaking, a customized linker script must be used when memory protection is involved.

In such systems, tasks present themselves as threads. Task functions may be reentrant. Each task uses a separate execution stack. The task code and the kernel code may or may not be compiled together. Tasks calling system functions usually use direct ordinary function calls; there is no concept of making system calls via software interrupts.

Such systems possess the concept of priority, and generally implement preemption between different priorities and timeslice round-robin scheduling among the same priority. Such systems possess a rudimentary memory management scheme, and this memory management scheme is generally based on SLAB and the buddy system.

Interrupts may be completely transparent to the operating system, in which case the operating system does not need to know whether an interrupt has arrived; if a context switch needs to be performed in an interrupt, the stack switching assembly must be inserted into that interrupt function, in which case the interrupt entry and exit need to be written in assembly code.

Typical such operating systems include RMP, RT-Thread, FreeRTOS, uC/OS, Salvo and ChibiOS.

1.2.4 Complex Application Systems

A complex application system is a system that exhibits most of the characteristics of an operating system, but its real-time metrics are generally not outstanding. It generally runs on 32-bit and 64-bit machines, and requires a system timer. It has a strict distinction between kernel mode and user mode, and requires that the hardware it runs on must have an MMU or MPU. It requires a fairly complex customized linker script, and requires extensive porting before it can run on a new architecture. Porting mainly involves the system timer, context/protection domain switching, interrupt management and coprocessor management, etc.

On such systems, tasks present themselves as processes, and one task can contain multiple threads. Because virtual memory is used, whether the task functions are reentrant has no impact. The kernel

code and the applications are compiled separately. System calls are made through software interrupts or dedicated system call instructions[2].

Such systems possess the concept of priority, but real-time performance is generally not guaranteed.

Such systems generally have two-level memory management. At the lower level is the management of pages, and at the upper level is the stack management provided by the system runtime library.

Interrupts are not transparent to such systems. The operating system requires that the context registers, coprocessors and MMU/MPU be maintained when entering and exiting an interrupt.

Typical such operating systems include Windows, Linux, Minix, FreeBSD, MacOS X and Amiga.

1.2.5 Complex Real-Time Application Systems

Complex real-time application systems are the most powerful category of all operating systems. The main characteristic of such systems is that, on top of complex application systems, guarantees of real-time responsiveness are added, and the execution timing of the kernel is completely predictable.

Typical such operating systems include RME, Composite, Fiasco.OC and various other L4 systems, RTLinux and VxWorks. RME is designed as such a system. Therefore, its design needs to take the most issues into account, and its design difficulty is also the greatest.

In the designs of the kernels described in 1.2.3 and 1.2.4, there have always been two approaches, namely the monolithic kernel (Monolithic Kernel) and the microkernel (Microkernel). The representative of the former is Linux, which compiles and links all system services into kernel space. The advantage of doing so is that all inter-module calls are simple function calls with no cross-protection-domain overhead, so it is faster; moreover, the kernel itself can provide more functionality. The representative of the latter is the various L4s, which place all functionality in user space as much as possible, and the kernel itself is only responsible for basic functions such as message forwarding and scheduling. In later microkernel development, even the scheduler and kernel memory management were further placed in user space. The advantage of doing so is that the functional modules of the operating system are isolated from one another as much as possible, achieving better maintainability, security and reliability, and the attack surface exposed to hackers is also smaller; the kernel itself can moreover undergo formal verification (Formal Verification) to guarantee its correctness[4]. The typical system architecture diagrams of these two approaches are shown below:

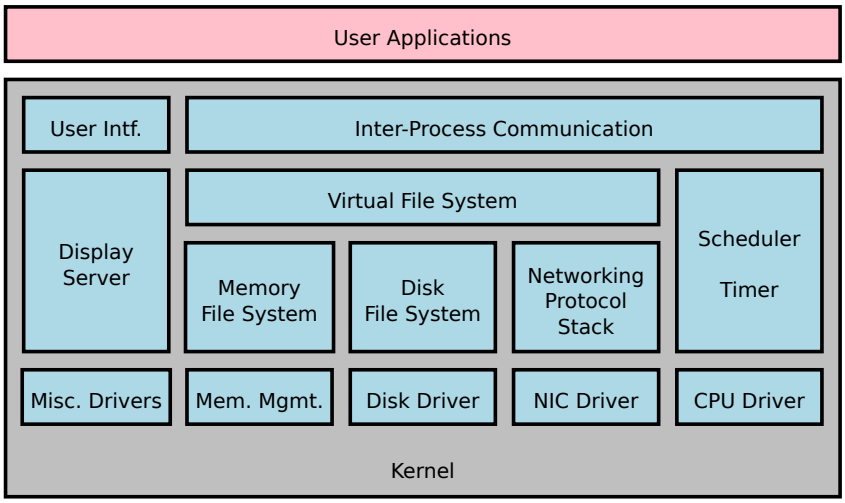


Figure 1-1 Monolithic Kernel Architecture

[2] Such as SYSCALL, SYSRET, etc. on x86-64

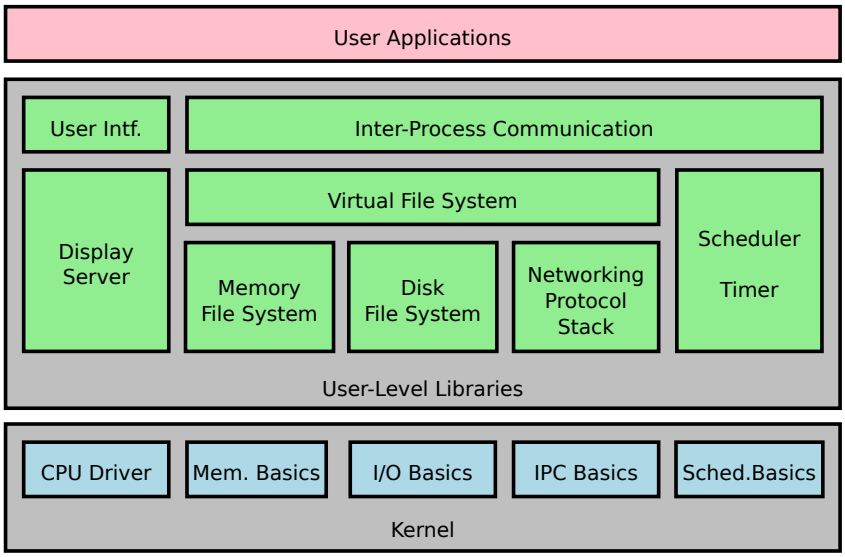


Figure 1-2 Microkernel Architecture

RME adopts a microkernel design. On the one hand, the advantage of monolithic kernels in not needing to frequently cross protection domains has become very insignificant on modern architectures[3], while on the other hand, microkernels, owing to the excellent locality of code and data and their simple internal structure, have achieved a decisive advantage in cache performance and concurrency performance, causing monolithic kernels to lose their only advantage——speed. A group of microkernel operating systems recently developed, represented by seL4[4], can be dozens of times better than Linux in inter-process communication, and their worst-case response time, depending on the configuration, can be even better. In terms of application performance, network function virtualization (Network Function Virtualization, NFV) applications based on microkernels have also achieved performance generally an order of magnitude higher than Linux, thoroughly refuting the assertion that monolithic kernels necessarily have high performance.

As for RME itself, on some single-core architectures[4] its average inter-process communication performance is 40 times that of Linux; even including the overhead caused by the user-mode library, it is still 25 times faster than Linux.

1.3 Performance Metrics of Real-Time Operating Systems and Their Components

There are hundreds of different kinds of RTOSes on the current market, and kernels developed by hobbyists and individuals are even more numerous. The performance of these systems often varies greatly. We need some metrics to measure the performance of these RTOSes. The metrics listed below can only be compared directly when the processor architecture, compiler and compilation options are the same. If different architectures, compilers or compilation options are used, the resulting data have no direct meaning, and are only for reference and not comparable. One recommended method is to evaluate using industry-standard ARM or MIPS series processors together with the GCC-O2 option. In addition, the Chronos simulator together with GCC-O2 can also be used for evaluation. When evaluating, note also that the load level of the system may affect these values, so these values can be compared only when the load level of the system is the same.

[3] For example, the dedicated system call instruction pair SYSCALL and SYSRET of the latest x86-64 consumes only 60 cycles, equivalent to one L3 cache miss

[4] Especially a group of microcontrollers represented by Cortex-M

1.3.1 Kernel Size

Kernel size is an important metric for measuring RTOSes. Because RTOSes are usually deployed on devices with extremely limited memory, a small kernel size is critical. Kernel size is mainly measured from two aspects, the first being the read-only segment size and the second being the data segment size. The read-only segment includes the kernel's code segment and read-only data segment, and the data segment includes the size of the kernel's read-write data segment. In Flash-based microcontroller systems, the read-only segments consume Flash, while the data segments consume SRAM.[1]

Because RTOSes are highly configurable, their kernel size is often not fixed, but is closely related to the configuration chosen. Therefore, to measure this performance, one should examine the kernel size under the minimal kernel configuration, the common kernel configuration and the maximal kernel configuration.[1]

Obtaining kernel size data is very simple: just compile the kernel with a compiler, and then use a dedicated binary viewer[5] to inspect the size of each section of the target file.

1.3.2 Execution Time, Worst-Case Execution Time and Their Jitter

Execution time refers to the amount of time an RTOS system call takes. Worst-case execution time refers to the maximum length that the execution time can reach under the most unfavorable conditions. The worst-case execution time of an RTOS is usually reached under the following circumstances: the most time-consuming system call is executed, and a large number of cache misses and TLB misses occur in the process. RTOSes generally disable interrupts when executing system calls; the worst-case execution time is usually the longest time the system keeps interrupts disabled, so its impact on the system's real-time responsiveness is enormous.

Worst-case execution time can be divided into two categories: the first is the worst-case execution time of kernel system calls, and the other is the worst-case execution time of inter-process communication and inter-thread synchronization.

To obtain the first type of worst-case execution time, before calling a certain system call, the timer can record the timestamp T_s at this time, and then after the system call ends, the timer is called again to record the timestamp T_e at this time. Then, the timer is called twice in succession, recording two timestamps T_{ts} and T_{te}, yielding the extra cost of calling the timer as T_{te}-T_{ts}. At this point, the execution time is T_e-T_s-(T_{te}-T_{ts}). The worst-case execution time is the largest execution time among all the system call tests.

To obtain the second type of worst-case execution time, the timer can be called once at the sending end of the communication mechanism to record the timestamp T_s at this time, and called once at the receiving end of the communication mechanism to record the timestamp T_e at this time. The measurement of the cost of calling the timer is similar to that for the first type of worst-case execution time. The finally obtained T_e-T_s-(T_{te}-T_{ts}) is the execution time. The worst-case execution time is the largest execution time among all the communication tests.

The jitter of execution time is also very important. When measuring the execution and communication time of the same system multiple times, we often obtain a distribution. The mean of this distribution is the average execution time, and its standard deviation[6] is called the execution time jitter.

For an RTOS, we generally consider that the execution time, the worst-case execution time and the jitter are all better the smaller they are. Execution time can be further divided in detail into the following categories:[1]

1.3.2.1 Intra-Process Thread Switch Time

Intra-process thread switch time refers to the time consumed to switch from one thread to another within the same process. We measure it using the method shown below. In the measurement, in addition to

[5] Such as Objdump
[6] Sometimes we also use the range

the method using T_e-T_s, the difference between two consecutive T_s values divided by 2 can also be used. Thread switching includes two cases: one is switching between threads of the same priority, and the other is waking up a higher-priority thread from a lower-priority thread.[2]

In the first case, we assume that the two threads in the figure have the same priority, and that at the start of the measurement we are executing thread A, which has just been switched to by thread B.

Table 1-1 The First Case of Intra-Process Thread Switching

Process 1: Thread A	Process 1: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Switch to thread B;	» Switch to thread A;
}	}

In the second case, we assume that thread B in the figure has a higher priority, and that at the start of the measurement we are executing thread A, which has just been switched to by thread B.

Table 1-2 The Second Case of Intra-Process Thread Switching

Process 1: Thread A	Process 1: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Wake up B (conventional) or switch to B (microkernel);	» Sleep (conventional) or switch to A (microkernel);
}	}

1.3.2.2 Inter-Process Thread Switch Time

Inter-process thread switch time refers to the time consumed to switch from one thread to another between different processes. The measurement method and the two possible cases are completely identical to those described in the previous section; the only difference is that the threads participating in the test now belong to two processes.[2]

In the first case, we assume that the two threads in the figure have the same priority, and that at the start of the measurement we are executing

thread A, which has just been switched to by thread B.

Table 1-3 The First Case of Inter-Process Thread Switching

Process 1: Thread A	Process 2: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Switch to thread B;	» Switch to thread A;
}	}

In the second case, we assume that thread B in the figure has a higher priority, and that at the start of the measurement we are executing thread A, which has just been switched to by thread B.

Table 1-4 The Second Case of Inter-Process Thread Switching

Process 1: Thread A	Process 2: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Wake up B (conventional) or switch to B (microkernel);	» Sleep (conventional) or switch to A (microkernel);
}	}

1.3.2.3 Intra-Process Inter-Thread Synchronous Communication Time

Intra-process inter-thread synchronous communication time refers to the total time used to perform synchronous communication between different threads within the same process. The following measurement assumes that thread B has already blocked at the receiving end, thread A performs the send, and thread B has a higher priority than thread A. For systems that communicate using thread migration techniques[7], the measurement of this parameter is meaningless, because synchronous communication is not needed within a process.[2]

Table 1-5 Intra-Process Inter-Thread Synchronous Communication

Process 1: Thread A	Process 1: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Send a signal to synchronous endpoint P;	» Receive a signal from synchronous endpoint P;
}	}

1.3.2.4 Inter-Process Synchronous Communication Time

Inter-process synchronous communication time refers to the total time used to perform synchronous communication between different processes. The following measurement applies to traditional operating systems, and assumes that thread B has already blocked at the receiving end, thread A performs the send, and thread B has a higher priority than thread A.[2]

Table 1-6 Inter-Process Synchronous Communication of Traditional Operating Systems

Process 1: Thread A	Process 2: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Send a signal to synchronous endpoint P;	» Receive a signal from synchronous endpoint P;
}	}

For operating systems such as RME and L4 that employ thread migration techniques, the measurement is as follows. Note that when thread A migrates and invokes function F, we are still running inside thread A, except that the code segment being executed and the data segment being used have moved into another process, and an additionally allocated execution stack is used inside that process.

[7] Such as RME and some L4 series

Table 1-7 Inter-Process Synchronous Communication with Thread Migration Techniques

Process 1: Thread A	Process 2: Thread A
Loop forever	Function F
{	{
» Record time T_s;	Record time T_e;
Migrate and invoke function F;	Return;
}	}

1.3.2.5 Intra-Thread Asynchronous Communication Time

Intra-thread asynchronous communication time refers to the total time used to send and receive asynchronous signals within the same thread. Common asynchronous signals may include the asynchronous signal primitive (Signal) provided by the RME system, or the semaphore (Semaphore), mailbox (Mailbox), message queue (Queue), pipe (Pipe), etc. provided by other operating systems. We measure it using the method shown below.[2] In the final measurement results, T_i-T_s yields the time consumed by the send, T_e-T_i yields the time consumed by the receive, and T_e-T_i is the communication time.

Table 1-8 Intra-Thread Asynchronous Communication

Thread A
Loop forever
{
Record time T_s;
Send a signal to asynchronous endpoint P;
Record time T_i;
Receive a signal from asynchronous endpoint P;
Record time T_e;
}

1.3.2.6 Intra-Process Inter-Thread Asynchronous Communication Time

Intra-process inter-thread asynchronous communication time refers to the total time used to send and receive asynchronous signals between different threads within the same process. We measure it using the method shown below. We assume that thread B has already blocked at the receiving end, thread A performs the send, and thread B has a higher priority than thread A.[2]

Table 1-9 Intra-Process Inter-Thread Asynchronous Communication

Process 1: Thread A	Process 1: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Send a signal to asynchronous endpoint P;	» Receive a signal from asynchronous endpoint P;
}	}

1.3.2.7 Inter-Process Asynchronous Communication Time

Inter-process asynchronous communication time refers to the total time used to send and receive asynchronous signals between different processes. The measurement method is completely identical to that

described in the previous section; the only difference is that the threads participating in the test now belong to two processes.[2]

Table 1-10 Inter-Process Asynchronous Communication

Process 1: Thread A	Process 2: Thread B
Loop forever	Loop forever
{	{
» Record time T_s;	Record time T_e;
Send a signal to asynchronous endpoint P;	» Receive a signal from asynchronous endpoint P;
}	}

1.3.2.8 Page Table Operation Time

Page table operation time refers to the time taken to operate on a page table entry. Because different operating systems differ greatly in the level of functionality they provide and in complexity in this respect, they cannot be compared directly. Usually, a microkernel provides system calls that can directly construct and map the underlying page tables, while a monolithic kernel provides operations that map pages into a process's address space. Some kernels used on microcontrollers adopt fixed-block allocation operations. In general, the method for timing such operations is very simple: read the timer when making such a system call, and read it again after the call finishes.[1][2]

Table 1-11 Page table operations

Thread A
Loop forever
{
Take timestamp T_s;
Perform the operation;
Take timestamp T_e;
}

1.3.3 Interrupt Response Time, Worst-Case Interrupt Response Time and Their Jitter

Interrupt response time refers to the time between the occurrence of an interrupt and the RTOS invoking the processing thread corresponding to the interrupt. Worst-case interrupt response time refers to the maximum length that the interrupt response time can reach under the most unfavorable conditions. Worst-case interrupt response time is usually reached under the following circumstances: a large number of cache misses and translation lookaside buffer misses occur during interrupt processing. Interrupt response time is the most important metric of an RTOS; it can even be said that all RTOS design revolves around this metric. This metric is the most direct standard for an RTOS's response time to external stimuli.

To obtain the worst-case interrupt response time, one can read the timer in the first line of assembly code of the interrupt vector[8], obtaining a timestamp T_s, and read the timer at the first line of code of the interrupt handling thread, obtaining a timestamp T_e. The measurement of the timer overhead is the same as above. The resulting T_e-T_s-(T_{te}-T_{ts}) is the interrupt response time. The worst-case interrupt response time is the largest response time among all interrupt response tests.

[8] It cannot be deferred until inside a C function, because register and stack maintenance is also part of the interrupt response time

The jitter of the interrupt response time is also very important. When measuring the interrupt response time of the same system multiple times, we often obtain a distribution. The mean of this distribution is the average interrupt response time, and its standard deviation[9] is called the interrupt response time jitter.

For an RTOS, we generally consider that the interrupt response time, the worst-case interrupt response time and the jitter are all better when smaller. The measurement of interrupt response time is usually as follows[1][3]:

Table 1-12 Interrupt response time

Kernel	Thread A
Hardware interrupt vector	Loop forever
{	{
» Take timestamp T_s;	Take timestamp T_e;
Send a signal from the kernel to asynchronous endpoint P;	» Receive a signal from asynchronous endpoint P;
}	}

1.3.4 Realistic Interrupt Response Time, Worst-Case Realistic Interrupt Response Time and Their Jitter

Realistic interrupt response time refers to the time in a hardware/software system from an external signal input interrupt to the issuance of an I/O operation response. Worst-case realistic interrupt response time refers to the maximum length that the realistic interrupt response time can reach under the most unfavorable conditions. Among the factors affecting the worst-case realistic interrupt response time, in addition to those that can affect the worst-case interrupt response time, there are also factors inherent to the corresponding CPU and I/O hardware itself.

To obtain the realistic interrupt response time, we need some external hardware to support this kind of measurement. For example, if we need to measure the realistic interrupt response time of a system's I/O, we can connect one pin of an FPGA to an input pin of a CPU or motherboard, and then connect another pin to an output pin on the CPU or motherboard. First, the FPGA sends a signal to the input pin, at which point the high-precision timer inside the FPGA starts running; when the FPGA receives the signal on the output pin, the high-precision timer inside the FPGA stops running. The final time value of the FPGA's internal timer is the realistic interrupt response time of the system. The worst-case realistic interrupt response time is the largest response time among all tests.

[9] Sometimes we also use the range

For a hardware/software system, we generally consider that the realistic interrupt response time, the worst-case realistic interrupt response time and the jitter are all better when smaller. It is worth noting that the realistic interrupt response time is generally approximately equal to the worst-case execution time plus the worst-case interrupt response time plus the inherent latency of the system's CPU/I/O. For example, when an I/O input arrives just as a system has begun executing a system call, the hardware interrupt vector cannot be executed immediately and can only wait until that system call has finished executing. Once that system call has finished executing, the actual hardware interrupt vector begins to execute, switching to the processing thread for handling and producing the output. The measurement of the realistic interrupt response time is usually as follows[1]:

Table 1-13 Realistic interrupt response time

FPGA (or oscilloscope)	System under test
Loop forever	Loop forever
{	{
» Send a signal and start the timer;	Receive a signal from the I/O;
Receive the signal;	Minimal internal processing routine;
Stop the timer;	Output a signal to the I/O;
}	}

1.3.5 Input/Output Performance Metrics

I/O performance metrics mainly apply to those operating systems that provide an I/O subsystem, especially those that provide virtualization support. Common I/O subsystems include disk systems, network systems, serial and parallel port systems and various data acquisition card systems; on microcontrollers, they manifest more as GPIO, PWM generators and LCD controllers, etc. The evaluation criteria for the many I/O subsystems are often different, but usually they all include two parts: throughput and latency. Throughput refers to the I/O system data transfer rate that the system can achieve within a certain period of time, and latency refers to the time required from issuing an I/O request to receiving a response.

1.3.6 Virtualization Performance Metrics

For those operating systems that support virtualization[10] of other operating systems[11], evaluating virtualization performance metrics is also very important. Virtualization performance metrics generally include two parts: one is the virtual machine functionality part, and the other is the virtual machine performance part. In addition, the communication overhead between virtual machines is also an important metric.

In terms of virtual machine functionality, we generally evaluate whether each function of the virtual machine is correctly implemented and supported. The more functions that are correctly implemented and supported, the better the functionality metric.

In the evaluation of the virtual machine performance part, the virtual machine performance metrics to be evaluated are the same as the various metrics listed in 1.3.2. In addition, the extra performance consumption and storage resource consumption introduced by virtualization must also be measured. The less extra performance consumption and the less storage resource consumption caused by virtualization, the better the virtualization performance metric.

The communication overhead between virtual machines is also a very important topic. The communication overhead between virtual machines is generally larger than the communication overhead within a virtual machine, and is often supported through dedicated drivers or virtual networks. The measurement of the communication overhead between virtual machines is consistent with the measurement method for I/O performance, and is not repeated here.

[10] Including full virtualization and para-virtualization
[11] Such as RME, NOVA[5] and some L4 systems

1.4 RME System Call Format

System calls are the only way to use the functionality provided by the system. For RME, this is the only way to use its system functionality. Generally speaking, system calls are implemented through software interrupts[12], and can also be implemented through dedicated system call instructions[13]. In the software interrupt implementation, once the software interrupt instruction is invoked, the system jumps to the software interrupt handling vector to perform interrupt handling. The system call parameters are passed to the kernel by some means[14], and the kernel then handles the call. In the dedicated instruction implementation, the kernel switches to a preconfigured kernel stack and jumps directly to the entry of the system call handling function, while parameter passing is the same as in the software interrupt implementation.

RME supports both methods at the same time. On the x86-64 architecture we use the latter, while on the ARM architecture we use the former.

1.4.1 System Call Basics

The basic method of an RME system call is to first place four machine-word-length parameters into four registers in order, and then invoke the software interrupt instruction or the dedicated kernel call instruction. RME always uses four registers to pass parameters on any architecture, because this is the maximum number of parameters allowed to be passed without using the stack in the C language function calling conventions of most current CPU architectures[15].

None of RME's system calls use parameters longer than 4 registers. In addition, RME does not pass parameters through memory[16]. Passing parameters through memory may cause a user-mode pointer to be dereferenced in kernel mode, resulting in a memory segmentation fault in kernel mode, which is often very difficult to handle and has a chance of causing a kernel crash or privilege theft.

1.4.2 Parameter Passing and Parameter Position Encoding

The parameters of RME system calls are passed in registers. However, some registers are longer than the parameters we need to pass, so using a register to pass only one parameter is too wasteful, and we can pass more parameters. Therefore, we cut the registers into multiple sections for use. In RME, we cut a register into up to 8 sections for use. Taking a 32-bit system as an example, the definition and identifier of each section are shown in the following figure[17]:

Table 1-14 System call parameter position encoding

[31 32-bit machine word 0]							
D1	D0						
Q3	Q2	Q1	Q0				
O7	O6	O5	O4	O3	O2	O1	O0

RME also has a small number of special system calls that use other value passing methods. These methods will be described in detail at the relevant functions.

[12] Such as SWI and SVC on ARM
[13] Such as SYSCALL/SYSRET on x86-64
[14] Generally shared memory or registers
[15] For example, both MIPS and ARM pass the first four parameters through registers and the remaining parameters through the stack
[16] Linux and some early L4 systems do so
[17] A 32-bit machine is taken as an example here; 64-bit machines follow the same pattern

1.4.2.1 System Call Number

The system call number indicates the number of the system call. This number is always located at the D1 position of the first register (P0), and we also denote it as N. RME has a total of 34 system calls, numbered 0-33, as follows:

Table 1-15 List of RME system calls

System call name	Call number	Meaning
RME_SVC_INV_RET	0	Return from an invocation
RME_SVC_INV_ACT	1	Perform an invocation
RME_SVC_SIG_SND	2	Send a signal to a signal endpoint
RME_SVC_SIG_RCV	3	Receive a signal from a signal endpoint
RME_SVC_KFN	4	Perform a kernel special function call
RME_SVC_THD_SCHED_FREE	5	Release a thread from a certain CPU
RME_SVC_THD_EXEC_SET	6	Set the execution attributes of a thread[18]
RME_SVC_THD_SCHED_PRIO	7	Change the priority of a thread
RME_SVC_THD_TIME_XFER	8	Transfer a timeslice between threads
RME_SVC_THD_SWT	9	Switch to a thread
RME_SVC_CPT_CRT	10	Create a capability table
RME_SVC_CPT_DEL	11	Delete a capability table
RME_SVC_CPT_FRZ	12	Freeze a capability in a capability table
RME_SVC_CPT_ADD	13	Perform capability delegation
RME_SVC_CPT_REM	14	Remove a capability from a capability table
RME_SVC_PGT_CRT	15	Create a page directory
RME_SVC_PGT_DEL	16	Delete a page directory
RME_SVC_PGT_ADD	17	Add a page table entry
RME_SVC_PGT_REM	18	Remove a page table entry
RME_SVC_PGT_CON	19	Construct a page table
RME_SVC_PGT_DES	20	Destruct a page table
RME_SVC_PRC_CRT	21	Create a process
RME_SVC_PRC_DEL	22	Delete a process
RME_SVC_PRC_CPT	23	Replace a process's capability table
RME_SVC_PRC_PGT	24	Replace a process's page table
RME_SVC_THD_CRT	25	Create a thread
RME_SVC_THD_DEL	26	Delete a thread
RME_SVC_THD_SCHED_BIND	27	Bind a thread to a certain processor
RME_SVC_THD_SCHED_RCV	28	Receive the scheduler information of a thread
RME_SVC_SIG_CRT	29	Create a signal endpoint
RME_SVC_SIG_DEL	30	Delete a signal endpoint
RME_SVC_INV_CRT	31	Create an invocation
RME_SVC_INV_DEL	32	Delete an invocation
RME_SVC_INV_SET	33	Set the execution attributes of an invocation[19]
Reserved	34-63	Not used for now; reserved for the future

[18] Entry and stack
[19] Entry and stack

1.4.2.2 Capability Table Capability Number

The capability table capability number indicates the capability table to be operated on. This number is always located at the D0 position of the first register P0, and we denote it as C. Since only some operations require a capability table capability, the value at this position is meaningful only in those cases.

1.4.2.3 Other Parameters

The first parameter is passed in the second register, the second parameter is passed in the third register, and the third parameter is passed in the fourth register. We denote these parameters as P1, P2 and P3 respectively. In the system call documentation, P1.D1 denotes the D1 section of P1, and so on.

1.4.3 Special Notes

1. When creating kernel objects, the kernel virtual address must be aligned to 2 to the power of RME_KOM_SLOT_ORDER.
2. System call numbers 0-8 may cause a potential thread switch; for the sake of separate optimization, they have been assigned consecutive call numbers.
3. All system calls return a non-negative value on success and a negative value on failure.
4. In this manual, the prefix "rme_" of all typedef types is omitted to save table space.

1.5 Manual Overview

In the following chapters, we will introduce each subsystem of RME respectively. Among them, Chapter 2 introduces capability tables and capability management, Chapter 3 introduces page tables and memory management, Chapter 4 introduces process and thread management, Chapter 5 introduces synchronous communication mechanisms and asynchronous communication mechanisms, Chapter 6 introduces the kernel function call mechanism. Afterwards, we will introduce the porting method of RME in Chapter 7, and the implementation standards of the kernel call functionality in RME in Chapter 8. Finally, we will introduce some known issues existing in RME and their solutions in Chapter 9.

This technical reference manual only describes the RME kernel itself. For the specific implementation details of RME on each architecture, please refer to the architecture manual of the corresponding architecture; for information about the RME user-level library, please refer to the usage manual of the RME user-level library[20].

This manual assumes that the reader has a general understanding of the basic concepts of operating systems. The more abstruse concepts related to the microkernel and RME will be introduced respectively at the beginning of each chapter.

1.6 RME Architecture Overview

The following figure is a brief architecture overview diagram laid out horizontally. Besides the RME kernel itself, the figure also includes its user-level library and the various components in its user-level library. In the figure, black dashed boxes represent logically independent software modules, black solid boxes represent hardware-protection-isolated software modules, gray solid boxes represent the boundaries of software packages, and boxes containing an ellipsis (……) represent subsystems that are omitted because their content is the same as the box immediately to their left. Due to limited space, only relatively important components are shown in the figure, and some components are not displayed; for detailed user-level architecture diagrams, please refer to the usage manual of the RME user-level library.

[20] That is, RVM (M7M2)

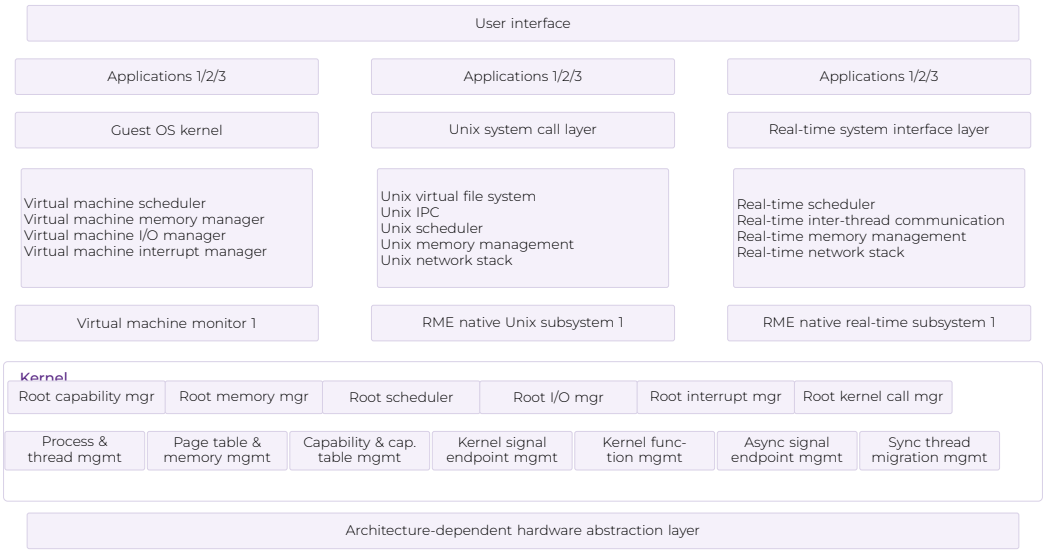


Figure 1-3 Brief architecture overview of RME

1.7 Bibliography of This Chapter

[1] T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," IEEE micro, vol. 29, 2009.

[2] R. P. Kar, "Implementing the Rhealstone real-time benchmark," Dr. Dobb's Journal, vol. 15, pp. 46-55, 1990.

[3] T. J. Boger, Rhealstone benchmarking of FreeRTOS and the Xilinx Zynq extensible processing platform: Temple University, 2013.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[5] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 209-222.

2. Capability Table and Capability Management

2.1 The Concept of Capability

A capability (Capability) is a kind of credential used to control access permissions that was first introduced in multi-user computer systems[1]. It is an unforgeable credential that uniquely identifies a certain resource as well as the operations that are permitted on that resource. For example, a Unix file descriptor is a kind of capability[2]; a Windows access permission is also a kind of capability. In a sense, a capability is a fat pointer (Fat Pointer) to a certain resource.

We use the following three principles to guarantee the security of the system[2]:

1. Capabilities cannot be forged and cannot be arbitrarily modified in user mode;
2. A process can only obtain capabilities through well-defined authorization interfaces;
3. Capabilities are only given to those processes that are responsible for managing the resource at system design time.

   Third-generation microkernels generally adopt the concept of capabilities to manage resources. In RME, all kernel resources are managed using capabilities, and all capabilities reside in capability tables in kernel space. Each process corresponds to one capability table (Capability Table), and when a system call is invoked, the system looks up kernel objects from that process's capability table. Each capability has a series of operation flag bits; if a certain capability possesses a certain operation flag bit, then that operation can be performed on the kernel object through this capability[3]. The vast majority of systems, such as seL4[5], have operation flag bits. However, for some other systems[1], a certain kernel object has only one operation, so a separate operation flag bit is not needed[4].

   Besides capabilities, there is another method called the access control list (Access Control List, ACL)[6]. It can also be used to manage permissions, but its access permission management granularity is relatively coarse, and the table easily becomes very large. Its advantage is that granting and revoking permissions is relatively simple[2].

   In RME, using the concept of capabilities, discretionary access control (Discretionary Access Control, DAC) and mandatory access control (Mandatory Access Control, MAC) can be readily implemented, multi-level security mechanisms can also be readily built, and the design of the whole user application can be planned according to the principle of least privilege.

2.2 Capability Table Operations and Capability States

A capability table is a kernel object used to store capabilities. In RME it is a linear array, and the size of each array position is a fixed 8 machine words, which can be used to store one capability. Inside a

[1] such as Composite
[2] especially the revocation operation

capability are recorded the exact type of the kernel resource, its exact location in kernel memory[3], the parent capability of the capability, the capability delegation reference count, and the current state of the capability. In addition, there is a timestamp counter used to guarantee the quiescence (Quiescence) of capability operations under multi-core parallelism.

2.2.1 Capability Types

In RME, there are altogether 8 types of capabilities[4], as shown in the table below. For information about capabilities of the corresponding types, please consult the corresponding chapters for reference. This chapter only describes the content related to capability table capabilities.

Table 2-1 Types of Capabilities

Capability type number	Capability type	Purpose
RME_CAP_TYPE_NOP	Empty capability	This position of the capability table is an empty capability.
RME_CAP_TYPE_KFN	Kernel function capability	The capability required to invoke special-function kernel functions.
RME_CAP_TYPE_KOM	Kernel memory capability	The capability required to create kernel objects using a region of kernel memory.
RME_CAP_TYPE_CPT	Capability table capability	Points to a capability table object; can be used for capability table management.
RME_CAP_TYPE_PGT	Page directory capability	Points to a page directory object; can be used for memory management.
RME_CAP_TYPE_PRC	Process capability	Points to a process object; can be used for process management.
RME_CAP_TYPE_THD	Thread capability	Points to a thread object; can be used for thread management.
RME_CAP_TYPE_INV	Invocation capability	Points to an invocation object; can be used for synchronous invocation.
RME_CAP_TYPE_SIG	Signal endpoint capability	Points to a signal endpoint object; can send and receive signals.

Each kind of capability represents the power to perform operations on that kind of kernel object, and also represents the functionality that the kernel object can implement. It is worth noting that a capability pointing to a capability table[5] is a meta-capability of the system, because this capability has the power to modify the capability table and can determine the contents of the capability table.

2.2.2 Capability Delegation Reference Count and Parent Capability

For those capabilities created in the process of creating kernel objects, we call them root capabilities; for those capabilities created by the capability delegation operation, we call them leaf capabilities. The mark of a root capability is that its attached attribute is set to ROOT, while the pointer used to record the root capability instead records the reference count, which is 0 at initial creation.

The delegation reference count and the root capability of a capability are used to track capability delegation relationships. A capability can be delegated from one capability table to another capability table; in this case we call the source capability the parent capability and the destination capability the child capability. The root capability pointer of the child capability must point to the root capability, and at

[3] a pointer into the kernel address space
[4] excluding the empty capability
[5] abbreviated as capability table capability

the same time the reference count of the root capability must be incremented by 1. That is, if the parent capability is a root capability, then the root capability pointer of the child capability points to the parent capability; if the parent capability is a leaf capability, then the root capability pointer of the child capability equals the root capability pointer of the parent capability. It is worth noting that some kernel objects also increment the reference count when they reference each other.

When removing capabilities, it is required that the leaf capabilities and all references to the kernel object be removed first, and only then can the root capability be deleted.[4]

2.2.3 Capability States

A capability has four states: empty, creating, valid, and frozen. These four states can be converted into each other through system calls.

To create a capability, invoke the corresponding kernel object creation function, use a capability table capability to designate the capability table being operated on[6], and designate the kernel virtual memory address of the kernel object to be created as well as some necessary parameters. At this point, the empty slot of the designated capability table is first changed to "creating" by an atomic compare-and-swap instruction; once creation succeeds, the state is changed to "valid"; otherwise it remains "empty" and the corresponding error code is returned. When creating a capability, if kernel memory needs to be allocated, then a kernel memory capability is also required. For kernel memory capabilities, please refer to the description in the next chapter.

There are two implementations for capability reclamation. One implementation is that of systems such as seL4 and Fiasco.OC, which support operating-system-level capability reclamation operations; that is, they only support the deletion operation, and during deletion they recursively traverse the entire capability delegation tree to reclaim all capabilities that were delegated. The RME system adopts the other implementation, which splits capability removal and deletion into two operations. The operating system does not perform the work of recursive traversal reclamation; this work must be done by the user mode. The benefit of doing so is that no kernel preemption points need to be introduced, but the drawback is that the user must track every capability delegation and take responsibility for capability reclamation themselves.[4]

To delete or remove a capability, the position must first be frozen. The freezing operation is performed through the capability freezing system call. At this point, the valid bit of the designated capability table is changed to "frozen". If freezing does not succeed, then the corresponding error code is returned. If freezing succeeds, then after the capability operation reaches quiescence, the function can be invoked to delete/remove it. Among these, the deletion operation deletes the corresponding kernel object itself while removing the capability; removal, however, only removes the capability without deleting the kernel object. After a capability is removed, it returns to the "empty" state. For leaf capabilities, only the capability removal operation can be used; for root capabilities, only the deletion operation corresponding to that kernel object can be used.

The deletion operation and the removal operation also have subtle behavioral differences. If the deletion operation detects that the kernel object cannot be deleted because references to it still exist, then it unfreezes the root capability and returns RME_ERR_CPT_REFCNT to make it convenient for the user to continue operating. If the user needs to delete, then the root capability must be frozen again before deletion. When the deletion operation detects that deletion is impossible for other reasons, it does not perform the unfreeze operation. Because the removal operation does not check references to the kernel object, it will not unfreeze the leaf capability even if the operation cannot proceed.

[6] Since the creation operation needs to add a capability into the capability table and thus modifies the capability table, a capability table capability is required; similarly, destroying a kernel object also requires a capability table capability.

2.2.4 Timestamp Counter and Multi-Core Parallelism Considerations

Execution flows in a multi-core environment are very complex. There are some kernel data structures that are not expected to be modified simultaneously in such complex operations; they require all operations to be atomic. Therefore, there are two solutions: the first solution is to take a lock for every operation, and the second solution is to use atomic operations when changing capability states. The locking approach introduces extra operations, is slower, and also causes cache line contention. Cache line contention refers to CPU1 attempting to modify the cache line while CPU2 attempts to query the state of that cache line, which causes CPU2's cache to be invalidated frequently; this is equivalent to greatly reducing the memory access speed and causes the efficiency of frequent capability operations to drop severely. The atomic operation approach has fewer of these problems, but brings extra implementation difficulty. In RME, we adopt the latter approach to achieve multi-core parallelism.

The RME kernel uses a large number of read-modify-write operations composed of compare-and-swap atomic operations and atomic increment operations to modify capabilities. The kernel also uses a timestamp counter to ensure that there are no conflicts between kernel operations, such as a kernel object being used on one CPU being deleted on another CPU.

If we want to delete or remove a capability, we must ensure that the capability has already been frozen for a period of time. We call this period the capability quiescence period. The length of this period can be configured in the kernel. It is required to be configured to be at least longer than twice the kernel's worst-case execution time. In this way, we can ensure that when the capability is deleted or removed, the capability has already settled down; that is, all operations involving this capability on all CPU cores have already completed, so there will be no mutually conflicting operations. For example, CPU1 is trying to delete capability A, while CPU2 simultaneously tries to use it. CPU2 may have already completed the validity check of the operation and be in the middle of performing that operation; if CPU1 deletes the capability directly without freezing, it may affect the execution of CPU2's corresponding operation. The timestamp counter inside each capability exists precisely to record the freeze time. It records the time at which the last freezing operation began, ensuring that no removal or deletion operation begins before the capability is completely frozen.

The complete capability state transition diagram is shown below. When creation begins, the CPU performing the creation performs a compare-and-swap operation on the empty slot, marking the start of creation; the slot is occupied until creation is complete, at which point the slot is marked as valid, and the capability can be used from then on; if the capability is not referenced and one wants to destroy it after use, then it must first be frozen. A newly created capability must wait for a quiescence period before it can be frozen, and a frozen capability must wait for a quiescence period before it can be removed or deleted.

When deleting or removing a capability, we must first complete all checks and confirm that the capability can be deleted or removed; afterwards, we change the state of the current slot to empty through a compare-and-swap atomic operation to guarantee that only the operation on one CPU core can proceed; finally we perform the real subsequent operations, namely kernel object destruction (deletion) or root capability dereferencing (removal). In RME, after the deletion or removal operation is complete, a new creation operation can begin immediately. The reason is that RME has already saved the information required for subsequent operations into local variables before modifying the slot state; after the modification it no longer accesses that capability slot, but instead uses the saved information to perform the subsequent operations. In Composite[4], the information in that slot may still be used in subsequent processing after the modification, so a quiescence period must also be inserted here to prevent this position from being overwritten by a newly created capability.

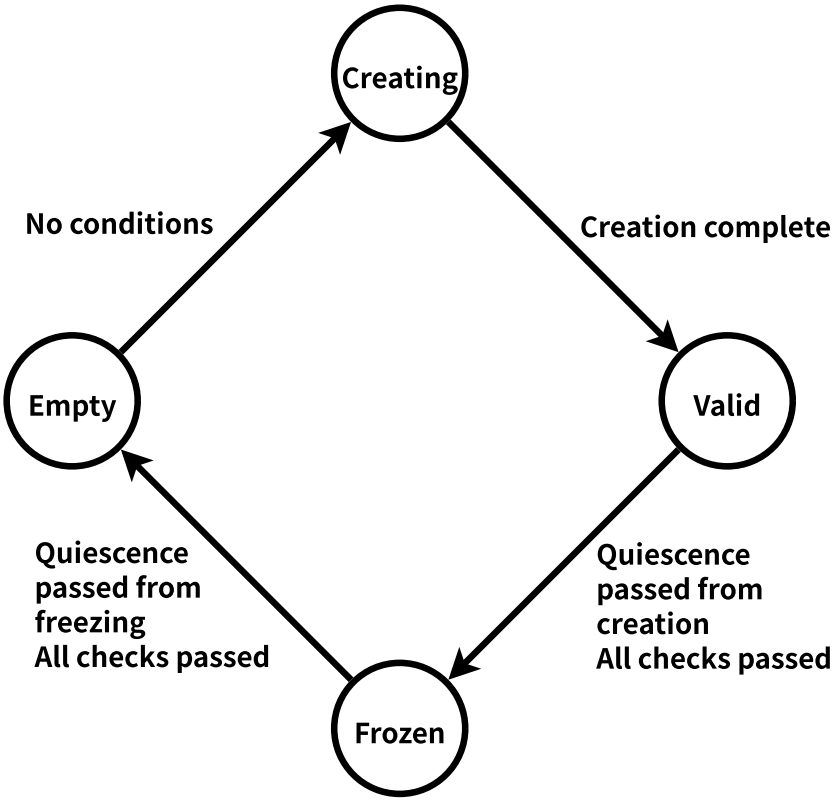


Figure 2-1 Life Cycle of Capabilities

The reason why the quiescence period is required to be at least twice the kernel's worst-case execution time is as follows: when invoking the capability freezing operation, we first update the capability's timestamp, and then mark the capability as frozen through an atomic operation. Therefore we have the following timeline:

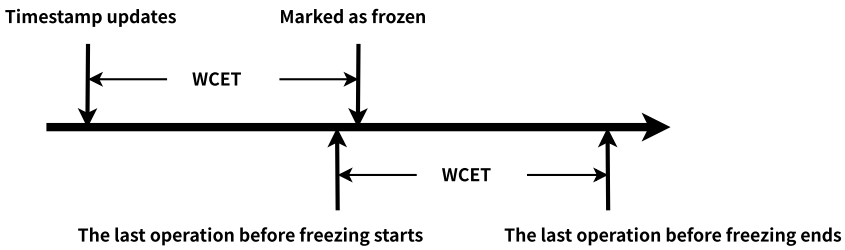


Figure 2-2 Timeline of Capability Freezing

It can be seen from this timeline that at most two kernel worst-case execution times may elapse from updating the timestamp to the end of the last operation before freezing. In practical use, it is recommended to configure the quiescence period to at least 20 times the kernel's worst-case execution time[7], because in engineering the estimation of the worst-case execution time is often accurate only in order of magnitude.

The principle explained above is the same for the creation——freezing quiescence period of capabilities, and is not repeated here.

[7] one order of magnitude higher is recommended

2.2.5 Capability Table Structure

Capability tables can be organized into a multi-level structure. The specific organization method is to place capability table capabilities inside capability tables, so that the capability tables can be organized into a radix tree. When performing operations on a kernel object, a capability number must be passed to designate the position of this capability within the capability table of the current process. A capability number can encode at most two levels of lookup. We call the range that can be reached by a one-level lookup the main capability table, and the range that can be reached only by a two-level lookup the expanded capability table. On 32-bit systems, the capability number is a 16-bit value; on 64-bit systems, the capability number is a 32-bit value. On systems with a higher number of bits, the pattern continues by analogy. The specific encoding method is as follows:

Table 2-2 Encoding of Capability Numbers

System bit width	Encoding type	Encoding method
32-bit system	One-level lookup encoding	[15:8] reserved [7] fixed at 0 [6:0] position
	Two-level lookup encoding	[15] reserved [14:8] subtable position [7] fixed at 1 [6:0] position within the subtable
64-bit system	One-level lookup encoding	[32:16] reserved [15] fixed at 0 [14:0] position
	Two-level lookup encoding	[32] reserved [31:16] subtable position [15] fixed at 1 [14:0] position within the subtable

It can be seen that on 32-bit systems, a single capability table can contain at most 2^7=128 capabilities, while on 64-bit systems this value is 2^{15}=32768. In RME this value is represented by the macro RME_CID_2L. Unlike systems such as seL4, RME's capability table does not support radix-tree-based lookup with an unlimited number of levels. That is, the capabilities in the capability table[8] pointed to by a capability table capability inside the expanded capability table are not considered to be within the expanded capability table and cannot be used directly through two-level lookup encoding[9]. To use them, they must be delegated into the expanded capability table through the capability delegation call before they can be invoked through two-level lookup encoding.

As shown in the figure below, in the main capability table there is a capability table capability pointing to an expanded capability table, and it is placed at the position of capability number 4. The capabilities in this expanded capability table can also be accessed through two-level lookup encoding. At the position of capability number 2 of this expanded capability table there is a capability table capability pointing to a second-level expanded capability table; this capability itself can be accessed using the two-level lookup encoding whose value is 4:2, but the capabilities in the second-level expanded capability table it points to cannot be directly accessed through two-level lookup encoding. The part of the figure that cannot be accessed from the main capability table is shown in gray.

[8] that is, the second-level expanded capability table
[9] that is, "the vassal of my vassal is not my vassal"

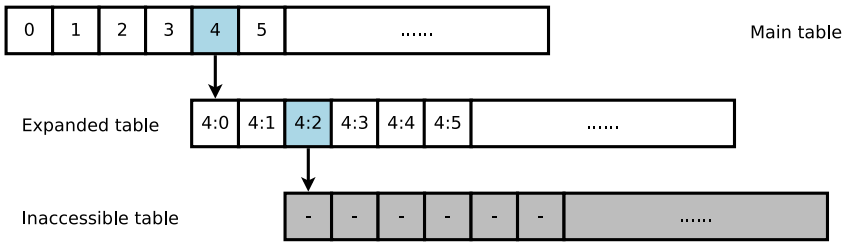


Figure 2-3 Capability Table Structure Diagram

2.3 Capability Table Function List

The kernel functions related to capability tables are as follows:

Table 2-3 Capability Table Related Kernel Functions

Call number	Type	Purpose
RME_SVC_CPT_CRT	System call	Create a capability table
RME_SVC_CPT_DEL	System call	Delete a capability table
RME_SVC_CPT_ADD	System call	Capability delegation
RME_SVC_CPT_FRZ	System call	Capability freezing
RME_SVC_CPT_REM	System call	Capability removal

The operation flags of capability table capabilities are as follows:

Table 2-4 Capability Table Operation Flags

Flag	Bit	Purpose
RME_CPT_FLAG_CRT	[0]	Allow capabilities to be created in this capability table.
RME_CPT_FLAG_DEL	[1]	Allow capabilities in this capability table to be deleted.
RME_CPT_FLAG_FRZ	[2]	Allow capabilities in this capability table to be frozen.
RME_CPT_FLAG_ADD_SRC	[3]	Allow this capability table to serve as the source table in capability delegation.
RME_CPT_FLAG_ADD_DST	[4]	Allow this capability table to serve as the destination table in capability delegation.
RME_CPT_FLAG_REM	[5]	Allow capabilities in this capability table to be removed.
RME_CPT_FLAG_PRC_CRT	[6]	Allow this capability table to be used as the process's capability table when creating a process.
RME_CPT_FLAG_PRC_CPT	[7]	Allow this capability table to replace the capability table of a process.

For bit[6] and bit[7] in the table above, please refer to4.4.1 and4.4.3 .

2.3.1 Capability Table Creation

This operation creates a capability table and places its capability into some existing capability table. A capability table is a kernel object, so the kernel memory it uses must be designated at creation time. In

RME, we adopt the method of a kernel memory capability plus a kernel-relative start virtual address to determine the real kernel virtual address of the kernel object. Each kernel memory capability designates an allowed range[10], and only kernel memory within this range can be used when creating kernel objects. The real kernel virtual address of the kernel object is the kernel start absolute virtual address of this kernel memory capability plus the kernel-relative start virtual address (Relative Address, Raddr) additionally designated when creating the kernel object. This point is the same for other kernel objects including capability tables, and will not be repeated in later chapters. For detailed information about kernel memory capabilities, please refer to3.4 .

The capability table creation operation requires the following parameters:

Table 2-5 Parameters Required for Capability Table Creation

Parameter name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CPT_CRT.
Cap_Cpt_Crt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created capability table capability. This capability number can be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	P1.D1	A kernel memory capability number; the kernel memory range it identifies must be able to hold the entire capability table, and it must have the RME_KOM_FLAG_CPT attribute. This capability number can be a one-level or two-level lookup encoding.
Cap_Crt	cid_t	P1.D0	A capability number corresponding to some position of the capability table that is to receive this newly created capability table capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Raddr	ptr_t	P2	The kernel-relative start virtual address to be used by the newly created capability table. This address is relative to the start virtual address of the kernel capability corresponding to Cap_Kom.
Entry_Num	ptr_t	P3	The number of entries contained in this capability table, which must be between 1 and RME_CID_2L. If RME_CPT_ENTRY_MAX[11] is defined as a value that is less than RME_CID_2L and not 0, then this number must be between 1 and RME_CPT_ENTRY_MAX.

[10] consisting of the kernel memory start absolute virtual address and the kernel memory end absolute virtual address
[11] see 7.3.2

The return values of this operation may be as follows:

Table 2-6 Possible Return Values for Capability Table Creation

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The passed parameter for the number of capability table capabilities is outside the allowed range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Crt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Crt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt_Crt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has already been frozen.
RME_ERR_CPT_FROZEN	Cap_Crt is frozen, or another core is creating a capability at that position.
RME_ERR_CPT_TYPE	Cap_Cpt_Crt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_FLAG	Cap_Cpt_Crt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Kom does not have the RME_KOM_FLAG_CPT attribute, or the range is incorrect.
RME_ERR_CPT_EXIST	Cap_Crt is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.

2.3.2 Capability Table Deletion

This operation deletes a capability table. The capability table being deleted must not contain capabilities; that is, all of its capability positions should be empty. Deleting a capability table requires the following parameters:

Table 2-7 Parameters Required for Capability Table Deletion

Parameter name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CPT_DEL.
Cap_Cpt_Del	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table that contains the capability table capability being deleted. This capability number can be a one-level or two-level lookup encoding.

Parameter name	Type	Field	Description
Cap_Del	cid_t	P1	A capability number corresponding to the capability table capability that is to be deleted. The capability corresponding to this capability number must be a capability table capability. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 2-8 Possible Return Values for Capability Table Deletion

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Del is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Del is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt_Del has already been frozen.
RME_ERR_CPT_FROZEN	Cap_Del has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt_Del is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Del is not a capability table capability.
RME_ERR_CPT_NULL	Cap_Del is an empty capability.
RME_ERR_CPT_NULL	Two cores simultaneously try to delete this capability table; the core that does not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt_Del does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Del is not quiescent.
RME_ERR_CPT_EXIST	The capability table corresponding to Cap_Del still contains capabilities.
RME_ERR_CPT_REFCNT	The reference count of Cap_Del is not 0, or it is not a root capability.

