GSC微控制器工程编译器
技术参考手册

版本 R1T1

天创·阳雨·暴风（七阶）

Genesia·Sunrain·Tempest (R.VII)

G7S2 (Tempest) R1T1

微控制器平台工程编译器（一版一型）

技术参考手册

系统特性

高层次描述到解决方案编译器

·生成可直接被主流 IDE 读取和进行编译的工程

·用户仅提供高层次系统特性描述文件，无需关心系统具体配置

全面的体系架构支持

·支持多核对称微控制器

·支持常见微控制器架构

完备的图形界面

·基于最新 H5 技术打造的现代 GUI

·0 代码实现系统配置和工程生成

支持 RME/RVM 工程完全生成

·生成 RME 和 RVM 的配置头文件

·生成 RME 的初始化内核对象创建文件

·生成 RVM 的初始化内核对象创建文件

·生成各个 RME 原生进程、RTOS 虚拟机和 POSIX 兼容进程本身

目录

表目录

图目录

版本历史

版本	日期（年-月-日）	说明
R1T1	2019-08-20	小范围预览发布

第 1 章 概述

1.1 简介

微控制器平台的软件复杂度在现代嵌入式系统中快速上升，其配置难度也越变越高。尤其对于微内核操作系统而言则更是如此：分配内存和权限，以及进行各个子系统的加载和重启需要大量的配置项。这些配置项之间往往存在着很大程度的关联，需要在配置时进行大量计算。这使得传统的微型 RTOS 使用的头文件配置方法已经不再适用于现代的微内核操作系统，尤其是在微控制器一类需要静态配置资源的场合。

本软件是适用于 RME（RTOS-Mutate-Eukaron）实时操作系统（Real-Time Operating System，RTOS）的专用方案编译器。该编译器器针对微控制器平台设计，可以根据用户图形界面输入的系统需求直接生成适合于某种集成开发环境（IDE）的工程。

本软件的许可证目前为单一制的 AGPLv3。

1.2 方案编译器的组成部分

和编程语言编译器类似，方案编译器的内核分为三个部分：输入分析模块、中间表示生成模块、工程输出模块。输入分析模块负责读取输入的系统配置到其内部数据结构；中间表示生成模块将内部的数据结构转变成一种中间表示，并且将它进行优化；工程输出模块则负责将中间表示映射到最后的所选平台。为了方便用户使用该配置器，在命令行界面之外，本方案编译器还额外提供了图形前端模块。

1.2.1 图形前端模块

本生成器的输入分析部分负责提取用户的输入并且生成一个固定格式的 XML 文件。该 XML 文件也可以由用户手写并通过命令行模式直接输入给方案编译器的输入分析模块。

1.2.2 输入分析模块

输入分析模块负责分析用户输入的 XML 文件，并且根据用户输入的 XML 文件去读取自身数据库中的被选中芯片的 XML 文件，将这些数据填充在内部数据结构中传递给中间表示生成模块。 生成对应于该芯片的可以被编译到对应于任何工具链的工程的中间工程格式。

1.2.3 中间表示生成模块

中间表示生成模块负责读取输入分析模块提供的内部信息并且决定各个内核对象的相关信息，各个内存段的地址以及工程的组织信息。在分配内存和组织工程时，中间表示生成模块将进行优化，确保资源消耗量被降低到最小。

1.2.4 工程输出模块

工程输出模块负责读取中间表示生成模块生成的中间表示，并且实际生成最终的初始化代码和用户所选工具链的工程结构。用户得到该工程后只要直接用工具链编译即可得到一个可用的系统，无需额外进行任何处理。

1.3 手册总览

在接下来的各个章节中，我们将分别介绍 GSG 的各个子系统。其中第 2 章介绍图形前端模块的使用，第 3 章介绍输入分析模块及其读取的 XML 的格式，第 4 章介绍中间表示生成模块的各个步骤，第 5 章介绍工程输出模块的各个步骤，第 6 章介绍用户应用程序相关内容。之后，我们会在第 7 章介绍 GSG 的添加新架构和工具链的方法，以及在第 8 章介绍虚拟机客户操作系统的实现规范。最后，我们会在第 9 章介绍 GSG 中存在的一些已知问题及其解决方案。

在本技术参考手册中介绍的仅仅是 GSG 的通用信息本身。关于各个架构和工具链上 GSG 的具体实现细节，请参看对应架构的相关手册；关于 RME 和 RVM 本身的相关信息，请参看 RME 和 RVM 的用户手册。

1.4 本章参考文献

无

第 2 章　图形前端模块

2.1　图形前端模块简介

由于输入 XML 来生成系统仍然十分麻烦并且容易出错，因此本编译器也提供一个图形化用户界面。藉由本章节对图形界面模块的描述，我们也介绍编译器的功能。

2.2　启动编译器

打开编译器的可执行文件并等待其加载完毕。在加载完毕后，如下的界面会弹出，用户需要选择生成工程的位置、使用的微控制器的架构以及微控制器本身的型号。

在用户选择完毕后，编译器正式启动，弹出其工作界面。工作区可以分为三个部分，分别是配置组选择区、配置项选择区和配置内容区。关于各个区域的用途和使用方式我们将在下面的章节分别介绍。

在下面的章节中，每一个标题都会对应展示 XML 文件中的相关域的名称。在域的名称前的标识符的意义如下：S 代表字符串，D 代表下拉选项框，C 代表多项选择框，H 代表十六进制数，N 代表整数。如果环绕标识符的是圆括号“()”，那么该项是可填充也可不填充的；如果环绕标识符的是方括号“[]”，那么该项必须得到填充。

2.2.1　配置类选择区

该区域负责选择配置大类。配置大类一共有七类，分别是工程配置、硬件配置、存储配置、RME 内核配置、RVM 用户态库配置、原生进程配置和虚拟机配置。其中，前五类配置每类只有一项，后两类配置的项目多少依赖于创建的进程和虚拟机的数量而定。这些配置大类都展示在左侧第一栏中，其中最下方的第八个按钮负责在被点击后生成工程。下图展示了一个典型的配置类选择区的状态。

2.2.2　配置对象选择区

对于原生进程和虚拟机配置类，每个原生进程和虚拟机对象都有一系列配置组。此时，左侧第二栏负责展示这些原生进程或虚拟机对象。当对应的对象被选中时，该对象的配置组会在配置组选择区被展示。下图展示了一个典型的配置对象选择区的状态。

2.2.3　配置组选择区

配置组选择区负责一个配置大类下的各个配置组。对于每个配置大类，其配置组的内容都不同，请参见具体章节。下图展示了一个典型的进程配置类的配置组选择区的状态。

2.2.4　配置项展示区

配置项展示区负责具体的配置项的展示和配置。同样地，针对每个配置组，其配置项的内容也不同，需要参见具体章节。下图展示了一个典型的内存块的配置项的状态。

2.3 工程配置类[S]<Project><Name>

工程配置类负责工程本身的基本信息。该类只有一项要配置，即为工程名称。在点击该选项卡时，会弹出一个字段文本框，只要在该文本框中设置工程的名称即可。

2.4 芯片配置类<Project>

芯片配置类负责芯片的选择。该类有四个项要用户手动选择，并没有配置组存在。

2.4.1 目标架构项[D]<Platform>

该配置项是一个必选的下拉选项框，要求用户选择一个架构。每个架构在最终的 XML 中用一个代号代表，比如 A7M 对应于 ARMv7-M 架构。

2.4.2 生产厂商项[D]

该配置项是一个必选的下拉选项框，要求用户选择一个生产厂商。XML 中没有直接的生产厂商的代表域。

2.4.3 芯片系列项[D]

该配置项是一个必选的下拉选项框，要求用户选择一个芯片系列。XML 中没有直接的芯片类别的代表域。

2.4.4 芯片型号项[D]<Chip_Full>

该配置项是一个必选的下拉选项框。在该选项框中，用户需要选定具体的、精确到最后一个字母的芯片型号。

2.4.5 芯片精确类别项[D]<Chip_Class>

该配置项会由编译器自动在选择了芯片型号后生成。每个芯片精确类别可以兼容一系列具体的芯片，比如 STM32F767IG 可以兼容 STM32F767IGT6 和 STM32F767IGK6。用户不需要关心 XML 中的该域。

2.5 硬件配置类

硬件配置类负责配置硬件平台的初始化、用户添加进系统的额外存储器以及系统中可以被多个进程共享的共享存储器。它有三个配置组，并且分别位于不同的选项卡上。

2.5.1 初始化配置组<RME>

初始化配置组包括了架构的初始化参数和芯片的初始化参数。这些参数是依芯片而异的，需要查看具体的架构手册。

2.5.1.1 架构配置组<Platform>

该配置组包含架构本身特有的配置项，其具体内容需要参见各个架构的架构手册。

2.5.1.2 芯片配置组<Chip>

该配置组包含每个芯片类特有的配置项，其具体内容需要参见各个架构的架构手册。

2.5.2 额外内存组<Extmem><En>

该配置组包含了要添加进系统的额外内存块的属性。额外内存是指微控制器可以访问的外部存储器，比如外扩的 SDRAM、SRAM 等等。如果需要生成器调用它们，那么就必须将它们的信息登记在该组内部。在工程刚刚建立时，该组的内容为空。若要添加额外内存，应当点击配置项展示区的“+”号来添加一个内存项。每一个额外内存项都有需要用户填充的如下属性。

2.5.2.1 额外内存段基址项[H]<Base>

该配置项是一个必填的十六进制值。用户应该填入该外部内存段的起始地址。

2.5.2.2 额外内存段长度项[H]<Size>

该配置项是一个必填的十六进制值。用户应该填入该外部内存段的长度。

2.5.2.3 额外内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该段存储器用于何种用途。

2.5.2.4 额外内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个内存段属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。编译器在分配内存时，仅会安排属性是此属性子集的内存段到这片内存上。比如，用户若指定额外内存段的属性不包括可写[1]，那么编译器将绝不会将可写内容分配在该内存段上。

2.5.3 共享内存组<Shmem><Sn>

该配置组包含了系统中存在的共享内存块的属性。共享内存是指可以被多个进程或虚拟机访问的内存块，比如缓冲区等等，它类似于 Linux 等系统中的共享内存。如果需要在声明进程或虚拟机时调用它

[1] 比如外挂 XIP SPI Flash

们，那么就必须将它们的信息登记在该组内部。在工程刚刚建立时，该组的内容为空。若要添加额外内存，应当点击配置项展示区的“+”号来添加一个内存项。每一个共享内存项都有需要用户填充的如下属性。

2.5.3.1 共享内存段名称项[S]<Name>

该配置项是一个必填的字段，要求用户输入该共享内存段的名称。该名称必须是合法的标识符，并且在各个不同类别的共享内存段之间必须不分大小写地不同。比如，如果已经有一个代码类共享内存段称为 A，那么不能再有一个代码类共享内存段称为 a，但是可以有一个数据类型共享内存段称为 A。

2.5.3.2 共享内存段基址项[H]<Base>

该配置项是一个可选的十六进制值。用户应该填入该共享内存段的起始地址。如果不填写，则代表该共享内存段的位置由编译器自动分配。

它对应于 XML 文件中的<Project><Shmem><Sn>标签。

2.5.3.3 共享内存段长度项[H]<Size>

该配置项是一个必填的十六进制值。用户应该填入该共享内存段的长度。

2.5.3.4 共享内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该共享内存段分配到哪一种存储器。

2.5.3.5 共享内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个内存段属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。编译器在检查在各个进程或虚拟机中声明的对共享内存的引用时，仅会允许这个属性的子集出现。比如，用户若指定该共享内存段的属性不包括可写，那么编译器会检查并确保后续的对共享内存的引用不包括写属性。

2.6 内核配置类<Project><RME>

工程配置类负责工程本身的基本信息，以及用户要向工程中额外添加的外部内存的声明。它还会继承用户在启动编译器时输入的若干信息，并予以显示。该配置类没有配置组，所有的配置项将会被直接展示。

2.6.1 内核代码内存基址项[H]<Code_Base>

该配置项是一个必填的十六进制值。用户应该填入内核所使用的代码段的基址。

2.6.2 内核代码内存长度项[H]<Code_Size>

该配置项是一个必填的十六进制值。用户应该填入内核所使用的代码段的长度。内核的代码段包括了内核的汇编代码以及其只读数据区，通常在某个架构上是一个固定值。

2.6.3 内核数据内存基址项[H]<Data_Base>

该配置项是一个必填的十六进制值。用户应该填入内核所使用的数据段的基址。

2.6.4 内核数据内存长度项[H]<Data_Size>

该配置项是一个必填的十六进制值。用户应该填入内核所使用的数据段的长度。内核的数据段包括了内核的可读写数据区、内核对象用内存区和内核栈区，通常要根据创建的内核对象的多寡进行调整。如果生成器报告内核数据内存耗尽而无法分配内核对象，那么可以尝试将这个值修改的大一些。

2.6.5 内核栈长度项[H]<Stack_Size>

该配置项是一个必填的十六进制值。用户应该填入内核所使用的内核栈段的长度。作为微控制器上运行的微内核，4kB 的内核栈对于任何用途都足够了。

2.6.6 额外内核对象用内存长度项[H]<Extra_Kmem>

该配置项是一个必填的十六进制值。用户应该填入在系统创建完所有与生成器生成的代码相关的的内核对象后，还额外需要保留给用户用来创建内核对象的额外内核内存长度。它只有在用户要手动在生成器生成的 RVM 启动代码中添加内核对象的创建代码时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

2.6.7 内核内存分配粒度项[D]<Kmem_Order>

该配置项是一个必选的下拉选项框，要求用户在 8 字节，16 字节和 32 字节这三项中勾选一个最适合自己的系统的内核内存分配粒度。系统在创建内核对象时会将所有的内核内存都对齐到这一粒度。如果系统本身的内存比较少，推荐使用 8 字节对齐分配以减少碎片；如果系统本身的内存较多，则可以采用 32 字节分配以降低内存分配登记表本身的大小。在默认状况下，设置为 16 字节分配即可。