2.3.3 Capability Delegation

This operation delegates a certain capability in one capability table into an empty position of another capability table. The parent capability of the newly created destination capability is the source capability, and the reference count of the source capability is incremented by 1. Performing capability delegation between capability tables requires the following parameters:

Table 2-9 Parameters Required for Capability Delegation

Parameter name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CPT_ADD.

Parameter name	Type	Field	Description
Cap_Cpt_Dst	cid_t	P1.D1	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_ADD_DST attribute; the capability corresponding to this capability number points to the destination capability table. This capability number can be a one-level or two-level lookup encoding.
Cap_Dst	cid_t	P1.D0	A capability number corresponding to the position that will receive the delegated capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Cpt_Src	cid_t	P2.D1	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_ADD_SRC attribute; the capability corresponding to this capability number points to the source capability table. This capability number can be a one-level or two-level lookup encoding.
Cap_Src	cid_t	P2.D0	A capability number corresponding to the capability to be delegated. The capability corresponding to this capability number must not be empty and must not be frozen. This capability number can only be a one-level lookup encoding.
Flags	ptr_t	P3	The operation flag attributes to be delegated. Only the operations permitted by this operation flag can be performed by the newly created capability.

It is worth noting that for kernel memory capabilities, additional bits placed in the system call number N and the capability table capability number C are also required during delegation to help determine their operation flag attributes. For the specific parameter passing method, please refer to3.4 .

The return values of this operation may be as follows:

Table 2-10 Possible Return Values for Capability Delegation

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Dst or Cap_Cpt_Src is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Dst or Cap_Src is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt_Dst or Cap_Cpt_Src is frozen.
RME_ERR_CPT_FROZEN	Cap_Src is frozen.
RME_ERR_CPT_TYPE	Cap_Cpt_Dst or Cap_Cpt_Src is not a capability table capability.
RME_ERR_CPT_NULL	Cap_Src is an empty capability.

Return value	Meaning
RME_ERR_CPT_FLAG	Cap_Cpt_Src does not have the RME_CPT_FLAG_ADD_SRC attribute.
RME_ERR_CPT_FLAG	Cap_Cpt_Dst does not have the RME_CPT_FLAG_ADD_DST attribute.
RME_ERR_CPT_FLAG	The operation flag attributes of Cap_Src conflict with the passed operation flag attributes; that is, the passed attributes include operations or operation ranges that Cap_Src does not permit.
RME_ERR_CPT_FLAG	The passed operation flag attributes are invalid, for example the upper and lower bounds of the operation range conflict, or no operation at all is permitted on the capability produced by the delegation.
RME_ERR_CPT_EXIST	Cap_Dst is not an empty capability.
RME_ERR_CPT_REFCNT	The reference count of Cap_Src exceeds the maximum range allowed by the system. On 32-bit systems the upper limit is $2^{23} - 1$, and on 64-bit systems the upper limit is $2^{46} - 1$. This is usually sufficient.

2.3.4 Capability Freezing

This operation freezes a capability in a capability table. If a capability is frozen, then after the quiescence period it is guaranteed that all operations on the kernel object pointed to by this capability that are initiated from this capability have stopped in the kernel, and at this point the capability can be deleted or removed. Note that this is not equivalent to all operations on the kernel object pointed to by this capability having stopped, because there may also be other capabilities pointing to this kernel object, and kernel object operations initiated from these capabilities can still proceed. Only if the root capability is frozen is it guaranteed that all operations on this kernel object have stopped, and only then can this capability and the kernel object be deleted. The capability freezing operation is also performed atomically by a compare-and-swap operation; however, when it fails, we know that some other core must have successfully helped us freeze the capability, so success is still returned.

Freezing a capability requires the following parameters:

Table 2-11 Parameters Needed for Capability Freezing

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CPT_FRZ.
Cap_Cpt_Frz	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_FRZ attribute; the capability referenced by this capability number points to the capability table containing the capability being frozen. This capability number may be a one-level or two-level lookup encoding.
Cap_Frz	cid_t	P1	A capability number corresponding to the capability to be frozen. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 2-12 Possible Return Values for Capability Freezing

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Frz is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Frz is out of range.
RME_ERR_CPT_FROZEN	The capability found by the one-level/two-level lookup of Cap_Cpt_Frz is already frozen.
RME_ERR_CPT_FROZEN	Cap_Frz is already frozen and does not need to be frozen again, or it is being created.
RME_ERR_CPT_TYPE	Cap_Cpt_Frz is not a capability table capability.
RME_ERR_CPT_NULL	Cap_Frz is an empty capability.
RME_ERR_CPT_FLAG	Cap_Cpt_Frz does not have the RME_CPT_FLAG_FRZ attribute.
RME_ERR_CPT_QUIE	Cap_Frz is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Frz is not 0.

2.3.5 Capability Removal

This operation removes a capability from a capability table. The capability being removed must not be a root capability[12]. Removing a capability does not cause the kernel object associated with it to be removed; only the capability itself is removed[13]. The capability removal operation is also performed atomically by a compare-and-swap operation; however, when it fails, we know that some other core must have successfully helped us remove the capability, so success is still returned.

[12] The deletion operation should be used for root capabilities

[13] The deletion operation, by contrast, can only be used on a root capability that is not referenced, and it deletes both the root capability and the kernel object at the same time

Removing a capability requires the following parameters:

Table 2-13 Parameters Needed for Capability Removal

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CPT_REM.
Cap_Cpt_Rem	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_REM attribute; the capability referenced by this capability number points to the capability table containing the capability being removed. This capability number may be a one-level or two-level lookup encoding.
Cap_Rem	cid_t	P1	A capability number corresponding to the capability to be removed. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 2-14 Possible Return Values for Capability Removal

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Rem is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Rem is out of range.
RME_ERR_CPT_FROZEN	The capability found by the one-level/two-level lookup of Cap_Cpt_Rem is already frozen.
RME_ERR_CPT_FROZEN	Cap_Rem is not frozen.
RME_ERR_CPT_TYPE	Cap_Cpt_Rem is not a capability table capability.
RME_ERR_CPT_NULL	Cap_Rem is an empty capability.
RME_ERR_CPT_NULL	Two cores attempt to remove this capability simultaneously; in this case the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt_Rem does not have the RME_CPT_FLAG_REM attribute.
RME_ERR_CPT_QUIE	Cap_Rem is not quiescent.
RME_ERR_CPT_ROOT	Cap_Rem is a root capability and cannot be removed; please use the deletion operation.

2.4 Bibliography of This Chapter

[1] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," Communications of the ACM, vol. 9, pp. 143-155, 1966.

[2] J. S. Shapiro, J. M. Smith, and D. J. Farber, EROS: a fast capability system vol. 33: ACM, 1999.

[3] R. J. Feiertag and P. G. Neumann, "The foundations of a provably secure operating system (PSOS)," in Proceedings of the National Computer Conference, 1979, pp. 329-334.

[4] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[6] B. W. Lampson, "Protection," ACM SIGOPS Operating Systems Review, vol. 8, pp. 18-24, 1974.

3. Page Tables and Memory Management

3.1 Memory Management Overview

Memory management (Memory Management) refers to the operations of allocating and reclaiming physical memory (Physical Memory) and the virtual address space (Virtual Address Space, VAS). To guarantee the safety of memory management, support at the hardware level is necessary. The memory management of the RME operating system supports memory protection unit (Memory Protection Unit, MPU) environments and memory management unit (Memory Management Unit, MMU) environments, and they are abstracted into the same page table data structure. The terms used in this chapter are defined as follows:

Table 3-1 Page Directory Terminology Definitions

Term	Definition
Page table (Page Table, PT)	Refers to the entire page table itself, including the top-level page directory and the page directories at each intermediate level; it is the general term for an address tree.
Page directory (Page Directory, PD)	Refers to one level of the page table, of which the topmost level is called the top-level page directory.
Page entry (Page Entry, PE)	Refers to an entry in the page directory table; it may point to a page, or it may point to the next-level page directory. Among these, the ones that point to a page are called page terminal entries (Page Terminal Entry, PTE), and those that point to a next-level page directory are called page intermediate entries (Page Intermediate Entry, PIE).

3.2 Page Table Operations and Structure

A page table is a multi-level radix trie (Radix Trie/Radix Tree) structure composed of a series of page directories. Each slot of a page directory is fixed to the length of one machine word. Each slot in a page directory can hold one of the following three kinds of data: the physical address of a lower-level page directory, the physical address and attributes of a page table entry, or an empty page table entry. If what is stored is the physical address of a lower-level page directory, then it means that a lower-level page directory exists here, and the mapping relationship of that part of the virtual address is to be determined by looking it up in that page directory; if what is stored is the physical address and attributes of a page table entry, then it means that a page table entry is mapped at this virtual address, and the access attributes of that page can be known; if what is stored is an empty page table entry, then it means that nothing is mapped at this virtual address. In RME, page tables must be constructed manually by the user, which is consistent with the solution approach of a series of microkernels such as Composite[1].

3.2.1 Kernel Memory and User Memory

The system memory of RME is divided into two parts: one part is kernel memory, and one part is user memory. Unlike microkernels such as Composite and L4, in the RME system the kernel memory mapping is fully established at the very beginning[1] and cannot be modified. This makes RME completely free of the need for a memory dynamic typing mechanism[2], and the memory management system is fully parallelizable, while at the same time completely eliminating the possibility of kernel memory content leakage. For MMU environments, when creating a top-level page directory, the system maps the kernel page directory created at boot time into the top-level page directory; for MPU environments, since kernel mode usually has access permission to the entire memory, we only need user pages in all page directories[3].

At system boot, all user physical memory pages are added to the page table of the boot process (Init). After this addition is complete, generally the system is no longer allowed to create physical memory page frames out of thin air, unless a custom kernel call is used[4]. When creating a new process, the page table entries of the new process's page table must be added from other processes. During the addition, the access attributes of this page can be specified, and the specified access attributes must be a subset of the access attributes of the parent page. In the RME system, the standard attributes for page access are as follows:

Table 3-2 Standard Page Access Attributes

Name	Identifier	Meaning
Readable	RME_PGT_READ	This page is readable.
Writable	RME_PGT_WRITE	This page is writable.
Executable	RME_PGT_EXECUTE	This page can be executed as code.
Cacheable	RME_PGT_CACHE	The contents of this page can be cached.
Bufferable	RME_PGT_BUFFER	Writes to this page can be buffered.
Static	RME_PGT_STATIC	This page is a static page that is always mapped. This means that in MPU environments or in MMU environments with manual TLB updates, this page is always mapped, rather than being mapped only when a page fault occurs.

It is worth noting that on some architectures, some of the above bits may not all be meaningful. For example, for the vast majority of MMU environments that automatically update the translation look-aside buffer (Trans Look-aside Buffer, TLB), the static attribute is meaningless; for some architectures, read and write are implemented together, so there is no separate read and write control.

To enable user mode to manage kernel memory when creating kernel objects, the kernel objects in the system are managed using a kernel object registry. The kernel object registry is a bitmap that stores the occupancy of virtual addresses by kernel objects. This bitmap guarantees that two kernel objects cannot exist on the same segment of kernel virtual address at the same time.

To prevent certain system components that have the right to create kernel objects from creating a large number of kernel objects when they fail or are compromised, thereby exhausting kernel memory and launching a denial-of-service (Denial of Service, DoS) attack, a kernel memory capability is therefore introduced to manage kernel memory. The concept of the kernel memory capability draws on the kernel object factory (Factory) of Fiasco.OC[4]. When creating any kernel object, a kernel memory capability is required; this kernel memory capability indicates the kernel virtual memory address range permitted for

[1] Static or semi-static allocation

[2] This mechanism is adopted by systems such as Composite and seL4[1][3]. In RME it can actually also be implemented by a trusted user-mode server

[3] Some architectures (such as Tricore) need to reserve one or more MPU register groups for kernel mode, but these pages are not reflected in the page table

[4] See Section 8.1

use in creating kernel objects, as well as which objects are permitted to be created on this memory. Only when the kernel object being created falls entirely within this range, and the flag bits of the kernel memory capability permit the creation of this kind of object, can the creation operation continue; otherwise an error is returned, thereby restricting certain infected malicious kernel components from launching denial-of-service attacks.

3.2.2 Page Directory Attributes

In the RME operating system, a page directory has four attributes. These four attributes uniquely determine the state of the page directory. We introduce these four attributes separately below.

3.2.2.1 Mapping Base Address

The mapping base address refers to the virtual address at which this level of page directory begins mapping. The physical memory of the first slot of this page directory, or the base address of the second-level page table it maps, is this virtual address. When we attempt to map a lower-level page directory to a certain position of a certain higher-level page directory, we may need to check whether the lower-level virtual address matches the higher-level virtual address, so as to determine whether the mapping can be performed. Of course, this check is necessary only in systems that use an MPU. In systems that use an MMU, since a page directory is often mapped into different positions of different page directories, this check can be configured not to be performed; in that case the mapping base address item is invalid, and the path-compressed page table (Path-Compressed Page Table, PCPT) format also cannot be used[5]. For the differences regarding the path-compressed page table format, MPU systems and MMU systems, please see the following chapters.

3.2.2.2 Top-Level Page Directory Flag

Indicates that this page directory is the topmost page directory. Only the topmost page directory can be used to create a process.

3.2.2.3 Page Directory Size Order

The page directory size order determines the size of the virtual address represented by each slot of the page directory. If the size order of a page directory is 12, then it means that each slot in this page directory corresponds to a page of size 2^{12}=4096 bytes.

3.2.2.4 Page Directory Number Order

The page directory number order determines the number of slots in the page directory. If the number order of a page directory is 10, then it means that this page directory has a total of 2^{10}=1024 slots.

3.2.3 Basic Page Directory Operations

There are altogether six basic operations on a page directory, as follows:

Table 3-3 Basic Page Directory Operations

Operation	Meaning
Create page directory (Create)	Create a new empty page directory.
Delete page directory (Delete)	Delete a page directory.
Map memory page (Add)	Add a physical memory page at a certain virtual address of the page directory.

[5] MMU systems generally do not support this format either

Operation	Meaning
Remove memory page (Remove)	Delete a physical memory page at a certain virtual address of the page directory.
Construct page directory (Construct)	Add a child page directory at a certain virtual address of a parent page directory.
Destruct page directory (Destruct)	Delete a child page directory mapped at a certain virtual address of the parent page directory.

The implementations of these six operations on MPU and MMU are very different, and they also have different restrictions. Please refer to the explanations in the following two sections to understand the specific differences.

3.2.4 Implementation of a Normal Multi-Level Page Table Under the Memory Management Unit

For the memory management unit, the implementation of a page directory is very simple: it is just a simple linear table. For example, for the x86-64 page table, its first-level page directory (PML4) has fixed 512 slots, each slot representing 2^{39} bytes; the second-level page directory (PDP) also has fixed 512 slots, each slot representing 2^{30} bytes; the third-level page directory (PGD) also has fixed 512 slots, each slot representing 2^{21} bytes; and the fourth-level page directory (PTE) also has fixed 512 slots, each slot representing 2^{12} bytes. The radix tree formed by these four levels of page directories is the entire page table, as shown in the figure below. Among them, blue slots represent entries pointing to child page directories, while gray slots represent mapped pages. In addition, the memory base addresses of these page directories should all be aligned to 4kB, so that they each occupy exactly one page.

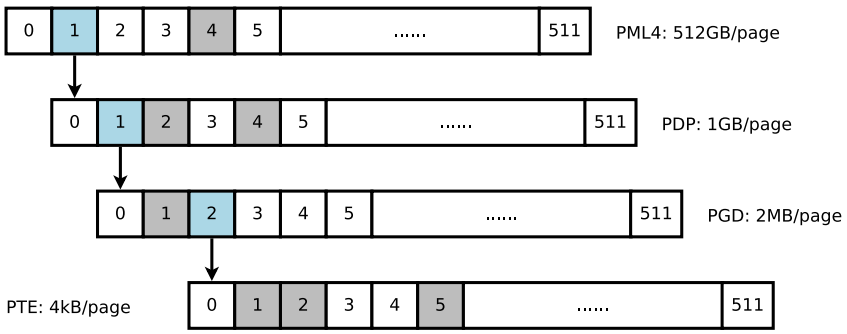


Figure 3-1 An Example of a Normal Multi-Level Page Table

In addition, if a page directory is allowed to be constructed into any virtual address slot of a higher-level page directory[6], then we can configure the kernel not to check whether the starting virtual address is appropriate. In this case, we are using a normal multi-level page table, and there is no need to implement path compression.

Since the processor has hardware that handles page tables directly, we do not need to generate page table metadata specifically for the processor. However, it should be noted that when the page table creation function is called, the created page table should conform to the requirements of the hardware page table lookup mechanism. For those MMUs whose TLB is filled purely by software, there is no such requirement; tree structures that logically conform to the form of a page table can be created at will, and these MMUs can also be regarded as region-based MMUs and handled in the way described in 3.2.5.

[6] As long as the size order and number order are appropriate

Some common features of MMUs are as follows:

Table 3-4 Features of Common MMUs

Processor	Page Table Levels	Page Size	Other Features
ARM926EJ-S	2 or 3 levels	1MB, 64kB, 4kB, 1kB	Manual locking of some TLB entries
x86-64 (AMD64)	3 levels	1GB, 2MB, 4kB	Additional segment-based memory management unit
Itanium (IA-64)	4 levels	256MB, 16MB, 4MB, 1MB, 256kB, 64kB, 8kB, 4kB	Partially manually fillable TLB
e200 (PowerPC)	Not applicable	All powers of 2 from 1kB to 4GB	Purely software-filled TLB
ARMv7-A (32-bit)	2 or 3 levels	4kB, 64kB, 1MB, 16MB	Manual locking of some TLB entries
ARMv8-M (64-bit)	3 or 4 levels	4kB, 16kB, 64kB	Optional stage-2 translation under virtualization
MIPS64	Not applicable	All powers of 4 from 1kB to 256MB	Purely software-filled TLB
TMS320C66X	Not applicable	All powers of 2 from 4kB to 4GB	Purely software-filled TLB

3.2.5 Implementation of a Path-Compressed Page Table Under the Memory Protection Unit

Under the memory protection unit[7], the processor often cannot directly recognize multi-level page tables. This obliges us to generate MPU metadata from the page table, to be used to set up the MPU efficiently at process switch. Moreover, in MPU environments there are also the following characteristics:

1. The number of MPU regions is often limited; for example, ARMv7-M has at most 16 regions, and each region can in turn be divided into 8 subregions. Therefore, in one process we can allow at most 128 regions to be mapped simultaneously, and these regions must also satisfy a series of stringent conditions such as address alignment. Therefore, we can consider dividing pages into two classes: one class is static pages, which are always mapped and can be used by applications requiring predictability; the maximum number of static pages is exactly the maximum number of MPU regions allowed by the processor. The other class is dynamic pages, which are mapped only when in use and are not guaranteed to be mapped at all times; the maximum number of dynamic pages is unlimited. If a dynamic page that is not currently mapped is accessed, the processor enters the page fault exception, and we then manually look up the page table to add this dynamic page to the MPU metadata; at this point, if there are not enough MPU regions, other dynamic pages may be replaced[2].
2. In a page table, often only one or two entries exist. For microcontrollers that use an MPU, maintaining the existence of multi-level page tables is an unnecessary waste of resources, so a way should be found to compress the page table. Compared with an ordinary page table, in a compressed page table the number of address bits translated by different page intermediate entries of the same page directory can be different. For example, at a certain virtual address there is a very small page that we need to add to the access range. For an ordinary page table, we need multiple levels of intermediate page

[7] Or a manually filled, region-based memory management unit; the same below

directories, and then at the last-level page directory we add this page. For a compressed page table, however, a single level of page directory is enough to address this page. We can state the base address, number order and size order of this page directory, and then directly construct it into the upper-level page directory. Of course, in this case we require that the virtual address range represented by this level of page table fall within the virtual address range permitted by the corresponding page table entry of the upper-level page directory. There is one exception to the above situation: when the architecture actually has a region-based MMU unit, we usually do not use a path-compressed page table.

3. The MPU cannot perform physical-address-to-virtual-address translation. Therefore, the virtual address is always equal to the physical address. This obliges us to check, when mapping a page, whether the mapping address of the page is equal to the physical address.

4. Since we need MPU metadata to speed up MPU filling, when we modify any level of page directory we must maintain consistency between the metadata and the page table. It is also possible not to maintain this consistency, but if we do so, we can only update the metadata through page faults. And updating the metadata through page faults is not always possible: some architectures do not provide key information such as the faulting address in some or all cases. For detailed information on this approach, please refer to 3.2.5.2.

   An example of a path-compressed page table on ARMv7-M is shown in the figure below. Green slots represent path-compressed entries pointing to child page directories, blue slots represent non-path-compressed entries pointing to child page directories, and gray slots represent mapped pages. The hexadecimal numbers in front of the page tables at each level represent the starting mapping address of that page table.

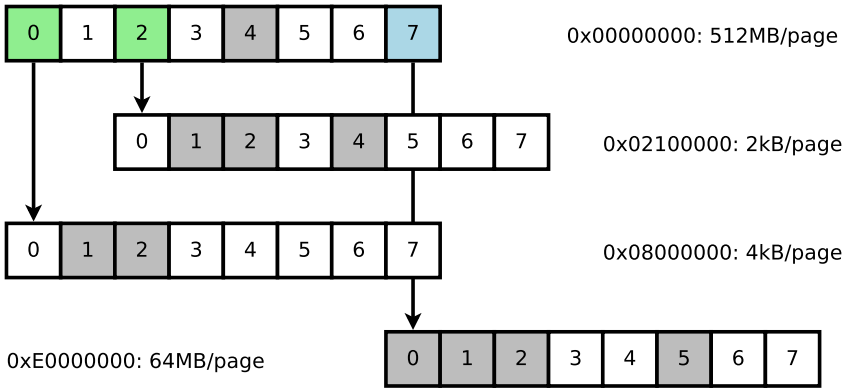


Figure 3-2 An Example of a Path-Compressed Page Table

Some common features of MPUs are as follows:

Table 3-5 Features of Common MPUs

Processor	Number of Regions	Region Organization	Size Range	Alignment Requirement	Other Features
ARM v6-M	0 to 16	Unified organization	128B-4GB	Aligned to size	8 subregions
ARM v7-M	0 to 16	Unified organization	128B-4GB	Aligned to size	8 subregions
ARM v8-M	0 to 16	Unified organization	128B-4GB	None	None

Processor	Number of Regions	Region Organization	Size Range	Alignment Requirement	Other Features
Tensilica L106	16 or 32	Unified organization	4kB-1GB	None	None
MIPS M14k	1 to 16	Unified organization	Arbitrary	None	Can be locked as read-only
PowerPC e200z4	32	16 each for code/data	Arbitrary	None	None
AVR32	8	Unified organization	4kB-4GB	Aligned to size	16 subregions
MSP430FRXX	3	Unified organization	Arbitrary	None	Three-segment segmented
Coldfire-MCF	4	2 each for code/data	16MB-4GB	Aligned to size	None
RISC-V	0 to 16	Unified organization	4B-4GB	Aligned to size	Can also be aligned to words

In summary, for the implementation of page tables under an MPU, the following two forms are common:

3.2.5.1 Immediately Update MPU Metadata When Updating the Page Table

This approach places the MPU metadata only in the top-level page directory, and requires construction to start from the top level. If any level of page directory does not have a top-level page directory and is not itself a top-level page directory, then no child page directory can be constructed into this page directory. In addition, any two page tables cannot share a page directory or a page directory tree. The restrictions of this implementation are as follows:

Table 3-6 Restrictions of Immediately Updating MPU Metadata When Updating the Page Table

Operation	Restriction or Disadvantage
Create process	No restriction.
Change process page table	No restriction.
Switch process	No restriction; simply use the MPU metadata in the top-level page directory.
Create page directory	No restriction.
Delete page directory	No restriction.
Map memory page	If it has a top-level page directory itself, update the MPU metadata of the top-level page directory.
Remove memory page	If it has a top-level page directory itself, update the MPU metadata of the top-level page directory.
Construct page directory	The parent page directory must itself have a top-level page directory, or must itself be a top-level page directory; the child page directory must not have a top-level page directory, and must not itself be a top-level page directory[8]. Add the mapped pages of the child page directory to the MPU metadata of the top-level page directory.

[8] And it is impossible for it to have child page directories either

Operation	Restriction or Disadvantage
Destruct page directory	The child page directory must have a top-level page directory, and must not contain any child page directories[9]. Remove the mapped pages of the child page directory from the MPU metadata of the top-level page directory.
Memory consumption	MPU metadata exists only in the top-level page directory.

This implementation is the recommended implementation on single-core systems. This implementation maximizes the efficiency of MPU-based microcontrollers, so that we can often use pre-generated MPU metadata to set the MPU registers in bulk, with only some loss of infrequently used functionality. When the page table structure changes, its MPU metadata is also always updated immediately, so there is no potential real-time problem caused by page faults.

On some architectures, it is sometimes[10] or always[11] impossible to determine the faulting address in a page fault, and impossible to dynamically map pages based on the address. Because of the lack of exception handling information, such architectures do not support dynamic pages in some or all cases.

3.2.5.2 Update MPU Metadata on Page Fault

This approach treats the MPU metadata placed at the top-level page directory as a software-filled TLB, using a software page table walk algorithm in the memory protection interrupt to fill it on every miss. It does not attempt to generate the MPU metadata corresponding to the entire page table at once, but instead chooses to generate it gradually. Whenever the page table structure changes or the page mapping changes, all MPU metadata is cleared, which is equivalent to a TLB flush under the MMU architecture. This approach also supports static pages and dynamic pages, and a static page incurs a page fault only on its first access, while subsequent accesses can guarantee real-time behavior.

Table 3-7 Restrictions of Updating MPU Metadata on Page Fault

Operation	Restriction or Disadvantage
Create process	No restriction.
Change process page table	Clear the MPU metadata, ready for rebuilding.
Switch process	No restriction.
Create page directory	No restriction.
Delete page directory	No restriction.
Map memory page	No restriction.
Remove memory page	Clear the MPU metadata, ready for rebuilding.
Construct page directory	No restriction.
Destruct page directory	Clear the MPU metadata, ready for rebuilding.
Memory consumption	MPU metadata exists only in the top-level page directory.

This implementation preserves the greatest compatibility with MMU systems, and is also the recommended implementation approach on multi-core MPU architectures. It imposes no requirement on the construction order of the page table, is also convenient for operations in multi-core environments, and multiple page tables can share a part in any manner, but its real-time performance is slightly worse than the scheme listed in 3.2.5.1.

[9] Its child page directories must be destructed from this page directory in advance

[10] Such as ARMv7-M: the page fault generated during hardware push and pop does not provide the faulting address, so dynamic pages cannot be used for the stack

[11] Such as ARMv6-M: because it is extremely streamlined, it does not provide any exception handling information under any circumstances, so it does not support dynamic pages at all

3.3 Page Table Function List

The kernel functions related to page tables are as follows:

Table 3-8 Page Table Related Kernel Functions

Call Number	Type	Purpose
RME_SVC_PGT_CRT	System call	Create page directory
RME_SVC_PGT_DEL	System call	Delete page directory
RME_SVC_PGT_ADD	System call	Map memory page
RME_SVC_PGT_REM	System call	Remove memory page
RME_SVC_PGT_CON	System call	Construct page directory
RME_SVC_PGT_DES	System call	Destruct page directory

The operation flags of a page table capability are as follows:

Table 3-9 Page Table Operation Flags

Flag	Bit	Purpose
RME_PGT_FLAG_ADD_SRC	[0]	Allows this page directory to act as the source directory in page frame passing.
RME_PGT_FLAG_ADD_DST	[1]	Allows this page directory to act as the destination directory in capability delegation.
RME_PGT_FLAG_REM	[2]	Allows removing page frames in this page directory.
RME_PGT_FLAG_CHILD	[3]	Allows this page directory to act as a child page directory in page table construction or destruction.
RME_PGT_FLAG_CON_PARENT	[4]	Allows this page directory to act as a parent page directory in page table construction.
RME_PGT_FLAG_DES_PARENT	[5]	Allows this page directory to act as a parent page directory in page table destruction.
RME_PGT_FLAG_PRC_CRT	[6]	Allows this page table to be used as the process page table when creating a process.
RME_PGT_FLAG_PRC_PGT	[7]	Allows this page table to be used to replace the page table of a process.
Other bits	Bit field	Operation range attributes.

Regarding bit [6] and bit [7] in the above table, please refer to 4.4.1 and 4.4.3. When filling in operation flags in the page-table-related kernel functions, the RME_PGT_FLAG(HIGH,LOW,FLAG) macro must be used for filling, where HIGH is the upper limit of the operation position Pos, and LOW is the lower limit of the operation position Pos; the closed interval formed by [HIGH, LOW] is the allowed range of Pos. FLAG consists of the respective allowed operation flags in bits [7:0].

3.3.1 Creating a Page Directory

This operation creates a page directory and places its capability into an existing capability table. The page directory creation operation requires the following parameters:

Table 3-10 Parameters Required for Creating a Page Directory

Parameter Name	Type	Field	Description
Svc_Num	ptr_t	N.Q0	Must be RME_SVC_PGT_CRT.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive the capability of this newly created page directory. This capability number may be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	P1.D1	A kernel memory capability number; the kernel memory range it identifies must be able to hold the entire page directory, and it must have the RME_KOM_FLAG_PGT attribute. This capability number may be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	P1.Q1	A capability number corresponding to a certain position in the capability table that receives the capability of this newly created page directory. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Raddr	ptr_t	P2	The kernel relative starting virtual address to be used by the newly created page directory. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.
Base	ptr_t	P3	The mapping base address of the newly created page directory; the last bit is the top-level flag, see below.
Is_Top	ptr_t	P3[0]	Whether this page directory is a top-level page directory. “1” means that this page directory is top-level.
Size_Order	ptr_t	P1.Q0	The size order of this page directory[12].
Num_Order	ptr_t	N.Q1	The number order of this page directory.

[12] Refers to the memory page size represented by each page table entry

The possible return values of this operation are as follows:

Table 3-11 Possible Return Values for Creating a Page Directory

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has been frozen.
RME_ERR_CPT_FROZEN	Cap_Pgt is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Kom has no RME_KOM_FLAG_PGT attribute, or the range is incorrect.
RME_ERR_CPT_EXIST	Cap_Pgt is not an empty capability.
RME_ERR_CPT_KOT	Allocation of kernel memory failed.
RME_ERR_PGT_HW	Underlying hardware restriction; creating such a page directory is not allowed.

3.3.2 Page Directory Deletion

This operation deletes a page directory. The page directory being deleted must not contain child page directories. Deleting a page directory requires the following parameters:

Table 3-12 Parameters Needed for Page Directory Deletion

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGT_DEL.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table containing the page directory capability being deleted. This capability number can be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	P1	A capability number corresponding to the page directory capability to be deleted. The capability corresponding to this capability number must be a page directory capability. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 3-13 Possible Return Values for Page Directory Deletion

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Pgt has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page directory capability.
RME_ERR_CPT_NULL	Cap_Pgt is an empty capability.
RME_ERR_CPT_NULL	Two cores attempt to delete this page directory at the same time; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Pgt is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Pgt is not 0, or it is not a root capability.
RME_ERR_PGT_HW	Restricted by the underlying hardware, deleting this page directory is not permitted. This may be because the page directory contains child page directories or for various other reasons.

3.3.3 Page Mapping

This operation transfers a part of a page table entry in one page directory to an empty position in another page directory. Transferring a page table entry between page directories requires the following parameters:

Table 3-14 Parameters Needed for Page Mapping

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGT_ADD.
Cap_Pgt_Dst	cid_t	P1.D1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_ADD_DST attribute; the capability corresponding to this capability number points to the destination page directory. This capability number can be a one-level or two-level lookup encoding.
Pos_Dst	ptr_t	P1.D0	A destination page table entry position in the destination page directory that is to receive the transfer. This page table entry must be empty.
Flag_Dst	ptr_t	C	The attribute of the destination page table entry. This attribute restricts the characteristics of the destination page table entry.
Cap_Pgt_Src	cid_t	P2.D1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_ADD_SRC attribute; the capability corresponding to this capability number points to the source page directory. This capability number can be a one-level or two-level lookup encoding.
Pos_Src	ptr_t	P2.D0	A source page frame position in the source page directory to be transferred. This page frame must be mapped.
Index	ptr_t	P3	The sub-position in the source page frame to be transferred.

The return values of this operation may be as follows:

Table 3-15 Possible Return Values for Page Mapping

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt_Dst or Cap_Pgt_Src is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt_Dst or Cap_Pgt_Src is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt_Dst or Cap_Pgt_Src is not a page directory capability.
RME_ERR_CPT_FLAG	Cap_Pgt_Src has no RME_PGT_FLAG_ADD_SRC attribute.
RME_ERR_CPT_FLAG	Cap_Pgt_Dst has no RME_PGT_FLAG_ADD_DST attribute.

Return Value	Meaning
RME_ERR_CPT_FLAG	The operation range attribute of Cap_Pgt_Dst or Cap_Pgt_Src does not permit this operation.
RME_ERR_PGT_ADDR	The size order of the destination page directory is greater than the size order of the source page directory, so the mapping cannot be performed.
RME_ERR_PGT_ADDR	Pos_Dst or Pos_Src exceeds the number of page table entries of the destination page directory or the source page directory.
RME_ERR_PGT_ADDR	Index exceeds the maximum number of sub-positions.
RME_ERR_PGT_ADDR	When the check that the physical address equals the virtual address is enabled, the mapped physical address differs from the destination virtual address.
RME_ERR_PGT_HW	The source page directory lookup failed. This may be because the position in the source page directory is empty.
RME_ERR_PGT_MAP	The mapping attempt failed for hardware reasons. The specific reason for the failure is hardware-dependent.
RME_ERR_PGT_PERM	The access control flags of the destination page are not a subset of the access control flags of the source page.

3.3.4 Page Removal

This operation removes a page table entry from a page directory, returning that position to the empty state. Removing a memory page requires the following parameters:

Table 3-16 Parameters Needed for Page Removal

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGT_REM.
Cap_Pgt	cid_t	P1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_REM attribute; the capability corresponding to this capability number points to the destination page directory. This capability number can be a one-level or two-level lookup encoding.
Pos	ptr_t	P2	A page table entry position in the destination page directory to be removed. This page table entry must be a mapped memory page.

The return values of this operation may be as follows:

Table 3-17 Possible Return Values for Page Removal

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page directory capability.
RME_ERR_CPT_FLAG	Cap_Pgt has no RME_PGT_FLAG_REM attribute.
RME_ERR_CPT_FLAG	The operation range attribute of Cap_Pgt does not permit this operation.
RME_ERR_PGT_ADDR	Pos exceeds the number of page table entries of the destination page directory.
RME_ERR_PGT_MAP	The removal attempt failed for hardware reasons. The specific reason for the failure is hardware-dependent.

3.3.5 Page Directory Construction

This operation places the physical address pointer pointing to the child page directory into an empty position of the parent page directory. If a compressed page table is used, the size of the child page directory must be less than or equal to one page of the parent page directory; otherwise, the size of the child page directory must exactly equal one page of the parent page directory. Constructing a page directory requires the following parameters:

Table 3-18 Parameters Needed for Page Directory Construction

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGT_CON.
Cap_Pgt_Parent	cid_t	P1.D1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_CON_PARENT attribute; the capability corresponding to this capability number points to the parent page directory. This capability number can be a one-level or two-level lookup encoding.
Pos	ptr_t	P2	A destination page table entry position in the destination page directory that is to receive the transfer. This page table entry must be empty.
Cap_Pgt_Child	cid_t	P1.D0	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_CHILD attribute; the capability corresponding to this capability number points to the child page directory. This capability number can be a one-level or two-level lookup encoding.

Parameter Name	Type	Domain	Description
Flag_Child	ptr_t	P3	The attribute when the child page directory is mapped. This attribute restricts the access permissions of all page directories below this mapping. For different architectures, the meaning of the value at this position is also different. For some architectures that do not support page directory attributes, this value is invalid.

The return values of this operation may be as follows:

Table 3-19 Possible Return Values for Page Directory Construction

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt_Parent or Cap_Pgt_Child is not a page directory capability.
RME_ERR_CPT_FLAG	Cap_Pgt_Parent has no RME_PGT_FLAG_CON_PARENT attribute.
RME_ERR_CPT_FLAG	Cap_Pgt_Child has no RME_PGT_FLAG_CHILD attribute.
RME_ERR_CPT_FLAG	The operation range attribute of Cap_Pgt_Parent does not permit this operation.
RME_ERR_PGT_ADDR	Pos exceeds the number of page table entries of the parent page directory.
RME_ERR_PGT_ADDR	The total size of the child page directory is greater than the size of one page of the parent page directory.
RME_ERR_PGT_ADDR	When the check that the physical address equals the virtual address is enabled[13], the mapped physical address is not equal to the destination virtual address.
RME_ERR_PGT_MAP	The construction attempt failed for hardware reasons. The specific reason for the failure is hardware-dependent; it may be that the hardware does not support this kind of mapping.

3.3.6 Page Directory Destruction

This operation removes a child page directory from a page directory, returning that position to the empty state. Destructing a page directory requires the following parameters:

Table 3-20 Parameters Needed for Page Directory Destruction

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGT_DES.

[13] Architectures based on a memory protection unit, such as ARMv7-M

Parameter Name	Type	Domain	Description
Cap_Pgt_Parent	cid_t	P1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_DES_PARENT attribute; the capability corresponding to this capability number points to the parent page directory. This capability number can be a one-level or two-level lookup encoding.
Pos	ptr_t	P2	A child page directory position in the destination page directory to be removed. This page table entry must be a mapped page directory.
Cap_Pgt_Child	cid_t	P3	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_CHILD attribute; the capability corresponding to this capability number points to the child page directory. This child page directory must be located at Pos of the parent page directory. This capability number can be a one-level or two-level lookup encoding.

The return values of this operation may be as follows:

Table 3-21 Possible Return Values for Page Directory Destruction

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt_Parent or Cap_Pgt_Child is not a page directory capability.
RME_ERR_CPT_FLAG	Cap_Pgt_Parent has no RME_PGT_FLAG_DES_PARENT attribute.
RME_ERR_CPT_FLAG	Cap_Pgt_Child has no RME_PGT_FLAG_CHILD attribute.
RME_ERR_PGT_ADDR	Pos exceeds the number of page table entries of the parent page directory.
RME_ERR_PGT_MAP	The destruction attempt failed for hardware reasons. The specific reason for the failure is hardware-dependent.

3.4 Kernel Memory Function List

There is only one kernel function related to kernel memory, namely the delegation of kernel memory capabilities. The initial kernel memory capability is created at system startup and cannot be deleted. The child capabilities produced by its delegation cannot be deleted, but can only be removed. A kernel memory capability has not only operation flags but also a virtual address range value aligned to 64 bytes. Among all kernel objects, only the four kinds of kernel objects listed in the table below require kernel memory.

The operation flags of the kernel memory capability are as follows:

Table 3-22 Kernel Memory Operation Flags

Flag	Bit	Purpose
RME_KOM_FLAG_CPT	[0]	Allows capability tables to be created on this segment of kernel memory.
RME_KOM_FLAG_PGT	[1]	Allows page directories to be created on this segment of kernel memory.
RME_KOM_FLAG_THD	[2]	Allows threads to be created on this segment of kernel memory.
RME_KOM_FLAG_INV	[3]	Allows synchronous invocations to be created on this segment of kernel memory.

When delegating a kernel memory capability, because a range parameter also needs to be passed in, the required information cannot be fully conveyed with only one parameter position P3. In this case, part of the system call number N and the capability table capability number C must be used to pass these parameters. The specific parameter passing rules are as follows:

Table 3-23 Parameters Needed for Kernel Memory Capability Delegation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N[5:0]	Must be RME_SVC_CPT_ADD.
Cap_Cpt_Dst	cid_t	P1.D1	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_ADD_DST attribute; the capability corresponding to this capability number points to the destination capability table. This capability number can be a one-level or two-level lookup encoding.
Cap_Dst	cid_t	P1.D0	A capability number corresponding to the capability that will receive the delegated capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Cpt_Src	cid_t	P2.D1	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_ADD_SRC attribute; the capability corresponding to this capability number points to the source capability table. This capability number can be a one-level or two-level lookup encoding.

Parameter Name	Type	Domain	Description
Cap_Src	cid_t	P2.D0	A capability number corresponding to the capability to be delegated. The capability corresponding to this capability number must be neither empty nor frozen. This capability number can only be a one-level lookup encoding.
Flag	ptr_t	P3	See below for the description.
Ext_Flag	ptr_t	N:C	See below for the description.

In the delegation of a kernel memory capability, P3 and N:C together determine the extended operation flag attribute of the newly produced kernel memory capability. N:C means combining the half-word N and the half-word C, where N is in the high half-word and C is in the low half-word, together forming one word. Since RME uses only the last 6 binary bits of N to represent the system call number, the remaining binary bits can be used to represent other information. Combined, N and C have X-6 binary bits in total (X is the machine word length measured in binary bits); together with the X binary bits provided by P3, there are 2X-6 binary bits in total. We reserve 6 binary bits for the operation flags, so the upper bound and lower bound of the kernel memory can each be allocated X-6 bits, which is exactly enough to represent a memory address aligned to 64 bytes.

The specific meaning of P3 (Flag) is as follows:

Table 3-24 Specific Meaning of P3 (Flag) in Kernel Memory Capability Delegation

Bit Field Range	Bit Field Meaning
High half-word (D1)	The high half-word of the upper bound of the kernel memory relative address.
Low half-word (D0)	The high half-word of the lower bound of the kernel memory relative address.

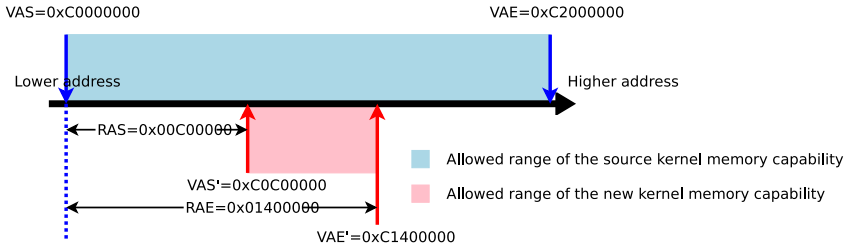


Figure 3-3 Example of Kernel Memory Capability Delegation

The specific meaning of N:C (Ext_Flag) is as follows:

Table 3-25 Specific Meaning of N:C (Ext_Flag) in Kernel Memory Capability Delegation

Bit Field Range	Bit Field Meaning
High half-word with its last six bits cleared ({D1[X/2-1:6]:0[5:0]})	The low half-word of the upper bound of the kernel memory relative address, aligned to 64 bytes.
Low half-word with its last six bits cleared ({D0[X/2-1:6]:0[5:0]})	The low half-word of the lower bound of the kernel memory relative address, aligned to 64 bytes.
The last six bits of the low half-word ({D0[5:0]})	The operation flag bits of the kernel memory capability.

It should be noted that the upper and lower bounds of the kernel memory address passed in are both relative to the source kernel memory capability. For example, the kernel start absolute virtual address of the source kernel memory capability is VAS=0xC0000000, and the kernel end absolute virtual address is VAE=0xC2000000, while the kernel start relative virtual address to be delegated is RAS=0x00C00000 and the kernel end relative virtual address is RAE=0x01400000; then the kernel start absolute virtual address of the new destination kernel memory capability is VAS'=VAS+RAS=0xC0000000+0x00C00000= 0xC0C00000, and the kernel end absolute virtual address is VAE'=VAS+RAE=0xC0000000+0x01400000=0xC1400000, as shown in the figure below.

In addition, the upper bound value of the kernel memory capability[14] does not include itself. In the above example, 0xC1400000 is not included in the legal addresses operable by the new kernel memory capability; that is, the kernel memory range actually permitted is 0xC0C00000-0xC13FFFFF. The upper bound must be greater than the lower bound; otherwise, an error is returned. When passed in, both the upper bound and the lower bound have their last six bits masked off, aligning them to 64 bytes. If the kernel memory registry table is configured to use slots larger than 64 bytes, the kernel automatically rounds the passed-in lower bound up and the upper bound down to align to the slot size.

The possible return values of this system call are completely identical to those of a general RME_SVC_CPT_ADD operation; please refer to 2.3.3.

3.5 Bibliography

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] Emcraft Systems. uCLinux(2017). https://github.com/EmcraftSystems/linux-emcraft

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[4] Fiasco.OC website (2017). http://os.inf.tu-dresden.de/fiasco

[14] That is, the kernel end absolute virtual address

4. Process and Thread Management

4.1 Introduction to Processes and Threads

Processes (Process) and threads (Thread) are the basic abstractions that RME provides for the programs executing on it. RME's process and thread abstractions are at a relatively low level, which is very different from the high-level abstraction implementations of traditional operating systems such as Linux. Compared with traditional operating systems, the implementation of RME possesses higher flexibility and performance.

4.1.1 Process Overview

In RME, a process refers to the smallest protection domain (Protection Domain) that possesses an independent address space[1] and a capability table. The page table determines the address space of the process, and the capability table determines the permissions of the threads[2] in the process. RME does not implement all the other information about processes in kernel mode; all of this information is left to user mode for implementation.

In fact, processes in RME have almost no similarities with the processes of traditional operating systems; they are actually ultra-lightweight virtual machines, or they can also be regarded as containers (Container). The capability tables of the individual processes each have mutually independent capability number namespaces; there is no system-global capability number. It is also permissible to regard multiple processes together as a mutually isolated but somewhat related software functional unit. When used as lightweight virtual machines, RME processes can provide performance close to that of a physical machine[4].

4.1.2 Thread Overview

In RME, a thread refers to the smallest schedulable entity that possesses an independent instruction stream and a stack. Threads in RME have multiple states that indicate their operating conditions. The threads referred to in this chapter are all kernel-mode threads; user-mode threads, being unrelated to the kernel itself, are not discussed here.

4.1.2.1 Design of the Context

In different systems, there are mainly three design strategies for the context (Context) of a thread, as follows:

1. Place the context of the thread on the user stack. This is a common strategy for small RTOSes. It is flexible enough that there is no need at all to consider the storage of thread contexts in the kernel, and thus it can be made very compact, and the efficiency of context saving and restoration is also very high. However, this design cannot be used for operating systems that possess a privileged mode: kernel mode cannot trust the stack pointer and buffers provided by user mode, so security cannot

[1] That is, an independent page table
[2] Or synchronous invocations; see Section 5.1.2

be ensured without checks, while imposing additional checks as Linux does would slow down the execution efficiency of the kernel.

2. Place the context of the thread on the kernel stack. This is a common strategy for monolithic kernels such as Linux. The efficiency of this method is acceptable, and programming is also simple, but (1) each thread requires an independent kernel stack in the kernel, wasting memory space, (2) in a microkernel it is very difficult to manage a kernel object such as a stack, whose usage changes dynamically, by means of capabilities, and (3) an additional kernel-mode context-switching assembly function needs to be written. Among these, (1) and (2) are fatal problems.

3. Place the context of the thread in the thread kernel object. This is the strategy adopted by RME: when switching contexts, we copy the context temporarily stored on the kernel stack into the kernel object of the current thread, and then copy the context in the kernel object of the next thread onto the kernel stack. When the interrupt exits, the system naturally switches to the new thread. Adopting this design adds register copy code, and an independent area is needed in the kernel object to save the register context, but each CPU only needs to prepare one fixed kernel stack, which is very suitable for embedded systems that are tight on resources, and is also conducive to improving data locality. Of course, this design means that the kernel itself cannot be preemptive (Preemptive); however, the lock-free microkernel is itself small and lean, and its system calls all possess very small WCETs, so it can achieve higher real-time responsiveness than a monolithic kernel even without being preemptive.

   When a coprocessor is present, one major negative effect of this design is the size of the thread kernel object. A coprocessor (Coprocessor) or floating-point unit (Floating-Point Unit, FPU) usually possesses a very large context; for example, AVX-512 of x86-64 possesses 32 512-bit registers, and its latest AMX matrix operation extension even possesses 8KiB matrix registers. If we configure such a large coprocessor context space for all threads without discrimination, this will cause a huge waste of kernel memory, because not every thread can make use of them. On those heterogeneous CPUs[3], there is even the situation that only a portion of the CPU cores support advanced coprocessors. To avoid waste, a context attribute Attr can be specified when creating a thread; it is a bit field that indicates which optional coprocessors the thread needs to enable. When allocating the thread kernel object, we allocate only the context storage space corresponding to the coprocessors of the enabled instruction sets, and do not allocate space for other coprocessors; when the thread is bound to a specific CPU core, we check whether that core supports the enabled coprocessors, and turn off the non-enabled coprocessors when running the thread. When a thread unexpectedly uses a coprocessor that has not been enabled, an undefined instruction exception will be generated; user mode can intercept this exception and allocate a new thread with a stronger context attribute to take over the work of the old thread.

   For such huge register contexts, saving and restoring them efficiently also requires skill. Even if a thread has declared multiple coprocessor contexts, they may not actually be used during real execution. If all contexts are saved and restored without discrimination, unnecessary switching delays will result. Therefore, when implementing coprocessor context switching, it is best to be able to determine in advance whether a coprocessor is actually used; if it is not used, there is no need to save and restore its context. To prevent leaking coprocessor information, if the state of a coprocessor changes from “in use” to “temporarily not in use”, the context of the coprocessor needs to be cleared.

   In actual use, it is recommended to schedule threads that require high real-time responsiveness and do not use coprocessors on one group of CPUs, and to schedule threads that require high computing power and use coprocessors on another group of CPUs. In this way, the former will not only avoid interference from the huge contexts of the latter, but will also not be affected by the current or power impact that the latter imposes on the CPU microarchitecture[4].

[3] Such as various ARM tri-cluster configurations, x86-64 big.LITTLE cores, etc.
[4] For example, enabling AVX-512 may cause the CPU to significantly reduce its frequency

For porters, if a coprocessor has multiple instruction subsets, whether to separate them into several independent context attributes is optional[5], but it is not recommended to divide them too finely, because the number of bits of the Attr parameter is limited; if the separated instruction subsets have dependency relationships, the kernel must check such dependency relationships when creating the thread and report an error when they are not satisfied[6]; if the registers of several coprocessors have overlapping parts, the kernel must make a declaration of a larger coprocessor automatically imply the declaration of the smaller coprocessor[7].

4.1.2.2 Design of Para-Virtualization Support

A very important use of a microkernel is to support the para-virtualization of other systems: the famous L4 supports virtualizing Linux and then running it as a group of user processes on L4, which is called L4Linux. To virtualize another operating system, it is necessary to correctly emulate the interaction between its kernel mode and user mode (or between the kernel and user programs, for those RTOSes), and the most important interaction among them is to allow kernel mode to modify the context of user-mode threads. Although systems such as L4 provide system calls to modify the context of a thread, this ultimately has to go through the microkernel, and is thus still one step slower than the original, non-virtualized system.

To solve this problem, RME allows a thread to be designated as virtual-machine-dedicated when it is created: when such a thread is bound to a CPU, a piece of kernel memory that both the microkernel and user mode can safely access needs to be specified, and thereafter the context of the thread will be saved there[8]; to save memory, the kernel object of this thread then no longer includes any context information. Obviously, if user mode were allowed to specify an arbitrary context save address, the integrity of the kernel would be compromised. Therefore, the specified address range must be configured at kernel compile time, and the kernel will, through some means[9], ensure that readable and writable pages are mapped at that address range in every process. On microcontrollers where everything is statically allocated, ensuring this is even easier.

Saving the context of a virtual-machine-dedicated thread to user mode may cause its context to be arbitrarily modified. For regular threads, user mode can also achieve the same purpose using the kernel calls listed in Section 8.10.2. If the register set of a thread contains registers related to privilege level transitions or coprocessor state[10], and the kernel does not check the modifications to these registers when exiting kernel mode and restoring registers, this will lead to privilege escalation of the user process or a corrupted coprocessor state. Therefore, every time kernel mode is exited, the critical registers of user mode must be checked, and corrupted values repaired.

4.1.2.3 Design of the Scheduler

In different systems, there are mainly four design strategies for the scheduler (Scheduler), as follows:

1. The scheduler is entirely in kernel mode. In this kind of implementation, the timeslices of the individual threads are filled in completely automatically, and the implementation of the scheduling policy is also entirely in kernel mode. Typical systems adopting this kind of implementation include Linux, all of L4, and VxWorks. Such systems are very traditional, and will not be described here.
2. The scheduler is entirely in user mode; even the priority determination and control logic are outside the kernel. When an interrupt occurs, the system switches to the thread corresponding to the

[5] For example, SSE includes multiple subsets such as SSE, SSE2, SSE3, SSSE3 and SSE4; whether to make them independent as different attributes is optional

[6] For example, SSE3 depends on SSE2; then if a thread declares that it uses SSE3 but does not use SSE2, the kernel must report that this attribute is illegal

[7] For example, AVX-512 includes AVX and SSE; then if a thread declares that it uses AVX-512, it does not need to additionally declare that it uses AVX and SSE

[8] This handling is somewhat similar to the VMCS area provided by hardware virtualization technologies such as VT-x

[9] The kernel's page table mapping code or a trusted user-mode memory manager

[10] Such as the LR of Cortex-M and the MSTATUS of RISC-V

interrupt to perform interrupt handling; as for the priority and which interrupt handler should be executed, these are entirely decided by the user-mode scheduler. This kind of implementation has extremely strong flexibility, and can very conveniently implement slack stealing scheduling (Slack Stealing Scheduling). This is the most primitive user-mode scheduling concept, but since it causes a large number of thread switches, it has excessively high additional overhead and thus no practical value.