2.6.8 内核优先级数量项[N]<Kern_Prios>

该配置项是一个必填的整数值，决定了系统中内核态优先级的数量。用户应该填入一个是处理器字长的整数倍的十进制数值。通常而言，只要填入处理器字长就足够了，比如是 32 位处理器则填写 32。

2.6.9 内核编译器优化等级项[D]<Compiler><Optimization>

该配置项是一个必选的下拉选项框，决定了编译内核时所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）和极限优化（O3）。

2.6.10 内核编译器优化方式项[D]<Compiler><Prioritization>

该配置项是一个必选的下拉选项框，决定了编译内核时编译器的优化方向。可选择的优化等级有大小优化（Size）和时间优化（Time）。当优化等级为不优化（O0）时，本项不起作用。

2.7 用户态库配置类<Project><RVM>

用户态库配置类负责用户态库和用户态共享内存的配置。该配置类没有配置组，所有的配置项将会被直接展示。

2.7.1 用户态库代码内存长度项[H]<Code_Size>

该配置项是一个必填的十六进制值。用户应该填入 RVM 所使用的代码段的长度。RVM 的代码段包括了其汇编代码以及其只读数据区，通常在某个架构上是一个固定值。需要注意的是，RVM 的代码起始地址永远直接位于内核代码段后面，因此代码内存基址是自动决定好的，无需用户手动设置。

2.7.2 用户态库数据内存长度项[H]<Data_Size>

该配置项是一个必填的十六进制值。用户应该填入 RVM 所使用的数据段的长度。RVM 的数据段包括了其可读写数据区和栈区，通常要根据是否开启虚拟机功能以及虚拟机的数量决定。如果在编译 RVM 时提示内存不够，那么可以尝试将这个值修改的大一些。需要注意的是，RVM 的数据起始地址永远直接位于内核数据段后面，因此数据内存基址是自动决定好的，无需用户手动设置。

2.7.3 额外权能表槽位数项[N]<Extra_Captbl>

该配置项是一个必填的整数值。用户应该填入在系统创建完所有与生成器生成的代码相关的内核对象后，还额外需要保留给用户用来创建内核对象的额外权能表槽位数。它只有在用户要手动在生成器生成的 RVM 启动代码中添加内核对象的创建代码时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

2.7.4 守护进程栈长度项[H]<Stack_Size>

该配置项是一个必填的十六进制值。用户应该填入各个守护进程的栈的长度。通常而言，1kB 对任何场合都足够了。

2.7.5 虚拟化优先级数[N]<Virt_Prios>

该配置项是一个必填的整数值。用户应该填入虚拟化优先级的数量。该优先级数量仅对虚拟机生效，而且必须是处理器字长的整数倍。通常而言，填充为处理器字长，如 32 位处理器则填充 32 就可以了。如果不使用虚拟机的话，这里可以填充为 0。

2.7.6 虚拟机事件源数[N]<Virt_Evts>

该配置项是一个必填的整数值。用户应该填入虚拟机的事件源的数量，它必须是处理器字长的整数倍，而且在使用虚拟机的情况下不能为 0。同时，事件源的数量不能超过 1024 的硬性限制。事件源是用于从原生进程到虚拟机以及虚拟机本身之间的沟通的，它可以被映射到虚拟机的虚拟中断向量。

2.7.7 虚拟机向量映射数[N]<Virt_Maps>

该配置项是一个必填的整数值。用户应该填入从事件源或物理中断源到虚拟中断向量映射的总数目，它必须大于事件源的数量，而且在使用虚拟机的情况下不能为 0。一个事件源或物理中断源可以被映射到多个虚拟向量，每一次从具体源到具体虚拟向量的映射都会消耗一个映射数。

2.7.8 用户态库编译器优化等级项[D]<Compiler><Optimization>

该配置项是一个必选的下拉选项框，决定了编译 RVM 时所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）和极限优化（O3）。

2.7.9 用户态库编译器优化方式项[D]<Compiler><Prioritization>

该配置项是一个必选的下拉选项框，决定了编译 RVM 时编译器的优化方向。可选择的优化等级有大小优化（Size）和时间优化（Time）。当优化等级为不优化（O0）时，本项不起作用。

2.8 原生进程配置类<Project><Process><Pn>

本类负责系统中的原生进程的配置。系统中可以有一个或多个原生进程，因此该类可以有一个或多个实例。这些实例均会被显示在左侧第二栏中。每一个实例都有十个配置组，它们分别占据一个选项卡。

2.8.1 进程基本信息组

该配置组包含进程的基本信息和配置。其具体包含的配置项请见下。

2.8.1.1 进程名称项[S]<Name>

该配置项是一个必填的字段，要求用户输入进程的名称。该名称必须是一个合法的标识符，并且和系统中的其他进程、虚拟机的名称不得重复。

2.8.1.2 进程额外权能表槽位数项[N]<Extra_Captbl>

该配置项是一个必填的整数值。用户应该填入额外需要保留给用户使用的该进程的权能表槽位数。它只有在用户要手动添加内核对象到该进程的权能表时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

2.8.1.3 进程编译器优化等级项[D]<Compiler><Optimization>

该配置项是一个必选的下拉选项框，决定了编译该进程时所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）和极限优化（O3）。

2.8.1.4 进程编译器优化方式项[D]<Compiler><Prioritization>

该配置项是一个必选的下拉选项框，决定了编译该进程时编译器的优化方向。可选择的优化等级有大小优化（Size）和时间优化（Time）。当优化等级为不优化（O0）时，本项不起作用。

2.8.2 进程内存组<Memory><Mn>

该配置组包含了该进程中存在的私有内存块的属性。该组中至少要包含一块代码内存和一块数据内存；声明的第一块代码内存将作为主代码内存来存放进程的全部代码，声明的第一块数据内存将作为主数据存储器来存放进程的全部数据区。在进程刚刚建立时，该组的内容为空。可以点击配置项展示区的“+”号来添加一个内存项。

2.8.2.1 进程内存段名称项(S)<Name>

该配置项是一个可选的字段，要求用户输入该进程内存段的名称。该名称必须是合法的标识符，并且在一个进程内部的内存段之间必须不分大小写地不同。如果用户不填写该字段，那么该内存段将作为匿名内存段存在。

2.8.2.2 进程内存段基址项(H)<Base>

该配置项是一个可选的十六进制值。用户应该填入该进程内存段的起始地址。如果不填写，则代表该进程内存段的位置由编译器自动分配。

2.8.2.3 进程内存段长度项[H]<Size>

该配置项是一个必填的十六进制值。用户应该填入该进程内存段的长度。

2.8.2.4 进程内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该进程内存段分配到哪一种存储器。

2.8.2.5 进程内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个内存段属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。

2.8.3 进程共享内存引用组<Shmem><Sn>

该配置组包含了该进程要引用的共享内存块的属性。在进程刚刚建立时，该组的内容为空。可以点击配置项展示区的“+”号来添加一个共享内存引用项。

2.8.3.1 进程共享内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。

2.8.3.2 进程共享内存段名称项[S]<Name>

该配置项是一个必选的下拉选项框，要求用户选择一个已经声明的共享内存段。一经填充，共享内存段的基址和长度将会自动决定。

2.8.3.3 进程共享内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个共享内存段映射到该进程中时应该具备的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。在共享内存段声明时的属性中不存在的属性在此不能选择。

2.8.4 进程中线程组<Thread><Tn>

该配置组包含了在该进程中运行的线程的属性。在进程刚刚建立时，该组的内容为空。可以点击配置项展示区的“+”号来添加一个线程。每一个进程中至少要声明有一个线程。

2.8.4.1 线程名称项[S]<Name>

该配置项是一个必填的字段，决定了线程的名称。该名称必须是一个合法标识符，而且在同一个进程内应当是唯一的。

2.8.4.2 线程栈大小项[H]<Stack_Size>

该配置项是一个必填的十六进制值，决定了线程栈的大小。

2.8.4.3 线程参数项[H]<Parameter>

该配置项是一个可选的十六进制值，决定了要交给线程的参数。如果不填写，那么默认为 0。

2.8.4.4 线程优先级项[N]<Priority>

该配置项是一个必填的整数值，决定了线程的实时优先级。该优先级必须在内核优先级数-2 和 5 之间。比如，如果内核优先级数是 32，那么该值可以在 5 和 30 之间挑选。该值越大，那么线程的优先级越高。

2.8.5 进程中迁移调用目标组<Invocation><In>

该配置组包含了在该进程中运行的迁移调用的相关信息。在进程刚刚建立时，该组为空。可以点击配置项展示区的“+”号来添加一个迁移调用目标。在进程中可以有迁移调用目标也可以没有；当其他进程内的线程激活迁移调用目标时，即跳转到该迁移调用目标处执行，执行完后再返回原进程。

2.8.5.1 迁移调用名称项[S]<Name>

该配置项是一个必填的字段，决定了迁移调用的名称。该名称必须是一个合法标识符，而且在同一个进程内应当是唯一的。

2.8.5.2 迁移调用栈大小项[H]<Stack_Size>

该配置项是一个必填的十六进制值，决定了迁移调用栈的大小。

2.8.6 进程中迁移调用入口组<Port><Pn>

该配置组包含了从该进程可以调用的迁移调用的相关信息。在进程刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个迁移调用入口。在进程中可以有迁移调用入口也可以没有；迁移调用入口一旦激活，激活它的线程就会跳转到迁移调用目标处执行，并在完成后返回自己的进程继续执行。

2.8.6.1 迁移调用目标入口所在进程项[D]<Process>

该配置项是一个必选的下拉选项框，决定了迁移调用目标所在进程。只有声明了迁移调用入口的进程在这个下拉列表中才会展示。

2.8.6.2 迁移调用目标入口项[D]<Name>

该配置项是一个必选的下拉选项框，决定了迁移调用要进入的具体目标。

2.8.7 进程中接收信号端点组<Receive><Rn>

该配置组包含了该进程中的线程可以阻塞接收信息的所有接收信号端点。在进程刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个接收信号端点。在进程中可以有接收信号端点也可以没有；一旦与接收信号端点对应的发送信号端点遭到激活，接收信号端点上的线程就会解除等待。

2.8.7.1 信号接收端点名称项[S]<Name>

该配置项是一个必填的字段，决定了接收端点的名称。该名称必须是一个合法标识符，而且在同一个进程内应当是唯一的。

2.8.8 进程中发送信号端点组<Send><Sn>

该配置组包含了该进程中的线程可以发送到的所有发送信号端点。在进程刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个发送信号端点。在进程中可以有发送信号端点也可以没有；一旦发送信号端点遭到激活，与之对应的接收信号端点上的线程就会解除等待。

2.8.8.1 对应的接收信号进程项[D]<Process>

该配置项是一个必选的下拉选项框，决定了接收信号端点所在进程。只有声明了接收信号端点的进程在这个下拉列表中才会展示。

2.8.8.2 对应的接收信号端点项[D]<Name>

该配置项是一个必选的下拉选项框，决定了该发送信号端点对应的具体接收信号端点。

2.8.9 进程中物理向量信号端点组<Vector><Vn>

本配置组包含了该进程中的线程可以阻塞接收的所有物理中断的向量信号端点。在进程刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个物理向量信号端点。在进程中可以有物理向量接收端点也可以没有；一旦该中断向量遭到激活，与之对应的物理向量端点上的线程就会解除等待。

2.8.9.1 物理向量名称项[S]<Name>/[N]<Number>

该配置项是一个必选的下拉选项框，可以选择物理向量的名称。只有该芯片具备的物理中断向量的名称才会在此展示。在选中名称后，编译器会自动从数据库读取该向量的向量号并填充在<Number>标签中，无需用户手动设置。

2.8.10 进程中内核功能组<Kernel><Kn>

本配置组包含了该进程中的线程可以激活的所有内核功能调用。在进程刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个内核功能调用。在进程中可以有内核功能调用也可以没有；进程中的线程可以通过激活内核调用来访问一些特殊的需要在内核态执行的操作。需要注意的是，如果内核功能被滥用，系统的安全性会遭到但严重损失。

2.8.10.1 内核功能名称项[S]<Name>

该配置项是一个必填的字段，决定了该内核功能调用的名称。该名称必须是一个合法标识符，而且在同一个进程内应当是唯一的。

2.8.10.2 内核功能主功能号起始值项[H]<Start>

该配置项是一个必填的十六进制值，决定了该内核功能调用允许的主功能号的开始点。

2.8.10.3 内核功能主功能号结束值项[H]<End>

该配置项是一个必填的十六进制值，决定了该内核功能调用允许的主功能号的结束点。这个值不能比起始值低。如一个内核功能调用的主功能号开始点为 0x10，结束点为 0x20，那么可以调用的内核功能的主功能号在 0x10 到 0x20 之间，包括了 0x10 和 0x20。

2.9 虚拟机配置类<Project><Virtual><Vn>

本类负责系统中的虚拟机的配置。系统中可以有一个或多个虚拟机，因此该类可以有一个或多个实例。这些实例均会被显示在左侧第二栏中。每一个实例都有六个配置组，它们分别占据一个选项卡。由于虚拟机本身也是一种进程，因此这些选项卡和原生进程的很相似，只是某些原生进程具备而虚拟机不具备的选项卡将不可点击。

2.9.1 虚拟机基本信息组

本配置组包含了虚拟机的基本信息。

2.9.1.1 虚拟机名称项[S]<Name>

该配置项是一个必填的字段，要求用户输入虚拟机的名称。该名称必须是一个合法的标识符，并且和系统中的其他进程、虚拟机的名称不得重复。

2.9.1.2 虚拟机操作系统选择项[D]<Guest>

该配置项是一个必选的下拉选项框，要求用户输入拟在虚拟机中运行的客户操作系统的名称。所有在此可以选择自动安装的操作系统都在此列表中出现。用户只需选择操作系统本身，即可自动安装操作系统的源代码到虚拟机中。

2.9.1.3 虚拟机额外权能表槽位数项[N]<Extra_Captbl>

该配置项是一个必填的整数值。用户应该填入额外需要保留给用户使用的该虚拟机的权能表槽位数。它只有在用户要手动添加内核对象到该进程的权能表时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

2.9.1.4 虚拟机工作线程栈大小项[H]<Stack_Size>

该配置项是一个必填的十六进制值，决定了虚拟机工作线程的栈的大小。通常而言，1kB 对任何情况都够用了。需要注意的是，这个工作线程的栈大小并不是虚拟机内操作系统管理的内部线程栈大小，因此可以设置得比较小。

2.9.1.5 虚拟机优先级项[N]<Priority>

该配置项是一个必填的整数值，决定了虚拟机的虚拟化优先级。该优先级必须在虚拟机优先级数-1 和 0 之间。比如，如果虚拟化优先级数是 32，那么该值可以在 0 和 31 之间挑选。该值越大，那么虚拟机的优先级越高。

需要注意的是，虚拟机优先级和原生进程中的线程优先级不是一回事；虚拟机的优先级比任何原生进程都低，它使用的优先级是虚拟化优先级。只有当所有原生进程中的线程都阻塞时，高优先级虚拟机才运行；当高优先级的虚拟机完全阻塞不运行时，才轮到低优先级虚拟机运行。

2.9.1.6 虚拟机时间片数[N]<Slices>

该配置项是一个必填的大于 0 的整数值，决定了虚拟机的时间片数量。在系统中，不同优先级的虚拟机之间为全抢占关系，而相同优先级的虚拟机之间则为时间片轮转调度关系。一般推荐将这个值设的比较大，比如 50 或 100。

2.9.1.7 虚拟机时钟中断周期[N]<Period>

该配置项是一个必填的大于 0 的整数值，决定了虚拟机的虚拟时钟中断的周期。这个虚拟时钟中断相当于一般的 RTOS 的定时器中断。一般推荐将这个值设的比较大，比如 50 或 100。

2.9.1.8 虚拟机看门狗超时时间[N]<Watchdog>

该配置项是一个必填的整数值，决定了虚拟机的看门狗超时时间。时间的单位是系统底层的定时器中断时基。一旦系统失去响应而导致看门狗超时，RVM 即会重启该超时的虚拟机。通常而言看门狗超时时间推荐设为 1 秒左右，这大概是 100 个或 1000 个底层系统定时器嘀嗒。如果将该值设置为 0，那么意味着看门狗对本虚拟机未启用。看门狗超时时间的计算是按照虚拟机运行的时间片数来计算的，而非按照系统中总经历的时间。如果看门狗超时时间设置为 1 秒，那么意味着虚拟机的净运行时间需要超过 1 秒且未喂狗才会导致看门狗超时。

2.9.1.9 虚拟机虚拟中断数量[N]<Vect_Num>

该配置项是一个必填的大于 0 的整数值，而且必须是处理器字长的整数倍。它决定了虚拟机的虚拟中断源的数量。

2.9.1.10 虚拟机编译器优化等级项[D]<Compiler><Optimization>

该配置项是一个必选的下拉选项框，决定了编译该虚拟机时所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）和极限优化（O3）。

2.9.1.11 虚拟机编译器优化方式项[D]<Compiler><Prioritization>

该配置项是一个必选的下拉选项框，决定了编译该虚拟机时编译器的优化方向。可选择的优化等级有大小优化（Size）和时间优化（Time）。当优化等级为不优化（O0）时，本项不起作用。

2.9.2 虚拟机内存组<Memory><Mn>

该配置组包含了该虚拟机中存在的私有内存块的属性。该组中至少要包含一块代码内存和一块数据内存；声明的第一块代码内存将作为主代码内存来存放虚拟机的全部代码，声明的第一块数据内存将作为主数据存储器来存放虚拟机的全部数据区。在虚拟机刚刚建立时，该组的内容为空。可以点击配置项展示区的“+”号来添加一个内存项。

2.9.2.1 虚拟机内存段名称项(S)<Name>

该配置项是一个可选的字段，要求用户输入该虚拟机内存段的名称。该名称必须是合法的标识符，并且在一个虚拟机内部的内存段之间必须不分大小写地不同。如果用户不填写该字段，那么该内存段将作为匿名内存段存在。

2.9.2.2 虚拟机内存段基址项(H)<Base>

该配置项是一个可选的十六进制值。用户应该填入该虚拟机内存段的起始地址。如果不填写，则代表该虚拟机内存段的位置由编译器自动分配。

2.9.2.3 虚拟机内存段长度项[H]<Size>

该配置项是一个必填的十六进制值。用户应该填入该虚拟机内存段的长度。

2.9.2.4 虚拟机内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该虚拟机内存段分配到哪一种存储器。

2.9.2.5 虚拟机内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个内存段属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。

2.9.3 虚拟机共享内存引用组

该配置组包含了该虚拟机要引用的共享内存块的属性。在虚拟机刚刚建立时，该组的内容为空。可以点击配置项展示区的“+”号来添加一个共享内存引用项。

2.9.3.1 虚拟机共享内存段类别项[D]<Type>

该配置项是一个必选的下拉选项框，要求用户选择一个内存段类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。

2.9.3.2 虚拟机共享内存段名称[S]<Name>

该配置项是一个必选的下拉选项框，要求用户选择一个已经声明的共享内存段。一经填充，共享内存段的基址和长度将会自动决定。

2.9.3.3 虚拟机共享内存段属性项[C]<Attribute>

该配置项是一个必选的多项选择框，要求用户勾选一个或者多个共享内存段映射到该虚拟机中时应该具备的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。在共享内存段声明时的属性中不存在的属性在此不能选择。

2.9.4 虚拟机中发送信号端点组

该配置组包含了该虚拟机中可以发送到的所有发送信号端点。在虚拟机刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个发送信号端点。在虚拟机中可以有发送信号端点也可以没有；一旦发送信号端点遭到激活，与之对应的原生进程中的接收信号端点上的线程就会解除等待。虚拟机中只有发送信号端点而无接收信号端点；这是虚拟机向原生进程主动进行通信的唯一方法。

2.9.4.1 对应的接收信号进程项[D]<Process>

该配置项是一个必选的下拉选项框，决定了接收信号端点所在进程。只有声明了接收信号端点的进程在这个下拉列表中才会展示。

2.9.4.2 对应的接收信号端点项[D]<Name>

该配置项是一个必选的下拉选项框，决定了该发送信号端点对应的具体接收信号端点。

2.9.5 虚拟机中内核功能组

本配置组包含了该虚拟机中可以激活的所有内核功能调用。在虚拟机刚刚建立时，该项为空。可以点击配置项展示区的“+”号来添加一个内核功能调用。在虚拟机中可以有内核功能调用也可以没有；虚拟机可以通过激活内核调用来访问一些特殊的需要在内核态执行的操作。需要注意的是，如果内核功能被滥用，系统的安全性会遭到但严重损失。

2.9.5.1 内核功能名称项[S]<Name>

该配置项是一个必填的字段，决定了该内核功能调用的名称。该名称必须是一个合法标识符，而且在同一个进程内应当是唯一的。

2.9.5.2 内核功能主功能号起始值项[H]<Start>

该配置项是一个必填的十六进制值，决定了该内核功能调用允许的主功能号的开始点。

2.9.5.3 内核功能主功能号结束值项[H]<End>

该配置项是一个必填的十六进制值，决定了该内核功能调用允许的主功能号的结束点。这个值不能比起始值低。如一个内核功能调用的主功能号开始点为 0x10，结束点为 0x20，那么可以调用的内核功能的主功能号在 0x10 到 0x20 之间，包括了 0x10 和 0x20。

2.10 工程生成按钮

工程生成按钮在界面左边栏的最下方。点击这个按钮后，会弹出一个界面，让用户选择所需要生成的工程对应的 IDE 和地址。每一个架构都有可能支持不同的 IDE；当这两者均选定后，点击生成工程按钮，即可生成最终的工程。在手机 APP 上，工程生成的功能不提供，而是生成可以被电脑版读取的 XML 文件。工程生成的界面如下。

2.11 一个典型的工程 XML 结构

该 GUI 模块生成的一个典型的将要送入输入分析模块的 XML 的结构如下表所示。本 XML 中含有两个进程和一个虚拟机。

表 2-1 一个典型的工程 XML 结构

内容	参见
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>	
<Project>	
<Name>Test</Name>	工程配置类[S]
<Platform>A7M</Platform>	目标架构项[D]
<Chip_Class>STM32F767IG</Chip_Class>	芯片精确类别项[D]
<Chip_Full>STM32F767IGT6</Chip_Full>	芯片型号项[D]
<RME>	
<Code_Base>0x08000000</Code_Base>	内核代码内存基址项[H]
<Code_Size>0x10000</Code_Size>	内核代码内存长度项[H]
<Data_Base>0x20000000</Data_Base>	内核数据内存基址项[H]
<Data_Size>0x10000</Data_Size>	内核数据内存长度项[H]
<Stack_Size>0x1000</Stack_Size>	内核栈长度项[H]
<Extra_Kmem>0x1000</Extra_Kmem>	额外内核对象用内存长度项[H]
<Kmem_Order>4</Kmem_Order>	内核内存分配粒度项[D]
<Kern_Prios>32</Kern_Prios>	内核优先级数量项[N]
<Compiler>	
<Optimization>O2</Optimization>	内核编译器优化等级项[D]
<Prioritization>Time</Prioritization>	内核编译器优化方式项[D]
</Compiler>	
<Platform>	架构配置组

内容	参见
<NVIC_Grouping>2-6</NVIC_Grouping>	
<Systick_Value>21600</Systick_Value>	
</Platform>	
<Chip>	芯片配置组
<XTAL>25</XTAL>	
<PLLM>25</PLLM>	
<PLLN>432</PLLN>	
<PLLP>2</PLLP>	
<PLLQ>9</PLLQ>	
<IRQ>WWDG,EXTI0</IRQ>	
</Chip>	
</RME>	
<RVM>	
<Code_Size>0x10000</Code_Size>	用户态库代码内存长度项[H]
<Data_Size>0x10000</Data_Size>	用户态库数据内存长度项[H]
<Stack_Size>0x1000</Stack_Size>	守护进程栈长度项[H]
<Extra_Captbl>32</Extra_Captbl>	
<Compiler>	
<Optimization>O2</Optimization>	用户态库编译器优化等级项[D]
<Prioritization>Time</Prioritization>	用户态库编译器优化方式项[D]
</Compiler>	
<Virt_Prios>32</Virt_Prios>	虚拟化优先级数[N]
<Virt_Evts>32</Virt_Evts>	虚拟机事件源数[N]
<Virt_Maps>128</Virt_Maps>	虚拟机向量映射数[N]
</RVM>	
<Extmem>	
<E1>	
<Base>0xC0000000</Base>	额外内存段基址项[H]
<Size>0x02000000</Size>	额外内存段长度项[H]
<Type>Data</Type>	额外内存段类别项[D]
<Attribute>RWBCS</Attribute>	额外内存段属性项[C]

内容	参见
</E1>	
</Extmem>	
<Shmem>	
<S1>	
<Name>Shared1</Name>	共享内存段名称项[S]
<Base>Auto</Base>	共享内存段长度项[H]
<Size>0x10000</Size>	共享内存段基址项[H]
<Type>Data</Type>	共享内存段类别项[D]
<Attribute>RWXBCS</Attribute>	共享内存段属性项[C]
</S1>	
</Shmem>	
<Process>	
<P0>	
<Name>ProP0</Name>	进程名称项[S]
<Extra_Captbl>9</Extra_Captbl>	进程额外权能表槽位数项[N]
<Compiler>	
<Optimization>O2</Optimization>	进程编译器优化等级项[D]
<Prioritization>Time</Prioritization>	进程编译器优化方式项[D]
</Compiler>	
<Memory>	
<M1>	
<Name>Main_Code</Name>	进程内存段名称项(S)
<Base>0x08020000</Base>	进程内存段基址项(H)
<Size>0x10000</Size>	进程内存段长度项[H]
<Type>Code</Type>	进程内存段类别项[D]
<Attribute>RXBCS</Attribute>	进程内存段属性项[C]
</M1>	
<M2>	
<Name>Main_Data</Name>	进程内存段名称项(S)
<Base>Auto</Base>	进程内存段基址项(H)
<Size>0x10000</Size>	进程内存段长度项[H]

内容	参见
<Type>Data</Type>	进程内存段类别项[D]
<Attribute>RWBCS</Attribute>	进程内存段属性项[C]
</M2>	
<M3>	
<Base>0xF0000000</Base>	进程内存段基址项(H)
<Size>0x10000</Size>	进程内存段长度项[H]
<Type>Device</Type>	进程内存段类别项[D]
<Attribute>RWS</Attribute>	进程内存段属性项[C]
</M3>	
</Memory>	
<Shmem>	
<S1>	
<Name>Shared1</Name>	进程共享内存段名称项[S]
<Type>Data</Type>	进程共享内存段类别项[D]
<Attribute>RBCS</Attribute>	进程共享内存段属性项[C]
</S1>	
</Shmem>	
<Thread>	
<T1>	
<Name>Thd1</Name>	线程名称项[S]
<Stack_Size>0x1000</Stack_Size>	线程栈大小项[H]
<Parameter>0x1234</Parameter>	线程参数项[H]
<Priority>17</Priority>	线程优先级项[N]
</T1>	
</Thread>	
<Invocation>	
</Invocation>	
<Port>	
<P0>	
<Name>Inv1</Name>	迁移调用目标入口项[D]
<Process>Proc2</Process>	迁移调用目标入口所在进程项[D]