3. The preemption priority is inside the kernel, and other parts, including the ready queue, are outside the kernel. When an interrupt arrives, preemption is performed and the interrupt handler thread is run immediately. The interrupt handler thread starts the interrupt post-processing thread, and then blocks at a receive point. If other lower-priority interrupts arrive during this time period, the lower-priority interrupts are converted into scheduler events sent to the corresponding scheduler. After the high-priority interrupt post-processing thread finishes handling the interrupt, it calls the scheduler, which processes all scheduler events that occurred during this period and decides the next thread to run. If, during this period, the timeslice allocated to the post-processing thread is exhausted, then we switch to any thread that still has a timeslice. Its disadvantage is that if a lower-priority thread becomes ready due to some interrupt while a higher-priority thread is executing, we have no way to execute the lower-priority thread immediately after the higher-priority one finishes. We must first switch to the user-mode scheduler, and then wait for the user-mode scheduler to repeatedly read all the events out of the system and judge the relative urgency of these events before the lower-priority interrupt or thread can be handled. This is intolerable in certain scenarios. A typical system adopting this kind of design is Composite[1]. It is worth mentioning that Composite extends the priority from a scalar (Scalar) into a vector (Vector); each scalar in the priority vector of each task corresponds respectively to the priority of that task in each subsystem, and preemption can occur only when all subsystems agree to the preemption[11][3], which is very suitable for crosstalk suppression in mixed-criticality systems. However, since every preemption requires a vector comparison, the execution time of this priority mechanism, although deterministic, is too slow, making it difficult to provide meaningful real-time responsiveness; in addition, storing priorities in vector form also consumes a large amount of kernel memory, making it difficult to deploy on microcontrollers. In systems equipped with accelerators[12], each task may also have different priorities on different accelerator schedulers, and preemption can occur only when all accelerators of all subsystems agree to the preemption[13]; thus it would inevitably be necessary to further extend the priority into a tensor (Tensor), and priority comparison would become an unimaginable tensor comparison.
4. The preemption priority and the run queue are in the kernel, and other parts, including timeslice management, are outside the kernel. This is the solution chosen by RME: it is closer to traditional systems yet implements user-mode scheduling. The advantage is that interrupt latency can be reduced, and it can ensure that the currently running thread is always the highest-priority one among the ready threads. The disadvantage is that every operation that may cause a context switch must process the kernel's run queue. Therefore, RME's kernel queue maintainer adopts a highly efficient design to minimize this impact. In addition, the kernel still has to send scheduler events to threads to cooperate with user-mode scheduling. Unlike Composite[1], RME implements the priority as a scalar. This makes it necessary for the system's priorities to be flattened (Flattened), which is not conducive to directly eliminating the interference among subsystems in the kernel, but it can greatly speed up the scheduling primitives, thereby making it convenient to implement in user mode vector or tensor priority scheduling algorithms suitable for mixed-criticality systems.

In RME, every thread has a preemption priority; the larger its value, the higher the priority[14]. The number of priorities is configured by the macro RME_PREEMPT_PRIO_NUM, and the priorities in the

[11] Every scalar element of the preempting vector is greater than every scalar element of the preempted vector
[12] GPU, NPU, ISP, DSP, etc.
[13] Every vector column of the preempting tensor is greater than every vector column of the preempted scalar
[14] This is the opposite of some systems; it is consistent with the common FreeRTOS

system range from 0 to RME_PREEMPT_PRIO_NUM-1. It can be configured up to 2 to the power of half the word length. For example, in a 32-bit system, the maximum number of priorities is 2^{16}=65536. In addition, each thread is assigned a priority ceiling when it is created; a thread cannot create, through a system call, a thread that possesses a higher priority ceiling. When performing a processor binding operation or a priority change operation on a thread, it is impossible to raise the priority of the operated thread above its priority ceiling. However, a thread with a low priority ceiling can raise the priority of another thread above the low-priority thread's own priority ceiling, as long as the priority of the operated thread is not raised beyond the priority ceiling of the operated thread. The implementation of the priority ceiling refers to seL4[2].

Unlike other systems that also use timeslice transfer, such as NOVA[5], the thread timeslices of RME are not automatically replenished by the system at some point, and this is also the reason why RME's scheduler is outside the kernel. If the timeslice of a thread is exhausted, then that thread will remain stopped from executing until some other thread transfers a timeslice to it.

Each thread possesses a user-determined thread identifier (Thread IDentifier, TID). This identifier is determined by the user when binding (Bind) the thread. In a 32-bit system, the range of this identifier is 0-2^{16}-1; considering that the vast majority of 32-bit devices today are embedded devices[15], this range is sufficient even if user mode requires globally unique TIDs in some situations. In a 64-bit system this range is 0-2^{32}-1, which will also be sufficient for the foreseeable future.

In addition, RME does not implement other functions such as thread local storage (Thread Local Storage, TLS) in kernel mode. These functions will be implemented by user-mode libraries.

4.2 Process Operations

4.2.1 Process Creation and Deletion

To create a process, a capability table and a page directory that can serve as the top level are required[16]. A process in RME merely serves as an isolated protection domain; it has no independent state. Destroying all the threads and synchronous invocation entries[17] in a process will not cause the process to be destroyed.

To delete a process, no threads may exist in the process, and no synchronous invocation entries[18] may exist. It suffices to specify the process to be deleted through a process capability.

4.2.2 Changing the Capability Table or Page Table of a Process

The capability table and page table of a process can be dynamically changed while the system is running, and a dynamic change always takes effect immediately. Dynamic change operations are typically used to implement the switching of virtual machines or containers in virtualization functionality, and can also be used to implement other functions such as fast cross-process communication.

[15] Basically microcontrollers or low-end microprocessors
[16] That is, a top-level page directory capability; see 3.2.2.2 for details
[17] See Section 5.1.1 for details
[18] See Section 5.1.1 for details

4.3 Thread Operations and States

4.3.1 Overview of Thread Operations

In RME, a thread needs to be bound to a certain CPU before it can be operated on, and only the CPU core to which it is bound can operate on it. If you want to change the CPU that can operate on the thread, then you need to modify its binding. In the system, a thread has the following states:

Table 4-1 States of Threads

State	Name	Description
Ready	RME_THD_READY	The thread is in the ready state; among them, the one with the highest priority on the current CPU is regarded as being in the running state. The running state does not directly correspond to any flag in the kernel.
Timeout	RME_THD_TIMEOUT	The thread's timeslice has been exhausted.
Blocked	RME_THD_BLOCKED	The thread is blocked at a certain receive point.
Exception	RME_THD_EXCPEND	An exception occurred during the execution of the thread, and it was forced to suspend.

These states can transition into one another. When a thread is created, it is in the RME_THD_TIMEOUT state, which means that it is not bound to any CPU and has not been allocated a timeslice. Next, we bind it to a certain core; at this point it is still in the RME_THD_TIMEOUT state. Then we set its entry and stack. Finally, we allocate a timeslice to it. It will be placed into the kernel ready queue and enter the RME_THD_READY state; if it is the highest-priority thread on that CPU, then it will preempt the current thread and actually run.

If a thread is blocked at some receive point during execution, the thread will transition to the RME_THD_BLOCKED state. In this case, when the blocking is lifted, the thread will return to the RME_THD_READY state.

If an exception occurs during the execution of a thread, then it will transition to the RME_THD_EXCPEND state and send a scheduler event to its parent thread[19]. If a signal endpoint[20] was also specified when the thread was bound, then that signal endpoint will also receive a signal. To clear the error state, it is necessary to reset its execution stack and entry, so as to place the thread in the RME_THD_TIMEOUT state.

If the thread exhausts all of its own timeslices while running, or transfers all of its own timeslices away in timeslice transfer, or chooses to give up all of its current timeslices when switching to another thread, then it will enter the RME_THD_TIMEOUT state and send a scheduler event to its parent thread. If a signal endpoint was also specified when the thread was bound, then that signal endpoint will also receive a signal.

When a thread is unbound from a certain CPU, the thread must have no child threads. During unbinding, the scheduler event of the parent thread corresponding to this thread, if it exists, will also be removed. If it is unbound while in RME_THD_BLOCKED, then the current blocking will directly return an RME_ERR_SIV_FREE error code[21]. In addition, when the thread is not in the RME_THD_EXCPEND state, if it is unbound, then the state of the thread will become the RME_THD_TIMEOUT state. If the thread is unbound while in the RME_THD_EXCPEND state, then the thread will remain in the RME_THD_EXCPEND state.

[19] Scheduler thread
[20] See Section 5.1.2 for details
[21] See Section 5.1.2 for details

4.3.2 Thread Creation and Deletion

When creating a thread, it is additionally necessary to specify the process in which the thread resides, the coprocessors it uses, and whether the thread is a virtual-machine-dedicated thread. Attributes such as the entry and stack are set through other kernel calls.

When a thread is deleted, it must be unbound. When deleting a thread, we clear its synchronous invocation stack[22].

4.3.3 Binding a Thread to a CPU and Unbinding It

After a thread is created, it needs to be bound to a certain CPU before it can be operated on; and if you want to change this binding, then you first need to unbind it from its current CPU.

Binding a thread to a certain CPU requires specifying the thread's priority, the thread's parent thread, and a TID. On whichever CPU the bind function is called, the thread will be bound to that CPU. The binding operation is performed by using a compare-and-swap atomic operation, ensuring that only one operation succeeds among operations performed simultaneously on multiple CPUs. In addition, there is a signal endpoint as an optional parameter; if a signal endpoint is specified at binding time, then that signal endpoint will receive the scheduler signal of this thread. It is worth noting that the TID does not need to be globally unique; user mode can decide the TID allocation policy.

Unbinding merely requires specifying the thread to be unbound. Once a thread has been unbound from a certain CPU, we can bind it to other CPUs, that is, the migration of threads between different processors is achieved. This is different from systems such as Composite[1] in which threads are permanently bound to a certain processor at creation time.

4.3.4 Setting the Execution Attributes of a Thread

After completing the binding of a thread, we need to set the thread's entry, stack and custom parameter. These three values will be passed to the register set of the thread; when the thread runs for the first time, the user-mode library uses the first two parameters to find the user-mode thread entry and thread stack. It should be noted that on architectures that distinguish virtual addresses from physical addresses, both of these values are virtual addresses.

4.3.5 Allocating Timeslices to a Thread, Changing the Priority, and Running

After the entry and stack have been set, we can allocate a timeslice to the thread, thereby starting the execution of the thread. The timeslice allocation of the RME system is organized by a user-mode scheduler tree, and every CPU has such a scheduler tree, used to manage the execution time allocation of this CPU. First, the Init thread on each CPU of the system allocates timeslices to the user-mode scheduler[23], and then these user-mode schedulers, according to their respective scheduling algorithms, transfer their timeslices to their respective child schedulers in appropriate proportions, and so on, allocating layer by layer, thereby completing the hierarchical scheduling of threads. This organization makes paravirtualizing other operating systems very easy. The Init thread on each CPU possesses infinite timeslices; that is, if there is no other thread that can run, we always run the Init thread on this CPU[24].

After the thread's timeslice allocation is completed, the thread is placed into the ready queue (Ready Queue) of each core, and its priority is compared with that of the currently running thread. If the currently running thread has a lower priority, then the thread will be put into operation immediately.

We can modify the priority of a thread that has already been bound to a certain CPU. After the priority

[22] See Section 5.1.1 for details
[23] The timeslice of the Init thread is RME_THD_INIT_TIME, which is infinite
[24] This is also the reason why the Init thread is not allowed to block on any endpoint; see Section 5.3 for details

is modified, if the thread's priority is the highest, and it is in the RME_THD_READY state, then it will be scheduled to run immediately.

The practice of transferring timeslices between threads draws on the TCap mechanism of Composite, and simplifies and improves it[3]. Classified by the amount of timeslice held and the thread creation time, there are three kinds of threads in the system, as shown in the following table:

Table 4-2 Categories of Threads

Category	Creation Time	Characteristics
Normal thread	After system boot	The timeslice is finite, and the total amount held does not exceed RME_THD_INF_TIME.
Infinite thread	After system boot	The timeslice is infinite; the timeslice holding amount is denoted as RME_THD_INF_TIME.
Initial thread	At system boot	The timeslice is infinite; the timeslice holding amount is denoted as RME_THD_INIT_TIME.

The meanings of several macros are shown in the following table:

Table 4-3 Meaning of Timeslice Macro Definitions

Macro Name	Meaning
RME_THD_INIT_TIME	The maximum positive integer value; in a 32-bit system it is 0x7FFFFFFF.
RME_THD_INF_TIME	RME_THD_INIT_TIME-1; in a 32-bit system it is 0x7FFFFFFE.

Among them, normal threads and infinite-timeslice threads are allowed to block, and are also allowed to lose all of their own timeslices; the initial thread is not allowed to do either. There are the following three kinds of timeslice transfer between threads:

The first is normal transfer; this kind of transfer transfers a finite number of timeslices to other threads;

The second is infinite transfer; this kind of transfer transfers an infinite number of timeslices to other threads;

The third is reclaim transfer; this kind of transfer transfers all of the source thread's timeslices to the target thread, and zeroes out the source thread's timeslices.

It is worth noting that timeslice transfer is atomic; that is, either the specified amount is transferred in full, or none of it is transferred; a partial transfer cannot occur. The rules of the three kinds of transfer are shown in the following tables.

Table 4-4 Rules of Normal Timeslice Transfers

Normal Transfer	Source Thread	Initial Thread	Infinite Thread	Normal Thread
Target Thread	Initial Thread	--	--	T-
	Infinite Thread	--	--	T-
	Normal Thread	-A	TA	TA

Table 4-5 Rules of Infinite Timeslice Transfers

Infinite Transfer Target Thread	Source Thread	Initial Thread	Infinite Thread	Normal Thread
	Initial Thread	--	--	S-
	Infinite Thread	--	--	S-
	Normal Thread	-I	-I	TA

Table 4-6 Rules of Reclaim Timeslice Transfers

Reclaim Transfer Target Thread	Source Thread	Initial Thread	Infinite Thread	Normal Thread
	Initial Thread	--	S-	S-
	Infinite Thread	--	S-	S-
	Normal Thread	-I	SI	TA

In the table:

"–" means that there is no effect on the source thread or the target thread;

"T" means that if all of the source thread's timeslices are transferred out, then the source thread will time out.

"S" means that the source thread will definitely time out.

"A" means that if the target thread's timeslice does not overflow, then these timeslices will be accepted.

"I" means that the target thread will definitely become an infinite thread.

4.3.6 Overview of Thread State Transitions

A complete thread state transition diagram is shown below:

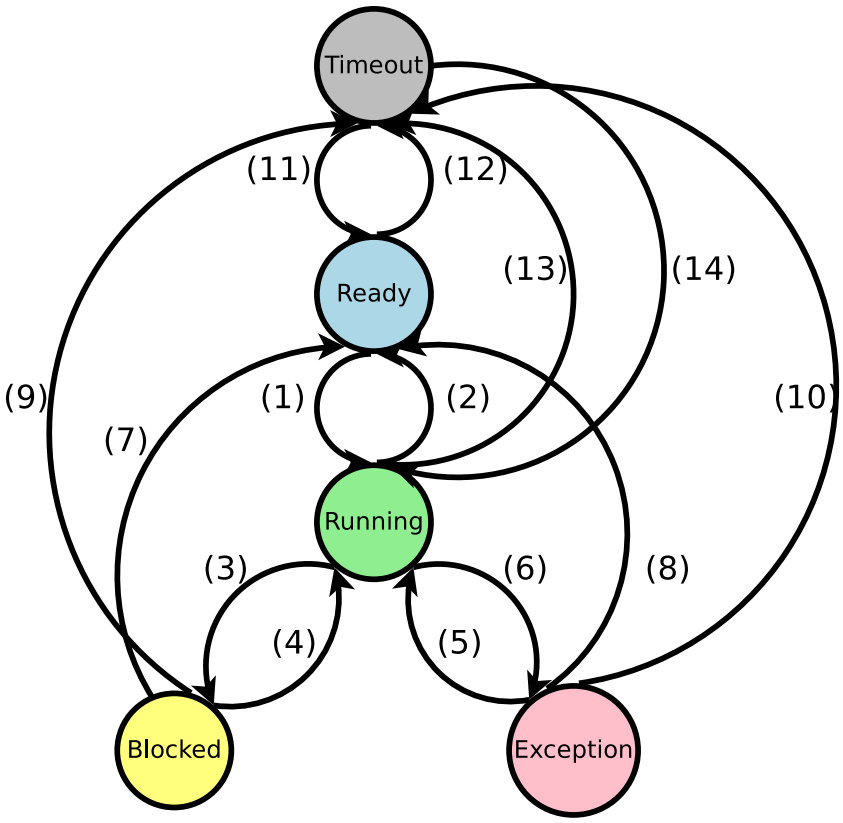


Figure 4-1 Thread State Transition Diagram

The meanings of the various numeric labels in the figure are as follows:

Table 4-7 Meaning of the Labels in the Thread State Transition Diagram

Label	Meaning
(1)	It is the highest-priority thread on the current core, so it transitions from the ready state to the running state.
(2)	A higher-priority thread on the current core interrupted its execution, so it transitions from the running state to the ready state.
(3)	The thread blocks at a signal endpoint, so it transitions from the running state to the blocked state.
(4)	The thread received a signal from the signal endpoint, was unblocked, and is the highest-priority thread on the current core, and its timeslice has not been exhausted, so it transitions from the blocked state to the running state.
(5)	After an error occurred, the thread was restored to a normally executable state by re-setting its execution information, and it is the highest-priority thread on the current core, and its timeslice has not been exhausted, so it transitions from the exception state to the running state.
(6)	An exception occurred during the execution of the thread, so it transitions from the running state to the exception state.
(7)	The thread received a signal from the signal endpoint, was unblocked, and its timeslice has not been exhausted, but it is not the highest-priority thread on the current core, so it transitions from the blocked state to the ready state.
(8)	After an error occurred, the thread was restored to a normally executable state by re-setting its execution information, and its timeslice has not been exhausted, but it is not the highest-priority thread on the current core, so it transitions from the exception state to the ready state.
(9)	While waiting at the signal endpoint, the thread had its own timeslices transferred out by other threads, zeroing its own timeslices; therefore, after receiving a signal and being unblocked, it transitions from the blocked state to the timeout state.
(10)	After an error occurred, the thread had its own timeslices transferred out by other threads, zeroing its own timeslices; therefore, after being restored to a normally executable state by re-setting its execution information, it transitions from the exception state to the timeout state.
(11)	The thread was transferred timeslices by other threads and re-entered a state in which it can run, but it is not the highest-priority thread on the current core, so it transitions from the timeout state to the ready state.
(12)	While in the ready state, the thread had its own timeslices transferred out by other threads, zeroing its own timeslices, so it transitions from the ready state to the timeout state.
(13)	The thread exhausted its own timeslices during execution, so it transitions from the running state to the timeout state.
(14)	The thread was transferred timeslices by other threads and re-entered a state in which it can run, and it is the highest-priority thread on the current core, so it transitions from the timeout state to the running state.

Among them, all transitions into the timeout state and the exception state, such as (6), (9), (10), (12) and (13), will cause a scheduler event to be sent.

4.4 Process Function List

The kernel functions related to processes are as follows:

Table 4-8 Process-Related Kernel Functions

Call Number	Type	Purpose
RME_SVC_PRC_CRT	System call	Create a process
RME_SVC_PRC_DEL	System call	Delete a process
RME_SVC_PRC_CPT	System call	Replace the capability table of a process
RME_SVC_PRC_PGT	System call	Replace the page table[25] of a process

The operation flags of a process capability are as follows:

Table 4-9 Process Operation Flags

Flag	Bit	Purpose
RME_PRC_FLAG_INV	[0]	Allows the creation of synchronous invocations within this process.
RME_PRC_FLAG_THD	[1]	Allows the creation of threads within this process.
RME_PRC_FLAG_CPT	[2]	Allows replacing the capability table of this process.
RME_PRC_FLAG_PGT	[3]	Allows replacing the page table[26] of this process.

Regarding bit [0] in the above table, please refer to 5.4.1.

4.4.1 Creating a Process

This operation creates a process and places its capability into an existing capability table. The newly created process references the capability table capability and the page directory capability; once a process has been created with a certain pair of capability table/page directory capabilities, this pair of capabilities cannot be removed/deleted before the process is deleted. A process capability has no separately allocated kernel object, so it can be created without a kernel memory capability.

Creating a process requires the following parameters:

Table 4-10 Parameters Needed for Process Creation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PRC_CRT.
Cap_Cpt_Crt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created process capability. This capability number can be a one-level or two-level lookup encoding.

[25] Top-level page directory
[26] Top-level page directory

Parameter Name	Type	Domain	Description
Cap_Prc	cid_t	P0	A capability number corresponding to a position in the capability table that is to receive this newly created process capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Cpt	cid_t	P1	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_PRC_CRT attribute; the capability corresponding to this capability number points to the capability table to be used by the newly created process. This capability number can be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	P2	A capability number corresponding to a page table capability that must have the RME_PGT_FLAG_PRC_CRT attribute; the capability corresponding to this capability number points to the page table[27] to be used by the newly created process. This capability number can be a one-level or two-level lookup encoding.
Raw_Pgt	ptr_t	P2	Once the user-mode page table control macro RME_PGT_RAW_USER is enabled at porting time, this parameter replaces Cap_Pgt. It is the physical address where the page table data structure directly recognizable by the processor resides, and the kernel will use it directly. Therefore, when user-mode page tables are used, user-mode processes with the process creation permission must be trusted.

[27] Top-level page directory

The return values of this operation may be as follows:

Table 4-11 Possible Return Values for Process Creation

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Crt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt_Crt has been frozen.
RME_ERR_CPT_FROZEN	The one-level/two-level lookup capability of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	The one-level/two-level lookup capability of Cap_Pgt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Prc has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt_Crt or Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page table capability.
RME_ERR_CPT_FLAG	Cap_Cpt_Crt has no RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_PRC_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Pgt has no RME_PGT_FLAG_PRC_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Prc is not an empty capability.

4.4.2 Deleting a Process

This operation deletes a process. The process being deleted must not contain threads or synchronous invocation calls (for the latter, see the synchronous communication mechanism). Deleting a process requires the following parameters:

Table 4-12 Parameters Needed for Process Deletion

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PRC_DEL.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table containing the process capability being deleted. This capability number can be a one-level or two-level lookup encoding.

Parameter Name	Type	Domain	Description
Cap_Prc	cid_t	P1	A capability number corresponding to the process capability to be deleted. The capability corresponding to this capability number must be a process capability. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 4-13 Possible Return Values for Process Deletion

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Prc has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_NULL	Cap_Prc is an empty capability.
RME_ERR_CPT_NULL	Two cores attempt to delete this process at the same time; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Prc is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Prc is not 0, or it is not a root capability[28].

[28] It internally has threads or synchronous invocation calls

4.4.3 Changing the Capability Table of a Process

This operation replaces the capability table of a process with another one. The replacement takes effect immediately after it is completed. Replacing the capability table of a process requires the following parameters:

Table 4-14 Parameters Needed for Changing the Capability Table of a Process

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PRC_CPT.
Cap_Prc	cid_t	P1	A capability number corresponding to a process capability that must have the RME_PRC_FLAG_CPT attribute; the capability corresponding to this capability number points to the process whose capability table is to be changed. This capability number can be a one-level or two-level lookup encoding.
Cap_Cpt	cid_t	P2	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_PRC_CPT attribute; the capability corresponding to this capability number points to the capability table to be used for the process. This capability number can be a one-level or two-level lookup encoding.

The return values of this operation may be as follows:

Table 4-15 Return Values Needed for Changing the Capability Table of a Process

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Prc has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_NULL	Cap_Prc or Cap_Cpt is an empty capability.
RME_ERR_CPT_FLAG	Cap_Prc has no RME_PRC_FLAG_CPT attribute.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_PRC_CPT attribute.
RME_ERR_PTH_CONFLICT	Two cores attempt to replace the capability table at the same time; in this case, the core that did not succeed returns this value.

4.4.4 Changing the Page Table of a Process

This operation replaces the page table of a process with another one. The replacement takes effect immediately after it is completed. Replacing the page table of a process requires the following parameters:

Table 4-16 Parameters Needed for Changing the Page Table of a Process

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PRC_PGT.
Cap_Prc	cid_t	P1	A capability number corresponding to a process capability that must have the RME_PRC_FLAG_PGT attribute; the capability corresponding to this capability number points to the process whose page table[29] is to be changed. This capability number can be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	P2	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_PRC_PGT attribute; the capability corresponding to this capability number points to the page table[30] to be used for the process. This capability number can be a one-level or two-level lookup encoding.
Raw_Pgt	ptr_t	P2	Once the user-mode page table control macro RME_PGT_RAW_USER is enabled at porting time, this parameter replaces Cap_Pgt. It is the physical address where the page table data structure directly recognizable by the processor resides, and the kernel will use it directly. Therefore, when user-mode page tables are used, user-mode processes with this permission must be trusted.

The return values of this operation may be as follows:

Table 4-17 Possible Return Values for Changing the Page Table of a Process

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Prc has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has been frozen.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page table capability.

[29] Top-level page directory
[30] Top-level page directory

Return Value	Meaning
RME_ERR_CPT_NULL	Cap_Prc or Cap_Pgt is an empty capability.
RME_ERR_CPT_FLAG	Cap_Prc has no RME_PRC_FLAG_PGT attribute.
RME_ERR_CPT_FLAG	Cap_Pgt has no RME_PGT_FLAG_PRC_PGT attribute.
RME_ERR_PTH_CONFLICT	Two cores attempt to replace the page table at the same time; in this case, the core that did not succeed returns this value.

4.5 Thread Function List

The kernel functions related to threads are as follows:

Table 4-18 Thread Related Kernel Functions

Call Number	Type	Purpose
RME_SVC_THD_CRT	System Call	Create a thread
RME_SVC_THD_DEL	System Call	Delete a thread
RME_SVC_THD_SCHED_BIND	System Call	Bind a thread to a certain CPU
RME_SVC_THD_SCHED_FREE	System Call	Free a thread from a certain CPU[31]
RME_SVC_THD_EXEC_SET	System Call	Set the execution attributes of a thread[32]
RME_SVC_THD_SCHED_PRIO	System Call	Change the priority of a thread
RME_SVC_THD_SCHED_RCV	System Call	Receive the scheduler events of a thread
RME_SVC_THD_TIME_XFER	System Call	Transfer a timeslice between threads
RME_SVC_THD_SWT	System Call	Switch to a thread of the same priority

The operation flags of the thread capability are as follows:

Table 4-19 Thread Operation Flags

Flag	Bit	Purpose
RME_THD_FLAG_EXEC_SET	[0]	Allow setting the execution attributes of this thread.
RME_THD_FLAG_SCHED_CHILD	[1]	Allow this thread to act as a child thread in a bind operation.
RME_THD_FLAG_SCHED_PARENT	[2]	Allow this thread to act as a parent thread in a bind operation.
RME_THD_FLAG_SCHED_PRIO	[3]	Allow changing the priority of this thread.
RME_THD_FLAG_SCHED_FREE	[4]	Allow removing the binding of this thread to a certain CPU.
RME_THD_FLAG_SCHED_RCV	[5]	Allow receiving the scheduler events of this thread.
RME_THD_FLAG_XFER_SRC	[6]	Allow this thread to act as the source in time transfer.
RME_THD_FLAG_XFER_DST	[7]	Allow this thread to act as the destination in time transfer.

[31] Top-level page directory
[32] Entry and stack

Flag	Bit	Purpose
RME_THD_FLAG_SWT	[8]	Allow a switch operation to switch to this thread.

4.5.1 Creating a Thread

This operation creates a thread and places its capability into an existing capability table. A newly created thread is not bound to any CPU, has no TID, and is in the RME_THD_TIMEOUT state. To prevent a thread from creating a thread that exceeds its own priority ceiling and thereby breaking the isolation between subsystems, no thread can create a thread with a priority ceiling higher than its own.

The thread references the process specified at creation time; once a thread has been created within a process, this process cannot be deleted before the thread is deleted. Considering that some architectures have a large number of vector registers, this operation allows a context attribute to be specified, indicating the instruction sets that the thread is permitted to enable, so as to save kernel memory usage. This operation also allows the thread to map its register set into user mode; if mapping into user mode is specified, a virtual machine monitor running in user mode can modify its context at any time, thereby greatly facilitating the programming of virtualization-related functions. If the kernel decides at configuration time not to use the virtualization mapping function[33], this function cannot be used. Creating a thread requires the following parameters:

Table 4-20 Parameters Needed for Thread Creation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N[5:0]	Must be RME_SVC_THD_CRT.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created thread capability. This capability number can be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	P1.D1	A kernel memory capability number whose identified kernel memory range must be able to hold the thread kernel object and must have the RME_KOM_FLAG_THD attribute. This capability number can be a one-level or two-level lookup encoding.
Cap_Thd	cid_t	P1.D0	A capability number corresponding to a position in the capability table that is to receive this newly created thread capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.

[33] See Section 7.3.2 for details

Parameter Name	Type	Domain	Description
Cap_Prc	cid_t	P2.D1	A capability number corresponding to a process capability that must have the RME_PRC_FLAG_THD attribute; the capability corresponding to this capability number points to the process that contains the newly created thread. This capability number can be a one-level or two-level lookup encoding.
Max_Prio	ptr_t	P2.D0	The priority ceiling of this thread.
Raddr	ptr_t	P3	The starting relative virtual address in kernel space to be used by the newly created thread kernel object. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.
Attr	ptr_t	N[X/2-1:7]	Specify the extended instruction set context attributes of the thread. For different architectures, this parameter has different meanings, but in all architectures 0 means that no instruction set extension is performed and no coprocessor register state is allocated, saved or restored. For 32-bit processors, this parameter has 9 bits; for 64-bit processors, this parameter has 25 bits.
Is_Hyp	ptr_t	N[6]	If set to 0, the system allocates space in the kernel to save the thread context, which is the normal case. If it is not 0, the thread will map its registers into user mode, in which case the specific mapping address needs to be set when binding the thread to a CPU.

The return values of this operation may be as follows:

Table 4-21 Possible Return Values for Thread Creation

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has been frozen.
RME_ERR_CPT_FROZEN	The one-level/two-level lookup capability of Cap_Prc has been frozen.
RME_ERR_CPT_FROZEN	Cap_Thd has been frozen, or another core is creating a capability there.

Return Value	Meaning
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Kom has no RME_KOM_FLAG_THD attribute, or the range is incorrect.
RME_ERR_CPT_FLAG	Cap_Prc has no RME_PRC_FLAG_THD attribute.
RME_ERR_CPT_EXIST	Cap_Thd is not an empty capability.
RME_ERR_CPT_KOT	Allocation of kernel memory failed.
RME_ERR_PTH_PRIO	Attempting to create a thread with a higher priority ceiling than its own, or the passed priority ceiling is too large and exceeds the limit allowed by the system configuration.

4.5.2 Deleting a Thread

This operation deletes a thread. The thread being deleted must already have been unbound. Deleting a thread requires the following parameters:

Table 4-22 Parameters Needed for Thread Deletion

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_DEL.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table containing the thread capability being deleted. This capability number can be a one-level or two-level lookup encoding.
Cap_Thd	cid_t	P1	A capability number corresponding to the thread capability to be deleted. The capability corresponding to this capability number must be a thread capability. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 4-23 Possible Return Values for Thread Deletion

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Thd has not been frozen.

Return Value	Meaning
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability.
RME_ERR_CPT_NULL	Cap_Thd is an empty capability.
RME_ERR_CPT_NULL	Two cores attempt to delete this thread at the same time; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Thd is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Thd is not 0, or it is not a root capability.
RME_ERR_PTH_INVSTATE	The thread is still in the bound state.

4.5.3 Binding a Thread to a CPU

This operation binds a thread to a certain CPU. The thread being set must not be bound to any CPU. The binding is completed on the CPU on which this function is called. For those threads that wish to map their register context into user mode for convenient modification, this operation also sets the mapping address of the context: the management of this address range is entirely the responsibility of a trusted daemon in user mode, and at boot time the kernel only guarantees that this address can be accessed both from kernel mode and from user mode, and that it will not cause a page fault. Binding a thread to a CPU requires the following parameters:

Table 4-24 Parameters Needed for Binding a Thread to a CPU

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_BIND.
Cap_Thd	cid_t	C	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SCHED_CHILD attribute; the capability corresponding to this capability number points to the thread being bound. This capability number can be a one-level or two-level lookup encoding.
Cap_Thd_Sched	cid_t	P1.D1	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SCHED_PARENT attribute; the capability corresponding to this capability number points to the parent thread of the thread being bound. This parent thread must already be bound to the CPU on which this function is called. This capability number can be a one-level or two-level lookup encoding.

Parameter Name	Type	Domain	Description
Cap_Sig	cid_t	P1.D0	A capability number corresponding to a signal endpoint capability that must have the RME_SIG_FLAG_SCHED attribute. When a scheduler event occurs[34], this endpoint will receive a signal. This capability number can be a one-level or two-level lookup encoding, and is also optional. If scheduler signals are not desired, RME_CID_NULL[35] can be passed in.
TID	tid_t	P2.D1	A user-provided TID. RME itself does not check whether two threads in the system have the same ID; whether to perform such a check is a matter for user mode.
Prio	ptr_t	P2.D0	The preemption priority of the thread being bound. In RME, thread priorities are counted from 0, and the larger the value, the higher the priority. This value cannot exceed the inherent priority ceiling of the thread.
Haddr	ptr_t	P3	The mapping address of the user-mode register context. This address must lie within the address range dedicated in the system to thread context mapping, that is, greater than or equal to the base address RME_HYP_VA_BASE and less than the end address RME_HYP_VA_BASE+ RME_HYP_SIZE, and it must be aligned to a word. For ordinary threads, setting it to NULL is sufficient.

The return values of this operation may be as follows:

Table 4-25 Possible Return Values for Binding a Thread to a CPU

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd or Cap_Thd_Sched is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd or Cap_Thd_Sched has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd or Cap_Thd_Sched is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd has no RME_THD_FLAG_SCHED_CHILD attribute.

[34] A thread timeout or error, where a timeout may be caused by the thread consuming all of its timeslice while running or by its timeslice being completely transferred to other threads

[35] A half-word-length INT_MIN (a negative value), which is 0x8000 on 32-bit systems and 0x80000000 on 64-bit systems

Return Value	Meaning
RME_ERR_CPT_FLAG	Cap_Thd_Sched has no RME_THD_FLAG_SCHED_PARENT attribute.
RME_ERR_CPT_FLAG	When Cap_Sig is passed in, it has no RME_SIG_FLAG_SCHED attribute.
RME_ERR_CPT_FLAG	The processor to be bound to does not support the Attr registered when this thread was created.
RME_ERR_PTH_HADDR	The Haddr passed in is not within the range allowed by the system and is invalid.
RME_ERR_PTH_NOTIF	An attempt was made to register itself as its own parent thread, which is invalid.
RME_ERR_PTH_PRIO	The preemption priority exceeds the inherent priority ceiling of this thread, which is invalid.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the bound state or Cap_Thd_Sched is in the unbound state.
RME_ERR_PTH_INVSTATE	Cap_Thd_Sched is bound to a CPU different from the CPU calling this function.
RME_ERR_PTH_CONFLICT	If two CPUs attempt to bind at the same time, this value is returned on the CPU that failed.

4.5.4 Unbinding a Thread from a CPU

This operation unbinds a thread from a CPU. The thread being unbound must not itself have child threads. If the thread being unbound has sent a scheduler event to its parent thread, then this event is revoked. If the thread being unbound blocks, then it simultaneously causes this blocking receive to be forcibly terminated and returns RME_ERR_SIV_FREE. Unbinding a thread from a CPU requires the following parameters:

Table 4-26 Parameters Needed for Unbinding a Thread from a CPU

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_FREE.
Cap_Thd	cid_t	P1	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SCHED_FREE attribute; the capability corresponding to this capability number points to the thread to be unbound. This capability number can be a one-level or two-level lookup encoding.

The return values of this operation may be as follows:

Table 4-27 Possible Return Values for Unbinding a Thread from a CPU

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.

Return Value	Meaning
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd has no RME_THD_FLAG_SCHED_FREE attribute.
RME_ERR_PTH_REFCNT	Cap_Thd is still the parent thread of some threads[36].
RME_ERR_PTH_INVSTATE	Cap_Thd is in the unbound state.
RME_ERR_PTH_INVSTATE	Cap_Thd is bound to a CPU different from the CPU calling this function.

[36] scheduler threads

4.5.5 Setting Thread Execution Attributes

This operation sets the execution attributes of a thread, that is, its entry, stack, and parameters. The thread being configured must already be bound to a CPU, and setting the execution attributes must be performed on this CPU. For a thread that is already in the RME_THD_EXCPEND state, setting the thread execution attributes sets it to the RME_THD_READY or RME_THD_TIMEOUT state depending on its current number of timeslices, where the latter sends a scheduler event. When both the Entry and Stack passed in are NULL, the execution attributes of this thread remain unchanged and only its state is changed, which is very useful in exception handling. Setting thread execution attributes requires the following parameters:

Table 4-28 Parameters Needed for Setting Thread Execution Attributes

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_EXEC_SET.
Cap_Thd	cid_t	C	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_EXEC_SET attribute; the capability corresponding to this capability number points to the thread whose execution attributes are being set. This capability number can be a one-level or two-level lookup encoding.
Entry	ptr_t	P1	The entry of this thread. This value is a virtual address inside the process in which the thread resides, and the thread will start executing from here. If NULL is passed in, no actual change is made.
Stack	ptr_t	P2	The execution stack of this thread. This value is a virtual address inside the process in which the thread resides, and the thread will use this address as the start of the stack. Whether the specific stack is an ascending stack or a descending stack is determined by the user-mode library. If NULL is passed in, no actual change is made.
Param	ptr_t	P3	The parameter to be passed to this thread.

The return values of this operation may be as follows:

Table 4-29 Possible Return Values for Setting Thread Execution Attributes

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd has no RME_THD_FLAG_EXEC_SET attribute.
RME_ERR_PTH_INVSTATE	The thread is in the unbound state.

4.5.6 Changing Thread Priority

This operation changes the priorities of up to four threads already bound to a CPU at one time. When more than one thread is specified, these priority change operations are transactional (Transactional): they either all succeed or all fail. When fewer than four threads are specified, the unused parameters may be left unfilled. All priority change operations must be performed on the same CPU. Changing thread priority requires the following parameters:

Table 4-30 Parameters Needed for Changing Thread Priority

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_PRIO.
Number	ptr_t	C	Specifies the number of threads whose priorities need to be modified; the allowed values are 1~4.
Cap_Thd0	cid_t	P1.D0	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SCHED_PRIO attribute; the capability corresponding to this capability number points to the thread whose preemption priority is to be modified. This capability number can be a one-level or two-level lookup encoding.
Prio0	ptr_t	P3.Q0	The new preemption priority of the thread. In RME, thread priorities are counted from 0, and the larger the value, the higher the priority. This value cannot exceed the inherent priority ceiling of this thread.
Cap_Thd1	cid_t	P1.D1	The capability number of the second thread (optional).
Prio1	ptr_t	P3.Q1	The new preemption priority of the second thread (optional).
Cap_Thd2	cid_t	P2.D0	The capability number of the third thread (optional).
Prio2	ptr_t	P3.Q2	The new preemption priority of the third thread (optional).
Cap_Thd3	cid_t	P2.D1	The capability number of the fourth thread (optional).
Prio3	ptr_t	P3.Q3	The new preemption priority of the fourth thread (optional).

The return values of this operation may be as follows:

Table 4-31 Possible Return Values for Changing Thread Priority

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of at least one of the four Cap_Thd is out of range.

Return Value	Meaning
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of at least one of the four Cap_Thd has been frozen.
RME_ERR_CPT_TYPE	At least one of the four Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	At least one of the four Cap_Thd has no RME_THD_FLAG_SCHED_PRIO attribute.
RME_ERR_PTH_PRIO	The priority of at least one thread exceeds the inherent priority ceiling of this thread, which is invalid.
RME_ERR_PTH_PRIO	The value of Number is not in the range 1 4.
RME_ERR_PTH_INVSTATE	At least one thread is in the unbound state.
RME_ERR_PTH_INVSTATE	At least one thread is bound to a CPU different from the CPU calling this function.

4.5.7 Receiving a Thread's Scheduler Event

This operation receives a scheduler event of a thread. This operation does not block; if the thread has no scheduler event for the time being, then we return a negative value. If the returned value is positive, then it may consist of three parts: the exception identifier RME_THD_EXCPEND_FLAG[37], the timeslice exhaustion identifier RME_THD_TIMEOUT_FLAG[38], and the thread identifier TID[39].

If RME_THD_EXCPEND_FLAG is set, it indicates that the thread is in the exception state; at this point, the kernel function call described in 8.10.4 can be used to query the cause of the exception, and RME_SVC_THD_EXEC_SET can be used to clear the exception and continue execution; if RME_THD_TIMEOUT_FLAG is set, it indicates that the thread is in the timeout state, and at this point RME_SVC_THD_TIME_XFER can be used to transfer more timeslices to it. These two identifiers may be present at the same time: when they are both present, it indicates that the thread is in the exception state and holds 0 timeslices; once its execution attributes are set to clear its exception state, the thread will immediately enter the timeout state[40].

Receiving the scheduler event of a thread requires the following parameters:

Table 4-32 Parameters Needed for Receiving a Thread's Scheduler Event

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_RCV.
Cap_Thd	cid_t	P1	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SCHED_RCV attribute; we attempt to receive the scheduler events sent from the child threads of this thread. This capability number can be a one-level or two-level lookup encoding.

[37] the second-to-last binary bit
[38] the third-to-last binary bit
[39] the other, subsequent binary bits
[40] This transition from the exception state to the timeout state also sends a scheduler event

The return values of this operation may be as follows:

Table 4-33 Possible Return Values for Receiving a Thread's Scheduler Event

Return Value	Meaning
Non-negative value	The operation succeeded. If the error identifier is 0, then what was received is a thread timeout event, indicating that the thread corresponding to the thread identifier exhausted its timeslice and stopped executing. If the error identifier is 1, then it indicates that the thread corresponding to the thread identifier encountered an error during execution and was forced to stop executing.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd has no RME_THD_FLAG_SCHED_RCV attribute.
RME_ERR_PTH_NOTIF	Cap_Thd has no scheduler event available to receive for the time being.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the unbound state.
RME_ERR_PTH_INVSTATE	Cap_Thd is bound to a CPU different from the CPU calling this function.

4.5.8 Transferring Run Time Slice Between Threads

This operation is used to transfer timeslices between threads. The source thread[41] and the destination thread[42] of the timeslice must be bound to the same CPU, and this system call must be performed on this CPU, to ensure that the transferred time is execution time on this CPU. The timeslice held by the source thread itself and the timeslice to be transferred must not be 0. If the source thread loses all of its timeslices when transferring, it immediately sends a scheduler event to its scheduler thread.

This function allows a low-priority thread to transfer run time slices to a high-priority thread, but does not allow a thread with a low priority ceiling to transfer run time slices to a thread with a high priority ceiling[43]: this is to ensure that timeslices always degrade (Degrade) during transfer, so that low-priority subsystems do not interfere with high-priority subsystems.

Transferring run time slices requires the following parameters:

Table 4-34 Parameters Needed for Transferring Run Time Slice Between Threads

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_TIME_XFER.
Cap_Thd_Dst	cid_t	P1	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_XFER_DST attribute; the capability corresponding to this capability number points to the destination thread. This capability number can be a one-level or two-level lookup encoding.

[41] source thread
[42] destination thread
[43] Except for the Init thread; as a special case, transferring timeslices to the Init thread to void time is permitted

Parameter Name	Type	Domain	Description
Cap_Thd_Src	cid_t	P2	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_XFER_SRC attribute; the capability corresponding to this capability number points to the source thread. This capability number can be a one-level or two-level lookup encoding.
Time	ptr_t	P3	The number of timeslices to transfer. The unit of this value is the timeslice; the size of a single timeslice is determined at system compile time. This value cannot be 0. Pass in RME_THD_INF_TIME for an infinite transfer. Pass in RME_THD_INIT_TIME for a revoking transfer. Pass in any other non-zero value for a normal transfer.

The return values of this operation may be as follows:

Table 4-35 Possible Return Values for Transferring Run Time Slice Between Threads

Return Value	Meaning
Non-negative value	The operation succeeded, and what is returned is the current number of timeslices of the destination thread. If an infinite transfer or a revoking transfer is performed and the destination thread has become an infinite thread, then RME_THD_INF_TIME is returned.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd_Dst or Cap_Thd_Src is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd_Dst or Cap_Thd_Src has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd_Dst or Cap_Thd_Src is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd_Dst has no RME_THD_FLAG_XFER_DST attribute.
RME_ERR_CPT_FLAG	Cap_Thd_Src has no RME_THD_FLAG_XFER_SRC attribute.
RME_ERR_PTH_INVSTATE	Cap_Thd_Src has no timeslice that can be transferred.
RME_ERR_PTH_INVSTATE	Cap_Thd_Dst or Cap_Thd_Src is in the unbound state.
RME_ERR_PTH_INVSTATE	Cap_Thd_Dst or Cap_Thd_Src is bound to a CPU different from the CPU calling this function.
RME_ERR_PTH_PRIO	The priority ceiling of Cap_Thd_Dst is higher than that of Cap_Thd_Src, and Cap_Thd_Dst is not an initial thread.

Return Value	Meaning
RME_ERR_PTH_OVERFLOW	The timeslice of the thread receiving this timeslice is already full. This error is very rare, because the timeslice generally transferred does not reach this value[44].

4.5.9 Switching to a Thread

This operation allows a user-mode scheduler to switch directly to a thread bound to the same CPU, facilitating the implementation of user-mode scheduling. The thread being switched to must have the same priority as the current thread, and must be in the ready (RME_THD_READY) state. To perform a thread switch, the following parameters are required:

Table 4-36 Parameters Needed for Switching to a Thread

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SWT.
Cap_Thd	cid_t	P1	A capability number corresponding to a thread capability that must have the RME_THD_FLAG_SWT attribute; the capability corresponding to this capability number points to the thread to switch to. This capability number can be a one-level or two-level lookup encoding. If you do not want to specify the thread to switch to and want the kernel to decide, then you may pass in RME_CID_NULL[45].
Is_Yield	ptr_t	P2	Whether this thread switch abandons all timeslices of the current thread. If this item is not 0, then this switch abandons all timeslices of the current thread and immediately sends a scheduler event to its scheduler thread. If it is called in the Init thread or an infinite thread, then this item has no effect, because the timeslices of the Init thread or an infinite thread are infinite.

[44] Receiving more would exceed or equal the maximum value RME_THD_INF_TIME allowed by the system
[45] A half-word-length INT_MIN (negative value), which is 0x8000 on a 32-bit system and 0x80000000 on a 64-bit system

The return values of this operation may be as follows:

Table 4-37 Possible Return Values for Switching to a Thread

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Thd has no RME_THD_FLAG_SWT attribute.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the unbound state.
RME_ERR_PTH_INVSTATE	Cap_Thd is bound to a CPU different from the CPU calling this function.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the blocked (RME_THD_BLOCKED) state.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the timeout (RME_THD_TIMEOUT) state.
RME_ERR_PTH_INVSTATE	Cap_Thd is in the exception (RME_THD_EXCPEND) state.
RME_ERR_PTH_PRIO	The priority of Cap_Thd is different from the priority of the current thread.

4.6 Bibliography of This Chapter

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[3] P. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: access control for time" in Real-Time Systems Symposium (RTSS), 2018 IEEE, 2018.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On, 2015, pp. 171-172.

[5] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2010, pp. 16-23.

5. Synchronous and Asynchronous Communication Mechanisms

5.1 Overview of Synchronous and Asynchronous Communication

5.1.1 Overview of Synchronous Communication

The synchronous (Synchronous) communication mechanism in an operating system is a mechanism used to synchronize two parallel or concurrent processes. In traditional operating systems it usually manifests as mechanisms such as pipes and semaphores. The greatest characteristic of the synchronous communication mechanism is its blocking nature, that is, after one side sends, if the other side does not receive, the sender blocks, and the sender's blocking is only released after the receiver receives. If the receiver receives first, then it also blocks, and the receiver's blocking is only released after the sender sends. Synchronous communication using this traditional implementation requires two threads in order to communicate between processes.

In RME, the synchronous communication model is further simplified into the thread migration (Thread Migration) invocation[1], which allows a thread to enter another process, execute a piece of code inside it, and then return to its own process to continue execution. This is one of the most efficient inter-process communication (Inter-Process Communication, IPC) implementations. Because this kind of invocation allows one execution flow to cross process boundaries, it can only be used when the two processes trust each other to a considerable degree. Its apparent result is as if a thread running inside one process directly invoked a function inside another process, so that only one thread is needed to perform inter-process communication. If this synchronous invocation is to use extra parameters, then these parameters are passed through shared memory, register sets, or coprocessor register sets. Synchronous invocations can be nested, and the number of nesting levels can be unlimited. An example of thread migration is shown below. This thread performs two migrations and alternately executes among three processes. In the figure, the red upright triangles represent invocation activation, and the red inverted triangles represent invocation return. In process 2 and process 3, the boxes on the thread's execution flow mark the thread's invocation stack, and the black triangle points to the top of the invocation stack. When returning, control always first returns into the process indicated by the top of the stack.

The design of the synchronous invocation refers to Composite[1] and Mach 3[2].

5.1.2 Overview of Asynchronous Communication

The asynchronous (Asynchronous) communication mechanism in an operating system is used to coordinate the producer-consumer (Producer-Consumer) problem in the operating system. Generally, such mechanisms include the mailbox (Mailbox), the message queue (Message Queue), and so on. The greatest characteristic of the asynchronous communication mechanism is its non-blocking nature, that is, after one side issues, if the other side does not receive, the information is first buffered, and then the sender returns directly. When receiving, if there is information, the receiver returns; if there is no information,

[1] or a local procedure call

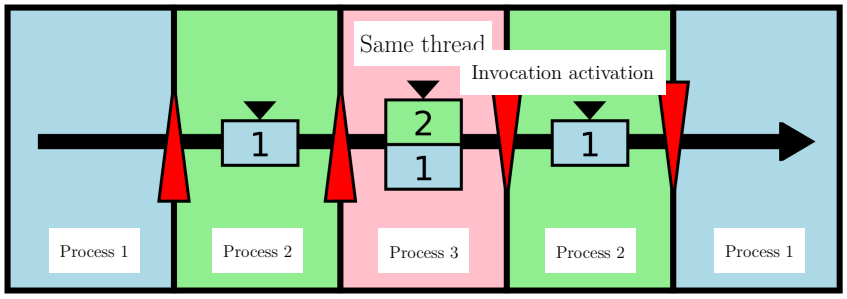


Figure 5-1 Example of Synchronous Thread Migration

then depending on the system call, the receiver may return directly, may block, or may block for a period of time until a timeout and then return.

In RME, the asynchronous communication model is further simplified into the signal endpoint (Signal Endpoint), whose main mechanism is to enable the sender to send to the signal endpoint and the receiver to receive from the signal endpoint. Sending is always non-blocking, receiving is also non-blocking when there is a signal, and receiving when there is no signal may either block or not block. A signal only carries quantity information[2], and does not carry any other information. If other information needs to be transferred, then the transfer of this information needs to be accomplished in user mode through shared memory (Shared Memory). An example of asynchronous communication is shown below. In it, thread 1, which has a lower priority, sends an asynchronous signal (red lightning bolt symbol) to thread 2, which has a higher priority; after thread 2 finishes processing, it blocks on the endpoint again (red rectangle symbol), so that thread 1 continues execution.

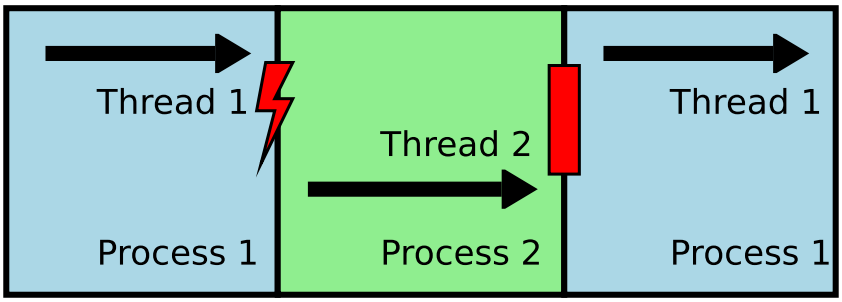


Figure 5-2 Example of Asynchronous Signal Sending and Receiving

5.2 Synchronous Communication Operations

To use the synchronous communication functionality, one must first create a synchronous invocation. When creating a synchronous invocation, the process needs to be specified. After creation, the entry of the invocation inside this process and the stack to be used by the thread need to be specified. Next, the synchronous invocation is performed, and the thread transitions from inside one thread to execute inside another process. If other synchronous invocations are used within this synchronous invocation, then the synchronous invocations will nest (Nesting). After the synchronous invocation finishes execution, the synchronous invocation return function needs to be called to return from this synchronous invocation. Compared with asynchronous communication, synchronous communication operations have immediacy, guaranteeing that a response will definitely be obtained at the fastest possible speed when the invocation is made, and when the invocation is made the thread merely crosses into another protection

[2] it has only one counter

domain to execute without changing its other attributes. Note that the synchronous invocation does not save any general-purpose registers or coprocessor registers, nor does it change the coprocessor's current state; it only changes those registers necessary for changing the program's execution flow, such as the stack pointer and the program counter. The values of the general-purpose registers and the coprocessor state will be carried directly to the thread migration entry and exit. If there are registers or coprocessor states that need to be saved, then user mode needs to do so itself. In addition, the coprocessor register set can also be used to pass extra parameters or receive extra return values.