内容	参见
</P0>	
</Port>	
<Receive>	
<R1>	
<Name>Recv1</Name>	信号接收端点名称项[S]
</R1>	
</Receive>	
<Send>	
</Send>	
<Vector>	
</Vector>	
<Kernel>	
<K1>	
<Name>KfunP0</Name>	内核功能名称项[S]
<Start>0x60</Start>	内核功能主功能号起始值项[H]
<End>0x90</End>	内核功能主功能号结束值项[H]
</K1>	
</Kernel>	
</P0>	
<P2>	
<Name>Proc2</Name>	进程名称项[S]
<Extra_Captbl>0</Extra_Captbl>	进程额外权能表槽位数项[N]
<Compiler>	
<Optimization>O3</Optimization>	进程编译器优化等级项[D]
<Prioritization>Size</Prioritization>	进程编译器优化方式项[D]
</Compiler>	
<Memory>	
<M1>	
<Base>Auto</Base>	进程内存段基址项(H)
<Size>0x10000</Size>	进程内存段长度项[H]
<Type>Code</Type>	进程内存段类别项[D]

内容	参见
<Attribute>RXBCS</Attribute>	进程内存段属性项[C]
</M1>	
<M2>	
<Name>Main_Data</Name>	进程内存段名称项(S)
<Base>Auto</Base>	进程内存段基址项(H)
<Size>0x10000</Size>	进程内存段长度项[H]
<Type>Data</Type>	进程内存段类别项[D]
<Attribute>RWXBCS</Attribute>	进程内存段属性项[C]
</M2>	
</Memory>	
<Shmem>	
<S1>	
<Name>Shared1</Name>	进程共享内存段名称项[S]
<Type>Data</Type>	进程共享内存段类别项[D]
<Attribute>RWBCS</Attribute>	进程共享内存段属性项[C]
</S1>	
</Shmem>	
<Thread>	
<T1>	
<Name>Thd1</Name>	线程名称项[S]
<Stack_Size>0x1000</Stack_Size>	线程栈大小项[H]
<Parameter>0x1234</Parameter>	线程参数项[H]
<Priority>16</Priority>	线程优先级项[N]
</T1>	
</Thread>	
<Invocation>	
<I1>	
<Name>Inv1</Name>	迁移调用名称项[S]
<Stack_Size>0x1000</Stack_Size>	迁移调用栈大小项[H]
</I1>	
</Invocation>	

内容	参见
<Port>	
</Port>	
<Receive>	
</Receive>	
<Send>	
<S1>	
<Name>Recv1</Name>	对应的接收信号端点项[D]
<Process>ProP0</Process>	对应的接收信号进程项[D]
</S1>	
</Send>	
<Vector>	
<V1>	
<Name>TIM1</Name>	物理向量名称项[S]
<Number>8</Number>	物理向量序号项[N]
</V1>	
</Vector>	
<Kernel>	
</Kernel>	
</P2>	
</Process>	
<Virtual>	
<V1>	
<Name>Virt1</Name>	虚拟机名称项[S]
<Guest>RMP</Guest>	虚拟机操作系统选择项[D]
<Extra_Captbl>0</Extra_Captbl>	虚拟机额外权能表槽位数项[N]
<Stack_Size>0x400</Stack_Size>	虚拟机工作线程栈大小项[H]
<Priority>5</Priority>	虚拟机优先级项[N]
<Slices>10</Slices>	虚拟机时间片数[N]
<Period>100</Period>	虚拟机时钟中断周期[N]
<Watchdog>1000</Watchdog>	虚拟机看门狗超时时间[N]
<Vect_Num>100</Vect_Num>	虚拟机虚拟中断数量[N]

内容	参见
<Compiler>	
<Optimization>O2</Optimization>	虚拟机编译器优化等级项[D]
<Prioritization>Time</Prioritization>	虚拟机编译器优化方式项[D]
</Compiler>	
<Memory>	
<M1>	
<Name>Main_Code</Name>	虚拟机内存段名称项(S)
<Base>Auto</Base>	虚拟机内存段基址项(H)
<Size>0x10000</Size>	虚拟机内存段长度项[H]
<Type>Code</Type>	虚拟机内存段类别项[D]
<Attribute>RXBCS</Attribute>	虚拟机内存段属性项[C]
</M1>	
<M2>	
<Name>Main_Data</Name>	虚拟机内存段名称项(S)
<Base>Auto</Base>	虚拟机内存段基址项(H)
<Size>0x10000</Size>	虚拟机内存段长度项[H]
<Type>Data</Type>	虚拟机内存段类别项[D]
<Attribute>RWXBCS</Attribute>	虚拟机内存段属性项[C]
</M2>	
</Memory>	
<Shmem>	
<S1>	
<Name>Shared1</Name>	虚拟机共享内存段名称[S]
<Type>Data</Type>	虚拟机共享内存段类别项[D]
<Attribute>RWBCS</Attribute>	虚拟机共享内存段属性项[C]
</S1>	
</Shmem>	
<Send>	
<S1>	
<Name>Recv1</Name>	对应的接收信号端点项[D]
<Process>ProP0</Process>	对应的接收信号进程项[D]

内容	参见
</S1>	
</Send>	
<Kernel>	
<K1>	
<Name>KfunP0</Name>	内核功能名称项[S]
<Start>0x100</Start>	内核功能主功能号起始值项[H]
<End>0x200</End>	内核功能主功能号结束值项[H]
</K1>	
</Kernel>	
</V1>	
</Virtual>	
</Project>	

2.12 本章参考文献

无

第 3 章　输入分析模块

3.1　输入分析模块概述

输入分析模块主要负责读取 XML，检查 XML 文件中是否存在显而易见的错误，以及将 XML 文件转化为内部的数据结构表示。我们在下面的章节中一一介绍输入分析模块的各个子模块的原理以及可能在各个阶段产生的报错信息。

3.2　命令行参数分析（C0000-C9999）

该模块分析用户输入的命令行参数。命令行参数有三个组成部分，分别是-i，-o 和-f。

-i 指定了用户 XML 的路径，-o 指定了输出工程的路径，-f 则指定了输出的工具链。

例如：-i path/to/input.xml -o path/to/output/folder/ -f makefile，意为 XML 文件存放在 path/to/input.xml 的位置，输出的工程则输出到 path/to/output/folder/，输出的工具链为 makefile 工程格式并使用 GCC 编译。

本工具支持的所有输出格式如下：

表 3-1　支持的输出工具链

输出工具链选项	工程组织	编译器
makefile	Makefile	GCC
eclipse	Eclipse	GCC
keil	Keil uVision	ARMCC 5.0

本步骤可能产生的所有报错信息如下（C0000 -- C9999）：

表 3-2　命令行参数分析可能产生的报错信息

错误编号	错误信息	原因
C0000	Too many or too few input parameters.	输入了过少或过多的命令行参数。
C0001	More than one input file.	指定了多于一个的输入文件。
C0002	More than one output path.	指定了多于一个的输出路径。
C0003	More than one output format.	指定了多于一个的输出格式。
C0004	Unrecognized command line argument.	未能识别的命令行参数。
C0005	No input file specified.	没有指定输入文件。
C0006	No output path specified.	没有指定输出路径。
C0007	No output project format specified.	没有指定输出工程格式。

3.3 工程 XML 文件分析（P0000-P9999）

该模块分析工程描述 XML 文件，也即 GUI 模块生成的 XML 文件。它首先按照输入的路径读取 XML 文件生成其 DOM，然后逐步分析该 DOM 中的各个组件，并最终生成方便引用的内部数据结构。下面将分章节讲解每个分步骤的具体内容。

3.3.1 工程配置初步分析（P0000-P0099）

在本步骤中，系统中的额外内存信息、共享内存信息和一些工程基本信息会经分析后直接填充在内部数据结构中。可能产生的所有报错信息如下（P0000 -- P0099）：

表 3-3 工程配置初步分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0000	Project XML parsing failed.	工程 XML 文件存在语法错误，无法解析。
P0001	Project XML is malformed.	工程 XML 中不存在<Project>标签。
P0002	Name section is missing.	工程名称项[S]不存在。
P0003	Name section is empty.	工程名称项[S]为空。
P0004	Platform section is missing.	目标架构项[D]不存在。
P0005	Platform section is empty.	目标架构项[D]为空。
P0006	Chip class section is missing.	芯片类别项[D]不存在。
P0007	Chip class section is empty.	芯片类别项[D]为空。
P0008	Chip fullname section is missing.	芯片型号项[D]不存在。
P0009	Chip fullname section is empty.	芯片型号项[D]为空。
P0010	RME section is missing.	工程 XML 中不存在<RME>标签。
P0011	RVM section is missing.	工程 XML 中不存在<RVM>标签。
P0012	Extra memory section is missing.	工程 XML 中不存在<Extmem>标签。
P0013	Extra memory section parsing internal error.	解析<Extmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0014	Extra memory type section is missing.	额外内存段类别项[D]不存在。
P0015	Extra memory type section is empty.	额外内存段类别项[D]为空。
P0016	Extra memory type is malformed.	额外内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0017	Shared memory section is missing.	工程 XML 中不存在<Shmem>标签。
P0018	Shared memory section parsing internal	解析<Shmem>标签及其内容时遇到未知

错误编号	错误信息	原因
	error.	错误。请检查该标签的完整性。
P0019	Shared memory type section is missing.	共享内存段类别项[D]不存在。
P0020	Shared memory type section is empty.	共享内存段类别项[D]为空。
P0021	Shared memory name cannot be omitted.	共享内存段名称项[S]是必填的，它不可以被省略不写。
P0022	Shared memory type is malformed.	共享内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0023	Process section parsing internal error.	解析<Process>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0024	Virtual machine section parsing internal error.	解析<Virtual>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0025	Project XML and chip XML platform does not match.	工程 XML 和芯片 XML 中的平台类型不对应。请检查工程 XML 是否指定了错误的平台。
P0026	Project XML and chip XML chip class does not match.	工程 XML 和芯片 XML 中的芯片类别不对应。请检查工程 XML 是否指定了错误的芯片类别。
P0027	The specific chip designated in project XML not found in chip XML.	工程 XML 中指定的芯片在芯片 XML 中不被兼容。请检查是否指定了错误的芯片型号。
P0028	The specific platform is currently not supported.	工程 XML 中指定的平台现在不被支持。请检查是否指定了错误的平台。

3.3.2 内核配置分析（P0100-P0199）

本阶段分析内核的基本配置信息，它们都位于<RME>标签下。可能产生的所有报错如下表所示（P0100 -- P0199）：

表 3-4 内核配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0100	Code base section is missing.	内核代码内存基址项[H]不存在。
P0101	Code base is not a valid hex integer.	内核代码内存基址项[H]不是十六进制。
P0102	Code size section is missing.	内核代码内存长度项[H]不存在。
P0103	Code size is not a valid hex integer.	内核代码内存长度项[H]不是十六进制。

错误编号	错误信息	原因
P0104	Data base section is missing.	内核数据内存基址项[H]不存在。
P0105	Data base is not a valid hex integer.	内核数据内存基址项[H]不是十六进制。
P0106	Data size section is missing.	内核数据内存长度项[H]不是十六进制。
P0107	Data size is not a valid hex integer.	内核数据内存长度项[H]不是十六进制。
P0108	Stack size section is missing.	内核栈长度项[H]不存在。
P0109	Stack size is not a valid hex integer.	内核栈长度项[H]不是十六进制。
P0110	Extra kernel memory section is missing.	额外内核对象用内存长度项[H]不存在。
P0111	Extra kernel memory is not a valid hex integer.	额外内核对象用内存长度项[H]不是十六进制。
P0112	Kernel memory order section is missing.	内核内存分配粒度项[D]不存在。
P0113	Kernel memory order is not a valid unsigned integer.	内核内存分配粒度项[D]不是十进制。
P0114	Priority number section is missing.	内核优先级数量项[N]不存在。
P0115	Priority number is not a valid unsigned integer.	内核优先级数量项[N]不是十进制。
P0116	Compiler option section is missing.	<Compiler>标签不存在。
P0117	Platform option section is missing.	<Platform>标签不存在。
P0118	Platform option section parsing internal error.	解析<Platform>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0119	Chip option section is missing.	<Chip>标签不存在。
P0120	Chip option section parsing internal error.	解析<Chip>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0121	Chip option mismatch in project option list.	<Chip>标签下所提供的某个选项在芯片 XML 中找不到，或者并非芯片 XML 中的所有选项都被指定。请检查是否指定了错误或多余的标签，或者漏掉了某些标签。
P0122	Value conversion failure.	某 Range 类型选项的设置值转换失败了。请检查工程 XML 中该选项是否被设置为无符号整数。
P0123	Value overflow.	某 Range 类型选项的设置值超出了允许范围的上界。请检查并重新设置。

错误编号	错误信息	原因
P0124	Value underflow.	某 Range 类型选项的设置值超出了允许范围的下界。请检查并重新设置。
P0125	Value not found.	某 Select 类型选项的设置值不在允许的选项列表中。请检查并重新设置。

3.3.3 用户态库配置分析（P0200-P0299）

本阶段分析用户态库的基本配置信息，它们都位于<RVM>标签下。可能产生的所有报错如下表所示（P0200 -- P0299）：

表 3-5 用户态库配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0200	Code size section is missing.	用户态库代码内存长度项[H]不存在。
P0201	Code size is not a valid hex integer.	用户态库代码内存长度项[H]不是十六进制。
P0202	Data size section is missing.	用户态库数据内存长度项[H]不存在。
P0203	Data size is not a valid hex integer.	用户态库数据内存长度项[H]不是十六进制。
P0204	Stack size section is missing.	守护进程栈长度项[H]不存在。
P0205	Stack size is not a valid hex integer.	守护进程栈长度项[H]不是十六进制。
P0206	Extra capability table size section is missing.	额外权能表槽位数项[N]不存在。
P0207	Extra capability table size is not a valid unsigned integer.	额外权能表槽位数项[N]不是十进制。
P0208	Compiler section is missing.	<Compiler>标签不存在。
P0209	Virtual machine priorities section is missing.	虚拟化优先级数[N]不存在。
P0210	Virtual machine priorities is not a valid unsigned integer.	虚拟化优先级数[N]不是十进制。
P0211	Virtual machine event number section is missing.	虚拟机事件源数[N]不存在。
P0212	Virtual machine event number is not a valid unsigned integer.	虚拟机事件源数[N]不是十进制。
P0213	Virtual machine mapping total number section is missing.	虚拟机向量映射数[N]不存在。
P0214	Virtual machine mapping total number	虚拟机向量映射数[N]不是十进制。

错误编号	错误信息	原因
	is not a valid unsigned integer.	