A synchronous invocation is not bound to a CPU. If threads on multiple CPUs attempt to activate one synchronous invocation at the same time, then only the thread on one CPU will succeed, and the other threads will all return error codes; if an attempt is made to activate a synchronous invocation again while it is already activated[3], then an error code will also be returned.

If nesting occurs within synchronous invocations, then returning also needs to be done level by level. If the thread is unbound while performing a synchronous invocation, then it remains in the invocation state, and after being bound to a new CPU it will continue executing from there; if an error occurs while the thread is performing a synchronous invocation, the thread's behavior is determined by the Fault_Ret_Flag parameter passed in when setting the invocation execution properties. If this parameter is not 0, then it does not enter the RME_THD_EXCPEND state, but instead returns directly from this synchronous invocation to the previous level, and the return value is an error code indicating that an error occurred. If this parameter is 0, then the thread enters the RME_THD_EXCPEND state and waits for the error to be repaired.

5.3 Asynchronous Communication Operations

To use the asynchronous communication functionality, one must first create a signal endpoint, then have the sender send a signal to it, and the receiver receives from this endpoint. When receiving, if the endpoint has no signal, then the receiver blocks; if the reception succeeds, then the return value will be the number of remaining signals at the time of the successful return.

A signal endpoint is not bound to a CPU. If threads on multiple CPUs simultaneously attempt to block on one endpoint, then only one will succeed, and the other threads will all return an error value. If, when a thread has already blocked on an endpoint, other threads also attempt to block[4], then the blocking of the other threads will fail. In practice, the usage of one signal endpoint with multiple receivers is not recommended, because this often requires fairly complex coordination.

Note that when a thread has already blocked on a signal endpoint, only a send operation on the same CPU as the blocked thread can release that thread's blocking. Although threads on other CPUs can also send to that endpoint, they can only increase the signal count and cannot release the blocking.

RME does not allow the initial (Init) thread to receive any signals. Because receiving a signal is a potentially blocking operation, and once the Init thread blocks, RME cannot guarantee that it will be able to find a thread to run when the CPU is idle. If it is necessary both to receive signals and for the receiving thread's timeslice to be infinite, then one can create a new thread and then perform an infinite transfer from Init to it, turning it into an infinite thread.

[3] regardless of whether on the same CPU or on a different CPU
[4] regardless of whether on the same CPU or on a different CPU

In order to make application programming more flexible, RME permits four different methods of receiving endpoint signals, which are as follows:

Table 5-1 Four Receiving Options of Endpoint Signals

Receiving Option	Meaning
RME_RCV_BS	Attempt to receive a single signal in a blocking manner. May cause the thread itself to block.
RME_RCV_BM	Attempt to receive multiple signals in a blocking manner. May cause the thread itself to block, and if there are signals on the endpoint, all signals on the endpoint will be obtained.
RME_RCV_NS	Attempt to receive a single signal in a non-blocking manner. This method returns immediately when there is no signal on the endpoint.
RME_RCV_NM	Attempt to receive multiple signals in a non-blocking manner. This method returns immediately when there is no signal on the endpoint, and if there are signals on the endpoint, all signals on the endpoint will be obtained.

5.4 List of Synchronous Communication Functions

The kernel functions related to synchronous invocations are as follows:

Table 5-2 Kernel Functions Related to Synchronous Communication

Call number	Type	Purpose
RME_SVC_INV_CRT	System call	Create a synchronous invocation
RME_SVC_INV_DEL	System call	Delete a synchronous invocation
RME_SVC_INV_SET	System call	Set the execution properties of the synchronous invocation[5]
RME_SVC_INV_ACT	System call	Activate[6] a synchronous invocation
RME_SVC_INV_RET	System call	Return from a synchronous invocation

The operation flags of the synchronous invocation capability are as follows:

Table 5-3 Synchronous Invocation Operation Flags

Flag	Bit	Purpose
RME_INV_FLAG_SET	[0]	Allow setting the execution properties of this synchronous invocation.
RME_INV_FLAG_ACT	[1]	Allow activating this synchronous invocation.

Since the synchronous invocation return operation does not require the synchronous invocation capability, there is no such operation flag bit either[7].

[5] entry and stack
[6] perform or invoke
[7] see Section 5.4.5 for details

5.4.1 Creating a Synchronous Invocation

This operation creates a synchronous invocation and places its capability into an existing capability table. The newly created synchronous invocation references the process specified at creation time; once a synchronous invocation has been created within a process, this process cannot be deleted before this synchronous invocation is deleted. Creating a synchronous invocation requires the following parameters:

Table 5-4 Parameters Needed for Synchronous Invocation Creation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_CRT.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive the capability of this newly created synchronous invocation. This capability number can be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	P1.D1	A kernel memory capability number; the kernel memory range identified by it must be able to hold the entire synchronous invocation object, and it must have the RME_KOM_FLAG_INV attribute. This capability number can be a one-level or two-level lookup encoding.
Cap_Inv	cid_t	P1.D0	A capability number corresponding to a position in the capability table that is to receive the capability of this newly created synchronous invocation. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Prc	cid_t	P2	A capability number corresponding to a process capability that must have the RME_PRC_FLAG_INV attribute; the capability corresponding to this capability number points to the process that contains the newly created synchronous invocation. This capability number can be a one-level or two-level lookup encoding.
Raddr	ptr_t	P3	The kernel starting relative virtual address to be used by the newly created synchronous invocation. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.

The return values of this operation may be as follows:

Table 5-5 Possible Return Values for Synchronous Invocation Creation

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Inv is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Prc has been frozen.
RME_ERR_CPT_FROZEN	Cap_Inv has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Kom has no RME_KOM_FLAG_INV attribute, or the range is incorrect.
RME_ERR_CPT_FLAG	Cap_Prc has no RME_PRC_FLAG_INV attribute.
RME_ERR_CPT_EXIST	Cap_Inv is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.

5.4.2 Deleting a Synchronous Invocation

This operation deletes a synchronous invocation. The synchronous invocation being deleted must not be in a state of being in use. Deleting a synchronous invocation requires the following parameters:

Table 5-6 Parameters Needed for Synchronous Invocation Deletion

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_DEL.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table containing the capability of the synchronous invocation being deleted. This capability number can be a one-level or two-level lookup encoding.

Parameter Name	Type	Domain	Description
Cap_Inv	cid_t	P1	A capability number corresponding to the capability of the synchronous invocation to be deleted. The capability corresponding to this capability number must be a synchronous invocation capability. This capability number can only be a one-level lookup encoding.

The return values of this operation may be as follows:

Table 5-7 Possible Return Values for Synchronous Invocation Deletion

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Inv is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Inv has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Inv is not a synchronous invocation capability.
RME_ERR_CPT_NULL	Cap_Inv is an empty capability.
RME_ERR_CPT_NULL	Two cores attempt to delete this synchronous invocation at the same time; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt has no RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Inv is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Inv is not 0, or it is not a root capability.
RME_ERR_SIV_ACT	This synchronous invocation is still in a state of being in use.

5.4.3 Setting Synchronous Invocation Execution Properties

This operation sets the execution properties of a synchronous invocation, namely its entry and stack. When setting them, we do not care whether this synchronous invocation is already in use. Setting the execution properties of a synchronous invocation requires the following parameters:

Table 5-8 Parameters Needed for Setting Synchronous Invocation Execution Properties

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_SET.
Cap_Inv	cid_t	P1.D0	A capability number corresponding to a synchronous invocation capability that must have the RME_INV_FLAG_SET attribute; the capability corresponding to this capability number points to the synchronous invocation whose execution properties are being set. This capability number can be a one-level or two-level lookup encoding.
Entry	ptr_t	P2	The entry to be used by this thread migration. This value is a virtual address inside the process in which this synchronous invocation resides; the synchronous invocation will start execution from here.
Stack	ptr_t	P3	The execution stack of this synchronous invocation. This value is a virtual address inside the process in which this synchronous invocation resides; the synchronous invocation will use this address as the start of the stack. Whether the specific stack is an ascending stack or a descending stack is decided by the user-mode library.
Is_Exc_Ret	ptr_t	P1.D1	If it is not 0, then once an error occurs during the synchronous invocation, a return from the synchronous invocation will be forced, and error repair will not be permitted. If it is 0, then the thread will enter the RME_THD_EXCPEND state and wait for error repair.

The return values of this operation may be as follows:

Table 5-9 Possible Return Values for Setting Synchronous Invocation Execution Properties

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Inv is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Inv has been frozen.
RME_ERR_CPT_TYPE	Cap_Inv is not a synchronous invocation capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Inv has no RME_INV_FLAG_SET attribute.

5.4.4 Activating a Synchronous Invocation

This operation performs a synchronous invocation. This synchronous invocation must not be occupied. Performing a synchronous invocation requires the following parameters:

Table 5-10 Parameters Needed for Activating a Synchronous Invocation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_ACT.
Cap_Inv	cid_t	P1	A capability number corresponding to a synchronous invocation capability that must have the RME_INV_FLAG_ACT attribute; the capability corresponding to this capability number points to the synchronous invocation to be activated. This capability number can be a one-level or two-level lookup encoding.
Param	ptr_t	P2	The parameter to be passed to this synchronous invocation.

The return values of this operation may be as follows:

Table 5-11 Possible Return Values for Activating a Synchronous Invocation

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Inv is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Inv has been frozen.
RME_ERR_CPT_TYPE	Cap_Inv is not a synchronous invocation capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Inv has no RME_INV_FLAG_ACT attribute.
RME_ERR_SIV_ACT	Can only occur when RME_INV_DEPTH_MAX[8] is not 0: if this synchronous invocation is activated, the total invocation depth of the current thread will exceed this upper limit.
RME_ERR_SIV_ACT	Cap_Inv is already in the activated state[9].
RME_ERR_SIV_ACT	Two CPUs attempt to perform this invocation at the same time; the CPU that fails returns this value.

[8] see 7.3.2
[9] another thread is performing an invocation

5.4.5 Returning from a Synchronous Invocation

This operation returns from a synchronous invocation. This is a special operation; it does not require any parameters other than the call number and the thread migration return value. If multiple synchronous invocations are nested, this function returns to the previous level of synchronous invocation. If an attempt is made to call this function when there is no synchronous invocation, an error code will be returned, indicating that the return was unsuccessful. Returning from a synchronous invocation requires the following parameters:

Table 5-12 Parameters Needed for Returning from a Synchronous Invocation

Parameter Name	Type	Domain	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_RET.

The return values of this operation may be as follows:

Table 5-13 Possible Return Values for Returning from a Synchronous Invocation

Return Value	Meaning
0	The operation succeeded. This synchronous invocation returns immediately, and this return value will not be received by the caller.
RME_ERR_SIV_EMPTY	An attempt was made to call this function when there is no synchronous invocation.

5.5 List of Asynchronous Communication Functions

The kernel functions related to signal endpoints are as follows:

Table 5-14 Kernel Functions Related to Asynchronous Communication

Call number	Type	Purpose
RME_SVC_SIG_CRT	System call	Create an asynchronous signal endpoint
RME_SVC_SIG_DEL	System call	Delete an asynchronous signal endpoint
RME_SVC_SIG_SND	System call	Send a signal to an asynchronous signal endpoint
RME_SVC_SIG_RCV	System call	Receive a signal from an asynchronous signal endpoint

The operation flags of the signal endpoint capability are as follows:

Table 5-15 Signal Endpoint Operation Flags

Flag	Bit	Purpose
RME_SIG_FLAG_SND	[0]	Allow sending to this signal endpoint.
RME_SIG_FLAG_RCV_BS	[1]	Allow receiving a single signal from this signal endpoint in a blocking manner.

Flag	Bit	Purpose
RME_SIG_FLAG_RCV_BM	[2]	Allow receiving multiple signals from this signal endpoint in a blocking manner.
RME_SIG_FLAG_RCV_NS	[3]	Allow receiving a single signal from this signal endpoint in a non-blocking manner.
RME_SIG_FLAG_RCV_NM	[4]	Allow receiving multiple signals from this signal endpoint in a non-blocking manner.
RME_SIG_FLAG_SCHED	[5]	Allow this endpoint to be bound by a thread for use as a scheduler signal endpoint.

For a description of bit [5], please see 4.5.7 .

5.5.1 Creating an Asynchronous Signal Endpoint

This operation creates a signal endpoint and places its capability into an existing capability table. Creating a signal endpoint requires the following parameters:

Table 5-16 Parameters Needed for Signal Endpoint Creation

Parameter Name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_CRT.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to accept the capability of this newly created signal endpoint. This capability number may be a one-level or two-level lookup encoding.
Cap_Sig	cid_t	PO	A capability number corresponding to a location in the capability table that is to accept the capability of this newly created signal endpoint. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.

The possible return values of this operation are as follows:

Table 5-17 Possible Return Values for Asynchronous Signal Endpoint Creation

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Sig is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Sig has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Sig is not an empty capability.

5.5.2 Deleting an Asynchronous Signal Endpoint

This operation deletes a signal endpoint. The signal endpoint being deleted must not be in use[10]. Deleting a signal endpoint requires the following parameters:

Table 5-18 Parameters Needed for Asynchronous Signal Endpoint Deletion

Parameter Name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_DEL.
Cap_Cpt	cid_t	C	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table that contains the capability of the signal endpoint being deleted. This capability number may be a one-level or two-level lookup encoding.
Cap_Sig	cid_t	P1	A capability number corresponding to the capability of the signal endpoint to be deleted. The capability corresponding to this capability number must be a signal endpoint capability. This capability number can only be a one-level lookup encoding.

[10] a thread is blocked on it

The possible return values of this operation are as follows:

Table 5-19 Possible Return Values for Asynchronous Signal Endpoint Deletion

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Sig is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has been frozen.
RME_ERR_CPT_FROZEN	Cap_Sig has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Sig is not a signal endpoint capability.
RME_ERR_CPT_NULL	Cap_Sig is an empty capability.
RME_ERR_CPT_NULL	Two cores attempted to delete the signal endpoint at the same time; the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_Sig is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_Sig is not 0, or it is not a root capability.
RME_ERR_SIV_ACT	The signal endpoint is still in use.
RME_ERR_SIV_CONFLICT	The signal endpoint is a kernel endpoint and cannot be deleted. See the next chapter for details.

5.5.3 Sending a Signal to an Asynchronous Signal Endpoint

This operation sends a signal to a signal endpoint. When a thread is blocked on the endpoint, only a send on the same CPU as that thread can wake the thread. Other CPUs can also send to the endpoint, but they can only increase the count value and cannot wake the thread. Sending a signal to a signal endpoint requires the following parameters:

Table 5-20 Parameters Needed for Sending a Signal to an Asynchronous Signal Endpoint

Parameter Name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_SND.
Cap_Sig	cid_t	P1	A capability number corresponding to a signal endpoint capability that must have the RME_SIG_FLAG_SND attribute; the capability corresponding to this capability number points to the endpoint to which the signal is to be sent. This capability number may be a one-level or two-level lookup encoding.
Number	ptr_t	P2	The count value to be sent to the signal endpoint. The upper limit of the amount that can be sent at one time is 2^{16} on 32-bit systems, 2^{32} on 64-bit systems, and so on. Besides the conventional use, when it is certain that all senders and the receiver reside on the same CPU core and that the receiver's priority is always higher than that of any sender, the receiver will immediately preempt the sender when it is unblocked; this number can be used to carry information about the sender's identity, so as to avoid unnecessary shared memory.

The possible return values of this operation are as follows:

Table 5-21 Possible Return Values for Sending a Signal to an Asynchronous Signal Endpoint

Return Value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Sig is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Sig has been frozen.
RME_ERR_CPT_TYPE	Cap_Sig is not a signal endpoint capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Sig does not have the RME_SIG_FLAG_SND attribute.
RME_ERR_SIV_FULL	The number sent exceeds the upper limit of the amount that can be sent at one time.
RME_ERR_SIV_FULL	The signal count of the signal endpoint is full, so no more signals can be sent to it. This is very rare, because the upper limit of the signal count is 2^{32-2} on 32-bit systems, 2^{64-2} on 64-bit systems, and so on.

5.5.4 Receiving a Signal from an Asynchronous Signal Endpoint

This operation receives a signal from a signal endpoint. If there is no signal on the signal endpoint, the thread will be blocked until a signal arrives. Receiving a signal from a signal endpoint requires the following parameters:

Table 5-22 Parameters Needed for Receiving a Signal from an Asynchronous Signal Endpoint

Parameter Name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_RCV.
Cap_Sig	cid_t	P1	A capability number corresponding to a signal endpoint capability that must have the RME_SIG_FLAG_RCV attribute; the capability corresponding to this capability number points to the endpoint from which the signal is to be received. This capability number may be a one-level or two-level lookup encoding.
Option	ptr_t	P2	The receive method option. The following four methods are available: RME_RCV_BS: block and attempt to receive a single signal (Blocking Single). RME_RCV_BM: block and attempt to receive multiple signals (Blocking Multi). RME_RCV_NS: do not block and attempt to receive a single signal (Non-blocking Single). RME_RCV_NM: do not block and attempt to receive multiple signals (Non-blocking Multi).

The possible return values of this operation are as follows:

Table 5-23 Possible Return Values for Receiving a Signal from an Asynchronous Signal Endpoint

Return Value	Meaning
Non-negative value	The operation succeeded. The value is the number of signals received in this receive.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Sig is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Sig has been frozen.
RME_ERR_CPT_TYPE	Cap_Sig is not a signal endpoint capability, or is an empty capability.
RME_ERR_CPT_FLAG	Cap_Sig does not have the attribute required by the corresponding receive option.
RME_ERR_CPT_FLAG	RME_RCV_BS requires the RME_SIG_FLAG_RCV_BS attribute.
RME_ERR_CPT_FLAG	RME_RCV_BM requires the RME_SIG_FLAG_RCV_BM attribute.
RME_ERR_CPT_FLAG	RME_RCV_NS requires the RME_SIG_FLAG_RCV_NS attribute.
RME_ERR_CPT_FLAG	RME_RCV_NM requires the RME_SIG_FLAG_RCV_NM attribute.
RME_ERR_SIV_BOOT	An attempt was made to have the Init thread receive a signal from the endpoint in a blocking manner.
RME_ERR_SIV_ACT	A thread is already blocked on the endpoint, so no signal can be received.
RME_ERR_SIV_ACT	The supplied receive option is incorrect.
RME_ERR_SIV_CONFLICT	Two cores attempted to receive on one endpoint at the same time; a conflict occurred and a retry is required.

Note that if the non-blocking method is chosen, then when there is no signal on the endpoint, the function returns 0 directly, indicating that nothing was received in this receive. In addition, if the receive option is to block and attempt to receive multiple signals, and this receive did in fact block, then when the block is released, regardless of how many signals already exist, the number of signals received is always equal to that of the send that released the block.

5.6 Bibliography of This Chapter

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] B. Ford and J. Lepreau, "Evolving mach 3.0 to a migrating thread model," presented at the Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, San Francisco, California, 1994.

6. Kernel Function Call Mechanism and Kernel Asynchronous Signal

6.1 Overview of the Kernel Call Mechanism

Since the architecture-independent part of the RME microkernel itself implements only the generic functionality common to multiple processors, if a certain microprocessor possesses a certain feature and that feature can be operated only in kernel mode, then to make use of that feature one must use RME's kernel call mechanism. Many other systems, such as Composite[1] and seL4[2], also have similar mechanisms.

The kernel function call (Kernel Function) mechanism introduces the kernel call capability, which allows invoking a user-defined, processor-architecture-dependent kernel function and causing it to run in kernel mode. This capability must be created during the boot sequence[1] and cannot be deleted. High-precision timers of the processor, inter-processor interrupts, and adjustment of low-power operating modes, among others, can all be exploited through the kernel call mechanism; and for processors that do not have these features, the kernel does not force users to implement them, so as to achieve maximum flexibility.

Each specific kernel function corresponds to a kernel function number, which must be passed in when making a kernel function call. For the implementation of the specific kernel call mechanism, please refer to the description in the next chapter.

6.2 Overview of Kernel Asynchronous Signal

In RME, since interrupt handling functions are registered in user mode, some mechanism is needed to convey these signals out. RME exports these interrupt functions by means of the kernel asynchronous signal endpoint (Kernel Asynchronous Signal Endpoint). A kernel asynchronous signal endpoint is the same as an ordinary signal endpoint; the only difference is that it is created at system boot and cannot be deleted throughout the entire course of system operation. If signals on a kernel asynchronous signal endpoint need to be received, one need only use the same receive function as for an ordinary endpoint to receive on that endpoint. For the same reason, RME does not implement timers in the kernel, but instead passes the clock interrupt to user mode for processing.

6.3 Kernel Call Mechanism Function List

The kernel functions related to the kernel call mechanism are as follows:

Table 6-1 Kernel Call Related Kernel Functions

Call number	Type	Purpose
RME_SVC_KFN	System call	Invoke a kernel function

[1] See 7.9.2.9

The operation flags of the kernel call capability are as follows:

Table 6-2 Kernel Call Operation Flags

Flag	Type	Purpose
All bits	Bit field	The range of allowed kernel function numbers. Note that the kernel function number must not be confused with the system call number. When transferring a kernel call capability, the macro RME_KFN_FLAG(HIGH,LOW) must be used to fill in the flag bits of the new kernel call capability, where HIGH is the upper limit of the function number and LOW is the lower limit of the function number; the closed interval formed by [HIGH, LOW] is the allowed range of kernel function numbers.

6.3.1 Initial Creation of the Kernel Call Mechanism

For the initial creation of the kernel call mechanism, please see 7.9.2.9 .

6.3.2 Kernel Call Activation

This operation executes a kernel call function. This operation may carry a sub-function number and may also carry two additional parameters. Activating a kernel call requires the following parameters:

Table 6-3 Parameters Needed for Kernel Call Activation

Parameter name	Type	Field	Description
Svc_Num	ptr_t	N	Must be RME_SVC_KFN.
Cap_Kfn	cid_t	C	A capability number corresponding to a kernel call capability. This capability number may be a one-level or two-level lookup encoding.
Func_ID	ptr_t	P1.D0	Kernel function number.
Sub_ID	ptr_t	P1.D1	Sub-function number.
Param1	ptr_t	P2	The first parameter passed in.
Param2	ptr_t	P3	The second parameter passed in.

The possible return values of this operation are as follows:

Table 6-4 Possible Return Values for Kernel Call Activation

Return value	Meaning
Non-negative value	The operation succeeded. The meaning of the return value is determined by the specific underlying implementation.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kfn is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kfn has already been frozen.
RME_ERR_CPT_TYPE	Cap_Kfn is not a kernel call capability, or is an empty capability.
RME_ERR_CPT_FLAG	The function number range of Cap_Kfn does not allow an invocation whose function number is Func_ID.

Return value	Meaning
RME_ERR_CPT_NULL	The kernel call failed because the underlying hardware does not support the passed parameters. This value is generally not returned; what is usually returned is a user-defined error code.

6.4 Kernel Asynchronous Signal Function List

6.4.1 Initial Creation of Kernel Signal Endpoint

For the initial creation of the kernel signal endpoint, please see 7.9.2.11 .

6.4.2 Sending a Signal to a Kernel Signal Endpoint

For sending a signal to a kernel signal endpoint, please see 7.6.2.1 .

6.4.3 Receiving a Signal from a Kernel Signal Endpoint

The method and call for receiving signals from a kernel endpoint are the same as those for an ordinary signal endpoint; please see 5.5.4 .

6.5 Bibliography of This Chapter

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

7. Porting RME to New Architectures

7.1 Porting Overview

Porting an operating system refers to the work of modifying an operating system so that it can run on a new architecture. Sometimes, we also call the work of making it possible to compile the operating system with a new compiler porting work. Compared with the porting of systems such as Linux, the porting of RME is very simple. All of RME's code is written in ANSI/ISO C89 code that conforms to the MISRA C specification to a very large extent[1] and contains only a minimal amount of assembly, so its porting work requires only a few steps.

Before porting, we must first do some preparatory work to determine that the porting can proceed; then, we write the corresponding porting code for each part respectively. Finally, some test cases can also be used to test whether the system has been ported successfully.

7.2 Pre-Porting Checks

7.2.1 Processor

RME requires that the selected processor be able to accommodate a complete, full-featured operating system. The processor must have reasonably good performance, so as to overcome the overhead of performing system calls and make a full-featured operating system meaningful in practice; the processor must also have a certain degree of memory access control capability, which can be implemented by a memory management unit (MMU) or a memory protection unit (MPU).

Ideally, such systems are generally 32-bit or wider single-core or multi-core microprocessor platforms with clock frequencies reaching the GHz level and RAM on the order of hundreds of MB or more. However, RME can also support 32-bit microcontrollers with only 16kB RAM and 64kB ROM.

RME cannot run on platforms with less than 32kB of memory, nor on processors narrower than 32 bits. Furthermore, if the selected processor has no memory protection function, then running RME is meaningless; in this situation, RMP is a better choice. Finally, because RME does not support hardware stack mechanisms[2], the stack must be implemented in software[3].

7.2.2 Compiler

RME requires the compiler to conform to the C89 standard and to be able to generate code according to a certain function calling convention. Because RME's code is very standard and does not use library functions from the C runtime library, it suffices for the compiler to conform to the ANSI C89 standard. Common compilers such as GCC, CLANG/LLVM, MSVC, ARMCC, ICC, IAR, and TASKING all meet this requirement. RME does not use compiler extensions whose implementations differ considerably between compilers, such as bit fields, enum, and structure alignment, nor does it use undefined operations in the C language[4], thereby guaranteeing the maximum degree of compatibility.

[1] To maintain engineering regularity, RME has some necessary deviations from this specification, see 9.8.1
[2] This is the typical implementation approach of a few architectures such as PIC microcontrollers
[3] That is, the stack pointer can be modified by the user, and the stack is implemented in memory
[4] Except for a small number of undefined behaviors that are unavoidable, see 9.8.1

When using low-quality compilers, take care to disable dead code elimination and link-time optimization, and preferably also disable the compiler's loop-invariant hoisting optimization. Some aggressive optimizations may cause interrupt handling vectors to be optimized away entirely[5], causing the kernel to fail to work, and may also cause other kernel functionality failures. Do not use any aggressive compilation optimization options; on ordinary compilers, the recommended optimization option is (for example, for GCC) -O2 or a comparable optimization level.

7.2.3 Assembler

RME requires the assembler to be able to import symbols from C and to call them according to the function calling convention; in addition, it also requires that the code produced by the assembler be exportable and callable from C according to the function calling convention. This is usually a very easy requirement to satisfy. If the compiler can perform inline assembly, then it is acceptable not to have an assembler.

7.2.4 Linker

RME requires the linker to have the ability to link multiple object[6] files into one intermediate object file, and requires that it accept customized linker scripts. Common linkers such as ld, cl, armlink, and ilink/xlink are all sufficient to meet this requirement. The linker scripts of each linker are often different, and this often has to be decided according to the syntax of each linker.

7.2.5 Debugger

RME has no special requirements for the debugger. If a debugger is available, that is of course best, but porting can also be done without a debugger. When a debugger is available, kernel variables can be inspected directly with the debugger; when no debugger is available, the lowest-level kernel function __RME_Putchar must first be implemented to realize the printing output of a single character, and then this print output can be used to output logs. For the implementation of this function, see the next section.

7.3 Introduction to the Architecture-Dependent Parts of RME

The source files of the architecture-dependent parts of RME are all placed under the corresponding architecture name in the Platform folder. For example, the folder for the ARMv7-M architecture is named Platform/A7M. Its header files are in Include/Platform/A7M, and other architectures follow the same pattern.

Each architecture contains one or more source files and one or more header files. When the kernel includes architecture-dependent header files, it always includes rme_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 RME, 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 header files of the low-level function implementations corresponding to 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 make a copy of this port and modify it while using it as a template.

[5] Because they are not called by any function in the function call graph, and some low-quality linkers do not include interrupt vectors in the function call graph for analysis

[6] Usually .o, .obj, or .elf files

7.3.1 Type Definitions

For each architecture/compiler, the first part that needs to be ported is RME's type definitions. RME has a total of five type definitions, as shown in the table below, and they all begin with the prefix rme_; for the sake of brevity, the rme_ prefix is omitted in this manual.

Table 7-1 Overview of 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. Example: `typedef long tid_t;`
ptr_t	The type of a pointer integer. This type should be typedef'd to an unsigned integer equal in size to the processor word. Example: `typedef unsigned long ptr_t;`
cnt_t	The type of a count variable. This type should be typedef'd to a signed integer that is at most one processor word long but at least 32 bits. Example: `typedef long cnt_t;`
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. Example: `typedef long cid_t;`
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. Example: `typedef long ret_t;`

7.3.2 General Macro Definitions

Next, what needs to be ported is RME's general macro definitions. RME has the following general macro definitions:

Table 7-2 Overview of General Macro Definitions

Macro Name	Purpose
RME_EXTERN	The compiler's extern keyword. Some compilers may have a non-standard extern keyword[7], in which case this macro definition is used to handle it. Example: #define RME_EXTERN extern #define RME_EXTERN extern "C"
RME_INLINE	The compiler's inline keyword. Some compilers may have a non-standard inline keyword[8], in which case this macro definition is used to handle it. Example: #define RME_INLINE inline #define RME_INLINE __forceinline

[7] Or it may be necessary to link some C++ code with the kernel
[8] Or it may be necessary to force inlining of some functions

Macro Name	Purpose
RME_LIKELY(X)	The compiler's likely keyword, used to guide branch prediction to indicate that this branch is very likely to be taken. If the compiler has this feature, define this keyword; if not, simply define it as (X). Example: #define RME_LIKELY(X) likely(X) #define RME_LIKELY(X) __builtin_expect(!!(X),1) #define RME_LIKELY(X) (X)
RME_UNLIKELY(X)	The compiler's unlikely keyword, used to guide branch prediction to indicate that this branch is very likely not to be taken. If the compiler has this feature, define this keyword; if not, simply define it as (X). Example: #define RME_UNLIKELY(X) unlikely(X) #define RME_UNLIKELY(X) __builtin_expect(!!(X),0) #define RME_UNLIKELY(X) (X)
RME_CPU_LOCAL()	This macro returns a pointer of type struct RME_CPU_Local* to CPU-local storage. For single-core processors, the CPU-local storage can be declared directly as a variable in the architecture-dependent layer, and this macro only needs to return the address of that variable; for multi-core processors, the CPU-local storage structure needs to be dynamically created at system initialization according to the number of processors in the system. Example: #define RME_CPU_LOCAL() &RME_XXX_CPU_Local (A certain single-core architecture, statically allocated) #define RME_CPU_LOCAL() asm("mov some_reg, r0") (A certain multi-core architecture, dynamically allocated, with the address stored in a dedicated segment register)
RME_WORD_ORDER	The power of 2 corresponding to the processor word length (counted in bits). For example, a 32-bit processor corresponds to 5, a 64-bit processor corresponds to 6, and so on. Example: #define RME_WORD_ORDER (5U)
RME_PGT_PHYS_ENABLE	Whether the processor requires that the virtual address always equal the physical address. Generally speaking, for MMU-based systems this item is always set to 0, in which case regular page tables are used; for MPU-based systems such as microcontrollers, this item is always set to 1, in which case path-compressed page tables are used. Example: #define RME_PGT_PHYS_ENABLE (1U)

Macro Name	Purpose
RME_QUIE_TIME	The length of the quiescence period, in units of timeslices. For a single processor, because there is no true parallelism, every capability always becomes quiescent immediately, so this item is set to "0"; for multiprocessor systems, in theory the value filled in here should exceed twice the kernel's worst-case execution time (WCET), while engineering practice recommends ten times. Generally speaking, the duration of one timeslice (on the order of at least 100us) far exceeds the kernel's WCET (on the order of 10us), so filling in 1 here is sufficient. Example: #define RME_QUIE_TIME (1U)
RME_TIMESTAMP	The system timestamp used to determine the quiescence period. It can be defined as a variable that is periodically incremented by the clock interrupt, or as a free-running counter. Example: #define RME_TIMESTAMP RME_Timestamp (The case where it is defined as a variable) #define RME_TIMESTAMP RME_X64_RDTSC (The case where it is defined as reading a free-running counter)
RME_CPT_ENTRY_MAX	The maximum length of a capability table, in units of the number of capabilities. Because a capability table must be traversed in its entirety both when it is created and when it is deleted, this macro can control the upper bound on the execution time of such traversal by controlling the size of the capability table. For situations where there is no overly high real-time requirement, or where the upper bound on capability table size is controlled by a trusted user-mode capability manager, this value can be set to 0; otherwise, it can be set to the upper bound on capability table size. If this value is not 0, the final upper bound on capability table size is determined by the smaller of this value and the upper bound permitted by the processor word length. Example: #define RME_CPT_ENTRY_MAX (0U) (The case with no limit) #define RME_CPT_ENTRY_MAX (128U) (The case where the capability table size is limited)

Macro Name	Purpose
RME_PGT_RAW_ENABLE	The switch for direct user-mode management of page tables. Once the user turns on this switch macro, the kernel will no longer be responsible for page tables and will hand all page-table-related matters over to a trusted user-mode process. When creating a process, the physical address of the page table can be specified directly, and the kernel will directly trust and use that physical address; this provides additional flexibility for physical address management and can be used in high-concurrency systems or microcontroller embedded systems. Specifically, the page table system calls and kernel function calls will disappear, the parameters of the RME_SVC_PRC_CRT and RME_SVC_PRC_PGT system calls will change, and the kernel will no longer call any page table low-level functions other than __RME_Pgt_Set and __RME_Pgt_Kom_Init; the parameters of __RME_Pgt_Set also change. For specific details, please refer to the relevant chapters. Example: #define RME_PGT_RAW_ENABLE (0U) (The case where the kernel manages page tables) #define RME_PGT_RAW_ENABLE (1U) (The case where a trusted user-mode process manages page tables directly)
RME_KOM_VA_BASE	The starting address of the user-allocatable kernel virtual memory. Simply fill in the starting address of the kernel virtual memory. When kernel objects are created, kernel memory is allocated starting from here, and these allocations are recorded in the kernel memory registration table. Example: #define RME_KOM_VA_BASE (0xC0000000U)
RME_KOM_VA_SIZE	The size of the address space of the user-allocatable kernel virtual memory. Simply fill in the size of the kernel virtual memory address space. For situations that require dynamic detection of the kernel virtual memory space size[9], fill in the minimum size permitted by the kernel here[10]. This macro is mainly used to indirectly specify the size of the kernel memory allocation table in situations where the kernel virtual memory size is fixed (such as determining the value of RME_KOT_WORD_NUM, etc.); for situations that dynamically detect the kernel virtual memory space size, it has no practical significance. Example: #define RME_KOM_VA_SIZE (0x30000000U)

[9] Such as x86-64, etc.

[10] If the available kernel memory virtual address space detected is smaller than this size, the kernel may refuse to start

Macro Name	Purpose
RME_KOM_SLOT_ORDER	The power of 2 corresponding to the kernel virtual memory allocation granularity. For example, if the minimum granularity of kernel memory allocation is 16Byte, then the number to fill in here is $\log_2(16) = 4$. Note that the minimum granularity of memory allocation cannot be less than one processor word. Example: #define RME_KOM_SLOT_ORDER (4U)
RME_HYP_VA_BASE	The starting address of the virtual memory dedicated to the virtual machine monitor. This memory is for the virtual machine monitor's use; a thread's virtualization attribute can be set to this address range so that its registers are saved here when the thread is switched. Example: #define RME_HYP_VA_BASE (0x01000000U)
RME_HYP_VA_SIZE	The size of the virtual memory dedicated to the virtual machine monitor. Simply fill in the actual size of the virtual machine dedicated virtual memory. If this feature is not used, then the size of this macro needs to be set to 0, in which case the macro RME_HYP_VA_BASE is also invalid. Example: #define RME_HYP_VA_SIZE (0x00010000U)
RME_KSTK_VA_BASE	The initial virtual address of the kernel stack segment. This is the stack used when the kernel starts up and does not always represent the runtime stack; on multi-core systems, the kernel code running on each CPU may be assigned a new stack. Example: #define RME_KSTK_VA_BASE (0xF0000000U)
RME_KSTK_VA_SIZE	The initial length of the kernel stack segment. Example: #define RME_KSTK_VA_SIZE (0x1000U)
RME_PREEMPT_PRIO_NUM	The maximum number of preemption priorities supported by the kernel. This number must be an integer multiple of the processor word length (counted in bits), and must not exceed the power of 2 corresponding to one quarter of the processor word length, for example no more than 2^8 for a 32-bit processor and no more than 2^{16} for a 64-bit processor. Generally speaking, defining this value as the processor word length is sufficient. Example: #define RME_PREEMPT_PRIO_NUM (32U)

Macro Name	Purpose
RME_PGT_SIZE_NOM(X)	The size of a non-top-level page directory on the processor. This macro takes one argument, whose meaning is the power of 2 corresponding to the number of entries in that page directory. If the page directory contains 1024 entries, then the value of X is 10, and the macro is RME_PGT_SIZE_NOM(10), which returns the size of that page table in bytes. If each page table entry is 4 bytes in size and the additional prefix data[11] attached to the page table is 4096 bytes in size, then the macro should return 8192.
RME_PGT_SIZE_TOP(X)	The size of a top-level page directory on the processor. This macro takes one argument, whose meaning is the power of 2 corresponding to the number of entries in that page directory. This macro is actually equal to RME_PGT_SIZE_NOM(X) plus the size of the extra additional data at the top level[12].
RME_KOT_VA_BASE	The kernel virtual memory address where the kernel memory registration table resides. For most architectures, simply define this macro as the base address of an array of size RME_KOT_WORD_NUM; for the small number of architectures with extremely large amounts of memory[13], because compilers such as GCC by default place the kernel within the top 2GB at most, the size of the kernel memory registration table is limited to 2GB, at which point at most 1TB of kernel memory is supported. Therefore, in this case the registration table needs to be relocated, pointing it to an address that is not so limited; this address can be specified by the linker script. Example: #define RME_KOT_VA_BASE RME_XXX_Kot (Small kernel memory allocation table) #define RME_KOT_VA_BASE ((ptr_t*)0xFFFF800001000000U) (Large kernel memory allocation table)
RME_INV_DEPTH_MAX	The limit on the maximum depth of thread invocations. This limit exists to ensure that the worst-case execution time of thread deletion is controllable, because it must peel off, layer by layer, the invocations that the thread is currently performing. For situations where there is no overly high real-time requirement or where threads are never deleted, this value can be set to 0; otherwise, it can be set to the upper bound on invocation depth. Example: #define RME_INV_DEPTH_MAX (0U) (The case with no limit) #define RME_INV_DEPTH_MAX (32U) (The case where the invocation depth is limited)

[11] Present only when an MPU is used
[12] Present only when an MPU is used
[13] For example, the latest x86-64 can have hundreds of TB of memory

Macro Name	Purpose
RME_ASSERT_FAILED(F,L,D,T)	Assertion failure handling. Once an assertion failure occurs in the kernel, it means that the kernel has completely crashed and must be fully reset and restarted; the specific handling method is architecture-dependent. The four parameters of this macro are __FILE__, __LINE__, __DATE__, and __TIME__ provided by the C preprocessor; if the user wishes to use these four parameters, they can be read from here. Example: #define RME_ASSERT_FAILED(F,L,D,T) __reboot_processor()
RME_COP_NUM	The number of coprocessors in the system. If the kernel supports coprocessors, simply set it to a value greater than 0. The kernel code uses whether this value is 0 to determine whether coprocessor-related functions need to be linked, but at runtime the kernel does not care about the specific value of this number. #define RME_COP_NUM (1U)

7.3.3 Special Macro Definitions

These macro definitions appear only on microcontroller platforms and are meaningless for microprocessor platforms. They are used in conjunction with the RVM virtual machine monitor, and these macro definitions can be filled in by the project generator that comes with RVM. For information about RVM, please consult the RVM manual.

Table 7-3 Overview of Special Macro Definitions

Macro Name	Purpose
RME_RVM_INIT_CPT_SIZE	The size of the initial capability table created at kernel startup. Example: #define RME_RVM_INIT_CPT_SIZE (54U)
RME_RVM_CPT_BOOT_FRONT	The initial capability table watermark after the kernel has booted. Beyond this watermark, all capabilities are empty, and RVM will use them to create kernel objects. The kernel itself does not use this macro; the user can check after kernel startup whether the occupied capabilities exceed this watermark, and report an error if they exceed it. Example: #define RME_RVM_CPT_BOOT_FRONT RVM_CAP_NUM
RME_RVM_KOM_BOOT_FRONT	The kernel memory watermark after kernel startup, a value relative to RME_KOM_VA_BASE. The kernel itself does not use this macro; the user can check after kernel startup whether the occupied kernel memory exceeds this watermark, and report an error if it does. #define RME_RVM_KOM_BOOT_FRONT RVM_KOM_ADDR

Macro Name	Purpose
RME_RVM_CPT_DONE_FRONT	The initial capability table watermark after RVM has completed creating all kernel objects. Beyond this watermark, all capabilities are empty, and the user can use them to create kernel objects. The kernel itself does not use this macro; this macro is provided for the user's convenience when writing startup code. Example: #define RME_RVM_CPT_DONE_FRONT USER_CAP_NUM
RME_RVM_KOM_DONE_FRONT	The kernel memory watermark after kernel startup, a value relative to RME_KOM_VA_BASE. The kernel itself does not use this macro; this macro is provided for the user's convenience when writing startup code. The user can check after startup whether the occupied kernel memory exceeds this watermark, and report an error if it does. #define RME_RVM_KOM_DONE_FRONT USER_KOM_ADDR
RME_RVM_PHYS_VCT_NUM	The number of physical interrupts. This macro will be filled in by the project generator according to the actual situation of the chip.
RME_RVM_PHYS_VCTF_BASE	The starting position and size of the physical interrupt flag storage area. It is used to notify the virtual machine monitor of the occurrence of physical interrupts. This macro will be filled in by the project generator.
RME_RVM_PHYS_VCTF_SIZE	
RME_RVM_VIRT_EVT_NUM	The number of event sources. This macro will be filled in by the project generator according to the project configuration.
RME_RVM_VIRT_EVTF_BASE	The starting position and size of the event source flag storage area. It is used to notify the virtual machine monitor of the occurrence of event source stimuli. This macro will be filled in by the project generator.
RME_RVM_VIRT_EVTF_SIZE	

7.3.4 Debug Switch Macro Definitions

These macro definitions relate to kernel debug printing. Their values determine whether kernel debugging features are enabled.

Table 7-4 Overview of Debug Switch Macro Definitions

Macro Name	Purpose
RME_ASSERT_ENABLE	Controls whether the assertion macro is enabled. If the RME_ASSERT macro needs to be enabled to detect errors, define it as 1; otherwise, define it as 0, in which case assertions will not be checked. Example: #define RME_ASSERT_ENABLE (1U)

Macro Name	Purpose
RME_DBGLOG_ENABLE	Controls whether the various debug print macros prefixed with RME_DBG_ are enabled. If debug printing is needed in the kernel, define it as 1. Example: #define RME_DBGLOG_ENABLE (1U)

7.3.5 Architecture-Dependent Structures

RME has three architecture-dependent structures in total, as follows:

Table 7-5 Overview of Architecture-Dependent Structures

Structure	Meaning
RME_Reg_Struct	The structure of the registers pushed onto the stack when entering an interrupt function, containing the various registers of the CPU.
RME_Exc_Struct	The structure related to exception handling (containing the exception cause and address, etc.), containing its various registers.
RME_Iret_Struct	The structure of the registers, related to the program execution flow, that need to be saved and restored in a synchronous invocation.

The implementation of these three structures is related to the implementation of the assembly functions for the entry sections of the system interrupt vectors.

7.3.6 Low-Level Assembly Functions

RME requires only four short low-level assembly functions to be implemented in assembly or inline assembly. The names and meanings of these functions are as follows:

Table 7-6 Overview of Low-Level Assembly Functions

Function Name	Meaning
Main function (main) or program entry point	The entry point of the kernel program.
__RME_Int_Disable	Disable processor interrupts.
__RME_Int_Enable	Enable processor interrupts.
__RME_User_Enter	Enter user mode for execution.

The specific implementation methods and implementation order of these functions will be explained in later chapters.

7.3.7 System Interrupt Vectors

At a minimum, RME requires only three interrupt vectors to be implemented in assembly or inline assembly. The names and meanings of these interrupt vectors are as follows:

Table 7-7 Overview of System Interrupt Vectors

Interrupt Vector Name	Meaning
System timer interrupt vector	Handle the system timer interrupt and manage timeslice usage.
System call interrupt vector	Used when handling system calls.
System fault interrupt vector	Used when memory access errors and other processor errors occur.

The specific implementation methods and implementation order of these interrupt vectors will be explained in later chapters.

7.3.8 Other Low-Level Functions

These low-level functions involve page tables, processor special functions, and other aspects. These functions can be implemented in assembly, or not in assembly, or partly in C and partly in inline assembly. These functions can be divided into the following categories:

7.3.8.1 Kernel Debug Print Function

Table 7-8 Overview of Kernel Debug Print Functions

Function	Meaning
__RME_Putchar	Print one character to the kernel debug console.

7.3.8.2 Processor Special Function Macro Definitions

Table 7-9 Overview of Processor Special Function Macro Definitions

Macro/Function	Meaning
RME_READ_ACQUIRE	Read-acquire operation.
RME_WRITE_RELEASE	Write-release operation.
RME_COMP_SWAP	Compare-and-swap atomic operation.
RME_FETCH_ADD	Fetch-and-add atomic operation.
RME_FETCH_AND	Fetch-and-AND atomic operation.
RME_MSB_GET	Get the most significant bit (MSB) position of a word.

7.3.8.3 Initialization and Boot Functions

Table 7-10 Overview of Initialization and Boot Functions

Function	Meaning
__RME_Lowlvl_Init	Low-level hardware initialization.
__RME_Boot	Create the initial kernel objects and start the system.

7.3.8.4 Register Set Related Functions

Table 7-11 Overview of Register Set Related Functions

Function	Meaning
__RME_Svc_Param_Get	Get the system call parameters from the register set.
__RME_Svc_Retval_Set	Set the system call return value in the register set.
__RME_Thd_Reg_Init	Initialize the register set of a thread or invocation.
__RME_Thd_Reg_Copy	Copy one register set to another register set.
__RME_Thd_Reg_Print	Print a thread's register set; for debugging only.
__RME_Thd_Cop_Init	Initialize a thread's coprocessor register set.
__RME_Thd_Cop_Swap	Save and restore a thread's coprocessor register set.
__RME_Inv_Reg_Save	Save the registers necessary for returning from a synchronous invocation.
__RME_Inv_Reg_Restore	Restore the registers necessary for returning from a synchronous invocation.
__RME_Inv_Retval_Set	Set the return value of a synchronous invocation in the register set.

7.3.8.5 Functions Related to the Coprocessor Register Set

Table 7-12 Overview of Functions Related to the Coprocessor Register Set

Function	Meaning
__RME_Thd_Reg_Init	Initializes the register set of a thread or an invocation.
__RME_Thd_Reg_Copy	Copies one register set to another register set.
__RME_Thd_Cop_Init	Initializes the coprocessor register set of a thread.
__RME_Thd_Cop_Swap	Saves and restores the coprocessor register set of a thread.

7.3.8.6 Kernel Function Call Functions

Table 7-13 Overview of Kernel Function Call Functions

Function	Meaning
__RME_Kfn_Handler	The implementation of kernel function calls.

7.3.8.7 Page Table Related Functions

Table 7-14 Overview of Page Table Related Functions

Function	Meaning
__RME_Pgt_Set	Switches the page table currently in use[14].
__RME_Pgt_Kom_Init	Initializes the kernel page table.
__RME_Pgt_Check	Checks whether the page directory parameters can be supported by this architecture.
__RME_Pgt_Init	Initializes a page directory.
__RME_Pgt_Del_Check	Checks whether this page directory can be deleted.
__RME_Pgt_Page_Map	Maps a page into a page directory.
__RME_Pgt_Page_Unmap	Removes the mapping of a page from a page directory.
__RME_Pgt_Pgdir_Map	Maps a child page directory into a parent page directory.
__RME_Pgt_Pgdir_Unmap	Removes the mapping of a child page directory from a parent page directory.
__RME_Pgt_Lookup	Looks up a physical page in a page directory according to its relative position.
__RME_Pgt_Walk	Starting from the top-level page directory, looks up the attributes of the physical page corresponding to a virtual address.

7.4 Porting of Type Definitions and Low-Level Assembly Functions

For type definitions, it is only necessary to determine how the processor's word length is expressed in the compiler and then define it with typedef. Note that for some architectures and compilers, the long (long integer) type corresponds to two machine words rather than one machine word; in this case, the int type should be used to express one machine word. For other architectures and compilers, int is half a machine word in length and long is one machine word in length; in this case, care should be taken to use long to define one machine word.

When necessary, several small programs can be written using the sizeof() operator to determine which standard the compiler's machine word actually follows.

To make low-level functions easier to write, it is recommended to use the following typedefs to define the fixed-width integers that are frequently used. When defining these integers, it is also necessary to determine how many machine words the compiler's char, short, int, long, etc. each occupy. Some compilers do not provide 64-bit or 128-bit integers, in which case these types can be omitted.

Table 7-15 Overview of Commonly Used Types

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;`

[14] the top-level page directory

Type	Meaning
s64_t	A signed 64-bit integer. For example: `typedef long s64_t;`
s128_t	A signed 128-bit integer. For example: `typedef long long s128_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	An unsigned 64-bit integer. For example: `typedef unsigned long u64_t;`
u128_t	An unsigned 128-bit integer. For example: `typedef unsigned long long u128_t;`

For macro definitions and structure type definitions, the decision must be made according to the specific system configuration. The specific method of determination is described in the previous section; simply fill in these definitions as the tables explain.

Next, the porting process for low-level assembly functions is described.

7.4.1 Implementation of the Kernel Entry Point

The kernel entry point performs a series of preparatory work and then calls the RME_Kmain function. For different architectures and toolchains, the kernel entry point is written differently and must be devised on your own. Before calling RME_Kmain, the following preparatory work is required:

1. Load each part of the kernel correctly into memory through the bootloader, and place the processor in privileged mode.
2. Establish the initial page table for system boot, and use this page table to switch the system into protected mode. This page table only needs to implement the mapping from the virtual addresses of kernel memory to the physical addresses of kernel memory. This temporary page table is used only once during the boot process and is not used again afterwards, so it can be deleted after the system boot is complete.
3. Set the correct kernel stack pointer according to RME_KSTK_VA_BASE and RME_KSTK_VA_SIZE.

7.4.2 Implementation of __RME_Int_Disable

This function needs to disable the processor's interrupts and then return. There is nothing in particular to note about the implementation; generally, it is sufficient to write a CPU register or a peripheral address to disable interrupts and then return.

Table 7-16 Implementation of __RME_Int_Disable

Prototype	void __RME_Int_Disable(void)
Meaning	Disables processor interrupts.
Return value	None.
Parameters	None.

7.4.3 Implementation of __RME_Int_Enable

This function needs to enable the processor's interrupts and then return. There is nothing in particular to note about the implementation; generally, it is sufficient to write a CPU register or a peripheral address to enable interrupts and then return.

Table 7-17 Implementation of __RME_Int_Enable

Prototype	void __RME_Int_Enable(void)
Meaning	Enables processor interrupts.
Return value	None.
Parameters	None.

7.4.4 Implementation of __RME_User_Enter

This function implements the switch from privileged mode to user mode and is called only at the very end of the system boot phase. After that, the system enters its normal running state. This function only needs to assign the value of Stack to the stack pointer, assign CPUID to the register of the first parameter as determined by the calling convention, and then jump directly to Entry and perform the processor state switch. This function will never return.

Table 7-18 Implementation of __RME_User_Enter

Prototype	void __RME_User_Enter(ptr_t Entry, ptr_t Stack, ptr_t CPUID)
Meaning	Enters user mode and begins executing the first process.
Return value	None.
Parameters	ptr_t Entry The entry virtual address of the first user-mode application. ptr_t Stack The stack virtual address of the first user-mode application. ptr_t CPUID The CPUID of the processor to which this thread belongs.

7.5 Porting of System Interrupt Vectors

The porting of system interrupt vectors mainly consists of two parts: one part is the assembly code for entering and exiting the interrupt vectors, and the other part is the system interrupt vectors themselves. RME requires a minimum of only three interrupt vectors to be implemented. The entry part of an interrupt vector requires saving the processor's registers onto the stack, and its exit part requires restoring these registers from the stack. The interrupt vectors may also involve saving and restoring the system coprocessor registers.

System interrupt vectors cannot nest within each other, so their preemption priorities must be set to the same value. For exceptional cases, please see 7.6 .

7.5.1 Interrupt Push and Pop

In the entry phase of an interrupt vector, the various processor registers that are to be saved by the interrupt need to be pushed onto the stack, and the order of pushing should be consistent with the defined register structure. After pushing is complete, the corresponding handling function needs to be called,

and a pointer to the register structure on the stack must be passed to it. In the exit phase of the interrupt vector, it is only necessary to pop the register set from the stack in reverse order. In an interrupt vector, if a thread switch is involved, the system determines whether the coprocessor register set needs to be saved and restored. If necessary, the coprocessor register set will be saved and restored. The coprocessor register set is not pushed onto the stack, so the coprocessor register structure only needs to include all the coprocessor's registers; there is no need to be concerned with order.

If the stack is a full descending stack, then after all pushes are complete, the stack pointer is the pointer to the structure;

If the stack is an empty descending stack, then after all pushes are complete, the stack pointer plus the processor word length (in bytes) is the pointer to the structure.

If the stack is a full ascending stack, then after all pushes are complete, the stack pointer minus the size of the register structure plus the processor word length (in bytes) is the pointer to the structure.

If the stack is an empty ascending stack, then after all pushes are complete, the stack pointer minus the size of the register structure is the pointer to the structure.

The timer interrupt handler, the system call handler and the memory fault handler all accept only one parameter, namely the pointer to the stack. Since these three functions are generally written in C, the parameters must be passed according to the C language calling convention.