3.3.4 进程配置分析（P0300-P0399）

本阶段分析各个进程的基本配置信息，它们都位于<Process>标签下。可能产生的所有报错如下表所示（P0300 -- P0399）：

表 3-6 进程配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0300	Name section is missing.	进程名称项[S]不存在。
P0301	Name section is empty.	进程名称项[S]为空。
P0302	Extra capability table size section is missing.	进程额外权能表槽位数项[N]不存在。
P0303	Extra capability table size is not a valid unsigned integer.	进程额外权能表槽位数项[N]不是十进制。
P0304	Compiler section is missing.	<Compiler>标签不存在。
P0305	Memory section is missing.	<Memory>标签不存在。
P0306	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0307	Memory section is empty.	<Memory>标签为空，这意味着无私有内存段。
P0308	Memory type section is missing.	进程内存段类别项[D]不存在。
P0309	Memory type section is empty.	进程内存段类别项[D]为空。
P0310	Memory type is malformed.	进程内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0311	Shared memory section is missing.	<Shmem>标签不存在。
P0312	Shared memory section parsing internal error.	解析<Shmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0313	Shared memory type section is missing.	进程共享内存段类别项[D]不存在。
P0314	Shared memory type section is empty.	进程共享内存段类别项[D]为空。
P0315	Shared memory type is malformed.	进程共享内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0316	Thread section is missing.	<Thread>标签不存在。
P0317	Thread section parsing internal error.	解析<Thread>标签及其内容时遇到未知错误。

错误编号	错误信息	原因
		请检查该标签的完整性。
P0318	Thread section is empty.	<Thread>标签为空，这意味着无线程存在。
P0319	Invocation section is missing.	<Invocation>标签不存在。
P0320	Invocation section parsing internal error.	解析<Invocation>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0321	Port section is missing.	<Port>标签不存在。
P0322	Port section parsing internal error.	解析<Port>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0323	Receive section is missing.	<Receive>标签不存在。
P0324	Receive section parsing internal error.	解析<Receive>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0325	Send section is missing.	<Send>标签不存在。
P0326	Send section parsing internal error.	解析<Send>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0327	Vector section missing.	<Vector>标签不存在。
P0328	Vector section parsing internal error.	解析<Vector>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0329	Kernel function section missing.	<Kernel>标签不存在。
P0330	Kernel function section parsing internal error.	解析<Kernel>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0331	No primary code section exists.	该进程中必须至少有一个私有的 Code 内存段作为主代码内存段，用来存放代码。
P0332	No primary data section exists.	该进程中必须至少有一个私有的 Data 内存段作为主数据内存段，用来存放数据。
P0333	Primary code section does not have RXS attribute.	进程的主代码段没有具备完整的 RXS 属性（可读可执行，且静态映射）。它必须具备该属性才能作为主代码段。
P0334	Primary data section does not have RWS attribute.	进程的主数据段没有具备完整的 RWS 属性（可读写且静态映射）。它必须具备该属性才能作为主数据段。
P0335	No thread exists.	进程中至少要有一个线程，现在没有。

3.3.5 虚拟机配置分析（P0400-P0499）

本阶段分析各个虚拟机的基本配置信息，它们都位于<Virtual>标签下。可能产生的所有报错如下表所示（P0400 -- P0499）：

表 3-7 虚拟机配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0400	Name section is missing.	虚拟机名称项[S]不存在。
P0401	Name section is empty.	虚拟机名称项[S]为空。
P0402	Guest operating system selection section is missing.	虚拟机操作系统选择项[D]不存在。
P0403	Guest operating system selection section is empty.	虚拟机操作系统选择项[D]为空。
P0404	Extra capability table size section is missing.	虚拟机额外权能表槽位数项[N]不存在。
P0405	Extra capability table size is not a valid unsigned integer.	虚拟机额外权能表槽位数项[N]不是十进制。
P0406	Stack size section is missing.	虚拟机工作线程栈大小项[H]不存在。
P0407	Stack size is not a valid hex integer.	虚拟机工作线程栈大小项[H]不是十六进制。
P0408	Priority section is missing.	虚拟机优先级项[N]不存在。
P0409	Priority is not a valid unsigned integer.	虚拟机优先级项[N]不是十六进制。
P0410	Timeslices section is missing.	虚拟机时间片数[N]不存在。
P0411	Timeslices is not a valid unsigned integer.	虚拟机时间片数[N]不是十进制。
P0412	Timeslices cannot be zero.	虚拟机时间片数[N]不能为 0。
P0413	Timer interrupt period section is missing.	虚拟机时钟中断周期[N]不存在。
P0414	Timer interrupt period is not a valid unsigned integer.	虚拟机时钟中断周期[N]不是十进制。
P0415	Timer interrupt period cannot be zero.	虚拟机时钟中断周期[N]不能为 0。
P0416	Watchdog timeout section is missing.	虚拟机看门狗超时时间[N]不存在。
P0417	Watchdog timeout is not a valid unsigned integer.	虚拟机看门狗超时时间[N]不是十进制。
P0418	Virtual vector number section is missing.	虚拟机虚拟中断数量[N]不存在。

错误编号	错误信息	原因
P0419	Virtual vector number is not a valid unsigned integer.	虚拟机虚拟中断数量[N]不是十进制。
P0420	Virtual vector number too large.	虚拟机虚拟中断数量[N]大于 768，不支持。
P0421	Virtual vector number cannot be zero.	虚拟机虚拟中断数量[N]不能为 0。
P0422	Compiler section is missing.	<Compiler>标签不存在。
P0423	Memory section is missing.	<Memory>标签不存在。
P0424	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0425	Memory section is empty.	<Memory>标签为空，这意味着无私有内存段。
P0426	Memory type section is missing.	虚拟机内存段类别项[D]不存在。
P0427	Memory type section is empty.	虚拟机内存段类别项[D]为空。
P0428	Memory type is malformed.	虚拟机内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0429	Shared memory section is missing.	<Shmem>标签不存在。
P0430	Shared memory section parsing internal error.	解析<Shmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0431	Shared memory type section is missing.	虚拟机共享内存段类别项[D]不存在。
P0432	Shared memory type section is empty.	虚拟机共享内存段类别项[D]为空。
P0433	Shared memory type is malformed.	虚拟机共享内存段类别项[D]必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0434	Send section is missing.	<Send>标签不存在。
P0435	Send section parsing internal error.	解析<Send>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0436	Kernel function section missing.	<Kernel>标签不存在。
P0437	Kernel function section parsing internal error.	解析<Kernel>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0438	No primary code section exists.	该虚拟机中必须至少有一个私有的 Code 内存段作为主代码内存段，用来存放代码。
P0439	No primary data section exists.	该虚拟机中必须至少有一个私有的 Data 内存段作为主数据内存段，用来存放数据。
P0440	Primary code section does not have RXS attribute.	虚拟机的主代码段没有具备完整的 RXS 属性（可读可执行，且静态映射）。它必须具备该

错误编号	错误信息	原因
		属性才能作为主代码段。
P0441	Primary data section does not have RWS attribute.	虚拟机的主数据段没有具备完整的 RWS 属性（可读写且静态映射）。它必须具备该属性才能作为主数据段。

3.3.6 内存段配置分析（P0500-P0599）

内存段配置分析会在分析额外内存（<Extmem>）、系统共享内存(<Project><Shmem>)和各个进程（<Process>）或虚拟机（<Virtual>）的内存（<Memory>）时被调用。它也会在分析芯片自带内存时被调用，因此这些错误也适用于芯片自带内存（<Chip><Memory>，见后文）的分析。可能产生的所有报错如下表所示（P0500 -- P0599）：

表 3-8 内存段配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0500	Base section is missing.	额外内存段基址项[H]、共享内存段基址项[H]、进程内存段基址项(H)、虚拟机内存段基址项(H)或芯片内存段基址项不存在。
P0501	Base is neither automatically allocated nor a valid hex integer.	额外内存段基址项[H]、共享内存段基址项[H]、进程内存段基址项(H)、虚拟机内存段基址项(H)或芯片内存段基址项不是十六进制，也没有注明自动分配。
P0502	Size section is missing.	额外内存段长度项[H]、共享内存段长度项[H]、进程内存段长度项[H]、虚拟机内存段长度项[H]或芯片内存段长度项不存在。
P0503	Size is not a valid hex integer.	额外内存段长度项[H]、共享内存段长度项[H]、进程内存段长度项[H]、虚拟机内存段长度项[H]或芯片内存段长度项不是十六进制。
P0504	Size cannot be zero.	额外内存段长度项[H]、共享内存段长度项[H]、进程内存段长度项[H]、虚拟机内存段长度项[H]或芯片内存段长度项不能为 0。
P0505	Size is out of bound.	额外内存段长度项[H]、共享内存段长度项[H]、进程内存段长度项[H]、虚拟机内存段长度项[H]或芯片内存段长度项的大小超过了 4GB。
P0506	Type section is missing.	额外内存段类别项[D]、共享内存段类别项[D]、进程内存段类别项[D]、虚拟机内存段类别项[D]

错误编号	错误信息	原因
		或芯片内存段类别项不存在。
P0507	Device-type memory cannot be automatically allocated.	设备（Device）类别的内存不能被设置为自动分配，必须手动指明其起始地址。
P0508	Attribute section is missing.	额外内存段属性项[C]、共享内存段属性项[C]、进程内存段属性项[C]、虚拟机内存段属性项[C]或芯片内存段属性项不存在。
P0509	Attribute section is empty.	额外内存段属性项[C]、共享内存段属性项[C]、进程内存段属性项[C]、虚拟机内存段属性项[C]或芯片内存段属性项为空。
P0510	Attribute does not allow any access and is malformed.	额外内存段属性项[C]、共享内存段属性项[C]、进程内存段属性项[C]、虚拟机内存段属性项[C]或芯片内存段属性项不允许任何形式的访问（不可读、不可写且不可执行）。

3.3.7 共享内存段引用配置分析（P0600-P0699）

本节介绍的共享内存段引用指位于各个进程（<Process>）或虚拟机（<Virtual>）内的对之前声明的共享内存（<Shmem>）的引用，而非共享内存段的声明本身。可能产生的所有报错如下表所示（P0600 -- P0699）：

表 3-9 共享内存段配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0600	Name section is missing.	进程共享内存段名称项（进程）[S]或虚拟机共享内存段名称（虚拟机）[S]不存在。
P0601	Name section is empty.	进程共享内存段名称项（进程）[S]或虚拟机共享内存段名称（虚拟机）[S]为空。
P0602	Attribute section is missing.	进程共享内存段属性项（进程）[C]或虚拟机共享内存段属性项（虚拟机）[C]不存在。
P0603	Attribute section is empty.	进程共享内存段属性项（进程）[C]或虚拟机共享内存段属性项（虚拟机）[C]为空。
P0604	Attribute does not allow any access and is malformed.	进程共享内存段属性项（进程）[C]或虚拟机共享内存段属性项（虚拟机）[C]不允许任何形式的访问（不可读、不可写且不可执行），这是不允许的。

3.3.8 线程配置分析（P0700-P0799）

线程配置分析产生会在分析各个进程（<Process>）的线程（<Thread>）时使用。可能产生的所有报错如下表所示（P0700 -- P0799）：

表 3-10 线程配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0700	Name section is missing.	线程名称项[S]不存在。
P0701	Name section is empty.	线程名称项[S]为空。
P0702	Stack size section is missing.	线程栈大小项[H]不存在。
P0703	Stack size is not a valid hex integer.	线程栈大小项[H]不是十六进制。
P0704	Stack size cannot be zero.	线程栈大小项[H]不能为 0。
P0705	Parameter section is missing.	线程参数项[H]不存在。
P0706	Parameter is not a valid hex integer.	线程参数项[H]不是十六进制。
P0707	Priority section is missing.	线程优先级项[N]不存在。
P0708	Priority is not a valid unsigned integer.	线程优先级项[N]不是十进制。

3.3.9 迁移调用目标配置分析（P0800-P0899）

迁移调用目标配置分析产生会在分析各个进程（<Process>）的迁移调用目标（<Invocation>）时使用。可能产生的所有报错如下表所示（P0800 -- P0899）：

表 3-11 迁移调用目标配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0800	Name section is missing.	迁移调用名称项[S]不存在。
P0801	Name section is empty.	迁移调用名称项[S]为空。
P0802	Stack size section is missing.	迁移调用栈大小项[H]不存在。
P0803	Stack size is not a valid hex integer.	迁移调用栈大小项[H]不是十六进制。
P0804	Stack size cannot be zero.	迁移调用栈大小项[H]不能为 0。

3.3.10 迁移调用入口配置分析（P0900-P0999）

迁移调用入口配置分析产生会在分析各个进程（<Process>）的迁移调用入口（<Port>）时使用。可能产生的所有报错如下表所示（P0900 -- P0999）：

表 3-12 迁移调用入口配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0900	Name section is missing.	迁移调用目标入口项[D]不存在。
P0901	Name section is empty.	迁移调用目标入口项[D]为空。
P0902	Process name section is missing.	迁移调用目标入口所在进程项[D]不存在。
P0903	Process name section is empty.	迁移调用目标入口所在进程项[D]为空。

3.3.11 接收信号端点配置分析（P1000-P1099）

接收信号端点配置分析产生会在分析各个进程（<Process>）的接收信号端点（<Receive>）时使用。可能产生的所有报错如下表所示（P1000 -- P1099）：

表 3-13 接收信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1000	Name section is missing.	信号接收端点名称项[S]不存在。
P1001	Name section is empty.	信号接收端点名称项[S]为空。

3.3.12 发送信号端点配置分析（P1100-P1199）

发送信号端点配置分析产生会在分析各个进程（<Process>）的发送信号端点（<Receive>）时使用。可能产生的所有报错如下表所示（P1100 -- P1199）：

表 3-14 发送信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1100	Name section is missing.	对应的接收信号端点项（进程）[D]或对应的接收信号端点项（虚拟机）[D]不存在。
P1101	Name section is empty.	对应的接收信号端点项（进程）[D]或对应的接收信号端点项（虚拟机）[D]为空。
P1102	Process name section is missing.	对应的接收信号进程项（进程）[D]或对应的接收信号进程项（虚拟机）[D]不存在。
P1103	Process name section is empty.	对应的接收信号进程项（进程）[D]或对应的接收信号进程项（虚拟机）[D]为空。

3.3.13 物理向量信号端点配置分析（P1200-P1299）

物理向量信号端点配置分析产生会在分析各个进程（<Process>）的物理向量信号端点（<Vector>）时使用。它也在分析芯片的物理信号端点时被使用（<Chip><Vector>，见后文）可能产生的所有报错如下表所示（P1200 -- P1299）：

表 3-15 物理向量信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1200	Name section is missing.	物理向量名称项[S]或芯片物理向量名称项不存在。
P1201	Name section is empty.	物理向量名称项[S]或芯片物理向量名称项为空。
P1202	Number section is missing.	物理向量号项[N]或芯片物理向量号项不存在。
P1203	Number is not a valid unsigned integer.	物理向量号项[N]或芯片物理向量号项不是十进制。

3.3.14 内核功能配置分析（P1300-P1399）

内核功能配置分析会在分析各个进程（<Process>）或虚拟机（<Virtual>）的内核功能（<Kernel>）时使用。可能产生的所有报错如下表所示（P1300 -- P1399）：

表 3-16 内核功能配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1300	Name section is missing.	内核功能名称项（进程）[S]或内核功能名称项（虚拟机）[S]不存在。
P1301	Name section is empty.	内核功能名称项（进程）[S]或内核功能名称项（虚拟机）[S]为空。
P1302	Starting kernel function number section is missing.	内核功能主功能号起始值项（进程）[H]或内核功能主功能号起始值项（虚拟机）[H]不存在。
P1303	Starting kernel function number is not a valid hex integer.	内核功能主功能号起始值项（进程）[H]或内核功能主功能号起始值项（虚拟机）[H]不是十六进制。
P1304	Ending kernel function number section is missing.	内核功能主功能号结束值项（进程）[H]或内核功能主功能号结束值项（虚拟机）[H]不存在。
P1305	Ending kernel function number is not a valid hex integer.	内核功能主功能号结束值项（进程）[H]或内核功能主功能号结束值项（虚拟机）[H]不是十六进制。
P1306	Ending kernel function number is smaller than the starting kernel function number.	结束值项比起始值项大，这是不允许的。结束值项至少应当等于起始值项。

3.4 芯片 XML 文件分析（D0000-D9999）

该模块分析芯片描述 XML 文件。这些文件是编译器自带的。下面将分章节讲解每个分步骤的具体内容。

3.4.1 芯片信息初步分析（D0000-D0099）

在本步骤中，系统中的额外内存信息、共享内存信息和一些工程基本信息会经分析后直接填充在内部数据结构中。可能产生的所有报错信息如下（D0000 -- D0099）：

表 3-17 芯片信息初步分析阶段可能产生的报错信息

错误编号	错误信息	原因
D0000	Chip XML parsing failed.	芯片 XML 文件存在语法错误，无法解析。
D0001	Chip XML is malformed.	芯片 XML 中不存在<Chip>标签。
D0002	Class section is missing.	芯片兼容类别项不存在。
D0003	Class section is empty.	芯片兼容类别项为空。
D0004	Compatible variant section is missing.	芯片兼容名称项不存在。
D0005	Compatible variant section is empty.	芯片兼容名称项为空。
D0006	Vendor section is missing.	芯片厂商项不存在。
D0007	Vendor section is empty.	芯片厂商项为空。
D0008	Platform section is missing.	芯片架构项不存在。
D0009	Platform section is empty.	芯片架构项为空。
D0010	Cores section is missing.	芯片 CPU 数项不存在。
D0011	Cores is not an unsigned integer.	芯片 CPU 数项为空。
D0012	Regions section is missing.	芯片 MPU 区域数项不存在。
D0013	Regions is not an unsigned integer.	芯片 MPU 区域数项为空。
D0014	Attribute section missing.	架构相关信息组不存在。
D0015	Attribute section parsing internal error.	解析架构相关信息组及其内容时发生未知错误。请检查<Attribute>标签及其内容。
D0016	Memory section is missing.	芯片 XML 中不存在<Memory>标签。
D0017	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
D0018	Memory section is empty.	<Memory>标签内部为空。
D0019	Memory type section is missing.	芯片内存段类别项不存在。

错误编号	错误信息	原因
D0020	Memory type section is empty.	芯片内存段类别项为空。
D0021	Memory type is malformed.	芯片内存段类别项必须是 Code，Data 或 Device 三者之一。当前值无法识别。
D0022	No code section exists.	芯片中没有可用于代码段的内存。
D0023	No data section exists.	芯片中没有可用于数据段的内存。
D0024	No device section exists.	芯片中没有可用于设备段的内存。
D0025	Option section is missing.	芯片 XML 中不存在<Option>标签。
D0026	Option section parsing internal error.	解析<Option>标签及其内容时遇到未知错误。请检查该标签的完整性。
D0027	Vector section missing.	芯片 XML 中不存在<Vector>标签。
D0028	Vector section parsing internal error.	解析<Vector>标签及其内容时遇到未知错误。请检查该标签的完整性。

3.4.2 芯片内存分析（P0500-P0599）

本步骤分析芯片本身的内存存储器。本部分所可能产生的报错（P0500 -- P0599）已经在前面完成介绍，因此在这里我们将其略去。

3.4.3 芯片选项分析（D0100-D0199）

本步骤分析芯片本身的选项。可能产生的所有报错信息如下（D0100 -- D0199）：

表 3-18 芯片选项分析阶段可能产生的报错信息

错误编号	错误信息	原因
D0100	Name section is missing.	选项名称项不存在。
D0101	Name section is empty.	选项名称项为空。
D0102	Type section is missing.	选项类别项不存在。
D0103	Type section is empty.	选项类别项为空。
D0104	Type is malformed.	选项类别项必须是 Select 或 Range 两者之一。当前值无法识别。
D0105	Macro section is missing.	选项宏定义名称项不存在。
D0106	Macro section is empty.	选项宏定义名称项为空。
D0107	Range section is missing.	选项范围项不存在。
D0108	Range section is empty.	选项范围项为空。

错误编号	错误信息	原因
D0109	Range section have an empty value.	选项范围项之中出现了一个空值。空值是指两个逗号之间没有内容，或者第一个逗号之前没有内容，或者最后一个逗号之后没有内容。
D0110	Range typed option cannot have more or less than two ends specified.	Range 类型的选项的选项范围项之中必须正好有两个值标注其起止点。多或者少都是不允许的。
D0111	Starting point or ending point conversion failure.	某 Range 类型选项的选项范围项中有不是整数的数值。请检查其起止点确实为无符号整数。
D0112	Range typed option starting point must be smaller than the corresponding ending point.	Range 类型的选项的选项范围项的起点值必须严格小于止点值，否则选项将无可选择。
D0113	Select typed option's possible values must all be valid C identifiers or numbers.	Select 类型的选项的选项范围项的所有可能值必须全部是合法的 C 标识符，或者是一个数字。

3.4.4 芯片向量分析（P1200-P1299）

本步骤分析芯片本身的中断向量。本部分所可能产生的报错（P1200 -- P1299）已经在前面完成介绍，因此在这里我们将其略去。

3.5 一个典型的芯片 XML 结构

本编译器另外需要的、描述每个芯片的 XML 的结构如下表所示。这些 XML 文件是编译器自带的，虽然用户也可以按照此格式自行编写并添加。我们将在接下来的章节中介绍各个标签的意义。

表 3-19 一个典型的芯片 XML 结构

内容	参见
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>	
<Chip>	
<Class>STM32F767IG</Class>	芯片兼容类别项
<Compatible>STM32F767IGK6,STM32F767IGT6</Compatible>	芯片兼容名称项
<Vendor>STMicroelectronics</Vendor>	芯片厂商项
<Platform>A7M</Platform>	芯片架构项
<Cores>1</Cores>	芯片 CPU 数项

内容	参见
<Regions>8</Regions>	芯片 MPU 区域数项
<Attribute>	架构相关信息组
<CPU_Type>Cortex-M7</CPU_Type>	
<FPU_Type>Double</FPU_Type>	
<Endianness>Little</Endianness>	
</Attribute>	
<Memory>	
<M1>	
<Base>0x08000000</Base>	芯片内存段基址项
<Size>0x00100000</Size>	芯片内存段长度项
<Type>Code</Type>	芯片内存段类别项
<Attribute>RXBCS</Attribute>	芯片内存段属性项
</M1>	
<M2>	
<Base>0x20000000</Base>	芯片内存段基址项
<Size>0x00080000</Size>	芯片内存段长度项
<Type>Data</Type>	芯片内存段类别项
<Attribute>RWXBCS</Attribute>	芯片内存段属性项
</M2>	
<M3>	
<Base>0x40000000</Base>	芯片内存段基址项
<Size>0x20000000</Size>	芯片内存段长度项
<Type>Device</Type>	芯片内存段类别项
<Attribute>RWS</Attribute>	芯片内存段属性项
</M3>	
<M4>	
<Base>0xE0000000</Base>	芯片内存段基址项
<Size>0x20000000</Size>	芯片内存段长度项
<Type>Device</Type>	芯片内存段类别项
<Attribute>RWS</Attribute>	芯片内存段属性项
</M4>	
</Memory>	

内容	参见
<Option>	
<O1>	
<Name>XTAL</Name>	选项名称项
<Type>Range</Type>	选项类别项
<Macro>RME_A7M_STM32F767IG_XTAL</Macro>	选项宏定义名称项
<Range>8,25</Range>	选项范围项
</O1>	
<O2>	
<Name>PLLM</Name>	选项名称项
<Type>Range</Type>	选项类别项
<Macro>RME_A7M_STM32F767IG_PLLM</Macro>	选项宏定义名称项
<Range>2,63</Range>	选项范围项
</O2>	
<O3>	
<Name>PLLN</Name>	选项名称项
<Type>Range</Type>	选项类别项
<Macro>RME_A7M_STM32F767IG_PLLN</Macro>	选项宏定义名称项
<Range>50,432</Range>	选项范围项
</O3>	
<O4>	
<Name>PLLP</Name>	选项名称项
<Type>Select</Type>	选项类别项
<Macro>RME_A7M_STM32F767IG_PLLP</Macro>	选项宏定义名称项
<Range>2,4,6,8</Range>	选项范围项
</O4>	
<O5>	
<Name>PLLQ</Name>	选项名称项
<Type>Range</Type>	选项类别项
<Macro>RME_A7M_STM32F767IG_PLLQ</Macro>	选项宏定义名称项
<Range>2,15</Range>	选项范围项
</O5>	
</Option>	

内容	参见
<Vector>	
<V1>	
<Name>TIM1</Name>	芯片物理向量名称项
<Number>8</Number>	芯片物理向量号项
</V1>	
<V2>	
<Name>TIM2</Name>	芯片物理向量名称项
<Number>9</Number>	芯片物理向量号项
</V2>	
</Vector>	
</Chip>	

3.5.1 芯片基本信息类<Chip>

本类负责记载芯片的基本信息。这些基本信息将被生成器用来判断芯片的具体型号和兼容性。

3.5.1.1 芯片兼容类别项<Class>

该项是一个字段，决定了芯片的类别。凡是属于这个类别的芯片都使用这个芯片 XML 进行描述。通常而言，会放在一个类别内的各个芯片仅仅具有管脚数量、外设数量的细微差异，其存储器布局都应当完全一致。

3.5.1.2 芯片兼容名称项<Compatible>

该项是一个或多个用逗号隔开的字段，决定了本 XML 兼容哪些具体的芯片。只有列出在该字段内的芯片才可以被兼容。

3.5.1.3 芯片厂商项<Vendor>

该项是一个字段，决定了芯片的制造厂商。它在生成某些工具链的工程时会用到。

3.5.1.4 芯片架构项<Platform>

该项是一个字段，决定了芯片的架构。

3.5.1.5 芯片 CPU 数项<Cores>

该项是一个十进制值，决定了芯片的 CPU 数量。目前这个数字总是 1。

3.5.1.6 芯片 MPU 区域数项<Regions>

该项是一个十进制数，决定了芯片的 MPU 区域数量。

3.5.1.7 架构相关信息组<Attribute>

本组决定了架构相关的一些信息。这些信息随着每个架构都有可能不同，通常是该架构通用的一些配置项。

3.5.2 芯片固有存储器类<Chip><Memory><Mn>

本类负责记载芯片的片上存储器信息。所有能够用到的片上存储器必须被登录在此。

3.5.2.1 芯片内存段基址项<Base>

该项是一个十六进制值，决定了该内存段的起始地址。

3.5.2.2 芯片内存段长度项<Size>

该项是一个十六进制值，决定了该内存段的长度。

3.5.2.3 芯片内存段类别项<Type>

该项是一个字段，决定了内存段的类别。可能的类别有三种：代码（Code）、数据（Data）和设备（Device）。

3.5.2.4 芯片内存段属性项<Attribute>

该配置项是一个字段，决定了内存段的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。

3.5.3 芯片固有选项类<Chip><Option><On>

本类负责记载芯片本身具备的固有选项。编译器会加载这些固有选项供用户配置。

3.5.3.1 选项名称项<Name>

该项是一个字段，决定了该选项的内部名称。这一项不会被反映在输出的工程的任何地方而是被生成器内部使用。

3.5.3.2 选项类别项<Type>

该项是一个字段，决定了该选项的性质。类别只有两种，一种是 Select（选择），另外一种是 Range（范围）。前者代表可以在<Range>标签列出的各个值之中选择一个作为其值，后者代表<Range>标签列出的两个值之间的所有值都可以。