7.5.2 Timer Interrupt Vector

The timer interrupt vector is responsible for maintaining the system's basic timing. In a multiprocessor system, timer interrupts are generated in two ways: one way is to generate a timer interrupt on the master processor and increment the timestamp variable, and then send an inter-processor interrupt (Inter-Processor Interrupt, IPI) to the timer interrupt vectors of the other slave processors; this way is applicable only to tick-based kernels; the other way is to generate timer interrupts on all processors, with the timestamp obtained by reading an independent one-way incrementing counter; this way is applicable to both tick-based kernels and tickless kernels. Whichever way is adopted, the following function must be called in the timer interrupt handling vector:

Table 7-19 Implementation of the Timer Interrupt Vector

Prototype	void _RME_Tim_Handler(struct RME_Reg_Struct* Reg, ptr_t Slice)
Meaning	Performs timer interrupt handling.
Return value	None.
Parameters	struct RME_Reg_Struct* Reg The processor register set pushed onto the stack during the entry phase. ptr_t Slice In a tick-based kernel, filled with 1 on each clock interrupt; in a tickless kernel, filled with the number of timeslices that have elapsed since the last call to _RME_Tim_Elapse or _RME_Tim_Handler. This timeslice count is at least 1.

This function is already implemented by the system and need not be implemented by the user. For information on tickless kernels and _RME_Tim_Elapse, please see 9.1.3 .

7.5.3 System Call Interrupt Vector

The system call interrupt vector is responsible for accepting and responding to system calls. In the system call interrupt vector, the following function needs to be called:

Table 7-20 Implementation of the System Call Interrupt Vector

Prototype	void _RME_Svc_Handler(struct RME_Reg_Struct* Reg)
Meaning	Performs system call handling.
Return value	None.
Parameters	struct RME_Reg_Struct* Reg The processor register set pushed onto the stack during the entry phase.

This function is also already implemented by the system and need not be implemented by the user.

7.5.4 Exception Handling Interrupt Vector

The exception handling interrupt vector is responsible for handling runtime exceptions of processes. In the exception handling interrupt vector, an exception handling function provided by the user needs to be called. The implementation of this function is closely related to the architecture and therefore must be redesigned during porting. The name of this function may be decided by the user, but its prototype must be as follows.

Table 7-21 Implementation of the Exception Interrupt Vector

Prototype	void _RME_Exc_Handler(struct RME_Reg_Struct* Reg)
Meaning	Performs system exception handling.
Return value	None.
Parameters	struct RME_Reg_Struct* Reg The processor register set pushed onto the stack during the entry phase.

In this function, it is necessary to determine whether the exception that occurred is a recoverable exception or an unrecoverable exception. If it is a recoverable exception that requires no user-mode intervention[15], the kernel can handle the exception and exit the interrupt vector immediately after completing the corresponding handling and recovery work; if the recovery from a recoverable exception fails, or an unrecoverable exception[16] occurs, then the exception cause register set of the thread needs to be filled in, and then the following function provided by the system needs to be called to kill the thread. This function will notify user mode of this exception through a scheduler event and a scheduler signal[17].

Table 7-22 Kill the Current Thread

Prototype	void _RME_Thd_Fatal(struct RME_Reg_Struct* Reg)
Meaning	A fatal unrecoverable error occurred in this thread, or recovery failed, and the thread needs to be killed.
Return value	None.
Parameters	struct RME_Reg_Struct* Reg The processor register set pushed onto the stack during the interrupt entry phase.

[15] such as a software-filled TLB or MPU page fault
[16] such as division by zero, an undefined instruction, a memory access fault, etc.
[17] if the thread registered a signal endpoint when it was bound

The most typical recoverable fault is the miss interrupt of a software-filled TLB or MPU, in which the page table or page directory cache needs to be flushed. To facilitate this operation, the system provides the following function to obtain the page table of the current process of a certain thread:

Table 7-23 Obtaining the Page Table of the Current Process of a Thread

Prototype	struct RME_Cap_Pgt* _RME_Thd_Pgt(struct RME_Thd_Struct* Thd)
Meaning	A fatal unrecoverable error occurred in this thread, or recovery failed, and the thread needs to be killed.
Return value	struct RME_Cap_Pgt* The page table of the process in which this thread resides. If this thread has not performed an invocation, then it is the process in which this thread was created; otherwise it is the process to which this thread has currently been migrated.
Parameters	struct RME_Thd_Struct* Thd The structure of this thread.

On a very small number of architectures, there are sometimes cases where some exceptions cannot be accurately attributed to a thread. These exceptions may be asynchronous exceptions[18], which may occur in one thread but only be exposed in another thread; these exceptions may also be some special synchronous exceptions[19], which will be executed in a tail-chained manner after an interrupt vector or system call that may perform a thread switch, causing these exceptions to be incorrectly attributed to the thread that has just been switched to. We call this type of exception hard-to-attribute exceptions, and they must be ignored in exception handling. If they are not ignored and inaccurate exception attribution occurs, other normally executing threads (even including the Init thread) may be killed, causing serious system integrity and availability problems.

7.6 Writing Device Interrupt Vectors

Besides timer interrupts, system call interrupts and exception handling interrupts, there are two other kinds of interrupts in RME. The first kind is the transparent interrupt; the method of writing such an interrupt function is the same as that for writing an ordinary program without an operating system: there is no need to push registers to save the context according to RME's interrupt saving method, and it can nest arbitrarily with other interrupts. Therefore, the response of such an interrupt function is very fast. However, this kind of interrupt cannot call any kernel function and can at most read or write I/O, or modify a variable at some memory address. Therefore, this kind of interrupt is mainly suitable for writing kernel-mode drivers for devices that require fast response or strict timing. Typical devices of this kind are 1-Wire single-wire bus devices[20], frequency-shift keying (On-Off Keying, OOK) remote control devices[21], electromechanical control devices[22] and high-baud-rate serial interfaces[23].

The second kind is the cognizable interrupt. The entry and exit of this kind of interrupt require pushing registers according to RME's interrupt saving method, saving the thread context, and nesting is not allowed. This kind of interrupt can call some specific kernel functions to send some signals to a certain user thread. This kind of interrupt is suitable for user-mode drivers that need to send signals to applications and let them handle the data of the device. In addition, this kind of interrupt can also perform a

[18] such as BUSFAULT on ARMv7-M; in theory all exceptions are synchronous, but every rule has its exceptions
[19] such as errors related to automatic push/pop on ARMv7-M
[20] such as DS18B20, PGA300, DS2432, SHT-XX, etc.
[21] such as PT2262 software decoding, etc.
[22] such as pulse train output, xShot, etc.
[23] such as SPI, UART, etc.

context switch. In the following two sections, we mainly introduce the characteristics of cognizable interrupts, because transparent interrupts differ little from the interrupts of common bare-metal programs.

Whichever kind of interrupt vector it is, they are all a very important part of this system. The security of a system depends heavily on the security of the implementation of its interrupt vectors. The requirements on the priorities and nestability of these interrupts are: transparent interrupts may nest within each other, and their preemption priority must be higher than that of cognizable interrupts and system interrupts; cognizable interrupts may not nest within each other, and their preemption priority must be equal to or lower than that of system interrupts.

On some special architectures that do not distinguish interrupts from exceptions[24], system call interrupts and exception handling interrupts are no different from other types of interrupts: they are asynchronous and pending-capable, and once they occur together with other cognizable interrupts, they may be left pending because of insufficient non-preemptive priority. In this case, even if these two interrupts do not nest with a cognizable interrupt, a race hazard can arise: suppose thread A initiates a system call but has not yet had it handled, and at this point a cognizable interrupt occurs and, because its non-preemptive priority is higher, is handled ahead of the system call interrupt. If this cognizable interrupt switches to thread B and then exits, the system call will begin to be handled with the identity of thread B, causing information security and functional safety problems. Therefore, on these architectures, the non-preemptive priority of the system call interrupt and the exception handling interrupt must be higher than that of the timer interrupt and other cognizable interrupts. Sometimes, in order to make the system call interrupt and the exception handling interrupt appear synchronous, their preemption priorities can even be raised by an additional level.

7.6.1 Entry and Exit of Interrupt Vectors

The entry and exit of cognizable interrupt vectors are the same as the entry and exit of system interrupt vectors: both require pushing and popping in the order of the register structure, and when calling an interrupt handling function written in C, the register set needs to be passed as a parameter. The prototypes of interrupt functions written in C are all as follows:

Table 7-24 Prototype of Interrupt Functions Written in C

Prototype	void _User_Handler(struct RME_Reg_Struct* Reg)
Meaning	Performs cognizable interrupt handling.
Return value	None.
Parameters	struct RME_Reg_Struct* Reg The processor register set pushed onto the stack during the entry phase.

7.6.2 Specific Kernel Functions Callable in Interrupt Vectors

In (cognizable) interrupt vectors, some specific functions can be called to send signals to user-mode handling threads so as to make them ready, or to perform other operations. The list of these operation functions is as follows:

7.6.2.1 Sending a Signal to a Kernel Endpoint

This function is used to send a signal to a certain kernel endpoint. This is the most important function and is generally used for exporting signals out of cognizable interrupt vectors. This function can be called multiple times within one interrupt vector if there are multiple signal endpoints to send to.

[24] the most typical such architecture is ARMv7-M

Table 7-25 Parameters Required for Sending a Signal to a Kernel Endpoint

Prototype	ret_t _RME_Kern_Snd(struct RME_Cap_Sig* Cap_Sig, ptr_t Number)	
Parameter name	Type	Description
Cap_Sig	...	The type is struct RME_Cap_Sig*, a pointer directly to the root kernel signal capability. Calling this function will send a signal to this kernel signal endpoint.
Number	ptr_t	The number of counts to send to the signal endpoint. The upper limit of the number that can be sent at one time is 2^{16} in a 32-bit system and 2^{32} in a 64-bit system, and so on. Since the kernel has a higher priority than any thread, this value can only be used for counting and cannot be used to carry information about the interrupt vector number.

The possible return values of this function are as follows:

Table 7-26 Possible Return Values for Sending a Signal to a Kernel Endpoint

Return value	Meaning
0	The operation succeeded.
RME_ERR_SIV_FULL	The signal count of this signal endpoint is already full, and no further signals can be sent to it. This is very rare, because the upper limit of the signal count is $2^{32} - 2$ in a 32-bit system and $2^{64} - 2$ in a 64-bit system, and so on.

The difference between this function and the regular send function is that if a higher-priority thread is woken up during this sending process, we only make that thread ready and do not perform a context switch immediately. The reason for this is that within one interrupt vector we may send signals to multiple kernel endpoints, and if the priorities of the woken threads keep increasing, deferring the switch to the exit of the interrupt vector is a better approach. For the switch function that needs to be called when exiting an interrupt vector, please see 7.6.2.2 .

7.6.2.2 Performing a Thread Switch on Interrupt Vector Exit

This function is used to perform the final thread switch on interrupt vector exit. At the end of any interrupt vector that has called the _RME_Kern_Snd function, this function must be called.

Table 7-27 Parameters Required for Performing a Thread Switch on Interrupt Vector Exit

Prototype	void _RME_Kern_High(struct RME_Reg_Struct* Reg, struct RME_CPU_Local* Local)	
Parameter name	Type	Description
Reg	...	The type is struct RME_Reg_Struct*, a pointer to the register set. This parameter is passed in from the interrupt handling function.
Local	...	The type is struct RME_CPU_Local*, a pointer to the data structure of the current CPU.

This function has no return value.

7.7 Porting of the Kernel Debug Print Function

The low-level interface of the kernel debug print function consists of only one function, as follows:

Table 7-28 Implementation of __RME_Putchar

Prototype	ptr_t __RME_Putchar(char Char)
Purpose	Output one character to the console.
Return value	ptr_t Always returns 0.
Parameter	char Char The character to output to the system console.

In the implementation of this function, it suffices to redirect its output to some peripheral. The most common such device is the serial port.

7.8 Porting of Processor Special Function Macro Definitions

The processor special function macro definitions include atomic operations and most significant bit (MSB) search. Atomic operations are used to implement a lock-free kernel under multi-core conditions; MSB search can speed up the search for the highest priority. The functionality provided by these macro definitions can be implemented in the C language as functions, or implemented in assembly or inline assembly, depending on the circumstances. If it is implemented in assembly language, care must be taken to follow the C language calling convention, because these functions are to be called from the C language. If inline assembly, which is generally more error-prone, is used for the implementation, then care must be taken regarding the correctness of the implementation.

On a multi-core processor, all atomic operation macro definitions must include a compiler barrier (Compiler barrier) in front of them; this barrier should cause all writes initiated by the compiler before it to actually land in memory, and cause all reads initiated by the compiler after it to be re-fetched from memory.

7.8.1 Read-Acquire Operation

This macro implements a memory read-acquire barrier. On a sequentially consistent architecture or a single-core architecture, this macro can be implemented directly as a read from address X; in other cases, this value should be defined as a function that implements a read barrier. If the compiler has a built-in function that can perform this operation, using the built-in function to achieve this effect is also acceptable. This macro should ensure that memory operations below the macro are not executed before its own read operation has completed, which is equivalent to ensuring read-read and read-write consistency. It is worth noting that RME does not need to ensure write-read order consistency. If on some architectures only write-read order is inconsistent[25], this macro can also be implemented as a simple read.

Table 7-29 Implementation of RME_READ_ACQUIRE

Macro definition	ptr_t RME_READ_ACQUIRE(volatile ptr_t* PTR)
Purpose	Perform a read-acquire operation. This operation returns the value of the memory location pointed to by PTR, and prevents operations that are logically after it from being reordered before it.
Return value	ptr_t The value obtained by the read operation at this address.
Parameter	volatile ptr_t* PTR A pointer to the target operation address.

Some examples are as follows:
#define RME_READ_ACQUIRE(PTR) (*(PTR))
(the case of a simple read)
#define RME_READ_ACQUIRE(PTR) asm("LDA PTR":::memory)
(the case where the processor provides a read-acquire instruction)
#define RME_READ_ACQUIRE(PTR) asm("LDR PTR; DMB":::memory)
(the case where the processor provides a memory barrier)
#define RME_READ_ACQUIRE(PTR) __read_acquire(PTR)
(the case where a compiler built-in function is used)

7.8.2 Write-Release Operation

This macro implements a memory write-release barrier. On a sequentially consistent architecture or a single-core architecture, this macro can be implemented directly as a write to address X; in other cases, this value should be defined as a function that implements a write barrier. If the compiler has a built-in function that can perform this operation, using the built-in function to achieve this effect is also acceptable. This macro should ensure that its own write operation is not executed before the memory operations above the macro have completed, which is equivalent to ensuring read-write and write-write consistency. It is worth noting that RME does not need to ensure write-read order consistency. If on some architectures only write-read order is inconsistent[26], this macro can also be implemented as a simple write.

[25] such as x86-64
[26] such as x86-64

Table 7-30 Implementation of RME_WRITE_RELEASE

Macro definition	void RME_WRITE_RELEASE(volatile ptr_t* PTR, ptr_t VAL)
Purpose	Perform a write-release operation. This operation writes VAL into the memory location pointed to by PTR, and prevents operations that are logically before it from being reordered after it.
Return value	ptr_t The value obtained by the read operation at this address.
Parameter	volatile ptr_t* PTR A pointer to the target operation address. ptr_t VAL The value to be written by the write-release operation.

Some examples are as follows:

```
#define RME_WRITE_RELEASE(PTR,VAL) ((*(PTR))=(VAL))
(the case of a simple write)
#define RME_WRITE_RELEASE(PTR,VAL) asm("STL PTR,VAL":::memory)
(the case where the processor provides a write-release instruction)
#define RME_WRITE_RELEASE(PTR,VAL) asm("DMB;STR PTR,VAL":::memory)
(the case where the processor provides a memory barrier)
#define RME_WRITE_RELEASE(PTR,VAL) __write_release(PTR,VAL)
(the case where a compiler built-in function is used)
```

7.8.3 Compare-and-Swap Atomic Operation

This macro/function implements a basic compare-and-swap (Compare-And-Swap, CAS) atomic operation. On some architectures it has direct instruction support[27], in which case implementing this instruction in assembly or inline assembly may be considered. On some RISC architectures, one may also consider using exclusive load and exclusive store instructions for support[28]. For some newer processors, one may consider implementing it with exclusive acquire load and exclusive release store instructions[29]. The specific method of support varies from processor to processor. For single-core processors, RME provides a generic implementation _RME_Comp_Swap_Single, and this macro can be defined directly as the name of that function.

Table 7-31 Implementation of RME_COMP_SWAP

Macro definition	ptr_t RME_COMP_SWAP(volatile ptr_t* PTR, ptr_t OLD, ptr_t NEW)
Purpose	Perform a compare-and-swap atomic operation. This operation compares OLD with the value of *PTR: if OLD and *PTR are not equal, it returns 0; if OLD and *PTR are equal, it returns 1 and assigns the value of NEW to *PTR.
Return value	ptr_t The return value indicating whether this macro/function succeeded. Returns 1 on success and 0 on failure.
Parameter	volatile ptr_t* PTR A pointer to the target operation address. ptr_t OLD The old value participating in the comparison. ptr_t NEW The new value to be assigned to the target address if the old value equals the value at the target address.

Since the RME kernel uses CAS to set some flags after reading and writing certain data structures, and also reads and writes certain data structures after using CAS to set some flags, we do not want CAS to occur out of order with the other memory operations before and after it as observed by other processors. If the target processor is not sequentially consistent (Strongly-ordered)[30], and the atomic operation is not a memory access serializing instruction (Memory access serializing instruction), then when this macro/-function is implemented, a full memory barrier must be added both before and after the operation.

Strictly speaking, on processors that have exclusive load and exclusive store instructions, CAS is executed in two steps: one is the exclusive load, and the other is the exclusive store. With this in mind, the RME design allows the restriction to be relaxed a little: the exclusive load is allowed to be reordered with other operations above the CAS operation, and the exclusive store is allowed to be reordered with other operations below the CAS operation. Therefore, on processors that have exclusive acquire load and exclusive release store instructions, any additional memory barrier is unnecessary.

7.8.4 Fetch-and-Add Atomic Operation

This macro/function implements a basic fetch-and-add (Fetch-And-Add, FAA) atomic operation. During implementation it should be noted that when multiple processors perform this operation simultaneously, the set formed by the return values of this macro/function on the individual processors must be continuous, with no gaps or discontinuities. On some architectures it has direct instruction support[31], in which

[27] such as the PREFIX LOCK CMPXCHG instruction of x86-64
[28] such as the LDREX and STREX instructions of ARMv7
[29] such as the LDAEX and STLEX instructions of ARMv8
[30] such as PowerPC and ARMv7, ARMv8
[31] such as the PREFIX LOCK XADDL instruction of x86-64

case implementing this instruction in assembly or inline assembly may be considered. On some RISC architectures, one may also consider using exclusive load and exclusive store instructions for support[32]. The specific method of support varies from processor to processor. The RME design makes it unnecessary for this instruction to guarantee the effect of a full memory barrier. For single-core processors, RME provides a generic implementation _RME_Fetch_Add_Single, and this macro can be defined directly as the name of that function.

Table 7-32 Implementation of RME_FETCH_ADD

Macro definition	ptr_t RME_FETCH_ADD(volatile ptr_t* PTR, cnt_t ADDEND)
Purpose	Perform a fetch-and-add atomic operation. This operation adds ADDEND to the value of *PTR, then writes the result back to *PTR, and returns the value of *PTR before ADDEND was added.
Return value	ptr_t The value of *PTR before ADDEND was added.
Parameter	volatile ptr_t* PTR A pointer to the target operation address. cnt_t ADDEND The number to be added to the target operation address. This number may be either positive or negative.

RME assumes that this operation is wait-free (Wait-free). However, on processors that have only exclusive load and exclusive store instructions, this operation must be implemented with a loop of attempts, so that its wait-freedom is affected to some extent. However, in practice this will not break the system's real-time response: because RME's system calls have a minimum execution time, and this time limits the execution frequency of system calls, generally this frequency (even multiplied by the number of CPUs) is far lower than the frequency of loop attempts. Therefore, the number of loop attempts of this operation has an acceptable upper bound in practical applications and will not loop indefinitely.

[32] such as the LDREX and STREX instructions of ARMv7

7.8.5 Logical AND Atomic Operation

This macro/function implements a basic logical AND (Fetch-And-aNd, FAN) atomic operation. On some architectures it has direct instruction support[33], in which case implementing this instruction in assembly or inline assembly may be considered. On some RISC architectures, one may also consider using exclusive load and exclusive store instructions for support[34]. The specific method of support varies from processor to processor. The RME design makes it unnecessary for this instruction to guarantee the effect of a full memory barrier. For single-core processors, RME provides a generic implementation _RME_Fetch_Add_Single, and this macro can be defined directly as the name of that function.

Table 7-33 Implementation of RME_FETCH_AND

Macro definition	ptr_t RME_FETCH_AND(volatile ptr_t* PTR, ptr_t OPERAND)
Purpose	Perform a logical AND atomic operation. This operation logically ANDs the value of *PTR with OPERAND, then writes the result back to *PTR, and returns the value of *PTR before the logical AND with OPERAND.
Return value	ptr_t The value of *PTR before it was ANDed with OPERAND.
Parameter	volatile ptr_t* PTR A pointer to the target operation address. ptr_t OPERAND The unsigned number with which the target operation address is to be ANDed.

RME assumes that this operation is also wait-free. For specific details in this regard, please refer to the description of FAA in the previous section.

[33] such as the PREFIX LOCK ANDL instruction of x86-64
[34] such as the LDREX and STREX instructions of ARMv7

7.8.6 Getting the Most Significant Bit Position of a Word

This macro/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, and positions are counted starting from the LSB (the LSB is bit 0). For example, if the number is the 32-bit 0x12345678, then the first "1" appears at bit 28, and this function will return 28.

Table 7-34 Implementation of RME_MSB_GET

Macro definition	ptr_t RME_MSB_GET(ptr_t VAL)
Purpose	Obtain the most significant bit position of an unsigned number equal in length to the processor word, 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".
Parameter	ptr_t VAL The number whose most significant bit position is to be calculated.

Since this macro/function needs to be implemented efficiently, its implementation method differs greatly across processors. For architectures that provide a most-significant-bit calculation instruction, it suffices to implement this macro/function directly in assembly and use that instruction. For architectures that provide a leading-zero count instruction[35], one may also first calculate the number of leading zeros with an assembly function, and then subtract this value from the processor word size minus 1 (in units of bits). For example, 0x12345678 has three 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 upper half-word of a word is 0; if it is not 0, then binary-search within this upper half-word; if it is 0, then binary-search within the lower half-word, until the position of the first "1" is determined. When the halving reaches 16 or 8 bits, a lookup table can be used to directly map to the relative position of the first "1" within these 16 or 8 bits, so that no further halving is needed, and then the results of the individual halvings are combined to calculate the position of the first "1". If one does not want to write this assembly function by hand, RME also provides a generic version _RME_MSB_Generic, and this macro can be defined directly as the name of that function.

For architectures that need "to obtain the least significant bit of a word" during porting, RME additionally provides _RME_LSB_Generic. The kernel itself does not use it; it is provided merely for the convenience of users.

7.9 Porting of Initialization and Boot Functions

There are four initialization and boot functions in total, as shown below.

7.9.1 Implementation of __RME_Lowlvl_Init

This function needs to perform the initialization of low-level hardware other than the memory management unit, such as the processor clock and the cache. The reason why the memory management unit does not need to be initialized here is that the memory management unit has in fact already been initialized before _RME_Kmain exits.

[35] such as ARMv7, which provides the CLZ instruction

Table 7-35 Implementation of __RME_Lowlvl_Init

Prototype	void __RME_Lowlvl_Init(void)
Purpose	Perform initialization of the lowest-level hardware. This includes the initial setup of the processor clock, the initialization of other necessary hardware[36], and so on. Only after this function has finished running does the initialization of the kernel data structures begin.
Return value	None.
Parameter	None.

7.9.2 Implementation of __RME_Boot

This function is the most important function in the RME boot process. It runs in kernel mode, creates the capability table and page table of the Init process, adds all user-accessible pages to the page table of the Init process, and creates all kernel signal endpoints and kernel call capabilities. On a single-core system, this function needs to create a thread, set the execution attributes, and finally call _RME_Enter_User_Mode to switch to it for execution. On a multi-core system, the system needs to initialize the other processors and have them each create, on their respective CPU cores, a thread of their own, and then jump into that thread to run.

Table 7-36 Implementation of __RME_Boot

Prototype	void __RME_Boot(void)
Purpose	This function starts the first process in the system, Init, and initializes all kernel signal endpoints and kernel function call capabilities in the system, and is also responsible for adding all user-accessible pages in the system to Init's initial page table.
Return value	None.
Parameter	None.

The various functions that this function needs to call are as follows. Except for the last three functions listed, which are provided by the user, all the other functions are built-in functions of RME.

Table 7-37 Overview of the Functions That __RME_Boot Needs to Call

Function	Number of calls	Purpose
_RME_Kot_Init	Only needs to be called once for the whole system	Initializes the kernel object table at boot.
_RME_CPU_Local_Init	Called once per processor	Initializes the local storage of each processor at boot.
_RME_Cpt_Boot_Init	Only needs to be created once for the whole system	Creates the initial capability table at boot.
_RME_Cpt_Boot_Crt	Depends on the circumstances	Creates other capability tables at boot.
_RME_Pgt_Boot_Crt	Only needs to create one group for the whole system	Creates page directories at boot.

[36] such as the cache controller and interrupt controller, or other peripherals on the processor board that must be initialized in the early power-on stage

Function	Number of calls	Purpose
_RME_Pgt_Boot_Con	Only needs to be called for one group for the whole system	Constructs page directories at boot.
_RME_Pgt_Boot_Add	Only needs to be called for one group for the whole system	Adds pages to the page directories at boot.
_RME_Prc_Boot_Crt	Only needs to be created once for the whole system	Creates the first process at boot.
_RME_Kfn_Boot_Crt	Only needs to be created once for the whole system	Creates the kernel function call capability at boot.
_RME_Kom_Boot_Crt	Depends on the circumstances	Creates kernel memory capabilities at boot.
_RME_Sig_Boot_Crt	Depends on the circumstances	Creates kernel signal endpoints at boot.
_RME_Thd_Boot_Crt	Called once per processor	Creates the initial thread at boot.
__RME_Pgt_Set	Called once per processor	Sets the processor to use the current page table.
__RME_Int_Enable	Called once per processor	Enables interrupts.
__RME_User_Enter	Called once per processor	Enters user mode and begins execution.

The above functions are listed in the logical order of invocation. The introduction and calling methods of these functions are as follows. Once any one of these functions returns a failure, the entire system boot process must be stopped. Therefore, it is recommended to wrap these functions with the macro check RME_ASSERT(func(...) == 0); once a failure occurs, the system enters an infinite loop and prints kernel crash information.

7.9.2.1 Initializing the Kernel Object Table at Boot Time

This function is used to initialize the kernel object table at system boot. The initialization work consists of clearing the table to zero, representing that no kernel memory is occupied. The kernel calls this function once by default at boot, initializing the portion of the kernel object table that can be determined at compile time. For some architectures this is sufficient, and the function need not be called again[37]. However, on some other architectures the amount of kernel memory needs to be dynamically probed, so this work may need to be performed by the porter by calling this function again[38].

[37] such as Cortex-M and other architectures whose kernel object table size is determined at compile time
[38] such as x86-64 and other architectures whose kernel object table size is determined at run time

Table 7-38 Parameters Needed for Initializing the Kernel Object Table at Boot Time

Prototype	ret_t _RME_Kot_Init(ptr_t Word)	
Parameter name	Type	Description
Word	ptr_t	The size of the kernel object table, in units of the processor word size. This value must be calculated dynamically according to the kernel memory allocation granularity and the detected amount of kernel memory; the specific calculation method is to divide the size of the kernel memory address space by the kernel memory allocation granularity, and then divide by the number of bits of the processor.

The possible return values of this function are as follows:

Table 7-39 Possible Return Values for Initializing the Kernel Object Table at Boot Time

Return value	Purpose
0	The operation succeeded.
-1	The passed kernel object table size is smaller than the default minimum size. The default minimum size is calculated from the macro RME_KOM_SIZE.

7.9.2.2 Initializing the Local Storage of Each Processor at Boot Time

This function is used to initialize the local storage area of each processor at system boot. The local storage area contains the processor's CPUID, its timer signal endpoint and interrupt signal endpoint, and the processor's run queue. This function initializes these data structures. For a single-core architecture, it suffices to call this function once to initialize the storage area that is statically defined at compile time; for a multi-core architecture, each processor needs to call it once to initialize its own storage area.

Table 7-40 Parameters Needed for Initializing the Local Storage of Each Processor at Boot Time

Prototype	void _RME_CPU_Local_Init(struct RME_CPU_Local* CPU_Local, ptr_t CPUID)	
Parameter name	Type	Description
CPU_Local	...	The type is struct RME_CPU_Local*, and it is a pointer to the CPU local storage to be initialized.
CPUID	ptr_t	The CPUID of this CPU, which identifies the number of this CPU.

This function has no return value.

7.9.2.3 Creating the Initial Capability Table at Boot Time

This function is used to create the first capability table at system boot, and to place the capability pointing to this capability table into the specified capability slot in that capability table. Compared with the

system call that creates a capability table, the difference of this function is that it can be used only at system boot, and it does not require a parent capability table[39]. This function does not require a kernel memory capability. In addition, the size of the capability table created by this function is not limited by RME_CPT_ENTRY_MAX but only by the processor word size.

Table 7-41 Parameters Needed for Creating the Initial Capability Table at Boot Time

Prototype	ret_t _RME_Cpt_Boot_Init(cid_t Cap_Cpt, ptr_t Vaddr, ptr_t Entry_Num)	
Parameter name	Type	Description
Cap_Cpt	cid_t	The location that is to receive the generated capability table capability. This capability number can only be a one-level lookup encoding.
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the initial capability table.
Entry_Num	ptr_t	The number of entries contained in this capability table, which must be between 1 and RME_CID_2L.

The possible return values of this function are as follows:

Table 7-42 Possible Return Values for Creating the Initial Capability Table at Boot Time

Return value	Purpose
0	The operation succeeded.
RME_ERR_CPT_RANGE	The passed capability table capability number parameter exceeds the range allowed by the operating system.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Crt is out of range.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.

This function is called only once. It creates the very first capability table. This capability table will later be used to hold other capabilities created during kernel initialization.

7.9.2.4 Creating Other Capability Tables at Boot Time

This function is used to create other capability tables at system boot, and to place the capability pointing to this capability table into the specified capability table. Compared with the system call that creates the initial capability table, the difference of this function is that it requires a parent capability table to hold the capability pointing to itself, and does not place the capability pointing to itself into itself. This function does not require a kernel memory capability. In addition, the size of the capability table created by this function is not limited by RME_CPT_ENTRY_MAX but only by the processor word size.

[39] because there are no other capability tables in the system at this time

Table 7-43 Parameters Needed for Creating Other Capability Tables at Boot-time

Prototype	ret_t _RME_Cpt_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt_Crt, cid_t Cap_Crt, ptr_t Vaddr, ptr_t Entry_Num)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a solid pointer to the parent capability table kernel object. All capability numbers are with respect to this capability table.
Cap_Cpt_Crt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created capability table capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Cpt	cid_t	The location that is to receive the generated capability table capability. This capability number can only be a one-level lookup encoding.
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the newly created capability table.
Entry_Num	ptr_t	The number of entries contained in this capability table, which must be between 1 and RME_CID_2L.

The possible return values of this function are as follows:

Table 7-44 Possible Return Values for Creating Other Capability Tables at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The passed-in capability table entry count parameter exceeds the range allowed by the operating system.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Crt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Crt is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Cpt_Crt has already been frozen.
RME_ERR_CPT_FROZEN (unlikely to return this value)	Cap_Crt is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt_Crt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt_Crt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Crt is not an empty capability.

Return value	Meaning
RME_ERR_CPT_KOT	Failed to allocate kernel memory.

The number of calls to this function depends on the situation. If, in addition to the initial first capability table created by _RME_Cpt_Boot_Init, we also need other capability tables, then it needs to be called.

7.9.2.5 Creating Page Directories at Boot-time

This function is used to create a page directory at system boot time and to place the capability pointing to this page directory into the specified capability table. This function does not require a kernel memory capability.

Table 7-45 Parameters Needed for Creating Page Directories at Boot-time

Prototype	ret_t _RME_Pgt_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Pgt, ptr_t Vaddr, ptr_t Base, ptr_t Is_Top, ptr_t Size_Order, ptr_t Num_Order)	
Parameter name	Type	Description
CPT	…	The type is struct RME_Cap_Cpt*, which is a solid pointer to the parent capability table kernel object. All capability numbers are with respect to this capability table.
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created page directory capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	A capability number corresponding to a location in the capability table that is to accept this newly created page directory capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the newly created page directory.
Base	ptr_t	The starting address of the mapping of the newly created page directory; the last bit is the top-level flag, see below.

Prototype	ret_t _RME_Pgt_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Pgt, ptr_t Vaddr, ptr_t Base, ptr_t Is_Top, ptr_t Size_Order, ptr_t Num_Order)	
Parameter name	Type	Description
Is_Top	ptr_t	Whether this page directory is the top-level page directory. "1" means that this page directory is the top-level page directory.
Size_Order	ptr_t	The size order of this page directory.
Num_Order	ptr_t	The number order of this page directory.

The possible return values of this function are as follows:

Table 7-46 Possible Return Values for Creating Page Directories at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN (unlikely to return this value)	Cap_Pgt is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Pgt is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_PGT_HW	Restricted by the underlying hardware; such a page directory is not allowed to be created.

At boot time, as many page directories as are needed are created. Therefore, this function may be called multiple times, producing a group of page directories. In a typical boot process, only one processor needs to perform this function, so only one group of calls to this function is made in the entire system.

7.9.2.6 Constructing Page Directories at Boot-time

This function is used to construct page directories at system boot time, organizing the multiple page directories created in the previous step into a directory tree[40]. In the following steps, we will use the initial capability table and page table to create the first process.

[40] That is, a page table.

Table 7-47 Parameters Needed for Constructing Page Directories at Boot-time

Prototype	ret_t _RME_Pgt_Boot_Con(struct RME_Cap_Cpt* Cpt, cid_t Cap_Pgt_Parent, ptr_t Pos, cid_t Cap_Pgt_Child)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Cap_Pgt_Parent	cid_t	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_CON_PARENT attribute; the capability corresponding to this capability number points to the parent page directory. This capability number may be a one-level or two-level lookup encoding.
Pos	ptr_t	The position of the target page table entry in the target page directory that is to receive the transfer. This page table entry must be empty.
Cap_Pgt_Child	cid_t	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_CHILD attribute; the capability corresponding to this capability number points to the child page directory. This capability number may be a one-level or two-level lookup encoding.
Flags_Child	ptr_t	The attributes of the child page directory when it is mapped. These attributes limit the access permissions of all page directories below this mapping. For different architectures, the meaning of the value at this position is also different. For some architectures that do not support page directory attributes[41], this value is invalid.

[41] For example, all MPU-based systems.

The possible return values of this function are as follows:

Table 7-48 Possible Return Values for Constructing Page Directories at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Pgt_Parent or Cap_Pgt_Child is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt_Parent or Cap_Pgt_Child is not a page directory capability.
RME_ERR_PGT_ADDR	Pos exceeds the number of page table entries in the parent page directory.
RME_ERR_PGT_ADDR	The total size of the child page directory is larger than the size of one page of the parent page directory.
RME_ERR_PGT_ADDR	When the physical-address-equals-virtual-address check is enabled, the mapped physical address conflicts with the target virtual address.
RME_ERR_PGT_MAP	The attempt to construct failed for hardware reasons. The specific reason for the failure is hardware-dependent; it may be that the hardware does not support this kind of mapping.

At boot time, this function is called as many times as page directories need to be constructed. In a typical boot process, only one processor needs to perform this function, so only one group of calls to this function is made in the entire system.

7.9.2.7 Adding Pages to Page Directories at Boot-time

This function is used to add pages to already constructed page directories, and this operation ignores whether the page directory permits add operations. After boot, these pages will constitute all user-addressable space. This function is the only opportunity to add new physical memory pages to the system; the memory space accessible at user addresses in the future can only be produced from these pages. Moreover, when these pages are mapped, an attribute must also be provided; in subsequent page mapping operations, this physical memory page cannot possess more attributes.

Table 7-49 Parameters Needed for Adding Pages to Page Directories at Boot-time

Prototype	ret_t _RME_Pgt_Boot_Add(struct RME_Cap_Cpt* Cpt, cid_t Cap_Pgt, ptr_t Paddr, ptr_t Pos, ptr_t Flag)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Paddr	ptr_t	Physical memory address.
Pos	ptr_t	The position of the target page table entry in this page directory to be filled. This page table entry must be empty.

Prototype	ret_t _RME_Pgt_Boot_Add(struct RME_Cap_Cpt* Cpt, cid_t Cap_Pgt, ptr_t Paddr, ptr_t Pos, ptr_t Flag)	
Parameter name	Type	Description
Flag	ptr_t	The attributes of the page table entry. These attributes limit the characteristics of the page table entry.

The possible return values of this function are as follows:

Table 7-50 Possible Return Values for Adding Pages to Page Directories at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Pgt is frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page directory capability.
RME_ERR_PGT_ADDR	Pos exceeds the number of page table entries in the page directory.
RME_ERR_PGT_ADDR	When the physical-address-equals-virtual-address check is enabled, the mapped physical address differs from the target virtual address.
RME_ERR_PGT_MAP	The attempt to map failed for hardware reasons. The specific reason for the failure is hardware-dependent.

At boot time, this function is called as many times as physical memory pages need to be added. In a typical boot process, only one processor needs to perform this function, so only one group of calls to this function is made in the entire system. At the same time, during this process it is not required that each physical memory page be mapped only once. If a page is mapped multiple times, these multiple mappings will exist simultaneously, and all of them are legal.

7.9.2.8 Creating the First Process at Boot-time

This function is used to create the first process at boot time and to place the capability pointing to the process into the specified capability table. This function does not require a kernel memory capability.

Table 7-51 Parameters Needed for Creating the First Process at Boot-time

Prototype	ret_t _RME_Prc_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt_Crt, cid_t Cap_Prc, cid_t Cap_Cpt, cid_t Cap_Pgt, ptr_t Vaddr)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Cap_Cpt_Crt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created process capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Prc	cid_t	A capability number corresponding to a location in the capability table that is to accept this newly created process capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_PRC_CRT attribute; the capability corresponding to this capability number points to the capability table to be used by the newly created process. This capability number may be a one-level or two-level lookup encoding.
Cap_Pgt	cid_t	A capability number corresponding to a page table capability that must have the RME_PGT_FLAG_PRC_CRT attribute; the capability corresponding to this capability number points to the page table to be used by the newly created process[42]. This capability number may be a one-level or two-level lookup encoding.

[42] Top-level page directory.

Prototype	ret_t _RME_Prc_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt_Crt, cid_t Cap_Prc, cid_t Cap_Cpt, cid_t Cap_Pgt, ptr_t Vaddr)	
Parameter name	Type	Description
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the newly created process kernel object.

The possible return values of this function are as follows:

Table 7-52 Possible Return Values for Creating the First Process at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt_Crt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt_Crt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_FROZEN	Cap_Prc is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt_Crt or Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page table capability.
RME_ERR_CPT_FLAG	Cap_Cpt_Crt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_PRC_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Pgt does not have the RME_PGT_FLAG_PRC_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Prc is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_CPT_REFCNT (unlikely to return this value)	The reference count of Cap_Cpt or Cap_Pgt exceeds the maximum range allowed by the system.

This function only needs to be called once by one core during the entire system boot.

7.9.2.9 Creating Kernel Function Capability at Boot-time

This function is used to create a kernel function call capability at system boot time and to place this kernel function call capability into the specified capability table. The kernel function call capability can only be

created during kernel boot; thereafter, all newly generated kernel function call capabilities are obtained by passing down this capability.

Table 7-53 Parameters Needed for Creating Kernel Function Capability at Boot-time

Prototype	ret_t _RME_Kfn_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Kfn)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created kernel function call capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Kfn	cid_t	A capability number corresponding to a location in the capability table that is to accept this newly created kernel function call capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.

The possible return values of this function are as follows:

Table 7-54 Possible Return Values for Creating Kernel Function Capability at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Kfn is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN (unlikely to return this value)	Cap_Kfn is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Kfn is not an empty capability.

This function only needs to be called once by one core during the entire system boot.

7.9.2.10 Creating Kernel Memory Capability at Boot-time

This function is used to create a kernel memory capability at system boot time and to place this kernel memory capability into the specified capability table. The kernel memory capability can only be created during kernel boot; thereafter, all newly generated kernel memory capabilities are obtained by passing down this capability.

Table 7-55 Parameters Needed for Creating Kernel Memory Capability at Boot-time

Prototype	ret_t _RME_Kom_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Kom, ptr_t Begin, ptr_t End, ptr_t Flags)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created memory capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	A capability number corresponding to a location in the capability table that is to accept this newly created kernel memory capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Begin	ptr_t	The starting virtual address of the kernel memory. When the address is passed in, it is automatically aligned to the granularity of the kernel memory allocation table.
End	ptr_t	The end virtual address of the kernel memory. When the address is passed in, it is automatically aligned to the granularity of the kernel memory allocation table minus 1.
Flags	ptr_t	The flag bits of this kernel memory capability, indicating which kernel objects are allowed to be created in this range of memory. This value cannot be 0; otherwise the kernel will crash directly.

The possible return values of this function are as follows:

Table 7-56 Possible Return Values for Creating Kernel Memory Capability at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN (unlikely to return this value)	Cap_Kom is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Kom is not an empty capability.

Generally speaking, this function only needs to be called once by one core during the entire system boot. If the available kernel memory of the system is divided into many segments, or different segments have different properties, then multiple kernel memory capabilities may be created.

7.9.2.11 Creating Kernel Signal Endpoints at Boot-time

This function is used to create kernel signal endpoints at boot time. Kernel signal endpoints are used to handle interrupts: in the interrupt vector, a signal is sent to the kernel signal endpoint so as to wake up the corresponding user-mode thread to perform interrupt handling. Since any one kernel signal endpoint can have only one thread blocked on it at any moment, the number of kernel signal endpoints that need to be created is the number of "interrupt vector-handling thread" pairs.

Kernel signal endpoints can only be created during kernel boot, and cannot be deleted. All kernel signal endpoints newly generated thereafter are obtained by passing down this capability. This function does not require a kernel memory capability.

Table 7-57 Parameters Needed for Creating Kernel Signal Endpoints at Boot-time

Prototype	ret_t _RME_Sig_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Sig, ptr_t Vaddr)	
Parameter name	Type	Description
CPT	...	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.

Prototype	ret_t _RME_Sig_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Sig, ptr_t Vaddr)	
Parameter name	Type	Description
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created kernel signal endpoint capability. This capability number may be a one-level or two-level lookup encoding.
Cap_Sig	cid_t	A capability number corresponding to a location in the capability table that is to receive this newly created kernel signal endpoint capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the newly created kernel signal endpoint kernel object.

The possible return values of this function are as follows:

Table 7-58 Possible Return Values for Creating Kernel Signal Endpoints at Boot-time

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Sig is out of range.
RME_ERR_CPT_FROZEN (unlikely to return this value)	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN (unlikely to return this value)	Cap_Sig is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_EXIST	Cap_Sig is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.

There are two modes for calling this function. In the first mode, a single processor core creates all the kernel signal endpoints. In the second mode, each processor separately creates the kernel signal endpoints that it needs to use. The second mode is faster, but the program is relatively more complex. Usually, using the first method is sufficient.

In a multi-core system, if the first mode is used, then after this step is completed, an IPI needs to be sent to the other processors, so that the other processors each execute the following functions and complete their own initialization. If the second mode is used, then the processors must be separated before this step is executed. All the steps after this step need to be executed once separately by each processor.

At least two signal endpoints must be created for each core: the first signal endpoint is used to receive the timer signal, and the pointer to it should be assigned to the Tick_Sig member of that CPU's local storage; the second endpoint is used to receive all other peripheral interrupts by default, and the pointer to it should be assigned to the Int_Sig member of that CPU's local storage.

7.9.2.12 Creating Initial Threads at Boot-time

This function is used to create the initial thread, that is, the Init thread, at system boot time. Once created, the Init thread is bound to a processor and possesses an infinite timeslice. In the Init process, each processor core has one Init thread. The Init thread cannot be killed, cannot be unbound from its processor, and cannot be blocked on any signal endpoint, but its priority can be changed. The priority ceiling of the Init thread is specified by the system as RME_PREEMPT_PRIO_NUM-1. The TID of the Init thread is always 0.

Unlike the system call that creates threads, this function allows the processor to which the thread is bound to be specified through the CPU_Local parameter. This is to make it convenient for the boot processor core to create all the kernel objects in a multiprocessor system, and this is also the recommended practice. This function does not require a kernel memory capability.

Table 7-59 Parameters Needed for Creating Initial Threads at Boot-time

Prototype	ret_t _RME_Thd_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Vaddr, ptr_t Prio, struct RME_CPU_Local* CPU_Local)	
Parameter name	Type	Description
CPT	…	The type is struct RME_Cap_Cpt*, which is a pointer to the parent capability table capability. All capability numbers are with respect to this capability table.
Cap_Cpt	cid_t	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created initial thread capability. This capability number may be a one-level or two-level lookup encoding.

Prototype	ret_t _RME_Thd_Boot_Crt(struct RME_Cap_Cpt* Cpt, cid_t Cap_Cpt, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Vaddr, ptr_t Prio, struct RME_CPU_Local* CPU_Local)	
Parameter name	Type	Description
Cap_Thd	cid_t	A capability number corresponding to a location in the capability table that is to receive this newly created thread capability. The capability corresponding to this capability number must be empty. This capability number can only be a one-level lookup encoding.
Cap_Prc	cid_t	A capability number corresponding to a process capability that must have the RME_PRC_FLAG_THD attribute; the capability corresponding to this capability number points to the process that contains the newly created thread. This capability number may be a one-level or two-level lookup encoding.
Vaddr	ptr_t	The starting virtual address in kernel space to be used by the newly created initial thread kernel object.
Prio	ptr_t	The preemption priority of the initial thread. In RME, thread priorities start from 0, and a larger value means a higher priority. This value cannot exceed the maximum value allowed by the system.
CPU_Local	...	The type is struct RME_CPU_Local*, which is a pointer to the CPU local storage of the CPU to which this thread is to be bound.

The possible return values of this function are as follows:

Table 7-60 Possible Return Values for Creating Initial Threads at Boot-time

Return value	Meaning
Non-negative value	The operation succeeded; the thread identifier (TID) is returned.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Prc is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Prc has already been frozen.
RME_ERR_CPT_FROZEN	Cap_Thd is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Prc is not a process capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	Cap_Prc does not have the RME_PRC_FLAG_THD attribute.
RME_ERR_CPT_EXIST	Cap_Thd is not an empty capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_PTH_PRIO	The specified initial thread priority exceeds RME_PREEMPT_PRIO_NUM-1.

This function needs to be called once by each processor, creating on that processor an Init thread bound to itself. Of course, it is also possible for a single processor to create Init threads for all the other processors.

7.9.2.13 Setting the Current Page Table, Enabling Interrupts and Entering User Mode to Begin Execution

Each processor calls __RME_Pgt_Set, __RME_Int_Enable and __RME_User_Enter once in turn, and then enters user mode to begin execution. For information about __RME_Pgt_Set, please refer to7.13.1 ; for information about __RME_Int_Enable and __RME_User_Enter, please refer to7.4 .

7.10 Porting of Register Set Related Functions

In RME, there are the following 8 functions related to register sets. These 8 functions are all very short, and involve only the register context. The implementations of these functions are often related to architecture-related structures, and are also related to the way the user-mode library uses these registers.

7.10.1 Implementation of __RME_Svc_Param_Get

This function is used to extract the parameters of a system call from the register set.

Table 7-61 Implementation of __RME_Svc_Param_Get

Prototype	void __RME_Svc_Param_Get(struct RME_Reg_Struct* Reg, ptr_t* Svc, ptr_t* Cid, ptr_t* Param)
Meaning	Extracts the parameters of the system call and puts them into the respective return values.
Return value	None.
Parameter	struct RME_Reg_Struct* Reg A pointer to the register set. ptr_t* Svc This parameter is used for output; it outputs the half-word-length system call number (N). ptr_t* Cid This parameter is used for output; it outputs the half-word-length capability number (C). ptr_t* Param This parameter is used for output; it outputs the three-word-length parameters (P1-P3).

7.10.2 Implementation of __RME_Svc_Retval_Set

This function is used to store the return value of a system call into the register set.

Table 7-62 Implementation of __RME_Svc_Retval_Set

Prototype	void __RME_Svc_Retval_Set(struct RME_Reg_Struct* Reg, ret_t Retval)
Meaning	Stores the return value of the system call into the register set.
Return value	None.
Parameter	struct RME_Reg_Struct* Reg This parameter is used for output; it is a pointer to the register set. ret_t Retval The return value returned by the system call.

7.10.3 Implementation of __RME_Thd_Reg_Init

This function is used to initialize the register set of a thread or of an invocation. It is called both when thread execution attributes are set and when an invocation is made.

Table 7-63 Implementation of __RME_Thd_Reg_Init

Prototype	void __RME_Thd_Reg_Init(ptr_t Attr, ptr_t Entry, ptr_t Stack, ptr_t Param, struct RME_Reg_Struct* Reg)
Meaning	Initializes the register set of a thread or of an invocation using the entry address, the stack address and the parameter.
Return value	None.

Prototype	void __RME_Thd_Reg_Init(ptr_t Attr, ptr_t Entry, ptr_t Stack, ptr_t Param, struct RME_Reg_Struct* Reg)
Parameter	ptr_t Attr The context attribute of the coprocessor. The specific meaning of this value is determined by the architecture. ptr_t Entry The entry address of the thread. ptr_t Stack The address of the thread stack. ptr_t Param The parameter to be assigned to the thread. struct RME_Reg_Struct* Reg This parameter is used for output; it is a pointer to the register set structure of this thread.

7.10.4 Implementation of __RME_Thd_Reg_Copy

This function is used to copy the register set of a thread. Sometimes this function needs to be implemented in assembly to improve efficiency.

Table 7-64 Implementation of __RME_Thd_Reg_Copy

Prototype	void __RME_Thd_Reg_Copy(struct RME_Reg_Struct* Dst, struct RME_Reg_Struct* Src)
Meaning	Copies one register set data structure to another.
Return value	None.
Parameter	struct RME_Reg_Struct* Dst This parameter is used for output; it is a pointer to the destination register set data structure. struct RME_Reg_Struct* Reg A pointer to the source register set data structure.

7.10.5 Implementation of __RME_Thd_Reg_Print

This function is used to print the register set of a thread. This function is only used for debug printing, and is called only when RME_DBGLOG_ENABLE is defined as 1. If this feature is not needed, it can be implemented as an empty function.

Table 7-65 Implementation of __RME_Thd_Reg_Print

Prototype	void __RME_Thd_Reg_Print(struct RME_Reg_Struct* Reg)
Meaning	Prints the register set of the thread.
Return value	None.
Parameter	struct RME_Reg_Struct* Reg A pointer to the register set data structure to be printed.

7.10.6 Implementation of __RME_Inv_Reg_Save

This function is used to save the necessary registers into the synchronous invocation structure to facilitate returning. It is sufficient to save only those registers that are necessary for recovering the program's execution flow[43].

Table 7-66 Implementation of __RME_Inv_Reg_Save

Prototype	void __RME_Inv_Reg_Save(struct RME_Iret_Struct* Ret, struct RME_Reg_Struct* Reg)
Meaning	Saves the necessary registers into the synchronous invocation structure.
Return value	None.
Parameter	struct RME_Iret_Struct* Ret This parameter is used for output; it is a pointer to the necessary register structure. struct RME_Reg_Struct* Reg A pointer to the register set.

7.10.7 Implementation of __RME_Inv_Reg_Restore

This function is used to restore the necessary registers from the synchronous invocation structure in order to return. It is sufficient to restore only those registers that are necessary for recovering the program's execution flow[44].

Table 7-67 Implementation of __RME_Inv_Reg_Restore

Prototype	void __RME_Inv_Reg_Restore(struct RME_Reg_Struct* Reg, struct RME_Iret_Struct* Ret)
Meaning	Restores the necessary registers from the synchronous invocation structure.
Return value	None.
Parameter	struct RME_Reg_Struct* Reg This parameter is used for output; it is a pointer to the register set. struct RME_Iret_Struct* Ret A pointer to the necessary register structure.

7.10.8 Implementation of __RME_Set_Inv_Retval

This function is used to store the return value of a synchronous invocation into the register set. It is also used when the scheduler event of a thread is received, to return the reason for the thread's error.

Table 7-68 Implementation of __RME_Set_Inv_Retval

Prototype	void __RME_Set_Inv_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
Meaning	Stores the return value of the synchronous invocation into the register set.
Return value	None.

[43] for example, for x86-64, SP and IP need to be saved; for Cortex-M, LR and SP need to be saved
[44] for example, for x86-64, SP and IP need to be restored; for Cortex-M, LR and SP need to be restored

Prototype	void __RME_Set_Inv_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
Parameter	struct RME_Reg_Struct* Reg This parameter is used for output; it is a pointer to the register set. ret_t Retval The return value returned by the synchronous invocation.

7.11 Porting of Coprocessor Register Set Related Functions

In RME, there are the following 4 functions related to coprocessors. These 4 functions are all very short, and involve only the coprocessor context. If coprocessors are not used, these functions need not be implemented.

7.11.1 Implementation of __RME_Thd_Cop_Check

This function is used when binding a thread to a processor to check whether the coprocessors used by the thread can be supported by the current processor. On certain asymmetric processors, some CPUs may lack the instruction set extensions required by certain threads, which makes it impossible for them to bind these threads; in this case, it suffices for this function to detect and reject such a situation during the check.

Table 7-69 Implementation of __RME_Thd_Cop_Check

Prototype	ret_t __RME_Thd_Cop_Check(ptr_t Attr)
Meaning	Checks whether the current CPU supports this coprocessor attribute.
Return value	ret_t If the current CPU is compatible with this context attribute, 0 is returned; otherwise RME_ERR_HAL_FAIL (-1) is returned.
Parameter	ptr_t Attr The context attribute of the coprocessor. The specific meaning of this value is determined by the architecture.

7.11.2 Implementation of __RME_Thd_Cop_Size

This function is used to query the context size of a certain coprocessor extension. When allocating a thread kernel object, in order to avoid wasting space, the size we allocate is always just enough to hold the context space that all coprocessors need to use; therefore, the space used by the coprocessor context needs to be queried in advance.

Table 7-70 Implementation of __RME_Thd_Cop_Size

Prototype	ptr_t __RME_Thd_Cop_Size(ptr_t Attr)
Meaning	Checks whether the current CPU supports this coprocessor attribute.
Return value	ptr_t The extra coprocessor context size corresponding to this coprocessor context attribute.

Prototype	ptr_t __RME_Thd_Cop_Size(ptr_t Attr)
Parameter	ptr_t Attr The context attribute of the coprocessor. The specific meaning of this value is determined by the architecture.

7.11.3 Implementation of __RME_Thd_Cop_Init

This function is used to initialize the coprocessor register set of a thread when setting the thread's execution attributes.

Table 7-71 Implementation of __RME_Thd_Cop_Init

Prototype	void __RME_Thd_Cop_Init(ptr_t Attr, struct RME_Reg_Struct* Reg, void* Cop)
Meaning	Initializes the thread's coprocessor register set.
Return value	None.
Parameter	ptr_t Attr The context attribute of the coprocessor. The specific meaning of this value is determined by the architecture. struct RME_Reg_Struct* Reg A pointer to the register set data structure. This parameter is used to assist in initializing the coprocessor registers. void* Cop This parameter is used for output; it is a pointer to the coprocessor register set.

7.11.4 Implementation of __RME_Thd_Cop_Swap

This function is used to save and restore the coprocessor register set of a thread as needed. Sometimes this function needs to be implemented in assembly to improve efficiency.

Table 7-72 Implementation of __RME_Thd_Cop_Swap

Prototype	void __RME_Thd_Cop_Swap(ptr_t Attr_New, ptr_t Is_Hyp_New, struct RME_Reg_Struct* Reg_New, void* Cop_New, ptr_t Attr_Cur, ptr_t Is_Hyp_Cur, struct RME_Reg_Struct* Reg_Cur, void* Cop_Cur)
Meaning	Saves the coprocessor register set of a thread.
Return value	None.
Parameters	ptr_t Attr_New The coprocessor context attributes of the new thread. The specific meaning of this value is determined by the architecture.

Prototype	void __RME_Thd_Cop_Swap(ptr_t Attr_New, ptr_t Is_Hyp_New, struct RME_Reg_Struct* Reg_New, void* Cop_New, ptr_t Attr_Cur, ptr_t Is_Hyp_Cur, struct RME_Reg_Struct* Reg_Cur, void* Cop_Cur)
	ptr_t Is_Hyp_New A flag bit indicating whether the old thread is dedicated to para-virtualization. If it is not 0, the old thread is dedicated to para-virtualization. struct RME_Reg_Struct* Reg_New A pointer to the register set data structure of the new thread. This parameter is used to help determine whether the new thread uses a coprocessor. void* Cop_New A pointer to the coprocessor register set of the new thread. The coprocessor context should be restored from this pointer. ptr_t Attr_Cur The coprocessor context attributes of the old thread. The specific meaning of this value is determined by the architecture. ptr_t Is_Hyp_Cur A flag bit indicating whether the old thread is dedicated to para-virtualization. If it is not 0, the old thread is dedicated to para-virtualization. struct RME_Reg_Struct* Reg_Cur A pointer to the register set data structure of the old thread. This parameter is used to help determine whether the old thread uses a coprocessor. void* Cop_Cur A pointer to the coprocessor register set of the old thread. The coprocessor context should be saved to this pointer.

It should be noted in particular that the coprocessor register set can sometimes be used as a covert channel with a very strong transmission capability. Therefore, in those implementations that attach importance to information security, if it is detected that the new thread does not use the coprocessor register set, the coprocessor register set should be filled with meaningless characters (such as 0x00) to prevent the information of the old thread from leaking to the new thread; alternatively, the coprocessor register set can always be saved and restored regardless of whether the thread uses the coprocessor register set.

7.12 Porting of Kernel Function Call Functions

Kernel function call functions are a set of functions implemented by the user that can run in the kernel mode of the operating system. These functions are determined at compile time. These functions are described as follows:

Table 7-73 Implementation of Kernel Function Call Functions

Prototype	ret_t __User_Func(struct RME_Cap_Cpt* Cpt, struct RME_Reg_Struct* Reg, ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)
Meaning	Implements a user-defined kernel-mode operation.
Return value	ret_t If it fails, it must return a negative value; if it succeeds, it must return a non-negative value. In addition, if this function succeeds, it is this function's responsibility to set its return value into the register set.
Parameters	struct RME_Cap_Cpt* Cpt A pointer to the capability table of the current process. struct RME_Reg_Struct* Reg This parameter can be used for input or output and is a pointer to the register set. ptr_t Func_ID The major function number. ptr_t Sub_ID The sub-function number. ptr_t Param1 The first parameter of this function. ptr_t Param2 The second parameter of this function.

This is a kernel-mode function that accepts two user-defined parameters, performs some operations, and then returns. Generally speaking, these functions are used to implement some processor-specific features[45]. The implementations of these functions should all be short and concise, and they should be guaranteed to complete within a certain time limit; otherwise, the real-time performance of calls to these functions cannot be guaranteed.

The implementation ideas of several of the most common kernel functions are introduced below for reference.

7.12.1 Implementation of the High-Precision Timer System

To use a high-precision timer, it is only necessary to implement a few kernel function calls and give them the ability to operate the timer. The interrupt generated by the timer can be delivered directly to the corresponding target thread through a kernel signal endpoint, or it can be handled by another management thread, which then delivers the timer interrupt to other threads.

7.12.2 Implementation of Inter-Processor Interrupts

Since in RME an asynchronous signal issued from one CPU cannot be delivered directly to another CPU so as to wake up a thread on it, a kernel function call is needed to implement inter-processor interrupts

[45] For example, some peripherals built into the CPU, special coprocessor instructions, or other I/O operations that must be implemented in kernel mode

and to indicate that a thread on another core needs to wake up some other thread.

7.12.3 Implementation of Cache Operations

Processor cache operations are generally privileged instructions as well. Therefore, these operations can be implemented separately as kernel function calls.

7.13 Porting of Page Table Related Functions

In RME, there are the following 11 functions related to page tables. The implementation of these functions is closely related to the processor architecture, and in a multi-core environment they are also responsible for checking for conflicts between parallel operations. The security and reliability of these functions greatly affect the security and reliability of the system, so they are the most important part of system porting. In what follows, we explain the functionality of these functions and the points to note when porting them one by one.

7.13.1 Implementation of __RME_Pgt_Set

This function is responsible for setting the page table currently used by the processor. What is passed into this function is a virtual address; in this function it is often necessary to first translate the virtual address into a physical address, and then assign the physical address to the corresponding register of the processor.

Table 7-74 Implementation of __RME_Pgt_Set

Prototype	void __RME_Pgt_Set(struct RME_Cap_Pgt* Pgt) void __RME_Pgt_Set(ptr_t Pgt)
Meaning	Sets the page table used by the processor.
Return value	None.
Parameters	struct RME_Cap_Pgt* Pgt The page directory root capability that points to the top-level page directory and contains all the information of this page directory. The macro RME_CAP_GETOBJ can be used to obtain the virtual address of the page directory, and this virtual address can be converted into a physical address by an architecture-dependent method and then loaded into the processor. In cases where additional information[46] is needed to assist the setting, the values stored in the capability can also be read. ptr_t Pgt Once the user-mode page table control macro RME_PGT_RAW_USER is enabled during porting, this parameter replaces Pgt. It is the physical address at which the page table data structure directly recognizable by the processor resides.

For the MMU architecture, this function makes the top-level page directory pointer register[47] point to the top-level page directory, and this operation also flushes the TLB cache at the same time. For the MPU architecture, this function copies the metadata of the top-level page table into the relevant MPU registers to complete the protection region setup. Since an MPU has a relatively large number of registers, it may be advisable to implement this function in assembly, thereby achieving the effect of setting the page table quickly.

[46] Such as the ASID, etc.
[47] Such as CR3 in x86-64

7.13.2 Implementation of __RME_Pgt_Kom_Init

This function is responsible for establishing the initial kernel page table at system startup.

Table 7-75 Implementation of __RME_Pgt_Kom_Init

Prototype	ret_t __RME_Pgt_Kom_Init(void)
Meaning	Establishes the initial kernel page table.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	None.

Once the kernel mappings established by this function are in place, they will not be changed by users, and these physical addresses will be used as kernel memory permanently for as long as the system exists. This kernel page table[48] will be mapped into the top-level page directory of every process, and its privilege attribute will be defined as the kernel level. The initial kernel page table should include two parts: one part is the virtual space occupied by the kernel, and the other part is the memory space used by the kernel virtual machine. As for when this kind of mapping is performed, please refer to the relevant part on __RME_Pgt_Init.

In an MPU environment, this function generally only needs to return success directly. This is because, generally speaking, a processor in privileged mode can access all memory space, and there is no need to add entries about kernel addresses to the page table and page table metadata.

7.13.3 Implementation of __RME_Pgt_Check

This function is responsible for checking whether the various parameters used to create a page directory can be supported by the underlying architecture.

Table 7-76 Implementation of __RME_Pgt_Check

Prototype	ret_t __RME_Pgt_Check(ptr_t Base, ptr_t Is_Top, ptr_t Size_Order, ptr_t Num_Order, ptr_t Vaddr)
Meaning	Checks whether the passed-in page directory creation parameters can be supported by the underlying hardware.
Return value	ret_t Returns 0 on success (supported by the hardware), or RME_ERR_HAL_FAIL (-1) on failure (not supported by the hardware).
Parameters	ptr_t Base The starting virtual address of the page directory mapping. This parameter is valid only in an MPU environment. ptr_t Is_Top Whether the page directory is a top-level page directory. 1 means top-level, and 0 means not top-level. ptr_t Size_Order The size order of the page table entries of the page directory. ptr_t Num_Order The number order of the page table entries of the page directory. ptr_t Vaddr The kernel virtual address at which the page directory kernel object itself resides.

[48] Or these kernel page directories

This function is called before the kernel call that creates the page table, to confirm that this kind of page table can be created, so that the validity of the page table parameters is checked before kernel memory is allocated. This function needs to be written strictly in accordance with the processor hardware's requirements for page tables, so that it returns 0 only for the page table forms supported by the processor, and returns RME_ERR_HAL_FAIL (-1) for all other combinations.

7.13.4 Implementation of __RME_Pgt_Init

This function is responsible for initializing a page directory that has just been created.

Table 7-77 Implementation of __RME_Pgt_Init

Prototype	ret_t __RME_Pgt_Init(struct RME_Cap_Pgt* Pgt_Op)
Meaning	Initializes the page directory that has just been created.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory.

This function is called when a page directory is created, and it is also the key function of page directory creation. This function must initialize the page directory into a usable form, for example, initializing all of its gaps as empty entries. In addition, if a top-level page directory is being created in an MMU environment, all the kernel entries created by __RME_Pgt_Kom_Init also need to be mapped into this page directory. Since the page directory parameters have already been checked by __RME_Pgt_Check beforehand, this function can skip these checks.

7.13.5 Implementation of __RME_Pgt_Del_Check

This function is responsible for checking whether a page directory can be safely deleted.

Table 7-78 Implementation of __RME_Pgt_Del_Check

Prototype	ret_t __RME_Pgt_Del_Check(struct RME_Cap_Pgt* Pgt_Op)
Meaning	Checks whether a page directory can be safely deleted.
Return value	ret_t Returns 0 on success (can be deleted), or RME_ERR_HAL_FAIL (-1) on failure (cannot be deleted).
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory.

This function is a mandatory check function in the page directory deletion operation. In this function, we need to check whether this page directory can be deleted. The kernel itself already guarantees that a page directory that has a parent page directory or a child page directory cannot be deleted, so this check is usually unnecessary. This check is needed only if some architectures impose requirements on the page mapping situation inside the page directory being deleted.

7.13.6 Implementation of __RME_Pgt_Page_Map

This function is responsible for mapping a page into a page directory. If this mapping cannot be completed because the passed-in parameters are incorrect[49], an error should be returned.

Table 7-79 Implementation of __RME_Pgt_Page_Map

Prototype	ret_t __RME_Pgt_Page_Map(struct RME_Cap_Pgt* Pgt_Op, ptr_t Paddr, ptr_t Pos, ptr_t Flag)
Meaning	Maps a page into the page directory.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory. ptr_t Paddr The physical page frame address that needs to be mapped. ptr_t Pos The position number of the page directory entry to which this page needs to be mapped. ptr_t Flag The RME standard page flags of this page.

In the above table, "page directory entry position number" refers to the slot number of the mapped page in the page directory. For example, if each entry of a page directory represents a page frame of 4kB, then 12kB is the starting point of its 3rd slot[50]. "RME standard page flags" are the abstract page flags used by the RME system, not those page flags used in a specific page table; please refer to the description in Chapter 3 for details. This function needs to convert these page flags into the page flags of page table entries that the processor can directly recognize, and then write them into the page table. For those processors that do not support some page flags, those unsupported page flags can simply be ignored. For example, for those MMU architectures whose hardware updates the TLB, the "static (RME_PGT_STATIC)" page flag may not be implemented.

In a multi-core environment, this function needs to guarantee that two CPUs will not simultaneously map two pages at one position at the same time. If this situation occurs, this function can use a read-modify-write[51] atomic operation to ensure that such a conflict does not occur in a multi-core environment. In an MPU environment, this function is also responsible for updating the top-level page table metadata of the MPU to add the mapping of this page.

7.13.7 Implementation of __RME_Pgt_Page_Unmap

This function is responsible for unmapping a page within a page directory. If this operation cannot be performed because the passed-in parameters are incorrect[52], an error should be returned.

[49] For example, the position number is out of range, the physical address alignment does not meet the requirement, there is an unsupported flag bit, or there is already a mapping at that position

[50] The slot number is counted from 0

[51] That is, compare-and-swap, CAS

[52] For example, the position number is out of range or there is no page at the position

Table 7-80 Implementation of __RME_Pgt_Page_Unmap

Prototype	ret_t __RME_Pgt_Page_Unmap(struct RME_Cap_Pgt* Pgt_Op, ptr_t Pos)
Meaning	Unmaps a page from a page directory.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory. ptr_t Pos The position number of the page directory entry whose mapping needs to be removed.

This function is the inverse operation of the above function; it is only necessary to remove the mapping of this page. In a multi-core environment, it is also necessary to guarantee that no conflict occurs when two CPUs attempt to remove the mapping at the same time. In an MMU environment, this function is also responsible for using a TLB flush instruction to flush the entire TLB cache, or it may use a single-entry TLB flush operation[53] when the position of the page mapping is known for certain. It is also acceptable not to flush the cache, or to flush the cache periodically with a special kernel function call; in this case, page mapping quiescence processing[54] needs to be performed in user mode. In an MPU environment, it is responsible for updating the top-level metadata of the MPU to remove the mapping of this page.

7.13.8 Implementation of __RME_Pgt_Pgdir_Map

This function is responsible for mapping a child page directory into a parent page directory. If this operation cannot be performed because the passed-in parameters are incorrect[55], an error should be returned.

Table 7-81 Implementation of __RME_Pgt_Pgdir_Map

Prototype	ret_t __RME_Pgt_Pgdir_Map(struct RME_Cap_Pgt* Pgt_Parent, ptr_t Pos, struct RME_Cap_Pgt* Pgt_Child, ptr_t Flag)
Meaning	Maps a child page directory into the parent page directory.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	struct RME_Cap_Pgt* Pgt_Parent The page directory capability that points to the parent page directory and contains all the information of the parent page directory. ptr_t Pos The position number of the parent page directory entry to which this child page directory needs to be mapped. struct RME_Cap_Pgt* Pgt_Child The page directory capability that points to the child page directory and contains all the information of the child page directory.

[53] Such as INVLTB in x86-64

[54] That is, a page whose mapping has been removed will take a while to actually disappear from the TLB

[55] For example, the position number is out of range, the physical address alignment does not meet the requirement, the relationship between the virtual addresses is incorrect, there is already a mapping at that position, the passed-in flag bits are incorrect, or certain special constraints are not satisfied in an MPU environment

Prototype	ret_t __RME_Pgt_Pgdir_Map(struct RME_Cap_Pgt* Pgt_Parent, ptr_t Pos, struct RME_Cap_Pgt* Pgt_Child, ptr_t Flag)
	ptr_t Flag The RME standard page flags of this child page directory. These page flags determine the access permission restrictions of the child page directory and of all page directories below it. For architectures that do not allow page directory attributes to be set, or for MPU-based architectures, this value is invalid.

This function often behaves differently under an MMU system and under an MPU system. Under an MMU system, this kind of mapping does not need to check whether the starting virtual address is compliant, but the address range contained in the child page directory must be exactly the size of one slot of the parent page directory. Under an MPU system, since a compressed page table can be used, the address range contained in the child page directory can be smaller than one slot of the parent page directory, but its starting virtual address must be guaranteed to be compliant.

Under an MPU system, due to certain inherent properties of the MPU[56], the parent page directory may be required to possess[57] a top-level page directory, and the child page directory is required not to be a top-level page directory itself and not to possess a top-level page directory. In addition, after the mapping is completed, if the child page directory contains pages that have been mapped, the MPU metadata contained at the top-level page directory needs to be updated to add the mappings of these pages.

7.13.9 Implementation of __RME_Pgt_Pgdir_Unmap

This function is responsible for unmapping a child page directory within a parent page directory. If this operation cannot be performed because the passed-in parameters are incorrect[58], for example, if that position of the parent page directory is not this child page directory, an error should be returned.

Table 7-82 Implementation of __RME_Pgt_Pgdir_Unmap

Prototype	ret_t __RME_Pgt_Pgdir_Unmap(struct RME_Cap_Pgt* Pgt_Parent, ptr_t Pos, struct RME_Cap_Pgt* Pgt_Child)
Meaning	Unmaps a child page directory from the parent page directory.
Return value	ret_t Returns 0 on success, or RME_ERR_HAL_FAIL (-1) on failure.
Parameters	struct RME_Cap_Pgt* Pgt_Parent The page directory capability that points to the parent page directory and contains all the information of the parent page directory. ptr_t Pos The position number of the child page directory entry whose mapping needs to be removed. struct RME_Cap_Pgt* Pgt_Child The page directory capability that points to the child page directory and contains all the information of the child page directory.

This function is the inverse operation of the above function; it is only necessary to remove the mapping

[56] Please refer to Section 3.2.5

[57] Or be itself

[58] For example, the position number is out of range, there is no child page directory at the position, or certain special constraints are not satisfied in an MPU environment

of the child page directory. In a multi-core environment, a compare-and-swap atomic operation is also needed to guarantee that no conflict occurs when two CPUs attempt to remove the mapping at the same time. In an MMU environment, this function is also responsible for using a TLB flush instruction to flush the entire TLB cache. It is also acceptable not to flush the cache; the page mapping quiescence processing mentioned in 7.13.7 can also be used. In an MPU environment, it may be responsible for updating the top-level metadata of the MPU to remove the mappings of the pages contained in the child page directory.

7.13.10 Implementation of __RME_Pgt_Lookup

This function is responsible for looking up the information of a certain page within a page directory.

Table 7-83 Implementation of __RME_Pgt_Lookup

Prototype	ret_t __RME_Pgt_Lookup(struct RME_Cap_Pgt* Pgt_Op, ptr_t Pos, ptr_t* Paddr, ptr_t* Flag)
Meaning	Looks up the information of the page at a certain position in a page directory and returns it.
Return value	ret_t On success[59] it returns 0, and on failure[60] it returns RME_ERR_HAL_FAIL (-1).
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory. ptr_t Pos The position number of the page directory entry to be looked up. ptr_t* Paddr This parameter is used for output and is a pointer to the physical page frame address of this page. ptr_t* Flag This parameter is used for output and is a pointer to the RME standard page flags of this page.

This function only needs to look up the corresponding information on this page and output it[61]. As for the page flags, care should be taken to convert the processor-recognizable page flags into RME standard page flags before outputting them. In addition, both output parameters should be implemented as optional: when only one of them needs to be looked up, it is sufficient to pass 0 (NULL) for the other parameter, in which case only that one kind of information is queried.

7.13.11 Implementation of __RME_Pgt_Walk

This function is responsible for looking up whether a virtual address in the entire page table[62] is mapped, as well as its information. This function should only accept page table lookups initiated from the top-level page directory; any attempt to start a page table lookup from another page directory should return an error.

[59] The page is found
[60] The page is not found, or the entry mapped at that position is a page directory
[61] Write it into the variable pointed to by the pointer
[62] Page directory tree

Table 7-84 Implementation of __RME_Pgt_Walk

Prototype	ret_t __RME_Pgt_Walk(struct RME_Cap_Pgt* Pgt_Op, ptr_t Vaddr, ptr_t* Pgt, ptr_t* Map_Vaddr, ptr_t* Paddr, ptr_t* Size_Order, ptr_t* Num_Order, ptr_t* Flag)
Meaning	Looks up whether a virtual address in the page table[63] is mapped, as well as its information, and returns it.
Return value	ret_t On success[64] it returns 0, and on failure[65] it returns RME_ERR_HAL_FAIL (-1).
Parameters	struct RME_Cap_Pgt* Pgt_Op The page directory capability that points to this page directory and contains all the information of this page directory. This page directory must be top-level. ptr_t Vaddr The virtual address to be queried. ptr_t* Pgt This parameter is used for output and is a pointer to the storage virtual address of the page directory kernel object in which this virtual address resides. ptr_t* Map_Vaddr This parameter is used for output and is a pointer to the mapping start virtual address of the page frame in which this virtual address resides. ptr_t* Paddr This parameter is used for output and is a pointer to the mapping start physical address of the page frame in which this virtual address resides. ptr_t* Size_Order This parameter is used for output and is a pointer to the size order of the page directory in which this virtual address resides. ptr_t* Num_Order This parameter is used for output and is a pointer to the number order of the page directory in which this virtual address resides. ptr_t* Flag This parameter is used for output and is a pointer to the RME standard page flags of the page to which this virtual address belongs.

This function needs to look up the (possibly compressed) page table tree according to the passed-in virtual address, determine whether the passed-in virtual address is in this page table, and if it exists, also determine the virtual address of the page directory kernel object itself in which it resides and which slot of this page directory it is in. This function is only allowed to look up user pages; looking up the information of memory pages is not allowed, and an error must be returned.

As for the page flags, care should be taken to convert the processor-recognizable page flags into RME standard page flags before outputting them. For those architectures that allow access control flags to be set on page directories, the flag bits of page directories at all levels should also be taken into account. In addition, all six output parameters should be implemented as optional: when only a few of them need to be looked up, it is sufficient to pass 0 for each of the other parameters, in which case only those kinds of information are queried.

[63] Page directory tree
[64] The page is found
[65] The page is not found

7.14 Other Function Descriptions

In the process of writing low-level drivers and debugging kernel code, there are several commonly used helper functions available. The kernel provides these functions, so that decoupling from the compiler's own C runtime library is achieved as far as possible. The definitions of these functions are all located in kernel.h; simply include kernel.h when they need to be used. The list of these functions is as follows:

7.14.1 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 7-85 Parameters Needed for Variable Clearing

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

7.14.2 Comparing Two Memory Blocks

This function is used to compare whether two memory blocks are identical. This function is essentially equivalent to the memcmp of the C language runtime library.

Table 7-86 Parameters Needed for Comparing Two Memory Blocks

Prototype	ret_t _RME_Memcmp(const void* Ptr1, const void* Ptr2, ptr_t Num)	
Parameter name	Type	Description
Ptr1	const void*	A pointer to the first memory block involved in the comparison.
Ptr2	const void*	A pointer to the second memory block involved in the comparison.
Num	ptr_t	The length of the memory to be compared, in bytes.

If the two memory blocks are completely identical within the specified length range, 0 is returned; if they are not identical, a non-zero value is returned.

7.14.3 Copying a Memory Block

This function is used to copy the contents of one memory block to another region. This function is essentially equivalent to the memcpy of the C language runtime library. The two memory regions must not overlap; otherwise, the behavior of this function is undefined.

Table 7-87 Parameters Needed for Copying a Memory Block

Prototype	void _RME_Memcpy(void* Dst, void* Src, ptr_t Num)	
Parameter name	Type	Description
Dst	void*	The destination address of the copy.
Src	void*	The source address of the copy.
Num	ptr_t	The length of the memory to be copied, in bytes.

It should be noted that the three functions listed in 7.14.1 -7.14.3 are all byte-by-byte implementations of their functionality and take no optimization into account, so do not use them in large memory operations. This is for maximum compiler and architecture compatibility (some architectures have special requirements such as alignment for word operations; alternatively, special instructions are required for efficient operation; alternatively, their compiler's built-in fast implementation may use FPU registers. These three situations must be avoided as much as possible in the kernel). The architecture-independent part of RME does not use any of these three functions; large memory operations should also be avoided as much as possible in the hardware abstraction layer. If large memory operations must be used, then you may consider writing them yourself, or use the versions provided by the compiler. In any case, the operations used must definitely not use FPU registers or cause memory access misalignment errors; this point deserves extra attention when using library functions provided by the compiler.

7.14.4 Printing a Signed Integer

This function is used to print a signed integer in decimal (base 10), mainly for kernel debugging. Printing is blocking; the function returns only after printing is complete. The printing includes the sign bit. This function has a macro with the same functionality, RME_DBG_I, which is controlled by a debug switch.

Table 7-88 Parameters Needed for Printing a Signed Integer

Prototype	cnt_t RME_Int_Print(cnt_t Int)	
Parameter name	Type	Description
Int	cnt_t	The signed integer to be printed.

The return value of this function is the length of the successfully printed string.

7.14.5 Printing an Unsigned Integer

This function is used to print an unsigned integer in hexadecimal (base 16), mainly for kernel debugging. Printing is blocking; the function returns only after printing is complete. The printing does not include the "0x" prefix, and the A-F in hexadecimal are all uppercase. This function has a macro with the same functionality, RME_DBG_H, which is controlled by a debug switch.

Table 7-89 Parameters Needed for an Unsigned Integer

Prototype	cnt_t RME_Hex_Print(ptr_t Uint)	
Parameter name	Type	Description
Uint	ptr_t	The unsigned integer to be printed.

The return value of this function is the length of the successfully printed string.

7.14.6 Printing a String

This function is used to print a string, mainly for kernel debugging. Printing is blocking; the function returns only after printing is complete. This function has a macro with the same functionality, RME_DBG_S, which is controlled by a debug switch.

Table 7-90 Parameters Needed for Printing a String

Prototype	cnt_t RME_Str_Print(const s8_t* String)	
Parameter name	Type	Description
String	const s8_t*	The string to be printed.

The return value of this function is the length of the successfully printed string. This length does not include the "\0" terminator of the string.

7.15 Bibliography of This Chapter

None

8. Kernel Call Implementation Specifications

8.1 Overview of Kernel Call Implementation Specifications

The low-level functions of the processor are implemented in RME using kernel calls. These low-level functions are often highly diverse, and RME's kernel calls must support them all. If each architecture adopted its own implementation scheme, the interface semantics of the various ports would inevitably become mutually incompatible, which would greatly increase the maintenance workload and lead to fragmentation of the ecosystem. To avoid this phenomenon, RME specifies implementation methods for a series of commonly used kernel calls. For any kernel call within the scope described in this chapter, its implementation interface must be consistent with what is described in this chapter. For those additional functions that are not specified in this chapter but are provided by the processor or the system, their implementation is free, but it is recommended to maintain a style consistent with this chapter. In the following subsections, we will introduce each category of kernel calls and their implementation specifications respectively.

An RME kernel call can carry a half-word-length main function number, a half-word-length subfunction number, and two word-length parameters. However, RME does not prohibit the implementer from using the processor's register set to pass additional parameters. In some extension standards, passing additional parameters is allowed, and outputting additional parameters is also allowed. These permitted exceptions will be described in the specific chapters. Unless otherwise specified, the system call number N of all kernel calls is RME_SVC_KFN, all of them require passing in a valid[1] kernel call capability as the parameter at their C position, and the type of the returned parameter is always ret_t. In the following chapters, we omit the error return values related to the kernel call capability that have already been listed in Chapter 6, and do not describe them.

Some kernel calls require certain additional capabilities to be passed in[2]. In this case, we do not perform operation flag attribute checks on these capabilities and assume by default that operations can be performed.

[1] The permitted operation range includes the passed-in main function number

[2] Such as the page directory capability and the thread capability, etc.

8.2 Page Table Kernel Call Implementation Specifications (“P” Standard Extension)

The page table is the only reference for an operating system’s memory management, and is also one of the most important data structures in kernel porting. RME is designed to support almost all memory access control methods, including three categories: memory protection units, software-filled memory management units, and hardware-filled memory management units. In addition, page table management must also support multi-core operation, as well as the flushing of translation look-aside buffers or register caches. Therefore, the implementation of page-table-related functions is a key point and also a difficult point in kernel porting. RME specifies the following five page-table-related kernel calls in total:

Table 8-1 Overview of Page Table Kernel Calls

Main function number	Call name	Purpose
0xF000	RME_KFN_PGT_CACHE_CLR	Clear all cache of a certain page table on the current CPU.
0xF001	RME_KFN_PGT_LINE_CLR	Clear a certain line of the cache of a certain page table on the current CPU.
0xF002	RME_KFN_PGT_ASID_SET	Set the ASID of a certain page table.
0xF003	RME_KFN_PGT_TLB_LOCK	Lock a certain page in the TLB.
0xF004	RME_KFN_PGT_ENTRY_MOD	Query or modify the attributes of a certain page.

8.2.1 Four Recommended Forms of Page Tables

In RME, depending on the underlying hardware, the page table design strategy adopted may also differ. We divide all memory management hardware into four categories, as described in the table below:

Table 8-2 Overview of the Four Recommended Forms of Page Tables

Category	Path compression	Cache update	Cache	Construction/ destruction restriction
Memory protection unit in a single-core system	Yes	Updated on change	Metadata	No sub-page-directories, or even no pages
Memory protection unit in a multi-core system	Yes	Updated on page fault	Metadata	No restriction
Software-filled memory management unit	No	Updated on page fault	Metadata	No restriction
Hardware-filled memory management unit	No	Updated by hardware	TLB	No restriction

8.2.1.1 Memory Protection Unit in a Single-Core System

The implementation of this type of page table is characterized by supporting path compression[3], and the cache consists of metadata stored at the top-level page directory. Each level of page directory contains a pointer to the top-level page directory, and when mapping, unmapping, constructing, and destructing, changes are directly synchronized to the metadata contained inside the top-level page directory; therefore, the data inside the memory protection unit registers is always consistent with the actual content of the page table, and no additional cache maintenance work is required. Because each level of page directory can record only one pointer to the top-level page directory, this implementation does not allow page directories to be shared between page tables. Because cache metadata must be maintained during construction and destruction, this implementation requires, at both construction and destruction time, that there be no second-level sub-page-directories in the sub-page-directories[4], or even that they contain no pages[5]. Because multiple page tables are not allowed to share a page directory, there is also no need for this type of page table to support page directory attributes.

This implementation allows the number of static pages that can be mapped to be no more than "the number of regions available to the memory protection unit minus 1"[6], while no limit is imposed on the number of dynamic pages. No page fault interrupts occur on static pages, and on dynamic pages interrupts occur only during replacement. The implementation of the dynamic page replacement algorithm is decided by the porter; RME recommends using a pseudo-random replacement algorithm.

The page table of ARMv7-M is a typical example of this type of implementation. This type of implementation requires no page table kernel calls at all, because changes are always consistent with the cache metadata state. This enables this implementation to achieve very good real-time performance. In those situations where only some memory accesses can produce a precise faulting access address, this imple-

[3] Therefore, the validity of the start address needs to be checked at construction time
[4] When a sub-page-directory corresponds to one region, such as ARMv7-M
[5] When a sub-page-directory may correspond to multiple regions, such as ARMv8-M
[6] Because at least one region must be reserved for dynamic page replacement

mentation approach must also be adopted, because not all pages can be filled in when a page fault occurs[7].

8.2.1.2 Memory Protection Unit in a Multi-Core System

The implementation of this type of page table is characterized by supporting path compression[8], and the cache consists of metadata stored at the top-level page directory. The cache of each CPU is independent, and they are updated independently when a page fault interrupt occurs. This implementation allows the caches of different CPUs to contain different entries, which is conducive to sharing the work among the CPUs, and also does not cause mutual interference among the CPUs. When mapping, unmapping, constructing, and destructing, changes are not synchronized into the cache metadata. The addition and replacement of cache metadata entries are performed when a page fault interrupt occurs; the clearing of cache metadata, however, needs to be performed using kernel calls. Once unmapping or destruction has been performed, the corresponding operation must be invoked on all cores to reset the cache related to that page directory. If several page tables share that page directory, then the caches of these page tables must be reset on all cores.

Compared with the structure described in Section 8.2.1.1, although this design suffers a certain loss in real-time performance, page directories can be shared between page tables, and there is no restriction at all during construction and destruction. The loss in real-time performance can also be compensated for by the following method: once a static page has been updated into the cache metadata, it will not disappear until the number of static pages reaches the upper limit and there is a new static page that needs to be mapped; dynamic pages, however, can only replace dynamic pages and will never replace static pages. Thus, as long as the number of static pages accessed simultaneously by one CPU does not exceed the upper limit on static pages[9], subsequent accesses to static pages are all real-time except for the first access. The page replacement algorithm is also decided by the porter.

Whether this type of implementation supports page directory attributes is decided by the porter's implementation. If the user decides to implement page directory attributes, then the attribute value (Flags) returned by the __RME_Pgt_Walk function must be a common subset of the attribute values of the page directories at all levels, and the __RME_Pgt_Pgdir_Map function must also support page directory attributes. EDI does not recommend implementing page directory attribute support, because generally it is rarely used and is unnecessary.

The page table of TriCore is a typical example of this type of implementation. The real-time performance of this type of implementation is also relatively good, second only to the implementation described in Section 8.2.1.1. In this type of page table implementation, the kernel function calls that can be implemented are as follows:

Table 8-3 Kernel Calls That a Multi-Core Memory Protection Unit Needs to Implement

Name	Function category	Necessity of implementation
RME_KFN_PGT_CACHE_CLR	Clear cache	Necessary
RME_KFN_PGT_LINE_CLR	Clear cache	Optional

8.2.1.3 Software-Filled Memory Management Unit

The implementation of this type of page table is exactly the same as the implementation described in Section 8.2.1.2; the only difference is that path compression is not supported at this time, and the start

[7] For example, the exceptions generated by the hardware automatic stacking of ARMv7-M; when these exceptions occur, the processor does not fill in MMFAR, so it is difficult to locate the specific access address; even if it can be located, the corresponding register contents of the original thread, such as LR, have already been declared lost, and it is impossible to resume the original program execution flow after handling the error

[8] Therefore, the validity of the start address needs to be checked at construction time

[9] This upper limit is also "the number of regions available to the memory protection unit minus 1"

address item may also be left blank when creating a page directory. For a specific description, please refer to Section 8.2.1.2 .

The page table of TMS320C66X is a typical example of this type of implementation. The real-time performance of this type of implementation is exactly the same as that of the implementation described in Section 8.2.1.2. For the kernel function calls required to be implemented in this implementation, please also refer to the description in Section 8.2.1.2 .

8.2.1.4 Hardware-Filled Memory Management Unit

The implementation of this type of page table is characterized by not supporting path compression; the cache is a hardware-implemented translation look-aside buffer (Trans Look-aside Buffer, TLB), and cache updates are performed by hardware. During mapping and construction, no special consideration is needed, while during unmapping and destruction, the TLBs of all CPUs must be forcibly flushed[10]. Page directories may be shared arbitrarily between page tables.

In terms of real-time performance, because TLB updates are completely controlled by hardware, more kernel function calls are needed to assist with TLB locking and address space identifier (Address Space IDentifier, ASID) related operations to ensure real-time performance.

This type of implementation generally needs to support page directory attributes, because the relevant hardware support often exists, and such architectures are generally used in fairly complex systems that have a realistic demand for page directory attributes.

The page table of x86-64 is a typical example of this type of implementation. The degree of uncertainty in the real-time performance of this type of implementation is relatively large. If ASID support and TLB locking are implemented, then very good real-time performance can be obtained; if no form of TLB locking is implemented, or the processor does not support ASID, then the real-time performance is relatively poor. In this type of page table implementation, the kernel function calls that can be implemented are as follows:

Table 8-4 Kernel Calls That a Hardware-Filled Memory Management Unit Needs to Implement

Name	Function category	Necessity of implementation
RME_KFN_PGT_CACHE_CLR	Clear cache	Necessary
RME_KFN_PGT_LINE_CLR	Clear cache	Optional
RME_KFN_PGT_ASID_SET	Set attributes	Optional
RME_KFN_PGT_TLB_LOCK	Lock the TLB	Optional
RME_KFN_PGT_ENTRY_MOD	Query or modify attributes	Optional

If the architecture supports more forms of page-table-related operations, then adding additional kernel calls to implement them is also permitted.

8.2.2 Clearing All Cache of a Page Table on the Current CPU

This operation should be implemented as clearing all cache of a certain page table on the current CPU. In the case described in Section 8.2.1.1, this operation does not need to be implemented; in the cases described in Sections 8.2.1.2 and 8.2.1.3, this operation must be implemented, and the top-level cache metadata of that page table on the current CPU should be emptied; in the case described in Section 8.2.1.4, this operation must also be implemented. If the processor supports ASID, then all entries in the TLB of the ASID corresponding to that page table should be cleared[11]; if the processor does not support ASID and the current page table is this page table, then it suffices to flush the TLB directly; if the processor does not

[10] If the TLBs of these CPUs may contain stale entries
[11] Including locked TLB entries

support ASID and the current page table is not this page table, then no operation is needed and it suffices to return success directly.

Table 8-5 Parameters Needed to Clear All Cache of a Page Table on the Current CPU

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PGT_CACHE_CLR.
Cap_Pgt	cid_t	Sub_ID	A capability number corresponding to the top-level page directory capability whose cache is to be cleared. This capability number may be a one-level or two-level lookup encoding.

The return values of this operation are recommended to be implemented as follows:

Table 8-6 Suggested Return Value Implementation for Clearing All Cache of a Page Table on the Current CPU

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.

8.2.3 Clearing a Certain Line of the Cache of a Page Table on the Current CPU

This operation should be implemented as clearing a specific entry of a certain page table on the current CPU. In the case described in Section 8.2.1.1, this operation does not need to be implemented; in the cases described in Sections 8.2.1.2 and 8.2.1.3, this operation is optional to implement, and if implemented, one entry in the top-level cache metadata of that page table on the current CPU should be cleared; in the case described in Section 8.2.1.4, this operation is also optional to implement. If the processor supports ASID, then one entry of the ASID corresponding to that page table in the TLB should be cleared[12]; if the processor does not support ASID and the current page table is this page table, then it suffices to clear one entry in the TLB; if the processor does not support ASID and the current page table is not this page table, then no operation is needed and it suffices to return success directly.

Table 8-7 Parameters Needed to Clear a Certain Line of the Cache of a Page Table on the Current CPU

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PGT_CACHE_CLR.

[12] Including a locked TLB entry, if it is selected

Parameter name	Type	Position	Description
Cap_Pgt	cid_t	Sub_ID	A capability number corresponding to the top-level page directory capability whose cache line is to be cleared. This capability number may be a one-level or two-level lookup encoding.
Vaddr	ptr_t	Param1	The virtual address whose TLB entry is to be cleared.

The return values of this operation are recommended to be implemented as follows:

Table 8-8 Suggested Return Value Implementation for Clearing a Certain Line of the Cache of a Page Table on the Current CPU

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.

8.2.4 Setting the ASID of a Page Table

This operation should be implemented as setting the ASID of a certain page table. The issue of where the ASID of a page table is stored is resolved by the implementer. In the cases described in Sections 8.2.1.1, 8.2.1.2 and 8.2.1.3, this operation does not need to be implemented; in the case described in Section 8.2.1.4, if the processor supports ASID, then this operation is optional to implement, and should set the ASID of that page table; if the processor does not support ASID, then this operation does not need to be implemented.

Table 8-9 Parameters Needed to Set the ASID of a Page Table

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PGT_ASID_SET.
Cap_Pgt	cid_t	Sub_ID	A capability number corresponding to the top-level page directory capability whose ASID is to be set. This capability number may be a one-level or two-level lookup encoding.
ASID	ptr_t	Param1	The value to be set as the ASID of this page table.

The return values of this operation are recommended to be implemented as follows:

Table 8-10 Suggested Return Value Implementation for Setting the ASID of a Page Table

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.

8.2.5 Locking a Certain Page in the TLB

This operation should be implemented as locking a certain page into the TLB. In the cases described in Sections 8.2.1.1, 8.2.1.2 and 8.2.1.3, this operation does not need to be implemented; in the case described in Section 8.2.1.4, this operation is optional to implement, and the specified virtual address should be locked into the TLB. If the processor does not support ASID, then whether this locked page is cleared when switching address spaces is decided by the architecture and the implementer.

Table 8-11 Parameters Needed to Lock a Certain Page in the TLB

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PGT_TLB_LOCK.
Cap_Pgt	cid_t	Sub_ID	A capability number corresponding to the top-level page directory capability whose ASID is to be set. This capability number may be a one-level or two-level lookup encoding.
Vaddr	ptr_t	Param1	The virtual address to be locked into the TLB.

The return values of this operation are recommended to be implemented as follows:

Table 8-12 Suggested Return Value Implementation for Locking a Certain Page in the TLB

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.
RME_ERR_PGT_ADDR	The specified address cannot be found in the given page table, so TLB locking cannot be performed.
RME_ERR_PGT_MAP	The TLB capacity is full, so no new pages can be locked.

8.2.6 Querying or Modifying the Attributes of a Certain Page

This operation should be implemented as querying or modifying the attributes of a certain page. The most common page attribute is the "Dirty"[13] attribute, which is used to determine whether the page has

[13] Also called "Modified"

been written; some processors may have other page attributes that can be queried or modified. Note that this function cannot be used to modify any standard page attribute specified by RME in the page table operation interface, but can only be used to read them; reading and writing other architecture-specific attributes are both allowed. In the cases described in Sections 8.2.1.1, 8.2.1.2 and 8.2.1.3, this operation is optional to implement, and if implemented, it does not need to have the function of modifying attributes; in the case described in Section 8.2.1.4, this operation is optional to implement, and if implemented, both read and write functions should be implemented.

Table 8-13 Parameters Needed to Query or Modify the Attributes of a Certain Page

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PGT_ENTRY_MOD.
Cap_Pgt	cid_t	Sub_ID	A capability number corresponding to the top-level page directory capability whose page attributes are to be queried. This capability number may be a one-level or two-level lookup encoding.
Vaddr	ptr_t	Param1	The virtual address whose attributes are to be queried or modified.
Operation	ptr_t	Param2	The attribute to be queried or modified. The meaning of this parameter is determined by the implementer.

The return values of this operation are recommended to be implemented as follows:

Table 8-14 Suggested Return Value Implementation for Querying or Modifying the Attributes of a Certain Page

Return value	Meaning
Non-negative value	The operation succeeded, and the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.
RME_ERR_PGT_ADDR	The specified address cannot be found in the given page table, so the attribute modification or query cannot be performed.

8.3 Interrupt Controller Kernel Call Implementation Specifications ("I" Standard Extension)

Interrupts are an important means by which a system responds to external events; they are responsible for receiving external asynchronous signals and passing those signals to other parts of the system, and are an important component of real-time systems. RME manages each interrupt by means of kernel signal endpoints, and each interrupt is represented as a send to a certain kernel signal endpoint. Therefore, we need to know which interrupt should be sent to which kernel endpoint; if there are multiple CPUs in the system, we also need to know which CPU the hardware interrupt is to be sent to. RME specifies the following three interrupt-controller-related kernel calls in total, all of which are optional to implement:

Table 8-15 Overview of Interrupt Controller Kernel Calls

Main function number	Call name	Purpose
0xF100	RME_KFN_INT_LOCAL_MOD	Get or set the state of a certain local interrupt source.
0xF101	RME_KFN_INT_GLOBAL_MOD	Get or set the state of a certain global interrupt source.
0xF102	RME_KFN_INT_LOCAL_TRIG	Trigger a certain local interrupt source of a certain CPU.
0xF103	RME_KFN_EVT_LOCAL_TRIG	Trigger a certain local event source of a certain CPU.

8.3.1 Getting or Setting the State of a Certain Local Interrupt Source

This operation should be designed to get or set the state of a certain local interrupt source of the current CPU. Common local interrupt source states include the on/off switch of the local interrupt source, the preemptive and non-preemptive priorities, and their corresponding kernel signal endpoint. This kernel call generally modifies the settings of the CPU's local interrupt controller[14].

Table 8-16 Parameters Needed to Get or Set the State of a Certain Local Interrupt Source

Parameter name	Type	Position	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_INT_LOCAL_MOD.
Int_Num	ptr_t	Sub_ID	The number of the local interrupt source to be queried or modified.
Operation	ptr_t	Param1	The operation to be performed. The meaning of this parameter is determined by the implementer.
Param	ptr_t	Param2	Additional parameter. The meaning of this parameter is determined by the implementer.

[14] Such as the LAPIC of x86-64

It is recommended to implement the return value of this operation as follows:

Table 8-17 Recommended Return Value Implementation for Getting or Setting the State of a Local Interrupt Source

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.3.2 Getting or Setting the State of a Global Interrupt Source

This operation should be designed to get or set the state of a global interrupt source. Common global interrupt source states include the on/off state of the global interrupt source, the CPU to which it is to be connected, and the local interrupt source of that CPU to which it is to correspond. This kernel call generally modifies the settings of the system's global interrupt controller[15].

Table 8-18 Parameters Needed for Getting or Setting the State of a Global Interrupt Source

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_INT_LOCAL_MOD.
Int_Num	ptr_t	Sub_ID	The number of the global interrupt source to be queried or modified.
Operation	ptr_t	Param1	The operation to be performed. The meaning of this parameter is determined by the implementer.
Param	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-19 Recommended Return Value Implementation for Getting or Setting the State of a Global Interrupt Source

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.3.3 Triggering the Local Interrupt Source of a CPU

This operation should be designed to trigger a local interrupt source of a certain CPU. Common uses include triggering a local software interrupt or sending an inter-processor interrupt (Inter-Processor Interrupt, IPI) to other processors. This kernel call generally triggers the interrupt through the CPU's local

[15] such as the IOAPIC of x86-64

interrupt controller[16], and sometimes may also trigger the interrupt through the global interrupt controller[17]. The functions that this operation can implement include triggering one interrupt of one CPU, triggering several interrupts of one CPU, triggering one interrupt of a group of CPUs, and triggering several interrupts of a group of CPUs; the specific combination of functions to be implemented is decided by the user.

Table 8-20 Parameters Needed for Triggering the Local Interrupt Source of a CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_INT_LOCAL_TRIG.
CPUID	ptr_t	Sub_ID	The number of the CPU to be triggered.
Int_Num	ptr_t	Param1	The number of the interrupt to be triggered. The meaning of this parameter is determined by the implementer.
Param	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-21 Recommended Return Value Implementation for Triggering the Local Interrupt Source of a CPU

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.3.4 Triggering the Local Event Source of a CPU

This operation should be designed to trigger a local event source of a certain CPU. Unlike local interrupt sources, a local event source generally corresponds to some software-defined operations; these operations send signals to certain special endpoints created at system boot or modify certain memory regions, thereby accomplishing operations or functions defined by the user-mode library of some platforms. For each architecture, the meaning and purpose of this operation may all be different.

Table 8-22 Parameters Needed for Triggering the Local Event Source of a CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_EVT_LOCAL_TRIG.
CPUID	ptr_t	Sub_ID	The number of the CPU to be triggered.
Evt_Num	ptr_t	Param1	The number of the event to be triggered. The meaning of this parameter is determined by the implementer.

[16] such as the LAPIC of x86-64
[17] such as the IOAPIC of x86-64

Parameter name	Type	Location	Description
Param	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-23 Recommended Return Value Implementation for Triggering the Local Event Source of a CPU

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.4 Cache and Prefetching Kernel Call Implementation Specifications ("C" Standard Extension)

The cache and prefetch subsystem is an important guarantee of system performance. Caches can exploit the locality of programs to accelerate execution, while prefetching can hide memory latency by initiating memory operations in advance. For the following reasons, it is necessary for us to implement the cache- and prefetch-related kernel calls:

1. Hardware such as DMA often has no cache coherence with the CPU, so cache coherence operations must be performed manually.
2. Self-modifying code invalidates instruction prefetching and the instruction cache, so on some architectures the prefetch buffers and caches must be cleared manually.
3. On some architectures the cacheability of the address space is not controlled by the page table but by special registers, and requires additional configuration.
4. On some architectures the cache size is variable, and part of the cache may be used as scratchpad memory (Scratchpad Memory)[18], which requires additional configuration.
5. On some architectures the organization of the cache is variable[19], which requires additional configuration.
6. Some multi-core architectures allow the caches of the various processors to be incoherent over certain memory regions, which requires additional configuration.

[18] also called tightly coupled memory (Tightly Coupled Memory, TCM)
[19] ways, sets, or cache line size

RME specifies the following 5 cache- and prefetch-related kernel calls in total, all of which are optional to implement:

Table 8-24 Overview of Cache and Prefetching Kernel Calls

Main function number	Call name	Function
0xF200	RME_KFN_CACHE_MOD	Get or set the cache state.
0xF201	RME_KFN_CACHE_CONFIG	Get or set the cache configuration.
0xF202	RME_KFN_CACHE_MAINT	Cache maintenance.
0xF203	RME_KFN_CACHE_LOCK	Lock the cache.
0xF204	RME_KFN_PRFTH_MOD	Get or set the prefetch state.

8.4.1 Getting or Setting the Cache State

This call gets or sets the enable and disable state of a certain cache of the CPU. Generally speaking, this merely requires operating one register of the cache controller to enable and disable the cache. The cache being operated on may be the level-1 cache (L1), the level-2 cache (L2), and so on. Caches can be further subdivided into instruction caches and data caches. On some processors whose cacheability is not controlled by the page table, the cacheability of the address space can also be configured by this kernel call.

Before enabling a cache, whether to clear the contents of the cache is decided by the implementer; the recommended practice is to clear the cache contents. Before disabling a cache, whether to flush the contents of all modified cache lines to memory is likewise decided by the implementer; the recommended practice is to flush the cache contents to memory.

Table 8-25 Parameters Needed for Getting or Setting the Cache State

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_CACHE_MOD.
Cache_ID	ptr_t	Sub_ID	The identifier of the cache to be obtained or set. The meaning of this parameter is determined by the implementer.
Operation	ptr_t	Param1	The operation to be performed. The meaning of this parameter is determined by the implementer.
Param	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-26 Recommended Return Value Implementation for Getting or Setting the Cache State

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.4.2 Getting or Setting the Cache Configuration

Some processors support dynamic configuration of the cache organization. This may include allowing part of the cache to be used as tightly coupled memory[20] (Tightly Coupled Memory, TCM), or allowing the organizational structure of the cache to be modified, or allowing a general-purpose cache to be configured as a dedicated instruction cache or data cache, or allowing cache incoherence among the various CPUs, or reserving certain cache lines for a certain CPU. These operations are handled by this kernel call.

Table 8-27 Parameters Needed for Getting or Setting the Cache Configuration

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_CACHE_CONFIG.
Cache_ID	ptr_t	Sub_ID	The identifier of the cache to be configured. The meaning of this parameter is determined by the implementer.
Param1	ptr_t	Param1	Additional parameters. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-28 Recommended Return Value Implementation for Getting or Setting the Cache Configuration

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.4.3 Cache Maintenance

In many situations, such as when DMA or a peripheral updates memory, we need to invalidate a specific cache line or a part of the cache. Or, we need to flush the data in the cache to memory in a timely manner for use by certain peripherals. Typical such peripherals include DMA, network cards, graphics cards, and so on. Such operations are handled by this kernel call.

Table 8-29 Parameters Needed for Cache Maintenance

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_CACHE_MAINT.
Cache_ID	ptr_t	Sub_ID	The identifier of the cache to be forcibly invalidated. The meaning of this parameter is determined by the implementer.

[20] also called scratchpad memory (Scratchpad memory)

Parameter name	Type	Location	Description
Operation	ptr_t	Param1	The category of maintenance to be performed. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-30 Recommended Return Value Implementation for Cache Maintenance

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.4.4 Locking the Cache

In some situations, we want certain data to remain in the cache forever and never be replaced; at such times the cache needs to be locked. There are many possible implementation schemes for locking a cache: locking a certain line of the cache, locking a certain set of the cache, locking a certain way of the cache, or locking a certain region of the cache, and so on. Such operations are handled by this kernel call.

Table 8-31 Parameters Needed for Locking the Cache

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_CACHE_LOCK.
Cache_ID	ptr_t	Sub_ID	The identifier of the cache to be locked. The meaning of this parameter is determined by the implementer.
Param1	ptr_t	Param1	Additional parameters. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-32 Recommended Return Value Implementation for Locking the Cache

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.4.5 Getting or Setting the Prefetch State

This operation enables or disables the CPU's prefetching function and branch predictor, and is also responsible for querying their state. Generally speaking, this only requires configuring one register of the prefetch engine controller. On some processors, the prefetchability of the address space can also be configured by this kernel call.