3.5.3.3 选项宏定义名称项<Macro>

该项是一个字段，决定了该选项对应的宏在配置头文件中应有的名字。

3.5.3.4 选项范围项<Range>

该项是一个字段，当选项是 Select 类型时列出所有选项可能取的值，当选项是 Range 类型时列出选项可取值的范围。

3.5.4 芯片固有中断向量类<Chip><Vector><Vn>

本类记载芯片本身具备的固有终端向量。

3.5.4.1 芯片物理向量名称项<Name>

该项是一个字段，决定了该物理向量的名称。

3.5.4.2 芯片物理向量号项<Number>

该项是一个字段，决定了该物理向量的向量号。

3.6 本章参考文献

无

第 4 章　中间表示生成模块

4.1　中间表示生成模块概述

中间表示生成模块负责生成接下来可以被各个工程生成模块调用的工程的中间表示。该步骤分为内存分配、权能表分配和内核对象分配三个阶段。

4.2　内存分配（M0000-M0999）

该步骤确定各个自动分配的内存段的首地址，以及决定页表的具体构造方式。它又可以分成内存对齐、内存段分配、页表生成三个子部骤。

4.2.1　内存对齐（A0000-A9999）

该步骤主要检查并且对齐各个内存块。给 RME 和 RVM 保留的内存必须被对齐到某个由目标架构决定的粒度。各个线程的栈也要对齐到这个粒度。各个内存块的起始地址和大小也会被检查，如果有不符合该对齐粒度的情况出现，那么将会报错。这些报错随着各个架构都有所不同，具体所产生的错误需要参见该架构的相应文档。

4.2.2　内存段分配（M0000-M0099）

该步骤将那些没有指定起始地址的内存段分配固定的内存地址。在这一步骤中，我们先将外部内存段添加进芯片的内存布局中并检查是否有交错；如果没有，那么接下来决定各个未分配的代码和数据内存段的起始地址。

分配器使用一个高度智能化的算法决定各个内存段的位置。分配结束后，我们将检查每个进程的内存布局，确定没有交错，以及确定各个进程的主代码段是互相独立的[1]。最后，我们还要检查所有的用户指定了初始地址和长度的那些内存段都落在芯片内存布局中。

可能产生的所有报错信息如下（M0000 -- M0099）：

表 4-1　内存段分配阶段可能产生的报错信息

错误编号	错误信息	原因
M0000	Chip internal and/or extra memory section overlapped.	芯片的内存布局有重叠。比如，添加的额外内存和芯片本身提供的内存发生了交叠，这在物理上是不可实现的。
M0001	RME code section is invalid, either wrong range or wrong attribute.	RME 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可执行或不可静态映射。

[1] 主数据段则无该要求

错误编号	错误信息	原因
M0002	RVM code section is invalid, either wrong range or wrong attribute.	RVM 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可执行或不可静态映射。
M0003	Code section is invalid, either wrong range or wrong attribute.	某进程的代码段或共享代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者无法满足该代码段声明时的那些访问权限。
M0004	Code memory fitter failed.	代码段分配失败。这可能是由于使用了过多的代码存储，或者声明的代码段数过多导致代码内存碎片化。
M0005	RME data section is invalid, either wrong range or wrong attribute.	RME 的数据段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可写或不可静态映射。
M0006	RVM data section is invalid, either wrong range or wrong attribute.	RVM 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放数据，或者不可读、不可写或不可静态映射。
M0007	Data section is invalid, either wrong range or wrong attribute.	某进程的数据段或共享数据段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者无法满足该数据段声明时的那些访问权限。
M0008	Data memory fitter failed.	数据段分配失败。这可能是由于使用了过多的数据存储，或者声明的数据段数过多导致数据内存碎片化。
M0009	Name is duplicate or invalid.	同一种类型的共享内存重名或者同一进程内部对同一共享内存的引用重复。
M0010	Entry not found in global database.	进程或内对共享内存的引用在共享内存的声明中找不到。
M0011	Attributes not satisfied by global entry.	进程内对共享内存的引用
M0012	Private and/or shared memory section overlapped.	在同一进程内部发生了地址空间交叠。这可能是由于两段被指定了起始地址和长度的私有内存发生交叠，也有可能是由于共享内存和私有内存发生交叠。
M0013	Process primary code section overlapped.	不同进程间的私有主代码空间发生了地址交叠。这是不允许的，因为主代码空间

错误编号	错误信息	原因
		对于各个进程而言是独占的。
M0014	Device memory have wrong attributes.	芯片内存布局中的设备段无法满足该设备内存声明的访问权限。
M0015	Device memory segment is out of bound.	声明的设备内存无法在芯片的内存布局中找到。这可能是由于芯片的这个位置不允许存放设备内存。

4.2.3 页表生成（A0000-A9999）

该步骤将生成该工程中各个进程和虚拟机的页表。该部分完全是由具体的架构所决定的，因此产生的具体报错信息需要查看该架构对应的手册。

4.3 权能表分配（M1000-M1999）

该步骤自动分配各个内核对象的权能表权能号。权能号包括两个，一个是在 RVM 中创建该内核对象时使用的全局权能号，另一个是在各个进程中引用这些权能时使用的本地权能号。它又可以分为合规检查、本地权能号分配、全局权能号分配、权能宏名分配和权能配对五个步骤。

4.3.1 合规检查（M1000-M1099）

该步骤主要进行内核对象的不能在解析 XML 时确定的较为复杂的参数检查。可能产生的所有报错信息如下（M1000 -- M1099）：

表 4-2 合规检查阶段可能产生的报错信息

错误编号	错误信息	原因
M1000	Total number of kernel priorities must be a multiple of word size.	内核态优先级的总数必须是机器字长的整数倍。不是整数倍的值不允许。
M1001	Kernel memory allocation granularity order must be no less than 2^3=8 bytes.	内核内存分配粒度级数必须大于等于 3。
M1002	Kernel memory allocation granularity order must be no more than 2^5=32 bytes.	内核内存分配粒度级数必须小于等于 5。
M1003	There are neither virtual machines nor processes in the system.	系统中既没有虚拟机也没有原生进程。这是不允许的。
M1004	Virtual machine exists but the total number of virtual machine priorities is set to 0.	系统中存在虚拟机，但是虚拟机优先级数量却被设置为 0。这是不允许的。
M1005	Total number of virtual machine priorities must be a multiple of word size.	虚拟机优先级的总数必须是机器字长的整数倍。不是整数倍的值不允许。

错误编号	错误信息	原因
M1006	Virtual machine exists but the total number of virtual event sources is set to 0.	系统中存在虚拟机，但是虚拟机事件源的数量却被设置为 0。这是不允许的。
M1007	Total number of virtual event sources must be a multiple of word size.	虚拟机事件源的总数必须是机器字长的整数倍。不是整数倍的值不允许。
M1008	Total number of virtual event sources cannot exceed 1024.	虚拟机事件源的总数量不能超过 1024。
M1009	Total number of virtual event to interrupt mappings cannot be smaller than the virtual event source number.	总计的事件源或物理中断到虚拟机虚拟向量的映射允许上限必须大于虚拟机事件源的总数量。
M1010	Priority must be bigger than service daemons' priority (4).	用户进程中的线程的优先级必须大于各个服务守护进程的优先级，也即必须大于 4。
M1011	Priority must be smaller than safety daemon priority (Kern_Prios-1).	用户进程中的线程的优先级必须小于安全守护进程的优先级，也即必须小于内核优先级总数-1。
M1012	Priority must be smaller than total number of virtual machine priorities.	虚拟机的优先级必须小于虚拟机优先级的总数。
M1013	Extra capacity cannot be larger than the architectural limits.	额外权能表的表项数量不能超过架构允许的数量。在 32 位机器上这个限制为 128；在 64 位机器上这个限制为 32768。
M1014	Kernel function number range exceeded architectural limits.	内核功能的功能号超过了架构允许的范围。在 32 位机器上它的最大值为 2^{16}，在 64 位机器上它的最大值为 2^{32}。
M1015	Name is duplicate or invalid.	系统中存在重复的内核对象名称。有可能是进程的名称重复，也有可能是进程内同一类内核对象的名称重复。
M1016	Name/process name is duplicate or invalid.	同一进程内部的迁移调用入口或者信号发送端点声明的目标的名称和所在进程都相同。这是不允许的。
M1017	Name/number is duplicate or invalid.	中断向量的名称或者中断号重复。这是不允许的。

4.3.2 本地权能号分配（M1100-M1199）

该步骤主要分配各个进程的本地权能表中的权能号。可能产生的所有报错信息如下（M1100 -- M1199）：

表 4-3 本地权能号分配阶段可能产生的报错信息

错误编号	错误信息	原因
M1100	Capability table size exceeded architectural limits.	本地权能表的大小超过了架构允许的现值。这是不允许的。

4.3.3 全局权能号分配（M1200-M1299）

该步骤主要分配 RVM 在创建各个内核资源时的全局权能号。本阶段不会产生任何报错信息。

4.3.4 权能宏名分配（M1300-M1399）

该步骤主要分配供引用权能使用的权能宏名。本阶段不会产生任何报错信息。

4.3.5 权能配对（M1400-M1499）

该步骤会将信号发送端点和迁移调用入口分别与信号接收端点和迁移调用目标配对。如果编译器发现无法配对的信号发送端点和迁移调用入口，则会报错。该步骤也同时检查所有声明的物理向量名称是否能够在芯片 XML 中被找到，如果不能则报错。可能产生的所有报错信息如下（M1200 -- M1299）：

表 4-4 权能配对阶段可能产生的报错信息

错误编号	错误信息	原因
M1400	Invalid process name.	信号发送端点或迁移调用入口引用的进程不存在。
M1401	Invalid invocation name.	迁移调用入口引用的迁移调用目标不存在。
M1402	Invalid receive endpoint name.	信号发送端点引用的信号接收端点不存在。
M1403	Invalid vector number.	物理向量端点的向量号和芯片 XML 中记载的不一致。
M1404	Invalid vector name.	物理向量端点引用的向量名称在芯片 XML 中无记载。

4.4 内核对象分配（M2000-M2999）

该步骤分配各个内核对象的内核内存地址，以及各个进程内部的线程、迁移调用目标的内存布局。它又可以分为内核对象内存试分配、内核对象内存分配、内核其他内存分配、用户态库内存分配和进程内存分配五个步骤。

4.4.1 内核对象内存试分配和分配（M2000-M2099）

由于在开始分配内核对象时用户态库的主权能表的大小是未知的，因此需要经过一个试分配阶段。试分配阶段将分配所有的内核对象的内存地址，但不分配主权能表所占的大小。在决定了内核对象的数

量后，主权能表的大小随即决定，然后在进行一轮分配得到各个内核对象的真实地址。可能产生的所有报错信息如下（M2000 -- M2099）：

表 4-5 内核对象试分配和分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2000	RVM capability table size exceeded the architectural limit.	RVM 的权能表大小超过了架构允许的长度。请考虑减少 RVM 的额外权能表槽位数，或降低系统的复杂性。

4.4.2 内核其他内存分配（M2100-M2199）

本步骤分配内核中存在的其他内存，诸如内核栈等等。某些内存段是内核运行所必需的，另外一些内存段则是内核与用户态库交流所必需的。可能产生的所有报错信息如下（M2100 -- M2199）：

表 4-6 内核其他内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2100	RME data section is not big enough, unable to allocate vector flags.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2101	RME data section is not big enough, unable to allocate event flags.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2102	RME data section is not big enough, unable to allocate kernel stacks.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2103	RME data section is not big enough, unable to allocate kernel object memory.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。

4.4.3 用户态库内存分配（M2200-M2299）

本步骤分配用户态库内存，诸如各个守护进程所使用的栈等等。可能产生的所有报错信息如下（M2200 -- M2299）：

表 4-7 用户态库内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2200	RVM data section is not big	RVM 的数据段不够大，无法分配启动守护进程栈。请考

错误编号	错误信息	原因
	enough, unable to allocate init thread stack.	虑将 RVM 的数据段修改得大一些。
M2201	RVM data section is not big enough, unable to allocate safety daemon thread stack.	RVM 的数据段不够大，无法分配安全守护进程栈。请考虑将 RVM 的数据段修改得大一些。
M2202	RVM data section is not big enough, unable to allocate virtual machine monitor daemon thread stack.	RVM 的数据段不够大，无法分配超调用守护进程栈。请考虑将 RVM 的数据段修改得大一些。
M2203	RVM data section is not big enough, unable to allocate vector handling daemon stack.	RVM 的数据段不够大，无法分配向量守护进程栈。请考虑将 RVM 的数据段修改得大一些。
M2204	RVM data section is not big enough, unable to allocate timer handling thread stack.	RVM 的数据段不够大，无法分配时间守护进程栈。请考虑将 RVM 的数据段修改得大一些。

4.4.4 进程内存分配（M2300-M2399）

本步骤分配进程本身的内存，诸如各个线程和迁移调用目标的栈等等。如果该进程实际上是一个虚拟机，那么还要分配与虚拟机监视器相关的内存。可能产生的所有报错信息如下（M2300 -- M2399）：