Table 8-33 Parameters Needed for Getting or Setting the Prefetch State

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PRFTH_MOD.
Prefetch_ID	ptr_t	Sub_ID	The identifier of the prefetch engine to be enabled. The meaning of this parameter is determined by the implementer.
Operation	ptr_t	Param1	The operation to be performed. The meaning of this parameter is determined by the implementer.
Param	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return value of this operation as follows:

Table 8-34 Recommended Return Value Implementation for Getting or Setting the Prefetch State

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.5 Hardware Hotplug Kernel Call Implementation Specifications (“H” Standard Extension)

Hardware hotplug is a very important part of system availability and scalability. Hotplug allows the system to dynamically add or reduce computing resources[21] on the premise of continuous operation, which greatly increases the scale flexibility of server-class systems and mainframes; in more complex embedded applications, boards can also be dynamically added to increase computing power, or dynamically removed for repair and system maintenance. RME specifies the following 3 hardware hotplug-related kernel calls in total, all of which are optional to implement:

Table 8-35 Overview of Hardware Hotplug Kernel Calls

Main function number	Call name	Function
0xF300	RME_KFN_HPNP_PCPU_MOD	Get or set the state of a physical processor package.
0xF301	RME_KFN_HPNP_LCPU_MOD	Get or set the state of a logical processor.
0xF302	RME_KFN_HPNP_PMEM_MOD	Get or set the state of a physical memory package.

8.5.1 Getting or Setting the State of a Physical Processor Package

This operation gets or sets the state of a physical processor package. A “physical processor” refers to one processor chip package, or a processor board containing multiple processor chip packages. Common states that need to be obtained or set include power on/off, insertion detection, and so on.

Table 8-36 Parameters Needed for Getting or Setting the State of a Physical Processor Package

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_HPNP_PCPU_MOD.
PCPUID	ptr_t	Sub_ID	The identifier of the physical processor whose state is to be obtained or set.
Param1	ptr_t	Param1	Additional parameters. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

[21] mainly CPU and memory

It is recommended to implement the return value of this operation as follows:

Table 8-37 Recommended Return Value Implementation for Getting or Setting the State of a Physical Processor Package

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.5.2 Getting or Setting the State of a Logical Processor

This operation gets or sets the state of a logical processor. A "logical processor" refers to the mutually independent CPU cores that software can recognize; each core can run only one thread at any time. Note that a logical processor does not necessarily always correspond to a physical core or package. A multi-core processor may integrate multiple physical processors within one package, while hyper-threading technology simulates one physical processor as multiple logical processors. Common states that need to be obtained or set include processor initialization, shutdown, and so on.

Table 8-38 Parameters Needed for Getting or Setting the State of a Logical Processor

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_HPNP_LCPU_MOD.
LCPUID	ptr_t	Sub_ID	The identifier of the logical processor whose state is to be obtained or set.
Param1	ptr_t	Param1	Additional parameters. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-39 Recommended Return Value Implementation for Getting or Setting the State of a Logical Processor

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.5.3 Getting or Setting the State of a Physical Memory Package

This operation gets or sets the state of a physical memory package. A "physical memory package" refers to a physical memory chip, or a memory module containing multiple physical memory chips. Common states that need to be read or set include power on/off, insertion detection, and so on.

Table 8-40 Parameters Needed for Getting or Setting the State of a Physical Memory Package

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_HPNP_PMEM_MOD.
PMEMID	ptr_t	Sub_ID	The identifier of the physical memory package whose state is to be read or set.
Param1	ptr_t	Param1	Additional parameters. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Additional parameters. The meaning of this parameter is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-41 Recommended Return Value Implementation for Getting or Setting the State of a Physical Memory Package

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6 Clock and Voltage Kernel Call Implementation Specifications ("F" Standard Extension)

The clock frequency and voltage of a system directly determine the performance and power consumption of the system, and also have an important impact on aspects such as battery life and thermal headroom. RME specifies a total of 7 clock- and voltage-related kernel calls as follows; they are all optionally implemented:

Table 8-42 Overview of Clock and Voltage Kernel Calls

Main function number	Call name	Function
0xF400	RME_KFN_IDLE_SLEEP	Putting the processor to sleep temporarily.
0xF401	RME_KFN_SYS_REBOOT	Restarting the entire system.
0xF402	RME_KFN_SYS_SHDN	Powering off the entire system.
0xF403	RME_KFN_VOLT_MOD	Getting or setting voltage parameters.
0xF404	RME_KFN_FREQ_MOD	Getting or setting frequency parameters.
0xF405	RME_KFN_PMOD_MOD	Getting or setting power states.
0xF406	RME_KFN_SAFETY_MOD	Getting or setting safety protection states.

8.6.1 Putting the Processor to Sleep

This operation causes the processor to execute a sleep instruction[22] to enter the sleep state. Various architectures usually provide an instruction that causes the processor to temporarily stop operating until the next interrupt arrives, at which point it continues executing. Some architectures also provide multiple instructions; different instructions cause the CPU to enter sleep states of different depths, in order to strike a balance between power consumption and interrupt latency; the lower the power consumption of the sleep state, the higher the interrupt latency tends to be. For the implementation of correct sleep, please refer to 9.1.5.

Table 8-43 Putting the Processor to Sleep

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_IDLE_SLEEP.
Operation	ptr_t	Sub_ID	Operation option, which determines the operation to be performed.

[22] such as HLT on x86-64 and WFE on ARM

It is recommended to implement the return values of this operation as follows:

Table 8-44 Recommended Return Value Implementation for Putting the Processor to Sleep

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.2 Restarting the Entire System

This operation, when called on any processor, causes the hardware of the entire system to be completely reset and restarted. Some systems provide multiple restart modes; in this case the restart mode is passed in through an additional parameter.

Table 8-45 Restarting the Entire System

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_SYS_REBOOT.
Operation	ptr_t	Sub_ID	Operation option, which determines the operation to be performed.

It is recommended to implement the return values of this operation as follows:

Table 8-46 Recommended Return Value Implementation for Restarting the Entire System

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.3 Powering Off the Entire System

This operation, when called on any processor, powers off the entire system. This operation is not supported on some architectures[23], because these architectures do not power off. Some systems provide multiple shutdown modes; in this case the shutdown mode is passed in through an additional parameter.

Table 8-47 Powering Off the Entire System

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_SYS_SHDN.
Operation	ptr_t	Sub_ID	Operation option, which determines the operation to be performed.

[23] architectures commonly used for microcontrollers, such as ARMv7-M

It is recommended to implement the return values of this operation as follows:

Table 8-48 Recommended Return Value Implementation for Powering Off the Entire System

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.4 Getting or Setting Voltage Parameters

This operation gets or sets the supply voltage values of the various power rails of the CPU, memory, and other peripherals. The unit of voltage may be millivolts (mV) or microvolts (uV).

Table 8-49 Parameters Needed for Getting or Setting Voltage Parameters

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VOLT_MOD.
Dom_ID	ptr_t	Sub_ID	The identifier of the power domain whose voltage is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Voltage	ptr_t	Param2	The voltage value to be set. The unit of this parameter is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-50 Recommended Return Value Implementation for Getting or Setting Voltage Parameters

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.5 Getting or Setting Frequency Parameters

This operation gets or sets the operating frequency of each clock domain of the CPU, memory, and other peripherals. The unit of frequency may be millihertz (mHz), hertz (Hz), kilohertz (kHz), megahertz (MHz), or gigahertz (GHz).

Table 8-51 Parameters Needed for Getting or Setting Frequency Parameters

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_FREQ_MOD.

Parameter name	Type	Location	Description
Fdom_ID	ptr_t	Sub_ID	The identifier of the clock domain whose clock frequency is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Frequency	ptr_t	Param2	The frequency value to be set. The unit of this parameter is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-52 Recommended Return Value Implementation for Getting or Setting Frequency Parameters

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.6 Getting or Setting Power States

This operation gets or sets the power states of the CPU, memory, and other peripherals. A power state typically includes the on/off state of power domains and the on/off state of clock domains. On processors that support Dynamic Voltage Frequency Scaling (DVFS), this operation is also responsible for setting the state of this feature. This operation can also be used to implement restart and shutdown, but this is not recommended, because restart and shutdown have already been assigned dedicated function call numbers.

Table 8-53 Parameters Needed for Getting or Setting Power States

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PMOD_MOD.
Pdom_ID	ptr_t	Sub_ID	The identifier of the power domain whose power state is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.

It is recommended to implement the return values of this operation as follows:

Table 8-54 Recommended Return Value Implementation for Getting or Setting Power States

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.6.7 Getting or Setting Safety Protection States

This operation gets or sets the state of the power and frequency safety lock. When the safety lock has not been opened, certain voltages and frequencies are not allowed to be set, because this may cause a system crash and may even cause permanent damage to the hardware[24]. When the safety lock is opened, these restrictions are lifted, allowing the system to be set to more extreme states.

Table 8-55 Parameters Needed for Getting or Setting Safety Protection States

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_SAFETY_MOD.
Dom_ID	ptr_t	Sub_ID	The identifier of the power domain or clock domain whose safety protection state is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.

It is recommended to implement the return values of this operation as follows:

Table 8-56 Recommended Return Value Implementation for Getting or Setting Safety Protection States

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

[24] especially voltage adjustment

8.7 System Monitoring Kernel Call Implementation Specifications (“M” Standard Extension)

The system monitor is indispensable for identifying system functionality, confirming system status, and tuning system performance. RME specifies a total of 6 system monitoring kernel calls as follows; they are all optionally implemented:

Table 8-57 Overview of System Monitoring Kernel Calls

Main function number	Call name	Function
0xF500	RME_KFN_PERF_CPU_FUNC	Getting CPU functionality support statuses.
0xF501	RME_KFN_PERF_MON_MOD	Getting or setting performance monitor configurations.
0xF502	RME_KFN_PERF_CNT_MOD	Getting or setting the value of a counting performance counter.
0xF503	RME_KFN_PERF_CYCLE_MOD	Getting or setting the value of a cycle performance counter.
0xF504	RME_KFN_PERF_DATA_MOD	Getting or setting the value of a data performance counter.
0xF505	RME_KFN_PERF_PHYS_MOD	Getting or setting the value of a physical performance counter.
0xF506	RME_KFN_PERF_CUMUL_MOD	Getting or setting the value of a cumulative performance counter.

When implementing the read operation of some performance counters, because the length of these counters may exceed the allowed length of the return value of an RME system call[25], the implementer needs to use other registers themselves to return the remaining part of the counter.

8.7.1 Getting CPU Functionality Support Statuses

CPUs of the same architecture may support different subsets of functionality for various reasons. This call allows the user to query the functionality of the CPU to confirm its compatibility with the software.

Table 8-58 Parameters Needed for Getting CPU Functionality Support Statuses

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_CPU_FUNC.
Freg_ID	ptr_t	Sub_ID	The identifier of the function register to be obtained.

[25] one machine word

It is recommended to implement the return values of this operation as follows:

Table 8-59 Recommended Return Value Implementation for Getting CPU Functionality Support Statuses

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.2 Getting or Setting Performance Monitor Configurations

Generally speaking, each CPU has some performance counters to record various events in the system. The number of events in a system is often very large, while the number of performance counters in a CPU is limited, so only a subset of events can be selected for recording. This call allows the state of the performance counters to be configured and the event source of each counter to be specified.

Table 8-60 Parameters Needed for Getting or Setting Performance Monitor Configurations

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_MON_MOD.
Perf_ID	ptr_t	Sub_ID	The identifier of the performance counter whose configuration is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Param	ptr_t	Param2	The event identifier to be set for this counter, or other information, which may also be unused. The specific use is determined by the user.

It is recommended to implement the return values of this operation as follows:

Table 8-61 Recommended Return Value Implementation for Getting or Setting Performance Monitor Configurations

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.3 Getting or Setting the Value of a Counting Performance Counter

This operation gets or sets the value of a counting performance counter. A counting performance counter is responsible for recording the number of occurrences of certain events in the system. Common counting events include:

1. The number of cache replacements
2. The number of cache hits or misses

3. The number of times an upper-level cache misses and a lower-level cache hits or misses
4. The number of times an upper-level cache misses and a lower-level cache hits, where the hit does or does not involve a cache coherence operation
5. In a NUMA structure, the number of local or remote memory accesses
6. The number of total interrupts[26] received by the processor
7. The number of branch instructions correctly predicted or incorrectly predicted by the processor
8. The number of atomic operations successfully executed or unsuccessfully executed by the processor, and the number of serialized atomic operations
9. The number of total instructions[27] fetched or completed by the processor

Table 8-62 Parameters Needed for Getting or Setting the Value of a Counting Performance Counter

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_CNT_MOD.
Cnt_ID	ptr_t	Sub_ID	The identifier of the counting performance counter whose value is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set for the counter. Some counters are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-63 Recommended Return Value Implementation for Getting or Setting the Value of a Counting Performance Counter

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.4 Getting or Setting the Value of a Cycle Performance Counter

This operation gets or sets the value of a cycle performance counter. A cycle performance counter is responsible for recording the number of clock cycles during which the effects produced by certain events in the system last. Common cycle measurement events include:

1. The number of cycles elapsed since the processor was powered on
2. The number of cycles for which the pipeline is stalled due to cache misses
3. The number of cycles for which the pipeline is stalled due to TLB misses
4. The number of cycles in which the processor does not dispatch new instructions due to branch mis-prediction, instruction delay slots, or other reasons

[26] or a certain class of interrupts
[27] or a certain class, such as branches, memory accesses, I/O, multiplication and division, etc.

Table 8-64 Parameters Needed for Getting or Setting the Value of a Cycle Performance Counter

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_CYCLE_MOD.
Cycle_ID	ptr_t	Sub_ID	The identifier of the cycle performance counter whose configuration is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set for the counter. Some counters are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-65 Recommended Return Value Implementation for Getting or Setting the Value of a Cycle Performance Counter

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.5 Getting or Setting the Value of a Data Performance Counter

This operation gets or sets the value of a data performance counter. A data performance counter is responsible for recording the cumulative amount of data exchanged brought about by certain events in the system. Common data measurement events include:

1. The total amount of data flowing over a certain bus in the system[28]
2. The total amount of data caused by certain operations or certain instructions in the system

Table 8-66 Parameters Needed for Getting or Setting the Value of a Data Performance Counter

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_DATA_MOD.
Data_ID	ptr_t	Sub_ID	The identifier of the data performance counter whose configuration is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set for the counter. Some counters are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-67 Recommended Return Value Implementation for Getting or Setting the Value of a Data Performance Counter

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.6 Getting or Setting the Value of a Physical Performance Counter

This operation gets or sets the value of a physical performance counter. A physical performance counter is responsible for recording the values of certain physical quantities in the system, or related information about them. Common physical quantities include:

1. The temperature of a certain part of the circuit or the ambient temperature
2. The fan speed of the cooling system
3. The current magnitude of a certain clock domain
4. The total amount of power consumed by a certain clock domain or power domain

[28] or the amount of data that meets a certain condition, such as size, transfer direction, nature of the data, etc.

Table 8-68 Parameters Needed for Getting or Setting the Value of a Physical Performance Counter

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_PHYS_MOD.
Phys_ID	ptr_t	Sub_ID	The identifier of the physical performance counter whose configuration is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set for the counter. Some counters are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-69 Recommended Return Value Implementation for Getting or Setting the Value of a Physical Performance Counter

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.7.7 Getting or Setting the Value of a Cumulative Performance Counter

This operation gets or sets the value of a cumulative performance counter. A cumulative performance counter is responsible for recording the values of certain cumulative time quantities in the system, or related information about them. Common cumulative quantities include:

1. The total power-on time or the total number of power-ons of the system
2. The continuous working time of the system since this power-on
3. The remaining service life or useful life of the system
4. The health status and damage management status of the system

Table 8-70 Parameters Needed for Getting or Setting the Value of a Cumulative Performance Counter

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_PERF_CUMUL_MOD.
Cumul_ID	ptr_t	Sub_ID	The identifier of the cumulative performance counter whose configuration is to be read or set.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.

Parameter name	Type	Location	Description
Value	ptr_t	Param2	The value to be set for the counter. Some counters are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-71 Recommended Return Value Implementation for Getting or Setting the Value of a Cumulative Performance Counter

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.8 Full Virtualization Kernel Call Implementation Specifications ("V" Standard Extension)

Full virtualization (Full Virtualization) is indispensable for inheriting existing software stacks and related functionality, and is also an essential feature of new operating system designs. Here, "full virtualization" specifically refers to the virtualization extensions provided by the hardware; they greatly increase the efficiency and security of virtualization. The full virtualization kernel calls specified by RME are all optionally implemented, but as long as one decides to implement this feature, then all of them must be implemented. There are 10 such calls in total as follows:

Table 8-72 Overview of Full Virtualization Kernel Calls

Main function number	Call name	Function
0xF600	RME_KFN_VM_CRT	Creating a virtual machine.
0xF601	RME_KFN_VM_DEL	Deleting a virtual machine.
0xF602	RME_KFN_VM_PGT	Setting the page table used by a virtual machine.
0xF603	RME_KFN_VM_MOD	Getting or setting the attributes of a virtual machine.
0xF604	RME_KFN_VCPU_CRT	Creating a virtual CPU.
0xF605	RME_KFN_VCPU_BIND	Binding a virtual CPU to a virtual machine.
0xF606	RME_KFN_VCPU_FREE	Unbinding a virtual CPU from a virtual machine.
0xF607	RME_KFN_VCPU_DEL	Deleting a virtual CPU.
0xF608	RME_KFN_VCPU_MOD	Getting or setting the registers of a virtual CPU.
0xF609	RME_KFN_VCPU_RUN	Running a virtual CPU.

When implementing the read operation of some registers, because the length of these registers may exceed the allowed length of the return value of an RME system call[29], the implementer needs to use other registers themselves to return the remaining part of the register.

Because hardware virtualization extensions often require some special, hardware-defined data structures, the porter needs to allocate the virtual machine and virtual CPU (Virtual CPU, VCPU) kernel objects themselves. The underlying logic of these kernel objects and the definitions of the concrete capability structures also need to be completed by the porter. Because the RME kernel does not leave any operation flag bits for kernel extensions, apart from the capability table and page table, these operations do not undergo any operation flag permission checks and may be directly executed by default.

8.8.1 Creating a Virtual Machine

Before using the virtualization extensions, a virtual machine must first be created using this call. In RME, any virtual CPU can only be created after the virtual machine has been created.

[29] one machine word

Table 8-73 Parameters Needed for Creating a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VM_CRT.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created virtual machine capability. This capability number can be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	Param1.D1	A kernel memory capability number whose identified kernel memory range must be able to hold the entire virtual machine kernel object. This capability number can be a one-level or two-level lookup encoding. This kernel function does not check the operation flag bits of the kernel memory capability, and this operation is permitted by default.
Cap_VM	cid_t	Param1.D0	A capability number corresponding to a position in the capability table that is to receive this newly created virtual machine capability. The capability corresponding to this capability number must be blank. This capability number can only be a one-level lookup encoding.
Raddr	ptr_t	Param2	The starting relative virtual address in kernel space to be used by the newly created virtual machine. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.

It is recommended to implement the return values of this operation as follows:

Table 8-74 Recommended Return Value Implementation for Creating a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_VM is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has already been frozen.
RME_ERR_CPT_FROZEN	Cap_VM is frozen, or another core is creating a capability at that location.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.

Return value	Meaning
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	The range of Cap_Kom is incorrect.
RME_ERR_CPT_EXIST	Cap_VM is not a blank capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_PGT_HW	Due to underlying hardware constraints, the virtual machine kernel object cannot be created at this memory address.

8.8.2 Deleting a Virtual Machine

This operation deletes a virtual machine. The virtual machine must have been created by RME_KFN_VM_CRT, and there are currently no VCPUs inside it.

Table 8-75 Parameters Needed for Deleting a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VM_DEL.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table that contains the virtual machine capability being deleted. This capability number can be a one-level or two-level lookup encoding.
Cap_VM	cid_t	Param1	A capability number corresponding to the virtual machine capability to be deleted. The capability corresponding to this capability number must be a virtual machine capability. This capability number can only be a one-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-76 Recommended Return Value Implementation for Deleting a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_VM is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	Cap_VM has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_VM is not a virtual machine capability.
RME_ERR_CPT_NULL	Cap_VM is a blank capability.
RME_ERR_CPT_NULL	Two cores simultaneously attempt to delete this virtual machine; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_VM is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_VM is not 0, or it is not a root capability.
RME_ERR_PTH_REFCNT	Virtual CPUs still exist in this virtual machine, so it cannot be deleted.

8.8.3 Setting the Page Table Used by a Virtual Machine

A virtual machine requires a page table when running, which is responsible for mapping the real addresses of this machine to the real addresses of the virtual machine. When the virtual machine is running, this page table will be loaded. This operation is responsible for setting the page table of the virtual machine.

Table 8-77 Parameters Needed for Setting the Page Table Used by a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VM_PGT.
Cap_VM	cid_t	Sub_ID	A capability number corresponding to a virtual machine; the capability corresponding to this capability number points to the virtual machine whose page table[30] is to be modified. This capability number can be a one-level or two-level lookup encoding.

[30] top-level page directory

Parameter name	Type	Location	Description
Cap_Pgt	cid_t	Param1	A capability number corresponding to a page directory capability that must have the RME_PGT_FLAG_PRC_PGT attribute; the capability corresponding to this capability number points to the page table[31] to be used by the virtual machine. This capability number can be a one-level or two-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-78 Recommended Return Value Implementation for Setting the Page Table Used by a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VM is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VM has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has already been frozen.
RME_ERR_CPT_TYPE	Cap_VM is not a virtual machine capability.
RME_ERR_CPT_TYPE	Cap_Pgt is not a page table capability.
RME_ERR_CPT_NULL	Cap_VM or Cap_Pgt is a blank capability.
RME_ERR_CPT_FLAG	Cap_Pgt does not have the RME_PGT_FLAG_PRC_PGT attribute.
RME_ERR_CPT_REFCNT	The reference count of Cap_Pgt exceeds the maximum range allowed by the system.
RME_ERR_PTH_CONFLICT	Two cores simultaneously attempt to set the page table; in this case, the core that did not succeed returns this value.

8.8.4 Getting or Setting the Attributes of a Virtual Machine

On some architectures, a virtual machine has some state registers that act on the overall environment and need to be set. This operation is responsible for setting these registers that may exist.

Table 8-79 Parameters Needed for Getting or Setting the Attributes of a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VM_MOD.

[31] top-level page directory

Parameter name	Type	Location	Description
Cap_VM	cid_t	Sub_ID	A capability number corresponding to a virtual machine; the capability corresponding to this capability number points to the virtual machine whose registers are to be read or modified. This capability number can be a one-level or two-level lookup encoding.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set. Some registers are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-80 Recommended Return Value Implementation for Getting or Setting the Attributes of a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VM is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VM has already been frozen.
RME_ERR_CPT_TYPE	Cap_VM is not a virtual machine capability, or is a blank capability.
RME_ERR_PGT_HW	Due to hardware constraints, the operation cannot be completed, or the operation is invalid[32].

8.8.5 Creating a Virtual CPU

On a virtual machine, virtual CPUs are the carriers for running the programs of the virtual machine itself. This operation is responsible for creating these virtual CPUs that are essential for running.

Table 8-81 Parameters Needed for Creating a Virtual CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_CRT.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created virtual CPU capability. This capability number can be a one-level or two-level lookup encoding.

[32] top-level page directory

Parameter name	Type	Location	Description
Cap_Kom	cid_t	Param1.D1	A kernel memory capability number whose identified kernel memory range must be able to hold the entire virtual CPU kernel object. This capability number can be a one-level or two-level lookup encoding. This kernel function does not check the operation flag bits of the kernel memory capability, and this operation is permitted by default.
Cap_VCPU	cid_t	Param1.D0	A capability number corresponding to a position in the capability table that is to receive this newly created virtual CPU capability. The capability corresponding to this capability number must be blank. This capability number can only be a one-level lookup encoding.
Raddr	ptr_t	Param2	The starting relative virtual address in kernel space to be used by the newly created virtual CPU. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.

It is recommended to implement the return values of this operation as follows:

Table 8-82 Recommended Return Value Implementation for Creating a Virtual CPU

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_VCPU is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has already been frozen.
RME_ERR_CPT_FROZEN	Cap_VCPU has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	The range of Cap_Kom is incorrect.
RME_ERR_CPT_EXIST	Cap_VCPU is not a blank capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_PGT_HW	Due to underlying hardware constraints, the virtual CPU kernel object cannot be created at this memory address.

8.8.6 Binding a Virtual CPU to a Virtual Machine

A virtual CPU must be bound into a virtual machine before it can participate in the operation of the virtual machine. A virtual machine must first have a page table bound before a virtual CPU can be bound to it. This operation binds a virtual CPU to a virtual machine.

Table 8-83 Parameters Needed for Binding a Virtual CPU to a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_BIND.
Cap_VCPU	cid_t	Sub_ID	A capability number corresponding to a virtual CPU capability; the capability corresponding to this capability number points to the virtual CPU to be bound. This capability number can be a one-level or two-level lookup encoding.
Cap_VM	cid_t	Param1	A capability number corresponding to a virtual machine capability; the capability corresponding to this capability number points to the target virtual machine. This capability number can be a one-level or two-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-84 Recommended Return Value Implementation for Binding a Virtual CPU to a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VCPU or Cap_VM is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VCPU or Cap_VM has already been frozen.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual CPU capability, or is a blank capability.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual machine capability, or is a blank capability.
RME_ERR_PTH_INVSTATE	Cap_VCPU is in the bound state.
RME_ERR_PTH_INVSTATE	The virtual machine corresponding to this virtual CPU has no page table, so binding cannot be performed.
RME_ERR_PTH_CONFLICT	If two CPUs attempt to bind at the same time, this value is returned on the CPU that failed.

8.8.7 Unbinding a Virtual CPU from a Virtual Machine

A virtual CPU can be unbound from a virtual machine. When the binding is released, the virtual CPU must not be running. This operation is responsible for unbinding the virtual CPU.

Table 8-85 Parameters Needed for Unbinding a Virtual CPU from a Virtual Machine

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_FREE.
Cap_VCPU	cid_t	Sub_ID	A capability number corresponding to a virtual CPU capability; the capability corresponding to this capability number points to the virtual CPU to be unbound. This capability number can be a one-level or two-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-86 Recommended Return Value Implementation for Unbinding a Virtual CPU from a Virtual Machine

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VCPU is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VCPU has already been frozen.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual CPU capability, or is a blank capability.
RME_ERR_PTH_REFCNT	Cap_VCPU is still running, so it cannot be unbound.
RME_ERR_PTH_INVSTATE	Cap_VCPU is in the unbound state.

8.8.8 Deleting a Virtual CPU

This operation deletes virtual CPUs. These virtual CPUs must have been created by RME_KFN_VCPU_CRT and must not currently be bound to any virtual machine.

Table 8-87 Parameters Needed for Deleting a Virtual CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_DEL.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table that contains the virtual CPU capability being deleted. This capability number can be a one-level or two-level lookup encoding.

Parameter name	Type	Location	Description
Cap_VCPU	cid_t	Param1	A capability number corresponding to the virtual CPU capability to be deleted. The capability corresponding to this capability number must be a virtual CPU capability. This capability number can only be a one-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-88 Recommended Return Value Implementation for Deleting a Virtual CPU

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_VCPU is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	Cap_VCPU has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual CPU capability.
RME_ERR_CPT_NULL	Cap_VCPU is a blank capability.
RME_ERR_CPT_NULL	Two cores simultaneously attempt to delete this thread; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_VCPU is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_VCPU is not 0, or it is not a root capability.
RME_ERR_PTH_INVSTATE	The virtual CPU is still in the bound state.

8.8.9 Getting or Setting the Registers of a Virtual CPU

A virtual CPU generally has some registers. These registers may be the processor's own working registers, coprocessor registers, or registers of other working modes, and so on. This operation is responsible for getting and setting these registers.

Table 8-89 Parameters Needed for Getting or Setting the Registers of a Virtual CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_MOD.
Cap_VCPU	cid_t	Sub_ID	A capability number corresponding to a virtual CPU; the capability corresponding to this capability number points to the virtual CPU whose registers are to be read or modified. This capability number can be a one-level or two-level lookup encoding.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.

Parameter name	Type	Location	Description
Value	ptr_t	Param2	The value to be set. Some registers are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-90 Recommended Return Value Implementation for Getting or Setting the Registers of a Virtual CPU

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VCPU is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VCPU has already been frozen.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual machine capability, or is a blank capability.
RME_ERR_PGT_HW	Due to hardware constraints, the operation cannot be completed, or the operation is invalid.

8.8.10 Running a Virtual CPU

This operation is responsible for running a virtual CPU. After a thread calls this function, it executes the functionality of this virtual CPU. During runtime, each virtual CPU corresponds to an active thread. In addition, this thread does not use the page table of the process it originally belongs to in order to run this virtual CPU; instead, it uses the page table of the virtual machine to run this virtual CPU.

Table 8-91 Parameters Needed for Running a Virtual CPU

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_VCPU_RUN.
Cap_VCPU	cid_t	Sub_ID	A capability number corresponding to a virtual CPU capability; the capability corresponding to this capability number points to the virtual CPU to be run. This capability number can be a one-level or two-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-92 Recommended Return Value Implementation for Running a Virtual CPU

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the user.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_VCPU is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_VCPU has already been frozen.
RME_ERR_CPT_TYPE	Cap_VCPU is not a virtual CPU capability, or is a blank capability.
RME_ERR_PTH_REFCNT	Cap_VCPU is already running on another CPU.
RME_ERR_PTH_INVSTATE	Cap_VCPU is in the unbound state.

8.9 Security Monitor Kernel Call Implementation Specifications (“S” Standard Extension)

Security monitors (Security monitor) play an important role in modern information security. They can often place a small part of the application code inside a secure zone (Enclave), thereby isolating it from the vast majority of code—even privileged code such as the operating system and the virtual machine monitor—greatly reducing the software’s attack surface (Attack surface). Although it cannot guarantee the integrity (Integrity) and availability (Availability) of an application, it can ensure confidentiality (Confidentiality) to a considerable extent.

The secure/non-secure memory partitioning of a security monitor is always determined once and for all at system manufacturing or boot time: one part of the physical memory address space is designated as secure space, and another part of the physical memory address space is designated as non-secure space. Throughout the entire operation of the system, the secure space is absolutely never accessed at will by the non-secure space. Some security monitor implementations allow these memories to be partitioned dynamically at runtime, but this is a very dangerous behavior, and the current RME specification does not support this feature.

Common security monitors include Intel’s Software Guard Extension (SGX), AMD’s Secure Encrypted Virtualization (SEV), ARM’s TrustZone[33], and RISC-V’s Sanctum and Keystone. All these security monitors can create a small secure zone and guarantee the security of that secure zone.

The goal of this specification is to unify the operations of these security monitors using the same or similar interfaces. Because most security monitors require the creation of some additional data structures, the porter needs to allocate these kernel objects themselves. The underlying logic of these kernel objects and the definitions of the concrete capability structures also need to be completed by the porter. These kernel objects and capabilities themselves reside in the kernel mode of the non-secure zone, so the secure zone logic still needs to be responsible for the confidentiality of the secure zone itself when the kernel is compromised[34].

[33] which can be divided into two subcategories, used for Cortex-A and Cortex-M respectively

[34] At this point, integrity and availability cannot possibly be guaranteed, because the kernel may already have been compromised

The 7 security monitor kernel calls specified by RME are all optional to implement, and their list is as follows:

Table 8-93 Overview of Security Monitor Kernel Calls

Main function number	Call name	Function
0xF700	RME_KFN_ECLV_CRT	Creating a secure zone.
0xF701	RME_KFN_ECLV_DEL	Deleting a secure zone.
0xF702	RME_KFN_ECLV_MOD	Getting or setting the attributes of a secure zone.
0xF703	RME_KFN_ETHD_CRT	Creating a secure zone thread.
0xF704	RME_KFN_ETHD_DEL	Deleting a secure zone thread.
0xF705	RME_KFN_ETHD_MOD	Getting or setting the attributes of a secure zone thread.
0xF706	RME_KFN_ETHD_ACT	Running a secure zone thread.

8.9.1 Two Typical Security Monitor Implementations

Security monitors can be divided into two types: the first is the region type, that is, the system designates a memory region as the secure state, and accessing this secure-state memory must go through a specific interface[35]; the second is the partition type, that is, the system has a secure mode, and accessing resources in the secure mode must go through a software interrupt to switch to the secure state[36]. The second type of security monitor sometimes shares a set of underlying hardware mechanisms with virtualization extensions. Next, we describe the methods for adapting the security monitor kernel call interface under these two different hardware implementations.

8.9.1.1 Region Type

The adaptation of region-type monitors is often relatively easy, because they directly create secure regions within a process. In this case, it is only necessary to make a one-to-one correspondence between the RME interface and the functions provided by the hardware. Under this type of monitor, crossing protection domains often involves dedicated hardware instructions[37], so RME_KFN_ECLV_ACT does not need to be implemented. This type of security monitor usually allows code inside the secure zone to directly access the non-secure region, although these accesses are not guaranteed to be possible under a specific RME implementation.

RME_KFN_ECLV_CRT, RME_KFN_ECLV_DEL, and RME_KFN_ECLV_MOD must be implemented on this type of security monitor; RME_KFN_ETHD_CRT, RME_KFN_ETHD_DEL, and RME_KFN_ETHD_MOD sometimes need to be implemented; and RME_KFN_ETHD_ACT need not be implemented in the vast majority of cases.

8.9.1.2 Partition Type

The adaptation of partition-type monitors is often more complicated and involves many state transitions. The usage overhead of this type of monitor is often also relatively large, and the secure state also has some code logic responsible for additional privilege checks. The operations of creating, deleting, and modifying a secure zone will first jump to the kernel non-secure state and complete basic checks, then switch to the kernel secure state through a dedicated switching instruction and be subject to further privilege checks by the kernel secure state, and only then will they be executed. The operations of entering and exiting the secure zone are often also relatively complex, so RME_KFN_ECLV_ACT and RME_KFN_ECLV_RET

[35] such as Intel SGX and ARMv8-M TrustZone
[36] such as AMD SEV and ARMv8-A TrustZone
[37] such as EENTER and EEXIT of Intel SGX and SG and BXNS (BLXNS) of ARMv8-M

need to be implemented to achieve explicit jumps. This type of security monitor usually does not allow code inside the secure zone to directly access the non-secure region.

On this type of security monitor, all 7 calls must be implemented.

8.9.2 Creating a Secure Zone

Before using the security monitor extension, a secure zone must first be created using this call. Any other operation is allowed to be executed only after the secure zone has been created.

Table 8-94 Parameters Needed for Creating a Secure Zone

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ECLV_CRT.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created secure zone capability. This capability number can be a one-level or two-level lookup encoding.
Cap_Kom	cid_t	Param1.D1	A kernel memory capability number whose identified kernel memory range must be able to hold the entire secure zone kernel object. This capability number can be a one-level or two-level lookup encoding. This kernel function does not check the operation flag bits of the kernel memory capability, and this operation is permitted by default.
Cap_ECLV	cid_t	Param1.D0	A capability number corresponding to a position in the capability table that is to receive this newly created secure zone capability. The capability corresponding to this capability number must be blank. This capability number can only be a one-level lookup encoding.
Raddr	ptr_t	Param2	The starting relative virtual address in kernel space to be used by the newly created secure zone. This address is relative to the starting virtual address of the kernel capability corresponding to Cap_Kom.

It is recommended to implement the return values of this operation as follows:

Table 8-95 Recommended Return Value Implementation for Creating a Secure Zone

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Kom is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_ECLV is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Kom has already been frozen.
RME_ERR_CPT_FROZEN	Cap_ECLV has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_Kom is not a kernel memory capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	The range of Cap_Kom is incorrect.
RME_ERR_CPT_EXIST	Cap_ECLV is not a blank capability.
RME_ERR_CPT_KOT	Failed to allocate kernel memory.
RME_ERR_PGT_HW	Due to underlying hardware constraints, the secure zone kernel object cannot be created.

8.9.3 Deleting a Secure Zone

This operation deletes a secure zone. The secure zone must have been created by RME_KFN_ECLV_CRT and must currently be in an unused state.

Table 8-96 Parameters Needed for Deleting a Secure Zone

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ECLV_DEL.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table that contains the secure zone capability being deleted. This capability number can be a one-level or two-level lookup encoding.
Cap_ECLV	cid_t	Param1	A capability number corresponding to the secure zone capability to be deleted. The capability corresponding to this capability number must be a secure zone capability. This capability number can only be a one-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-97 Recommended Return Value Implementation for Deleting a Secure Zone

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_ECLV is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	Cap_ECLV has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_ECLV is not a secure zone capability.
RME_ERR_CPT_NULL	Cap_ECLV is a blank capability.
RME_ERR_CPT_NULL	Two cores simultaneously attempt to delete this secure zone; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_ECLV is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_ECLV is not 0, or it is not a root capability.
RME_ERR_PTH_REFCNT	The secure zone cannot be deleted for some reason.

8.9.4 Getting or Setting the Attributes of a Secure Zone

A secure zone usually has some attributes that need to be set after it is created. These attributes usually include the secure zone's memory allocation, initial contents, number of threads, final sealing, and so on. This operation is responsible for setting these attributes where they are present. Because some secure zone attributes may be long, this operation may need to use additional registers to pass in extra parameters. The method of passing these parameters can be customized for each architecture.

Table 8-98 Parameters Needed for Getting or Setting the Attributes of a Secure Zone

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ECLV_MOD.
Cap_ECLV	cid_t	Sub_ID	A capability number corresponding to the secure zone; the capability corresponding to this capability number points to the secure zone whose attributes are to be read or modified. This capability number can be a one-level or two-level lookup encoding.
Value1	ptr_t	Param1	The meaning is determined by the implementer.
Value2	ptr_t	Param2	The meaning is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-99 Recommended Return Value Implementation for Getting or Setting the Attributes of a Secure Zone

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_ECLV is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_ECLV has already been frozen.
RME_ERR_CPT_TYPE	Cap_ECLV is not a secure zone capability, or is a blank capability.
RME_ERR_PGT_HW	The operation cannot be completed due to hardware constraints, or the operation is invalid.

8.9.5 Creating a Secure Zone Thread

In some security monitor implementations, secure zone threads must be created in advance before they can be executed. This operation is used to create these secure zone threads. Because secure zone threads are located entirely within the secure zone, they do not occupy RME's kernel memory. Note that some security monitors may allow new threads to be created within a secure zone after the secure zone has entered the in-use state.

Table 8-100 Parameters Needed for Creating a Secure Zone Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ETHD_CRT.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_CRT attribute; the capability corresponding to this capability number points to the capability table that is to receive this newly created virtual CPU capability. This capability number can be a one-level or two-level lookup encoding.
Cap_ECLV	cid_t	Param1.D1	A capability number corresponding to a secure zone capability; the capability corresponding to this capability number points to the secure zone that is to contain this thread. This capability number can be a one-level or two-level lookup encoding.
Cap_ETHD	cid_t	Param1.D0	A capability number corresponding to a position in the capability table that is to receive the newly created secure zone thread capability. The capability corresponding to this capability number must be blank. This capability number can only be a one-level lookup encoding.

Parameter name	Type	Location	Description
Param	ptr_t	Param2	The meaning is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-101 Recommended Return Value Implementation for Creating a Secure Zone Thread

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_ECLV is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_ETHD is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_ECLV has already been frozen.
RME_ERR_CPT_FROZEN	Cap_ETHD has been frozen, or another core is creating a capability there.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_ECLV is not a secure zone capability.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_CRT attribute.
RME_ERR_CPT_FLAG	The range of Cap_Kom is incorrect.
RME_ERR_CPT_EXIST	Cap_ETHD is not a blank capability.
RME_ERR_PGT_HW	Due to underlying hardware constraints, the secure zone thread object cannot be created.

8.9.6 Deleting a Secure Zone Thread

This operation deletes secure zone threads. These secure zone threads must have been created by RME_KFN_ETHD_CRT and must not currently be in use. Note that some security monitors may not allow threads inside a secure zone to be deleted before the secure zone's in-use state is released.

Table 8-102 Parameters Needed for Deleting a Secure Zone Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ETHD_DEL.
Cap_Cpt	cid_t	Sub_ID	A capability number corresponding to a capability table capability that must have the RME_CPT_FLAG_DEL attribute; the capability corresponding to this capability number points to the capability table containing the secure zone thread capability that is being deleted. This capability number can be a one-level or two-level lookup encoding.

Parameter name	Type	Location	Description
Cap_ETHD	cid_t	Param1	A capability number corresponding to the secure zone thread capability to be deleted. The capability corresponding to this capability number must be a secure zone thread capability. This capability number can only be a one-level lookup encoding.

It is recommended to implement the return values of this operation as follows:

Table 8-103 Recommended Return Value Implementation for Deleting a Secure Zone Thread

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Cpt is out of range.
RME_ERR_CPT_RANGE	The one-level lookup of Cap_ETHD is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Cpt has already been frozen.
RME_ERR_CPT_FROZEN	Cap_ETHD has not been frozen.
RME_ERR_CPT_TYPE	Cap_Cpt is not a capability table capability.
RME_ERR_CPT_TYPE	Cap_ETHD is not a secure zone thread capability.
RME_ERR_CPT_NULL	Cap_ETHD is a blank capability.
RME_ERR_CPT_NULL	Two cores simultaneously attempt to delete this thread; in this case, the core that did not succeed returns this value.
RME_ERR_CPT_FLAG	Cap_Cpt does not have the RME_CPT_FLAG_DEL attribute.
RME_ERR_CPT_QUIE	Cap_ETHD is not quiescent.
RME_ERR_CPT_REFCNT	The reference count of Cap_ETHD is not 0, or it is not a root capability.
RME_ERR_PTH_INVSTATE	The secure zone thread cannot be deleted due to certain hardware constraints.

8.9.7 Getting or Setting the Attributes of a Secure Zone Thread

Secure zone threads generally have some attributes that need to be configured, namely their stack addresses, entry points, and so on. This operation is responsible for getting or setting these attributes.

Table 8-104 Parameters Needed for Getting or Setting the Attributes of a Secure Zone Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ETHD_MOD.
Cap_ETHD	cid_t	Sub_ID	A capability number corresponding to a secure zone thread; the capability corresponding to this capability number points to the secure zone thread whose attributes are to be read or modified. This capability number can be a one-level or two-level lookup encoding.

Parameter name	Type	Location	Description
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set. Some attributes are read-only; in this case, set operations are not allowed and only read operations are permitted.

It is recommended to implement the return values of this operation as follows:

Table 8-105 Recommended Return Value Implementation for Getting or Setting the Attributes of a Secure Zone Thread

Return value	Meaning
0	The operation succeeded.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_ETHD is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_ETHD has already been frozen.
RME_ERR_CPT_TYPE	Cap_ETHD is not a secure zone thread capability, or is a blank capability.
RME_ERR_PGT_HW	The operation cannot be completed due to hardware constraints, or the operation is invalid.

8.9.8 Running a Secure Zone Thread

This operation is responsible for jumping to a thread in the secure zone to continue its execution. This call only needs to be implemented on partition-type security monitors. If the user wishes to pass extra parameters to the secure zone or return extra parameters from the secure zone, the method of passing them is determined by the user.

Table 8-106 Parameters Needed for Running a Secure Zone Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_ETHD_RUN.
Cap_ETHD	cid_t	Sub_ID	A capability number corresponding to a virtual CPU capability; the capability corresponding to this capability number points to the virtual CPU to be run. This capability number can be a one-level or two-level lookup encoding.
Param1	ptr_t	Param1	The meaning is determined by the implementer.
Param2	ptr_t	Param2	The meaning is determined by the implementer.

It is recommended to implement the return values of this operation as follows:

Table 8-107 Recommended Return Value Implementation for Running a Secure Zone Thread

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the user.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_ETHD is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_ETHD has already been frozen.
RME_ERR_CPT_TYPE	Cap_ETHD is not a secure zone thread capability, or is a blank capability.
RME_ERR_PTH_REFCNT	Cap_ETHD is already running on another thread.
RME_ERR_PTH_INVSTATE	The secure zone containing Cap_ETHD is not yet ready for execution.

8.10 Debug and Trace Kernel Call Implementation Specifications ("D" Standard Extension)

Debugging and tracing are the basic means of program debugging and analysis. RME specifies the following 5 debug and trace kernel calls in total, all of which are optional to implement:

Table 8-108 Overview of Debug and Trace Kernel Calls

Main function number	Call name	Purpose
0xF800	RME_KFN_DEBUG_PRINT	Debug printing.
0xF801	RME_KFN_DEBUG_REG_MOD	Get or set the register set of a thread.
0xF802	RME_KFN_DEBUG_INV_MOD	Get or set the invocation register set of a thread.
0xF803	RME_KFN_DEBUG_MODE_MOD	Get or set the debug engine mode.
0xF804	RME_KFN_DEBUG_IBP_MOD	Get or set the instruction debug breakpoint state.
0xF805	RME_KFN_DEBUG_DBP_MOD	Get or set the data debug breakpoint state.

8.10.1 Debug Printing

Debug printing is the most basic program analysis tool. For this purpose, RME provides a dedicated kernel call to implement console character printing. Generally speaking, this call prints one character to the serial port and does not return until printing is complete. This call can also be implemented to print multiple characters; in this case, the extra character parameters need to be passed in by the implementer using free registers.

Table 8-109 Parameters Needed for Debug Printing

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_PRINT.
Char	char	Sub_ID	The character to be printed.

It is recommended to implement the return values of this operation as follows:

Table 8-110 Recommended Return Value Implementation for Debug Printing

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.10.2 Getting or Setting the Register Set of a Thread

Reading and modifying the register set of a thread is essential for analyzing the execution status of a program, so RME provides a kernel call to handle such operations. This call can also be used to change the execution flow of a program or to modify its coprocessor state. This kernel call is responsible for getting or setting the current register set of a thread. Note that RME kernel calls implement only two parameters and one return value by default, while the register set of a thread is often very large. To improve read/write efficiency, the user should use other registers to implement the passing of more parameters and return values themselves.

Table 8-111 Parameters Needed for Getting or Setting the Register Set of a Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_REG_MOD.
Cap_Thd	cid_t	Sub_ID	A capability number corresponding to the thread capability whose register set contents are to be read or set. This capability number can be a one-level or two-level lookup encoding.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Value	ptr_t	Param2	The value to be set to this register.

It is recommended to implement the return values of this operation as follows:

Table 8-112 Recommended Return Value Implementation for Getting or Setting the Register Set of a Thread

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has already been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is a blank capability.
RME_ERR_PTH_INVSTATE	The specified register does not exist or is not allowed to be read or written, or the value attempted to be written is invalid.

8.10.3 Getting or Setting the Invocation Register Set of a Thread

Reading and modifying the register set saved in an invocation can cause a thread, when returning from the invocation, to return not to the place where the invocation was originally initiated, but to another address to continue execution. This is very useful when implementing certain functions such as para-virtualization, so RME has reserved a dedicated kernel call for this function. Although the register set saved by an invocation is only a small subset of the register set of a thread, to improve read/write efficiency, the user should use other registers to implement the passing of more parameters and return values themselves.

Table 8-113 Parameters Needed for Getting or Setting the Invocation Register Set of a Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_INV_MOD.
Cap_Thd	cid_t	Sub_ID	A capability number corresponding to the thread capability whose register set contents are to be read or set. This capability number can be a one-level or two-level lookup encoding.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Reg	ptr_t	Param2	The value to be set to this register.

It is recommended to implement the return values of this operation as follows:

Table 8-114 Recommended Return Value Implementation for Getting or Setting the Invocation Register Set of a Thread

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has already been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is a blank capability.

Return value	Meaning
RME_ERR_SIV_EMPTY	The specified thread is not in an invocation, so its invocation register set cannot be read or written.
RME_ERR_PTH_INVSTATE	The specified register does not exist or is not allowed to be read or written, or the value attempted to be written is invalid.

8.10.4 Getting the Exception Register Set of a Thread

When an exception occurs in a thread, it is often necessary to read information such as the cause of the exception and the address at which the exception occurred, so as to complete exception handling. This kernel call is responsible for getting this exception information. The amount and type of information required for exception handling may differ for each architecture, so the user may need to use other registers to implement the passing of more parameters and return values themselves.

Table 8-115 Parameters Needed for Getting the Exception Register Set of a Thread

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_EXC_GET.
Cap_Thd	cid_t	Sub_ID	A capability number corresponding to the thread capability whose register set contents are to be read. This capability number can be a one-level or two-level lookup encoding.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.

It is recommended to implement the return values of this operation as follows:

Table 8-116 Recommended Return Value Implementation for Getting the Exception Register Set of a Thread

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Thd is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Thd has already been frozen.
RME_ERR_CPT_TYPE	Cap_Thd is not a thread capability, or is a blank capability.
RME_ERR_PTH_INVSTATE	The specified register does not exist or is not allowed to be read.

8.10.5 Get or Set Debug Mode

The debug engine module of a processor often includes some registers, through which the debug mode and state of the processor can be set. This kernel call is responsible for getting or setting these registers.

Table 8-117 Parameters needed for getting or setting the debug mode

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_MODE_MOD.
Operation	ptr_t	Sub_ID	Operation option, which determines the operation to be performed.
Param1	ptr_t	Param1	Extra parameter. The meaning of this parameter is determined by the implementer.
Param2	ptr_t	Param2	Extra parameter. The meaning of this parameter is determined by the implementer.

The return values of this operation are suggested to be implemented as follows:

Table 8-118 Suggested return value implementation for getting or setting the debug mode

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
Negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.10.6 Get or Set Instruction Debug Breakpoint State

Some processors may support hardware instruction breakpoints. When the processor executes to the instruction address corresponding to a hardware instruction breakpoint, a debug interrupt is generated, and the debugger can learn that the program has run to this point by intercepting this interrupt. This kernel call is responsible for driving the relevant hardware to get and set hardware instruction breakpoints and their states. The breakpoint functionality of some processors may support ASID, in which case a top-level page directory capability needs to be passed in additionally.

Table 8-119 Parameters needed for getting or setting the instruction debug breakpoint state

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_IBP_MOD.
Cap_Pgt	cid_t	Sub_ID	The ASID of this instruction breakpoint.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Address	ptr_t	Param2	The address at which the instruction breakpoint is to be set.

The return values of this operation are suggested to be implemented as follows:

Table 8-120 Suggested return value implementation for getting or setting the instruction debug breakpoint state

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.
Other negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.10.7 Get or Set Data Debug Breakpoint State

Some processors may support hardware data breakpoints. When the processor writes to or reads from a data address, a debug interrupt is generated, and the debugger can learn that the program is accessing a certain variable in that memory by intercepting this interrupt. This kernel call is responsible for driving the relevant hardware to get and set hardware data breakpoints and their states. The breakpoint functionality of some processors may support ASID, in which case a top-level page directory capability needs to be passed in additionally. The breakpoint functionality of some processors may support ASID, in which case a top-level page directory capability needs to be passed in additionally.

Table 8-121 Parameters needed for getting or setting the data debug breakpoint state

Parameter name	Type	Location	Description
Func_Num	ptr_t	Func_ID	Must be RME_KFN_DEBUG_DBP_MOD.
Cap_Pgt	cid_t	Sub_ID	The ASID of this data breakpoint.
Operation	ptr_t	Param1	Operation option, which determines the operation to be performed.
Address	ptr_t	Param2	The address at which the data breakpoint is to be set.

The return values of this operation are suggested to be implemented as follows:

Table 8-122 Suggested return value implementation for getting or setting the data debug breakpoint state

Return value	Meaning
Non-negative value	The operation succeeded; the meaning of the return value is determined by the implementer.
RME_ERR_CPT_RANGE	The one-level/two-level lookup of Cap_Pgt is out of range.
RME_ERR_CPT_FROZEN	The capability of the one-level/two-level lookup of Cap_Pgt has been frozen.
RME_ERR_CPT_TYPE	Cap_Pgt is not a top-level page directory capability, or is an empty capability.
Other negative value	The operation cannot be completed for hardware reasons; the meaning of the return value is determined by the implementer.

8.11 Chapter Bibliography

None

9. Appendix

9.1 Implementation of Special Kernel Features

RME can support many special features provided by certain other operating systems, such as CPU hotplug, memory hotplug, the separation kernel, para-virtualization, and so on. Their implementation ideas are briefly described below.

9.1.1 Implementation of CPU Hotplug

CPU hotplug is divided into two functions. One is hot plug-in, that is, inserting a new CPU to increase the number of CPUs; the other is hot plug-out, that is, pulling a CPU out of its socket to decrease the number of CPUs. RME's support for this feature depends on the hardware-level support provided by the underlying hardware platform, and both must be accomplished with customized kernel function calls.

For hot plug-in, after the insertion of a physical CPU package is detected, the processor can be initialized and the required Init threads can be created, after which these processor cores can be used.

Hot plug-out is the opposite: the activities of the corresponding processor must be stopped, and the other processor cores must no longer send IPIs to that processor. Only then can we remove this physical CPU package.

9.1.2 Implementation of Memory Hotplug

Memory hotplug is likewise divided into two parts: one part is adding memory, and the other part is removing memory. RME's support for this feature depends on the hardware-level support provided by the underlying hardware platform, and it too must be accomplished with customized kernel function calls.

In the case of adding memory, if user-mode memory is being added, it suffices to add the physical page frames corresponding to this memory to some user page table. If kernel memory needs to be added, then the other processors must first be paused, after which new pages are added to the kernel page table and the kernel page table is modified to reflect this change, and finally the other processors are resumed. After all operations are completed, a TLB flush operation must also be performed.

In the case of removing memory, if user-mode memory is being removed, the user-mode library must confirm that these physical pages are not currently mapped. At this point, the mappings of these pages can be removed directly and the memory module can be pulled out. If kernel memory is being removed, then the other processors must first be paused, after which the data in the memory to be removed is copied onto blank physical pages and the kernel page table is modified to reflect this change, and finally the other processors are resumed. After all operations are completed, a TLB flush operation must also be performed.

9.1.3 Implementation of the Separation Kernel

Because RME is a microkernel operating system, it is very convenient, just like Barrelfish[1], to run multiple instances of RME on a single physical machine. Each instance can manage one or more CPU cores, and communication can then be implemented in user mode through shared memory or a network among

the multiple operating systems. This approach does not require the caches among CPUs to be coherent, does not require the features contained in each CPU to be the same, and does not even require the instruction sets and architectures of the CPUs to be the same. If processors of multiple architectures are used in such a system, then RME and the user-mode library need to be ported separately for them.

For example, if there is a SoC or single-board computer as shown in the figure below[1], then it is possible to run RME on all of the cores, and then accomplish information transfer through shared memory among the different subsystems.

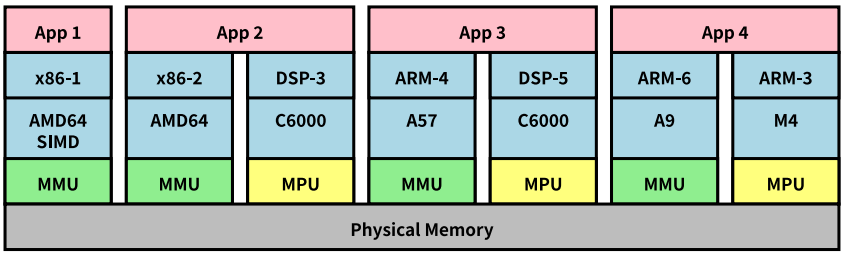


Figure 9-1 Typical Implementation of a Heterogeneous Multi-core Separation Kernel

The figure above shows an example in which different architectures run together. It is also feasible to run one instance of RME on each different NUMA node, as shown in the following figure:

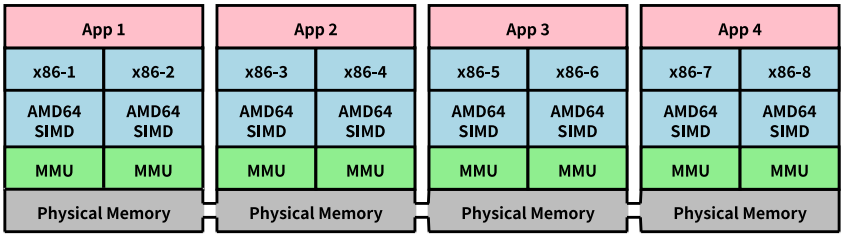


Figure 9-2 Typical Implementation Scheme of a Separation Kernel Based on NUMA

9.1.4 Implementation of a Tickless Kernel

A tickless kernel requires that every CPU core of the system have an independently programmable high-precision timer, and that this high-precision timer generate a variable scheduler timer interrupt. The ratio between a timeslice and the timer reload value can be freely decided by the implementer as needed. To facilitate the implementation of a tickless kernel, the system provides two functions, _RME_Tim_Future and _RME_Tim_Elapse; they may be ignored if a tickless kernel is not implemented.

In order for the timer to generate an interrupt at the potential next timeout point, after every change of the remaining timeslice of the current thread[2], this time must be set as the next timeout time of the hardware timer. Three situations may cause the remaining timeslice to change: (1) a context switch caused by any reason[3], (2) RME_SVC_THD_TIME_XFER is used in the system call vector and the timeslice of the current thread changes as a result, and (3) a timer interrupt occurs[4]. On some architectures, reprogramming the timer may take a relatively long time, and the occurrence of an interrupt does not necessarily cause the remaining timeslice to change, so one may query and determine the change of the current

[1] HMMU refers to a hardware-filled MMU, and SMMU refers to a software-filled MMU

[2] such as at the exit of the various interrupt vectors

[3] There is also the coincidence that the remaining timeslices of the threads before and after the context switch are exactly the same; in this case the timer does not need to be reprogrammed

[4] The occurrence of this interrupt does not necessarily mean that a thread has exhausted its own timeslice; it may also mean that the longest configurable timeout of the hardware timer is not long enough, so that one timing period must be completed in several segments

thread's timeslice before and after entering the interrupt by oneself to decide whether the timer needs to be reprogrammed; the timer interrupt is a special case, since it definitely requires one timer reprogramming, and thus this query can be omitted.

The kernel provides the function _RME_Tim_Future to obtain this timeout time. If the length of this period exceeds the range that the hardware timer can support, it suffices to set the timeout time of the hardware timer to the maximum allowed value; the design of RME guarantees that only the time that has actually elapsed is deducted from the remaining time of the thread, so there is no need to worry about the loss of the thread's remaining time.

Table 9-1 Obtaining the Number of Timeslices Until the Nearest Timeout

Prototype	rme_ptr_t _RME_Tim_Future(void)
Meaning	The number of timeslices from now to the nearest future timeout; the next timeout time of the system timer can be set according to this value.
Return value	rme_ptr_t The number of timeslices from now to the nearest future timeout.
Parameters	None.

In order for the time consumed by all threads to be precisely recorded, the time already consumed must be deducted from the current thread before every point at which a context switch may occur or the timeslice may be modified[5]. The kernel provides the function _RME_Tim_Elapse for this purpose to record the consumed time.

Table 9-2 Recording the Elapsed Time

Prototype	void _RME_Tim_Elapse(rme_ptr_t Slice)
Meaning	Records the number of timeslices already used by the current thread before initiating a potential context switch.
Return value	None.
Parameters	rme_ptr_t Slice The number of timeslices that have elapsed since the last call to _RME_Tim_Elapse or _RME_Tim_Handler.

It is worth noting that although a tickless kernel has lower CPU power consumption, its real-time performance is not necessarily better, because it inevitably includes the registration of elapsed time before a context switch and the reprogramming of the timer after a context switch. Both may increase the length of the context switch critical section and negatively affect the real-time performance of the system. Moreover, the signal that the kernel sends to the timer endpoint will also lose its periodicity, and user mode can no longer rely on it.

When a tickless kernel is used, the RME_TIMESTAMP macro must be defined as a read of some independent one-way incrementing counter[6], which makes the timestamp the globally latest value at all times, thereby guaranteeing the safety of the quiescence period of capability operations.

[5] At the entry of every interrupt vector except the timer interrupt (which calls _RME_Tim_Handler); strictly speaking, if it can be determined that this interrupt does not cause a thread switch and does not modify the timeslice of the current thread through RME_SVC_THD_TIME_XFER, then there is no need to call _RME_Tim_Elapse, but determining whether a thread switch occurs is very difficult, and the overhead of _RME_Tim_Elapse is very low, so it is recommended to call it at the entry of all interrupt vectors

[6] Such a counter can usually be found in the CPU performance monitoring unit (Performance Monitoring Unit, PMU) or in a global timer; some architectures such as x86 also provide a dedicated RDTSC instruction that increments constantly (CONSTANT_TSC) and never stops (NONSTOP_TSC)

9.1.5 Implementation of Processor Sleep

Processor sleep is notoriously difficult to implement correctly; a careless design may in mild cases cause lost wakeups and system stalls, and in severe cases cause permanent sleep and a machine freeze. The definitions of the sleep instruction adopted by processors can be divided into the "look-before-and-after" type, the "after-only" type, and the "trigger-only" type, and the three definitions differ greatly in implementation strategy. If there is concern that sleep cannot be implemented correctly, it can be omitted. The system's power consumption will be slightly higher; this is the only cost of not sleeping.

9.1.5.1 The "Look-Before-and-After" Type

This is the sleep instruction definition that is the most complex in hardware design and the simplest in software use[7]. This kind of sleep instruction behaves in exactly the same way regardless of whether the processor's global interrupt enable is set. It (1) always atomically checks, before execution, whether an interrupt is already pending; if so, it executes as a NOP and does not enter sleep; and (2) always checks, after execution, whether a new interrupt is currently pending or triggered; if so, it immediately exits sleep and resumes instruction execution.

For this type, it suffices to simply execute the sleep instruction when sleeping; there is no need to worry about lost interrupt wakeups.

9.1.5.2 The "After-Only" Type

This is a sleep instruction definition that is relatively complex in both hardware design and software use[8]. This kind of sleep instruction behaves in exactly the same way regardless of whether the processor's global interrupt enable is set. Unlike the "look-before-and-after" type, however, it does not check before execution whether an interrupt is already pending; it only checks after execution whether a new interrupt is currently pending or triggered, and if so, it immediately exits sleep and resumes instruction execution.

For this type, executing it with interrupts disabled requires vigilance against lost interrupt wakeups. If no special handling is performed and a new interrupt becomes pending after interrupts are disabled but before the sleep instruction is executed, the wakeups from these pending interrupts may be lost. A naive approach is to first determine whether there is a pending interrupt and enter sleep only if there is none: if no interrupt is pending at the time of the check, but a new interrupt becomes pending after the check has finished and before sleep is executed, the wakeup from this newly arrived interrupt will likewise be lost.

If lost interrupts are not a concern, it suffices to simply execute the sleep instruction when sleeping. If lost interrupts cannot be tolerated, it must be handled as the "trigger-only" type described in the next subsection. However, some processors have a configuration switch that converts the behavior of the sleep instruction into the "look-before-and-after" type described in the previous subsection; if such a switch can be found, it can simply be turned on.

9.1.5.3 The "Trigger-Only" Type

This is the sleep instruction definition that is the simplest in hardware design but the most complex in software use[9]. This kind of sleep instruction can only be used when the processor's global interrupt enable is set, and once it is used, the processor immediately stops and waits for a new interrupt to arrive. If such an instruction is mistakenly executed with interrupts disabled, the processor will lock up because it cannot accept interrupts, causing the machine to freeze permanently. This causes trouble for RME: RME always executes the sleep instruction in the system call interrupt context——where interrupts are necessarily disabled——because the sleep feature is provided through a kernel function call. Monolithic

[7] such as the WFE instruction of ARMv7-M after SEVONPEND is set

[8] such as the WFI instruction of some non-standard RISC-V processors; the standard WFI instruction is of the "look-before-and-after" type

[9] such as the HALT instruction of x86

kernels such as Linux do not have this problem, because they have a genuine kernel idle thread with interrupts enabled that is responsible for sleeping.

For this type, we need to forge an environment similar to a Linux "kernel thread" in which to invoke the sleep instruction. Specifically:

(1) Write a piece of code as the "kernel thread", in which the sleep instruction is called in a loop. It does not need to contain any other content, nor does it need a separate stack.

(2) Once a sleep request is detected in _RME_Svc_Handler, exit the interrupt context to this "kernel thread" in assembly[10], and enable interrupts at the same time as exiting to this "kernel thread". To this end, a return stack can be forged on the spot; the stack of this "kernel thread" still uses the current kernel stack, while the current kernel context is saved onto the current kernel stack.

(3) When the first interrupt after sleeping arrives, check whether the interrupt interrupted the "kernel thread". If so, record that this interrupt has occurred, but do not run its interrupt vector for the time being; instead, restore the kernel context and continue the execution of _RME_Svc_Handler. There is no need to be concerned about whether the "kernel thread" has finished running.

(4) After _RME_Svc_Handler finishes running, scan the interrupt occurrence record and run the deferred interrupt vectors. Finally, return to user mode in the usual manner.

In this way, the sleep instruction is invoked with interrupts enabled, so a sleep-related lockup will not occur; after the processor is woken up, it continues executing the previous kernel function call, and the interrupt vectors are processed only after that call has been fully handled, so kernel reentrancy will not occur on a uniprocessor.

9.1.6 Saving and Restoring the Coprocessor Context

Coprocessors refer to CPU extension modules such as floating-point processors and vector processors, and they also have their own contexts; regardless of whether multiple threads share the same coprocessor, its context must be handled carefully, otherwise, in addition to causing program corruption, it will also turn the coprocessor register set into a huge covert storage channel. This is very different from RMP, which cares only about temporal isolation and not about spatial isolation. In addition, RME has a class of para-virtualization dedicated threads whose register state is untrusted; the coprocessors of such threads need to be handled even more carefully, because the coprocessor state they report is also untrusted.

The main function in charge of coprocessor context switching is __RME_Thd_Cop_Swap, which is introduced in 7.3.8.5 and is responsible for saving and restoring the context of the coprocessor together during a thread context switch. The saving and restoration of the coprocessor are implemented in it.

According to the functional model that the processor hardware layer provides for coprocessor control, coprocessors can be classified into the "always-on" type, the "off-on" type, the "off-initial-used" type, and the "off-initial-clean-dirty" type, where each provides more control functionality than the preceding one in order to optimize its context switch performance, and its optimal switch strategy is correspondingly more complex. Of course, each type can use the switch strategy of the types preceding it at the cost of performance; when (1) the information security requirements are very high, (2) it is uncertain how the switch strategy should be implemented, or (3) the current strategy implementation is suspected of being wrong, the "always-on" switch strategy can be used uniformly.

9.1.6.1 The "Always-On" Type

This is the simplest coprocessor control model. Under this model, if the coprocessor is implemented at the hardware level, it will be permanently on when the hardware is powered up and cannot be turned off, or it is enabled by default in most software environments[11]. Such a coprocessor is a coprocessor in name only; it can even be treated as part of the general register set, so its switch strategy is self-evident: switch

[10] rather than the usual return to user mode

[11] such as the MMX coprocessor of x86; although it has a switch, it is basically standard equipment and can be treated as the "always-on" type

the coprocessor register set at all times. Of course, the context attribute Attr of every thread in the system must contain this coprocessor.

Under this strategy, no special handling of para-virtualization dedicated threads is required.

9.1.6.2 The "Off-On" Type

This is the second simplest coprocessor control model. Under this model, the hardware implements a coprocessor that is off by default at system startup; its switch must be turned on before it can be used, otherwise an exception will be generated[12]. In this case, the coprocessor context can be switched according to the following strategy:

First, (1) determine whether the context attribute Attr_Cur of the current thread includes this coprocessor; if it does, turn on the coprocessor and save its context into the current thread's context structure; if it does not, do nothing. Then, (2) determine whether the context attribute Attr_New of the new thread includes this coprocessor; if it does, turn on the coprocessor and restore the context from the new thread's context structure into the coprocessor; if it does not, turn off the coprocessor.

Under this strategy, no special handling of para-virtualization dedicated threads is required, unless the on/off state of the coprocessor is also reported by a register under its control. Since the coprocessor on/off state it reports is not trustworthy, the final on/off state of the coprocessor should be forcibly kept consistent with the context attribute. This also applies to the subsequent types and will not be repeated below.

It is not recommended to use the "turn off after switching"-"exception"-"turn on" cyclic strategy[13] on such hardware to implement lazy software saving of the coprocessor state, because this strategy worsens the worst-case execution time: first, coprocessors on all architectures have switch overhead of varying magnitude, and this strategy increases the number of switches; second, if two alternately switched threads both use the coprocessor frequently, then each thread switch generates an additional coprocessor enable exception. In fact, the saving of the coprocessor state can be postponed until the last moment——for example, until just before the new thread begins to use the coprocessor, or until just before the coprocessor state structure of the current thread must be updated[14]——and only then performed, but this requires recording which thread the current coprocessor state belongs to and tracking it at all times, which imposes an additional mental burden on the porter. This may be done only when average performance is far more important than the worst-case execution time, and when doing so one must have a clear understanding of the speculative execution (Speculative execution) mechanism of the processor microarchitecture, otherwise it may cause leakage of the coprocessor state[15].

However, it is encouraged to use the "initially off"-"first exception"-"permanently on" strategy to convert such hardware into the "off-initial-dirty" type and adopt its "switch strategy 1", because the exception is raised only once, when a thread first uses the coprocessor.

9.1.6.3 The "Off-Initial-Dirty" Type

This is a relatively common coprocessor control model. Under this model, the hardware not only implements a coprocessor switch, but also detects in real time whether the coprocessor has been used. If the coprocessor is not turned on, using it generates an exception; if the coprocessor is turned on but has not actually been accessed, the "dirty" bit will not be set by the hardware until the coprocessor is actually accessed for the first time[16]. In this case, there are two handling strategies for the context of this coprocessor:

The first strategy treats the "dirty" bit as an indication of the "initial state". It is the most commonly used strategy; it helps to skip unnecessary context saving and restoration for threads that declare the

[12] such as the COP1 floating-point coprocessor of MIPS, which provides a simple on/off control bit

[13] Quite a number of operating systems adopt this strategy

[14] Those kernel function calls that query registers may read the coprocessor state

[15] To avoid this situation, official ports never use one thread's coprocessor context to execute another thread, even if the coprocessor is off

[16] such as the FPv4 and FPv5 floating-point processors of ARMv7-M

coprocessor in their context attributes but have never touched it[17], and it is the strategy used in the official ports of RME; this strategy is similar to the strategy adopted by RMP. The specific operations of this strategy are shown in the following table.

Table 9-3 "Off-Initial-Dirty" Type Switch Strategy 1

Current thread	New thread	Operation
Does not include the coprocessor (not in Attr)	Does not include the coprocessor	Nothing needs to be done
	Coprocessor untouched	Turn on the coprocessor
	Coprocessor used	Turn on the coprocessor and restore the coprocessor context from the new thread's context structure
Coprocessor untouched (initial)	Does not include the coprocessor	Turn off the coprocessor
	Coprocessor untouched	Nothing needs to be done
	Coprocessor used	Restore the coprocessor context from the new thread's context structure
Coprocessor used (dirty)	Does not include the coprocessor	First save the coprocessor context into the current thread's context structure, then reinitialize the coprocessor context, and finally turn off the coprocessor
	Coprocessor untouched	Save the coprocessor context into the current thread's context structure and reinitialize the coprocessor context
	Coprocessor used	Save the coprocessor context into the current thread's context structure and restore the coprocessor context from the new thread's context structure

Under this strategy, special vigilance is required against para-virtualization dedicated threads, because the "dirty" bit they report is not reliable. After modifying the coprocessor context, such a thread can clear the "dirty" bit in user mode to report a forged "initial state"[18], and can thereby leak its own coprocessor state to the new thread. Therefore, as long as the current thread is a para-virtualization dedicated thread, its "dirty" bit state must be ignored and it must be considered to have used the coprocessor. However, when it reports that it is in the "initial state", it is not necessary to save the coprocessor context into its context structure; this does not harm honest para-virtualization dedicated threads.

The second strategy treats the "dirty" bit as an indication that the context "has been saved". It helps to skip unnecessary context saving during execution segments in which the coprocessor is not used by threads that use the coprocessor intermittently, and can improve average performance when all threads actually use the coprocessor, but it provides no benefit for the worst-case execution time. The specific operations of this strategy are shown in the following table.

[17] It should be declared as used even if the thread does not use the coprocessor throughout its entire lifetime, which helps to improve forward compatibility

[18] This situation occurs only when multiple CPU cores are present; other cores can modify this data structure in parallel

Table 9-4 "Off-Initial-Dirty" Type Switch Strategy 2

Current thread	New thread	Operation
Does not include the coprocessor (not in Attr)	Does not include the coprocessor	Nothing needs to be done
	Coprocessor saved	Turn on the coprocessor and restore the coprocessor context from the new thread's context structure
Coprocessor saved (clean)	Does not include the coprocessor	Turn off the coprocessor (and clear the coprocessor context)
	Coprocessor saved	Restore the coprocessor context from the new thread's context structure
Coprocessor not saved (dirty)	Does not include the coprocessor	First save the coprocessor context into the current thread's context structure, then clear the current thread's "dirty" bit, and finally turn off the coprocessor (and clear the coprocessor context)
	Coprocessor saved	First save the coprocessor context into the current thread's context structure, then clear the current thread's "dirty" bit, and finally restore the coprocessor context from the new thread's context structure

Under this strategy, no special handling of para-virtualization dedicated threads is required[19]. Even if the state of the "dirty" bit it reports is not trustworthy, the coprocessor context is always reloaded when switching to it, so it is impossible to leak its own context by maliciously manipulating this bit.

Note: in theory, the operating system can introduce an additional variable in the thread context structure to record the "initial" state, which is used to separately indicate whether the thread has ever touched the coprocessor; this can convert the "off-initial-dirty" type into the "off-initial-clean-dirty" type described in the next subsection to further improve performance. However, this introduces additional complexity and memory consumption, and introducing this variable imposes an additional mental burden on readers of the code, so the official ports of RME do not adopt this scheme.

[19] Under this strategy, the "dirty" bit of the new thread is necessarily cleared

9.1.6.4 The "Off-Initial-Clean-Dirty" Type

This is the most advanced and most complex coprocessor control model; it provides native, independent indications for all states of the coprocessor, and thus can achieve the best performance. If the coprocessor is not turned on, using it generates an exception; if the coprocessor is turned on but has not actually been accessed, the coprocessor stays in the "initial" state until the coprocessor is actually accessed for the first time[20], at which point the hardware makes the coprocessor enter the "dirty" state; after the coprocessor context saving during a thread switch is completed, the software does not change the "dirty" state to the "initial" state, but to the "clean" state; any coprocessor access in the "clean" state also causes the hardware to turn the coprocessor into the "dirty" state. Its switch strategy is shown in the following table:

Table 9-5 "Off-Initial-Clean-Dirty" Type Switch Strategy

Current thread	New thread	Operation
Does not include the coprocessor (not in Attr)	Does not include the coprocessor	Nothing needs to be done
	Coprocessor untouched	Turn on the coprocessor
	Coprocessor saved	Turn on the coprocessor and restore the coprocessor context from the new thread's context structure
Coprocessor untouched (initial)	Does not include the coprocessor	Turn off the coprocessor
	Coprocessor untouched	Nothing needs to be done
	Coprocessor saved	Restore the coprocessor context from the new thread's context structure
Coprocessor saved (clean)	Does not include the coprocessor	Reinitialize the coprocessor context and turn off the coprocessor
	Coprocessor untouched	Reinitialize the coprocessor context
	Coprocessor saved	Restore the coprocessor context from the new thread's context structure
Coprocessor not saved (dirty)	Does not include the coprocessor	First save the coprocessor context into the current thread's context structure, then change the current thread to the "clean" state, then reinitialize the coprocessor context, and finally turn off the coprocessor
	Coprocessor untouched	First save the coprocessor context into the current thread's context structure, then change the current thread to the "clean" state, and finally reinitialize the coprocessor context
	Coprocessor saved	First save the coprocessor context into the current thread's context structure, then change the current thread to the "clean" state, and finally restore the coprocessor context from the new thread's context structure

Under this strategy, special vigilance is required against para-virtualization dedicated threads, be-

[20] such as the F, D and Q standard extensions of RISC-V

cause any coprocessor state they report is not reliable[21]. After modifying the coprocessor context, such a thread can modify the coprocessor state register in user mode to report a forged "initial" state, and can thereby leak its own coprocessor state to the new thread. Therefore, as long as the current thread is a para-virtualization dedicated thread, the "initial" state it reports must be treated as the "clean" state.

9.1.7 Implementation of Direct User-Mode Page Table Management

Although the RME kernel does its utmost to provide efficient abstractions for page tables, out of considerations such as controllable worst-case execution time and maximum user-mode flexibility, each level of page directory in RME corresponds to one capability, so page table operations are fine-grained but inefficient. On the one hand, in microcontroller scenarios where the page table mappings generally do not need to be modified[22], a large number of page directories and their capabilities consume the already limited system memory; on the other hand, in microprocessor scenarios where page tables need to be modified frequently or in large quantities[23], the low efficiency of page table operations drags down the throughput and tail latency of the entire system. Considering that the requirements for memory management differ vastly across scenarios, RME believes that the authority and responsibility for memory management essentially belong to the user[24]: the user is responsible for implementing page table management itself when its requirements are special.

Defining RME_PGT_RAW_ENABLE as 1 enables the direct user-mode page table management feature. Once this feature is enabled, page table management is fully controlled by a trusted user-mode process, which is essentially part of the kernel[25], and differs from the kernel only in that (1) it ostensibly runs in user mode, and (2) for the preceding reason, it can be preempted and therefore does not affect the worst-case execution time of the system. In this case, the page-table-related system calls and related low-level functions change somewhat; for these changes, please refer to 7.3.2, and they will not be repeated here. The design of direct user-mode page table management in extreme scenarios will be elaborated in the next two subsections.

9.1.7.1 Deeply Embedded Scenarios

The devices most commonly used in deeply embedded scenarios are microcontrollers (MCUs), whose systems are characterized by dedication to a single purpose, real-time reliability, limited budgets, and tight resources; once a process is created, its address space mappings are basically not modified. In this case, unless the dynamic page feature is to be implemented[26], the significance of retaining the page directory data structure and its capabilities is very limited.

In deeply embedded scenarios, the recommended approach is to generate, at project compile time, tables that can be directly filled into the memory protection unit (MPU), and to initialize the MPU directly with these tables during system initialization so as to completely eliminate the kernel page table overhead; these tables are all constants and can therefore be placed in ROM to further reduce memory usage. In addition, this method can generate configuration data tailored to the characteristics of the MPU, and can adapt to a base-address-length MPU better than a path-compressed page table. The RVM virtual machine monitor that accompanies RME provides this feature; for how to enable this feature, please refer to the RVM manual.

[21] Under this strategy, the new thread cannot be in the "dirty" state

[22] The page tables of a memory protection unit are purely software-simulated; if the mappings do not need to be modified, they are not needed at all

[23] This is generally due to page faults, and page faults are the underlying means of implementing lazy requests; in operating systems that care about average performance and resource utilization (such as cloud computing), lazy requests in turn provide the foundational support for a series of efficiency optimizations

[24] All performance problems are essentially problems of authority and responsibility

[25] Compromising it has exactly the same consequences as compromising the kernel; in other words, it belongs to the trusted computing base (Trusted Computing Base, TCB)

[26] Some RISC-V microcontrollers adopt simplified microarchitectures for cost reasons and cannot generate precise memory access permission exceptions, and therefore cannot use the dynamic page feature; in this case, there is basically no reason not to enable direct user-mode page table management

9.1.7.2 Cloud-Native Service Scenarios

Compared with deeply embedded scenarios, cloud-native service scenarios are the other extreme; the leading role is played by standard rack-mounted servers based on microprocessors. They often have memory on the order of terabytes, so the memory space occupied by page tables is negligible. However, (1) microprocessor systems are highly dynamic: processes are created and destroyed and memory is allocated and freed very frequently, and the resulting page table operations are large-scale; sometimes hundreds of thousands or even millions of pages are mapped or unmapped at once. If a page table capability operation were required for every page mapping or unmapping, the resulting performance overhead would be non-negligible. Moreover, (2) a page directory must be zeroed by the kernel when it is created, which greatly slows down the kernel's worst-case execution time; if multiple page directories are created in succession, dense jitter may be introduced, thereby causing the tail latency to exceed the limit[27]. In addition, (3) page faults also introduce an enormous number of page table operations; even on systems with ample memory that do not use page swapping algorithms, page faults play a huge role in deferring data copies and synchronizing data on demand[28], so responding to them swiftly is still very necessary. Finally, (4) it is difficult to decide how much kernel-dedicated memory a server system should allocate, so its kernel object table Kotbl always covers the entire memory range; in this case, the dynamic demarcation between kernel memory and user memory, as well as the management of kernel objects, already require a trusted user-mode process to take care of everything, and adding a page table management feature on top of that is only natural.

The recommended practice in cloud-native service scenarios has no fixed rule, but the general principles are (1) to reduce the number of IPCs for the same amount of page table operation work, (2) to integrate page tables and other memory objects into easy-to-use coarse-grained entities and provide them to applications, (3) to avoid adding one layer after another when multiple management levels exist, and (4) to fully take the temporal and spatial isolation between processes into account when designing mechanisms. Of course, if the address space is not frequently operated on or there are no stringent performance requirements, there is nothing inappropriate about not using direct user-mode page table management.

9.2 Recommended Handling Scheme for the Memory Protection Unit

Memory protection units differ considerably among processors. These differences can be divided into three aspects: influence, region organization, and region format; the ways to deal with them are discussed in the following three subsections respectively.

9.2.1 Influence of the Memory Protection Unit

The difference in this aspect lies in whether the region attributes of the memory protection unit take effect for kernel mode.

9.2.1.1 All-Mode-Affecting Type

This is the most common type of memory protection unit[29], whose region attributes take effect for both kernel mode and user mode. This means that if the region attributes are not modified when entering the kernel, the access control restrictions and cache policies that user mode applies to certain memory regions will be imposed on the kernel. Once a driver in the kernel[30] needs to access these regions, it may

[27] On the client side this often manifests as program stutter and an unresponsive mouse, while on the server side it often manifests as large response fluctuations and poor service quality

[28] For example, the fork call of Linux tries its best to defer the memory copy until the corresponding page is actually modified; a similar mechanism also exists in mmap

[29] such as ARMv7-M

[30] RME does not encourage this, but when there are extreme real-time requirements it may be necessary to write a small part of the driver into a kernel-mode interrupt vector

be subject to random and unexpected restrictions[31], thereby causing a kernel failure.

In this case, the recommended approach is to turn off the memory protection unit after entering the kernel, and to turn it on just before leaving the kernel. There is no need to worry that turning off the memory protection unit will cause a kernel failure; this type of memory protection unit has a background region, and when the memory protection unit is off or does not include a certain region, kernel mode will instead use the access permissions of this background region.

All official ports of RME adopt the practice of temporarily turning off the memory protection unit. This means that the kernel always uses the background region when running, and its access permissions and cache policies are the same as when the memory protection unit is not enabled; this is very conducive to porting interrupt vectors in bare-metal drivers.

9.2.1.2 User-Mode-Only Type

This is the second most common type of memory protection unit; its regions can be configured to take effect only for user mode. Once the kernel is entered, all memory accesses are no longer restricted by it.

In this case, the ideal behavior is obtained without any special handling.

9.2.2 Region Organization of the Memory Protection Unit

The difference in this aspect lies in whether the memory protection unit prepares different regions for different types of memory.

9.2.2.1 Unified Type

The vast majority of processors adopt this design; any of its entries can be assigned to a memory region of any type.

In this case, the ideal behavior is obtained without any special handling.

9.2.2.2 Separate Type

A small number of processors adopt this design; their entries are divided into three parts: code, data and device, each having a certain number of dedicated entries.

RME and RVM currently do not support this design, and RME does not provide a corresponding official port. In fact, none of the microcontroller architectures that RME supports or plans to support include the separate design.

9.2.3 Region Format of the Memory Protection Unit

The difference in this aspect lies in the format of the individual regions in the memory protection unit.

9.2.3.1 Power-of-Two Type

This is the most common memory protection unit region format[32]. This format stipulates that the length of the memory area corresponding to each region must be a power of 2, and that the start address of this memory region must be aligned to this power of 2. Some designs also allow a region to be divided equally into 4, 8 or 16 independent subregions, each of which can be selectively enabled or disabled.

This region format is close to the "page" of a paged memory management unit, so these regions can be regarded as a software-filled TLB. The path-compressed page table algorithm built into RME copes well with this format.

[31] This depends on which process was interrupted before the interrupt vector was triggered
[32] such as ARMv7-M

9.2.3.2 Contiguous Type

This is the second most common memory protection unit region format[33]. This format stipulates that an arbitrary start address and end address can be specified for each region, as long as these addresses are aligned to the processor word length.

This region format is close to the "segment" of a segmented memory management unit, so the path-compressed page table algorithm built into RME does not match it well. Therefore, when porting, it is recommended to merge consecutive page mappings into the same region; to facilitate such merging by the kernel, user mode should deliberately map consecutive pages in address order.

9.2.3.3 Variable

This is the least common memory protection unit region format[34]. This format specifies that each logical region can be either power-of-two or contiguous: when the logical region is power-of-two, it occupies one physical region; when the logical region is contiguous, it occupies two physical regions. The total number of physical regions is limited.

This region format is very flexible, so great care must be taken when designing the metadata cache. To maximize the utilization of physical regions as much as possible, when adding contiguous pages, consideration can be given to merging three or more power-of-two logical regions into a single contiguous logical region; this kind of merging may also have to be performed when adding dynamic pages. To make it convenient for the kernel to perform such merging, user mode should intentionally map contiguous pages in address order.

9.3 Factors Limiting Multicore Scalability in RME

In RME, not all operations can be executed without affecting each other. A typical operation that cannot be executed in this way is multiple CPUs attempting to simultaneously create kernel objects in the same piece of memory. The operations that cannot be executed in parallel are listed below, together with their reasons.

9.3.1 Update of RME_TIMESTAMP

If RME_TIMESTAMP is defined as a variable, its update must be performed by one processor rather than multiple processors. On multicore processors, the other cores need to synchronize this variable into their own cache lines. Therefore, this operation cannot be parallelized well. However, this does not greatly affect the degree of parallelism of the system, because the clock interrupt frequency of large systems can be configured to be relatively low.

If RME_TIMESTAMP is defined as a read of some free-running counter[35], then this problem does not exist. It is worth noting multi-package systems: although the counters of the cores within the same physical CPU package may be consistent, the counters between different physical CPU packages may become inconsistent. In this case, the porter must solve this problem themselves.

9.3.2 Multiple Cores Simultaneously Creating Kernel Objects in the Same Kernel Memory Region

Since the creation of kernel objects always involves writing to the kernel object registry table, when multiple CPUs attempt to concurrently write to the same location in the table, a large number of cache line updates occur, making this operation not parallelize well.

[33] such as ARMv8-M
[34] such as RISC-V
[35] such as the x86 RDTSC

9.3.3 Multiple Cores Simultaneously Sending Signals to the Same Signal Endpoint

Since this is an atomic compare-and-swap operation, the CPU and the bus controller may lock the bus at the hardware level, so simultaneous accumulation by multiple cores is actually completed serially. Therefore, this case cannot be parallelized well. Of course, if the system has hardware that can perform accumulation in parallel[36], then this problem does not exist.

9.3.4 Multiple Cores Simultaneously Performing Capability Transfer or Removal on the Same Root Capability

Since the reference count of a capability is kept inside the root capability, multiple capability transfers or removals performed simultaneously on the same root capability will modify the reference count at the same time. This causes a large number of cache line updates and makes this operation not parallelize well. However, this problem generally does not exist in systems, first because capability transfer is usually a relatively rare operation, and second because in multicore systems user programs usually create dedicated kernel objects for each core.

Since the root capabilities of kernel memory capabilities (RME_CAP_TYPE_KOM) and kernel function call capabilities (RME_CAP_TYPE_KFN) are created only at system boot, any leaf capability of these two types in the system must come from the initial capabilities created at boot, which cannot be deleted. Fortunately, neither of these two types of capabilities has a kernel object, nor are they referenced by any other kernel object, so RME applies a special optimization to them: they do not record their own root capability and do not perform reference counting, thus avoiding this problem.

9.4 Factors Limiting RME on 32-Bit Systems

In the case of 32-bit processors, RME is constrained by the processor word length in the implementation of certain functions. The two constrained functions are thread creation and quiescence period calculation.

9.4.1 A Thread's Thread ID

In RME, each thread corresponds to a TID when it is bound, and its length is half a machine word. Therefore, on a 32-bit system there are only 2^{16} different TIDs in total[37]. If more threads than this number are to be created, then it means that the TIDs of some threads must overlap when they are bound. If user-mode software does not take this situation into account when it is designed, trouble will result.

However, this is not a problem on ordinary 32-bit systems. In the modern context, 32-bit systems are mainly embedded systems, so the number of threads is generally very small. Under this premise, user mode can always find an unused TID to assign to a new thread.

9.4.2 Calculation of the Quiescence Period

All quiescence periods in the system are calculated by comparison with the value of RME_TIMESTAMP. This value increments with each clock tick. The length of RME_TIMESTAMP is always one machine word, so the problem of overflow and wraparound exists. For example, when this value is 0x00000000, an operation occurs[38] that requires a quiescence period of 10 timeslices; then, after waiting until the value of RME_TIMESTAMP exceeds 0x0000000B or is higher, the next operation can be performed on that capability. However, if we go for a very long time without performing the next operation, but instead wait until the counter reaches its end and returns to 0 before performing the next operation, we will find that

[36] common modern architectures generally have such hardware
[37] on 64-bit systems this value is 2^{32}, so it is not a problem
[38] such as capability freezing

the location has entered a non-quiescent state again: in fact the location became quiescent long ago, and this non-quiescence is an illusion caused by variable overflow.

Therefore, on a 32-bit multicore system, if an object is found to be non-quiescent after the quiescence period has elapsed, one can wait another quiescence period and then retry immediately, until it succeeds. On 64-bit systems, since RME_TIMESTAMP never overflows, this problem does not occur.

9.5 Known Potential Covert Channels in RME

There are some known potential covert channels in RME. These covert channels fall into two types, listed below.

9.5.1 Covert Storage Channels

The covert storage channels present in RME mainly occur between two processes that share a capability, a kernel object, or a capability table. In the case of a shared capability or kernel object, the two processes can probe and change the state of the capability or kernel object through operations on it, thereby transferring information.

In addition, in systems with a coprocessor, if the part of the implementation that saves and restores the coprocessor context is improper, the coprocessor register file may be used as a covert storage channel with a very large bandwidth. In those cases where lazy saving and restoring of the contents of the coprocessor register file is implemented, even if the implementation is correct, a malicious task may be able to infer the coprocessor usage of other sensitive tasks[39], or use it as a timing channel. For this reason, in all default implementations of RME, the coprocessor register file currently in use by a thread is always saved and restored immediately upon a switch.

Improper implementation of kernel function calls has the same problem, and careful handling is also required in the design.

9.5.2 Covert Timing Channels

In RME, timeslices and scheduling are managed by the user, so if the user's scheduling algorithm or timeslice allocation algorithm is written improperly, a large number of covert timing channels will appear. To reduce the bandwidth of such channels, one can consider disabling the processor's user-mode high-precision timer instructions[40]. For this part, please consult the relevant material yourself; no detailed explanation is given here.

9.6 Memory Consistency Model

RME makes extensive use of lock-free data structures, so the memory consistency model is of vital importance. Memory consistency models can generally be divided into two categories: the strong sequential consistency model and the weak consistency model. In the strong sequential consistency model, all memory accesses on a multicore processor have a global order, that is, from the perspective of all processors, the order of other processors' operations on memory is always consistent; in the weak consistency model, the operations of other processors observed by one processor may be reordered, and if two processors both observe a third processor at the same time, the reordering they observe may also be different.

On a processor with a weak consistency model, the following four kinds of reordering may occur:

Read-read reordering, that is, there is no fixed order between two adjacent read operations.

Read-write reordering, that is, there is no fixed order between an adjacent read-write pair.

[39] CVE-2018-3665

[40] such as the x86 RDTSC

Write-read reordering, that is, there is no fixed order between an adjacent write-read pair.

Write-write reordering, that is, there is no fixed order between two adjacent write operations.

In reality, there is a continuous spectrum between the two consistency models. Some processors may reorder write-read but be ordered for other cases[41].

Reordering may cause some synchronization flags to fail. Generally speaking, a shared memory region is guarded by a synchronization flag, and the value of the synchronization flag represents the current state of the shared memory region. We first check and attempt to take possession of the synchronization flag, and then perform operations in the shared memory region; when the operations are complete, we first wrap up the work and then release the flag. When reordering occurs, operations inside the shared memory region may happen before the synchronization flag is taken possession of, and the wrap-up work may be delayed until after the synchronization flag is released. Both of these destroy the properties of the critical region. Therefore, a memory barrier should be added after acquiring the flag, to ensure that operations on the shared memory must begin only after the flag is acquired; a memory barrier should be added after the wrap-up work, to ensure that the release of the synchronization flag takes place only after the wrap-up work has finished. Although RME does not use the locks described above for kernel synchronization, memory consistency still affects the kernel's synchronization operations.

To eliminate the influence of memory reordering on the kernel, RME provides two macros, RME_READ_ACQUIRE() and RME_WRITE_RELEASE(), for synchronizing memory operations. In addition, compare-and-swap atomic operations all require a full memory barrier to be appended at the end. For specific implementation guidance, please refer to the description in 7.8.

9.7 Known Factors Affecting Real-Time Performance in RME

There are two known main factors affecting real-time performance in RME, namely the capability table and the page table.

9.7.1 Overly Long Capability Table

The influence of the capability table on real-time performance is mainly manifested in the creation and deletion of capability tables. Since RME does not adopt an implementation with kernel preemption points, a capability table accesses a large amount of memory continuously in one go during creation and deletion. If the capability table being created is relatively large, and RME_CPT_ENTRY_MAX in the kernel[42] is not restricted and therefore cannot prevent the creation, the kernel will briefly become unresponsive. To avoid this situation, RME_CPT_ENTRY_MAX needs to be set to limit the maximum size of capability tables, or only trusted user-mode root capability managers should be allowed to create and delete capability tables on processor cores that do not need to respond to real-time interrupts.

9.7.2 Page Faults

For the page table implementation described in 8.2.1.1, dynamic pages may generate page faults[43], so the real-time performance of data access to dynamic pages cannot be guaranteed; for the page table implementations described in 8.2.1.3 and 8.2.1.3, if the number of mapped static pages exceeds the number of regions supported by the hardware, page faults may be triggered even on static pages. In this situation, one solution is not to map too many static pages.

For the page table implementation described in 8.2.1.4, since the TLB is filled by hardware, there is greater uncertainty. If real-time performance needs to be obtained on a hardware-filled memory manage-

[41] such as x86-64
[42] see Section 7.3.2
[43] but static pages never will

ment unit, it is recommended to use a processor that supports TLB locking, so that the pages of real-time processes can be locked in the TLB[44]; if the processor does not support TLB locking, a TLB coloring algorithm can be adopted when mapping pages, trying to ensure that the TLB entries of real-time processes are not evicted by non-real-time processes, or that the total number of pages mapped in the system does not exceed the size of the TLB[45]; if the processor cannot even implement the TLB coloring algorithm, the data cache can also be colored so that part of the page table of real-time processes permanently resides in the data cache, so that once a TLB miss occurs, the lookup speed can be greatly accelerated[46].

9.7.3 Livelock Caused by Linked Load/Store Conditional

On many processors, no atomic instructions other than Linked Load (LL) and Store Conditional (SC) are provided. Although these two instructions are sufficiently flexible in semantics to implement almost all atomic operations, they may cause livelock, thereby causing the real-time guarantees and capability quiescence guarantees provided by RME to fail. Therefore, the following must be noted when using this pair of instructions:

1. If the processor provides a starvation-free guarantee for LL/SC, using them to implement all of RME's atomic operations will not cause problems.
2. If the processor does not provide a starvation-free guarantee for LL/SC but provides dedicated atomic operation instructions[47], then LL/SC should be avoided and the dedicated atomic instructions should be used instead.
3. If the processor does not provide a starvation-free guarantee for LL/SC and provides dedicated atomic instructions for operations other than CMPXCHG, then LL/SC should be used only to implement CMPXCHG, while the other instructions should use the dedicated atomic operations; moreover, when implementing CMPXCHG, it should exit once the number of loop attempts exceeds a certain threshold, thereby achieving the starvation-free property. Although this changes the semantics of CMPXCHG, so that it still has a chance of failing even when the success conditions are met, it is better than an infinite livelock loop. The design of RME can accept this modified CMPXCHG semantics.
4. If the processor does not provide any atomic instructions other than LL/SC, and LL/SC has no starvation-free guarantee, then it is recommended to limit the number of CPUs in the system, such as to dual-core or quad-core. In this way, because a system call has a large inherent overhead, the degree of contention among CPUs in system calls is controllable, thereby achieving starvation freedom in a roundabout way. In fact, modern multicore processors generally satisfy cases 1-3, so case 4 is very rare.

9.8 RME Deviations from the C Language Standard and the MISRA Standard

9.8.1 C Undefined Behavior: Type-Punning Pointer Aliasing May Exist and Does Not Follow the Strict Aliasing Rule

Type-punning pointer aliasing is a usage that widely exists in practical applications of the C language[48]; it allows two pointers of different types to reference the same block of memory. This is very common in network protocol stacks, because we often need to assemble or disassemble network packets, change their byte order, and so on. This is also very common in operating systems, especially converting back

[44] this is the recommended approach
[45] this is the second-best approach
[46] this is the worst approach
[47] such as CMPXCHG, XADD, AMOADD, AMOAND, etc.
[48] referring to C90, the same below

and forth between unsigned integer pointers and structure pointers to facilitate operations on objects. Unfortunately, the strict aliasing rule (Strict Aliasing Rule) of the C language does not allow two pointers of different types to reference the same block of memory, unless they are compatible types or one is converted from another type to char[2], and there is no general way to get around this. Generally speaking, there are three methods to avoid type punning, which are:

1. Use a union to perform conversions between different types. Unfortunately, this is defined behavior in C but not in C++, and RME's code may need to be compiled as C++ because it may be directly linked with some libraries written in C++[49], so it may still be undefined behavior. In addition, this method is powerless against array objects whose length is not determined at compile time, unless potential performance is sacrificed to convert them one by one. Finally, this use of a union does not conform to the design spirit of union and is an abuse of the language rules.
2. Use char* to perform the operations. Unfortunately, existing compilers optimize this very poorly; some rather good compilers even compile these operations into byte-by-byte access instructions, and this is even more so for low-quality embedded compilers. Therefore, this is a bad solution.
3. Use the memcpy function to copy contents between two incompatible pointers and then copy them back after the operation, hoping that the compiler can detect this behavior and optimize away the repeated memcpy. This cleverly exploits the fact that the parameters of memcpy are char*, and char* can always alias any pointer. The direct problem it brings is that some low-quality compilers may not be intelligent enough and sometimes will not optimize away the memcpy function, causing the generated code to actually call memcpy twice, resulting in a huge performance loss. Moreover, once memcpy is not optimized away, this may cause a local copy buffer declared inside the function to be allocated on the stack, greatly increasing stack consumption and causing a stack overflow. Finally, this kind of use of memcpy does not conform to the design intent of memcpy and is an abuse of a language loophole; the C standard does not encourage this practice.

   In view of the problems of the above three methods, RME does not use these methods, but directly uses type-punning pointer aliasing, while turning off the relevant optimization at compile time[50].

9.8.2 C Undefined Behavior: The Length of a Byte is Always 8 Bits, and the Size of a char Type is Always One Byte

The C language does not define the length of char; it only states that char must be equal to the width of a single character in the system's basic character set[2]. Theoretically, a char type can be 32 or 16 bits[51]. On all current mainstream compilers and architectures, the length of char is always 8 bits, so we also make this assumption in the code to avoid an unnecessary "char type length" configuration macro.

9.8.3 C Undefined Behavior: The Actual Representation of a Pointer is Always a Machine-Recognizable Address

The C language does not define how a pointer should be represented in memory. Theoretically, the actual representation of a pointer variable can be anything, as long as the value it returns when dereferenced is the value of the variable it points to according to C language semantics[2]. This makes it impossible to align a pointer to a certain memory granularity under the strict C standard, because it is impossible to use a mask on it to mask out some of its bits; it is also impossible to use some unused bits of the pointer to store other information. Therefore, we assume that the actual representation of a pointer is always a binary address directly recognizable by the machine; this holds on all architectures that RME supports or plans to support.

[49] if `extern "C"` is not desired
[50] such as GCC's -fno-strict-aliasing option
[51] some early embedded compilers did this

9.8.4 C Undefined Behavior: The Length of a Pointer is Always Less Than or Equal to One Machine Word

The C language does not stipulate the length of a pointer. The length of a pointer can be any value, as long as it works normally according to C language semantics. It is entirely possible for the length of a pointer to be longer than one machine word, so that it cannot be placed inside an unsigned integer value whose length is equal to one machine word[2]. This situation currently occurs only on almost all 8-bit microcontrollers and some 16-bit microcontrollers, and RME never intended to support these architectures from the beginning of its design, so we can assume that the length of a pointer is always less than or equal to one machine word. In fact, on all architectures that RME supports or plans to support, the length of a pointer is always equal to one machine word. In addition, we do not use types such as uintptr_t provided by the C library, because these types exist only in C99 and are even optional implementations.

9.8.5 C Undefined Behavior: Coercing a Pointer to an Unsigned Integer Value of One Machine Word Always Leaves Its Actual Representation Unchanged

The C language does not stipulate how the actual representation of a pointer should change after the pointer is converted to an unsigned integer. It only stipulates that if a pointer is first converted to an unsigned integer with sufficient space and then converted back, the pointer obtained after the round trip should be regarded as equal when compared with the original pointer[52][2]. If the representation of a pointer changes after conversion to an unsigned integer, then we will encounter a problem similar to that described in 9.8.3. Therefore, we assume that coercing a pointer to an unsigned integer value of one machine word always leaves its actual representation unchanged; this holds on all architectures that RME supports or plans to support.

9.8.6 C Undefined Behavior: The Length of External Identifiers is Extended to 31 Characters

C90 stipulates that at minimum only the first 6 characters of an external (extern) variable name are significant. This is too few for any operating system. Therefore, RME relaxes this restriction to 31 characters, the same as the restriction on internal variable names. All modern compilers support this.

9.8.7 C Undefined Behavior: Modifications to Non-volatile Variables are Always Preserved

The latest C23 stipulates that a write to a non-volatile variable is no longer a side effect that must be preserved[53]. In this case, if the compiler determines that a write to a data structure will not be read by any read operation within the "same thread", the write can be optimized away. For example, a sufficiently smart link-time optimization (Link-Time Optimization, LTO) compiler may mistakenly consider the creation of a kernel object to be redundant, because the object is no longer read by the current system call after it is created, so the actual creation operation can be optimized away. This optimization cannot be turned off by a compiler barrier, because a compiler barrier can only force the compiler to synchronize the state of those necessary variables with memory, but it cannot prevent the compiler from considering some writes to be useless. Fortunately, this theoretically feasible optimization does not exist in reality, because (1) for practical compiler authors, analyzing the invalidity of the side effects of writes to global variables is difficult and dangerous, laborious and unrewarding, and (2) for programmers, it would force all global variable write accesses to be performed in the form of volatile pointers[54], making programming

[52] this does not even prohibit the representation before conversion and after the round trip from being different, as long as the two values are always treated as equal in pointer comparison

[53] it once was in C17 and earlier

[54] if these global variables are not declared with volatile

very cumbersome. Given that many popular operating systems assume that writes are preserved[55], and the compilers used to compile these popular operating systems do not perform this optimization either, we also make this assumption.

9.8.8 Description of MISRA-C Deviations

RME has strictly checked all three categories of recommendations in MISRA-C:2012[3], and all deviations have been approved.

RME has no deviations from the Mandatory category described in MISRA-C:2012[3].

RME's deviations from the Required category described in MISRA-C:2012[3] are as follows:

Table 9-6 RME Deviations from the Required Section of MISRA-C:2012

Entry	Explanation
1.3	Some undefined behaviors as described above are necessary for implementing low-level software such as an operating system.
5.1	RME extends the effective length of external identifiers to 31 characters. This is necessary for the software engineering of an operating system.
11.3	Type-punning pointer aliasing is necessary for software such as an operating system. In fact, RME uses such type-punning pointer aliasing only for structures and substructures.
11.9	RME does not use NULL but uses RME_NULL. This is a very minor inconsistency.
14.3	In its process of checking whether the configuration is correct, RME makes extensive use of Boolean expressions whose results can be determined at compile time. These Boolean expressions are all fed directly into the RME_ASSERT macro to ensure that RME is correctly configured. This improves the quality of the software.

[55] such as Linux

RME's deviations from the Advisory category described in MISRA-C:2012[3] are as follows:

Table 9-7 RME Deviations from the Advisory Section of MISRA-C:2012

Entry	Explanation
2.5	In the hardware abstraction layer, especially in the firmware library, some macros appear as configuration options. They may be unused in the current configuration, but they must be retained to accommodate future software configuration adjustments, which helps maintain engineering consistency and improve code quality. These macros all have good comments and documentation for reference.
2.7	Since RME's hardware abstraction layer must define a unified interface for all supported hardware, it is inevitable that some hardware abstraction layer function parameters are redundant for some architectures.
11.4	Converting an unsigned number to a pointer is necessary for software such as an operating system.
12.4	On 32-bit systems, RME_TIMESTAMP may overflow when incrementing and restart from 0; this is the correct implementation. A detailed explanation of it is given in 9.4.2.
15.4	In RME's kernel memory registry table-related code, some loops have two breaks that can terminate them, and such an implementation also appears in the page table lookup functions of the hardware abstraction layer. Other forms of implementation may increase implementation complexity and be detrimental to engineering maintenance. In loops where this situation occurs, the comments detail the reason for doing so and the exit conditions of each branch, and all the break statements are listed side by side in order from top to bottom.
15.5	For the same reasons as 15.4, a function in RME may have multiple exits. These code locations are well commented, and other alternative implementations would violate other requirements or recommendations and would break the overall abstraction of the project.
20.5	To make the declaration of every function and variable occur only once (including its extern form), and to systematically prevent implicit declaration errors so as to improve software quality, RME uses an original segmented-inclusion header file format. This format requires the use of #undef. #undef is used only for this format, and this format is well followed throughout the system, and there is corresponding documentation that explains it in detail.

9.9 Chinese-English Terminology Quick Reference Table

Table 9-8 Chinese-English Terminology Quick Reference Table

English Term	Chinese Translation
Capability	权能
Component	组件
Coroutine	协程
Daemon	守护进程

English Term	Chinese Translation
Endpoint (Signal)	（信号）端点
Expandable/Expandability	可扩展/可扩展性
Invocation (Thread Migration)	（线程迁移）调用
Page Directory	页目录
Page Entry	页表项
Page Table	页表
Priority	优先级
Process	进程
Quiescence	安定
Scalable/Scalability	可伸缩/可伸缩性
Scheduler	调度器
Signal	信号
Thread	线程

9.10 Bibliography for This Chapter

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, et al., "The multikernel: a new OS architecture for scalable multicore systems," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 29-44.

[2] ISO, International Standards Organisation, "Programming Languages—C, International standard, ISO/IEC 9899: 1990 (E)", 1990.

[3] M. I. S. R. Association, MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013: Motor Industry Research Association, 2013.