表 4-8 用户态库内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2300	Data section size is not big enough, unable to allocate stack.	进程的数据段不够大，无法分配线程或迁移调用目标的栈。请考虑将进程的数据段修改得大一些。
M2301	Data section size is not big enough, unable to allocate virtual machine interrupt flags.	虚拟机的数据段不够大，无法分配虚拟向量标志内存区。请考虑将虚拟机的数据段修改得大一些。
M2302	Data section size is not big enough, unable to allocate virtual machine parameters.	虚拟机的数据段不够大，无法分配超调用参数区。请考虑将虚拟机的数据段修改得大一些。
M2303	Data section size is not big enough, unable to allocate virtual machine registers.	虚拟机的数据段不够大，无法分配虚拟化寄存器区。请考虑将虚拟机的数据段修改得大一些。

4.5 本章参考文献

无

第 5 章 最终工程生成模块

5.1 最终工程生成模块概述

最终工程生成模块负责将中间表示生成最终的工程。最终的工程会对应到非常具体的某个工具链的某种组态。该步骤分为内核工程生成、用户态库工程生成和用户进程工程生成三个阶段。

5.2 内核工程生成（G0000-G0999）

该步骤生成 RME 内核的工程。它又可以分为文件夹结构生成、配置头文件生成、启动序列源文件生成、用户可编辑源文件生成、架构相关文件生成五个步骤。

在文件夹结构生成步骤，内核所需的各个文件夹将被建立，还将填充内核源码到文件夹内。

在配置头文件生成步骤，生成选择架构所使用的配置头文件。

在启动序列源文件生成步骤，用于在启动时创建内核对象的头文件和源文件将被建立。

在用户可编辑源文件生成步骤，用户可编辑源文件将被建立。用户可编辑源文件主要负责内核态中断向量的填充。

在架构相关文件生成阶段，相关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤及其五个子部骤都没有独立的报错信息。

5.3 用户态库工程生成（G1000-G1999）

该步骤负责生成 RVM 用户态库的工程。它又可以分为文件夹结构生成、配置头文件生成、启动序列源文件生成、架构相关文件生成四个步骤。

在文件夹结构生成步骤，用户态库工程所需的各个文件夹将被建立，还将填充用户态库源码到文件夹内。

在配置头文件生成步骤，生成选择架构所使用的配置头文件。

在启动序列源文件生成步骤，用于在启动时创建内核对象的头文件和源文件将被建立。

在用户可编辑源文件生成步骤，用户可编辑源文件将被建立。用户可编辑源文件主要负责进一步初始化以及某些可以定制的处理逻辑的编写。

在架构相关文件文件生成阶段，相关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤可能产生的报错信息如下（G1000 -- G1999）：

表 5-1 用户态库工程生成时可能产生的报错信息

错误编号	错误信息	原因
G1000	Internal capability table computation failure.	内部权能表大小计算错误，无法生成内核对象创建代码。出现此错误说明编译器遇到内部故障，请向我们报告此

错误编号	错误信息	原因
		故障。

5.4 用户进程工程生成（G2000-G2999）

该步骤负责生成各个用户进程的工程。它又可以分为文件夹结构生成、进程源文件生成、架构相关文件生成三个步骤。

在文件夹结构生成步骤，进程工程所需的各个文件夹将被建立，还将填充进程调用系统服务所需的库源码到文件夹内。

在进程源文件生成步骤，进程中声明的可在本地引用的各个内核对象的宏定义将会被给出，方面用户引用；另外，用户声明的线程和迁移调用目标的空函数也将被创建，等待用户填充。

在架构相关文件生成步骤，关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤可能产生的报错信息如下（G2000 -- G2999）：

表 5-2 用户进程工程生成时可能产生的报错信息

错误编号	错误信息	原因
G2000	Virtual machine should not have threads other than Vect and User.	在虚拟机中只允许存在两个线程，一个是负责运行用户程序的 User 线程，另一个是负责接收中断的 Vect 线程。出现此错误说明编译器遇到内部故障，请向我们报告此故障。

5.5 工作空间生成（G3000-G3999）

该步骤负责生成工作空间。生成的各个子项目都将被置于工作空间之下，便于统一管理和编译。它又可以分为文件夹结构生成和架构相关文件生成两个步骤。

在文件夹结构生成步骤，工作空间的文件夹将被建立。

在架构相关文件生成步骤，我们则直接生成和架构相关的工作空间文件。

总的而言，本步骤及其两个子部骤都没有独立的报错信息。

5.6 文件系统操作组件（F0000-F9999）

在上面提到的工程生成步骤中，大量使用了文件系统读写操作。这些文件读写操作可能产生的报错信息如下（F1000 -- F1999）：

表 5-3 文件系统操作时可能产生的报错信息

错误编号	错误信息	原因

错误编号	错误信息	原因
F0000	Folder creation failed.	在输出文件夹中创建文件夹遇到问题。检查输出文件夹是否允许读写。
F0001	Cannot open source file.	不能打开作为输入的源文件。检查输出文件夹是否允许读写。
F0002	Cannot read file.	不能打开文件进行读取。检查输出文件夹是否允许读写。
F0003	Windows/Linux stat failed.	用来判别文件大小的 stat 系统调用失败。检查文件系统相关配置是否正确。
F0004	Cannot open destination file.	无法打开作为输出的目标文件。检查输出文件夹是否允许读写。

5.7 架构相关文件生成组件（A0000-A9999）

本章提到的所有生成步骤中独有架构相关文件的生成。这些文件的生成是架构相关的，因此其报错需要查看架构相关手册来判明。

5.8 本章参考文献

无

第 6 章　工程结构说明

6.1　工程架构概述

本编译器生成的工程由四个部分组成：内核工程、用户态库工程、原生进程工程、虚拟机工程。

由于单片机使用的工具链仅支持裸机工程，因此我们先将用户态库工程、原生进程工程、虚拟机工程编译为二进制映像，然后将它们转换为 C 数组，添加进内核工程之中进行编译组装生成最终的文件。因此，在编译顺序上，用户态库、原生进程和虚拟机必须先编译，然后再编译内核生成最终的二进制映像。

在执行时，原生进程工程中的线程永远具备较高的优先级，因此先于任何虚拟机执行。当原生进程全部执行完毕阻塞时，虚拟机进程开始执行。在虚拟机进程中，高优先级虚拟机永远有绝对优先的执行权，只有当高优先级虚拟机完全完成所有工作后才能轮到低优先级虚拟机执行。当那些哪怕是低优先级的虚拟机也阻塞的时候，系统则进入低功耗状态。

6.2　工作空间

为了方便管理生成的各个工程，这些工程均会被置于同一个工作空间下，方便统一管理。通常而言，只要打开工作空间本身，点击 IDE 的编译按钮就能完成对整个工程的构建任务。

通常而言，编译器并不输出针对工作空间的源代码，源代码全部被放置在工作空间下的各个工程中。下面我们将对每个工程的使用方法做说明。

在下面的说明中，<Name>代表某内核对象的名称，<name>代表其小写化后的名称，<NAME>代表其大写化后的名称。

6.3　内核工程

内核工程是所有工程的主工程。其他工程生成的二进制文件都要转换为 C 数组文件后加入内核工程进行编译。内核工程中含有 RME 的内核本身和一些用户可编辑的文件；这些用户可编辑的文件提供给用户一些更改内核逻辑的钩子。同时，编译器会建立一些宏定义帮助用户引用生成的内核对象。

下面我们将对该工程的组成部分进行进一步讲解。

6.3.1　固定文件：内核固有文件

这些文件是内核本身的固有文件。不属于下面所提到的文件的文件都可以划分在这一组，它们提供通用的内核功能。这些文件是编译器直接从内核源码中拷贝来的。对于这些文件，我们建议用户不要进行任何改动。

6.3.2　固定文件：rme_boot.h

本文件负责记载在内核初始化时的物理向量的权能号。这些权能号的宏定义的命名规则为 RME_BOOT_VECT_<NAME>，如果该向量的名称叫 Tim1，那么宏名称为 RME_BOOT_VECT_TIM1。

另外，盛放这些物理向量权能的权能表的宏定义则为 RME_BOOT_CTVECT<n>，n 为从 0 开始按顺序排列的自然数。

6.3.3 固定文件：rme_boot.c

本文件负责启动内核时的物理向量信号端点声明和创建。它是由编译器自动生成的，内含两个函数，分别如下。

6.3.3.1 RME_Boot_Vect_Init

本函数负责初始化物理向量信号端点。这些端点必须在系统启动时被创建和初始化。

表 6-1 RME_Boot_Vect_Init 函数

原型	ptr_t RME_Boot_Vect_Init(struct RME_Cap_Captbl* Captbl, ptr_t Cap_Front, ptr_t Kmem_Front)
意义	初始化各个物理向量信号端点。
返回值	ptr_t 返回创建完这些物理向量信号端点后的内核内存前端地址。
参数	struct RME_Cap_Captbl* Captbl 指向主权能表的权能。
	ptr_t Cap_Front 主权能表权能前端权能编号。
	ptr_t Kmem_Front 内核内存前端地址。

6.3.3.2 RME_Boot_Vect_Handler

本函数负责处理各物理向量的中断，调用用户在 rme_user.c 中填充的各个向量的响应函数，并向各个专用物理向量信号端点发送中断。在发送完成后，如果用户在响应函数中返回的值为 0，那么还要向通用向量信号端点发送一个信号。

表 6-2 RME_Boot_Vect_Handler 函数

原型	ptr_t RME_Boot_Vect_Handler(ptr_t Vect_Num)
意义	向各个物理向量中断端点发送中断。
返回值	ptr_t 返回在响应函数中用户返回的返回值。
参数	ptr_t Vect_Num

	需要响应的中断向量号。

6.3.4 用户可编辑文件：rme_user.c

本文件是用户可编辑文件，它含有四个钩子函数供用户填充定制化逻辑。现将这四个钩子函数列表如下。

6.3.4.1 RME_Boot_Pre_Init

本钩子用于在内核初始化其各数据结构之前初始化基本硬件。如果系统中有 SDRAM 控制器等存储器读写组件，那么它们需要在系统启动之前被初始化。该函数甚至会在系统设置处理器时钟频率和 PLL 之前被调用，而后续系统负责的初始化可能改写处理器时钟频率和 PLL 设置，这一点在编写该函数时应该考虑到。

表 6-3 RME_Boot_Pre_Init 函数

原型	void RME_Boot_Pre_Init(void)
意义	在系统启动之前初始化硬件。
返回值	无。
参数	无。

6.3.4.2 RME_Boot_Post_Init

本钩子用于在内核初始化其各数据结构之后初始化其他较复杂的硬件。如果系统中有某些硬件必须在系统启动后初始化，或者对系统默认初始化的时钟频率或 PLL 配置不满意，请在这里进行对它们进行最终修改。

表 6-4 RME_Boot_Post_Init 函数

原型	void RME_Boot_Pre_Init(void)
意义	在系统启动之后初始化硬件。
返回值	无。
参数	无。

6.3.4.3 RME_User_Kern_Func_Handler

本钩子用于处理用户定义的内核功能调用。用户只要在此函数中填充内容，判断要执行的是何种自定义内核功能调用，执行之并返回合适的返回值即可[1]。

[1] 当用户返回大于 0 的返回值时无需设置返回值到当前寄存器中，这个功能会由内核中调用本钩子的函数完成。有关具体信息请参见 RME 的内核手册。

表 6-5 RME_User_Kern_Func_Handler 函数

原型	ret_t RME_User_Kern_Func_Handler(ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)
意义	执行用户定义的内核功能调用函数。
返回值	ret_t 标志工作成功与否的状态码，如果成功请返回非负数，失败则返回负数。
参数	ptr_t Func_ID 主功能号。
	ptr_t Sub_ID 子功能号。
	ptr_t Param1 该函数的第一个参数。
	ptr_t Param2 该函数的第二个参数。

6.3.4.4 中断向量函数族

这些钩子是同一类钩子，在物理中断向量被触发时它们会先于任何中断端点被调用，从而做出对中断事件的立即响应。向专用物理向量信号端点和通用向量信号端点的发送在本钩子函数执行完成后才会得到执行，因此本钩子函数相当于裸机编程中的中断向量本身，或者 Linux 的中断向量上半段。

这些钩子函数的命名规则是 RME_Vect_<NAME>_User，比如如果中断向量的名称叫做 Tim1，那么钩子函数的名称就是 RME_Vect_TIM1_User。都有一个参数和一个返回值。

表 6-6 中断向量函数族（RME_Vect_<NAME>_User）

原型	ptr_t RME_Vect_<NAME>_User(ptr_t Vect_Num)
意义	在物理向量被触发后立即进行处理。
返回值	ptr_t 如果希望系统继续向通用中断端点发送信号，那么输入 0，否则输入一个非 0 值。需要注意的是，不论返回什么值，向专用物理向量信号端点的发送都一定会进行。
参数	ptr_t Vect_Num 本中断的中断向量号。

6.3.5 固定文件：链接器脚本

这是整个映像最终链接时使用的链接器脚本。编译器会自动生成它；用户无需改动，除非用户需要添加自己的代码到工程中。

6.4 用户态库工程

用户态库工程负责编译链接整个用户态库。它除了含有用户态库的一般源码，还包含了编译器生成的内核对象创建源码以及用户可编辑的源文件。

下面我们将对该工程的组成部分进一步讲解。

6.4.1 固定文件：用户态库固有文件

这些文件是内核本身的固有文件。不属于下面所提到的文件的文件都可以划分在这一组，它们提供通用的内核功能。这些文件是编译器直接从用户态库源码中拷贝来的。对于这些文件，我们建议用户不要进行任何改动。

6.4.2 固定文件：rvm_boot.h

本文件负责记载在用户态库初始化时的各类内核对象的权能号。这些权能号的宏定义的命名规则如下表所示。

表 6-7 内核对象权能号宏命名原则

内核对象	命名规则
进程	RVM_PROC_<PROC>，其中<PROC>是进程本身的名称。
页表	RVM_PGTBL_<PROC>_N#num，其中<PROC>是进程本身的名称，#num 是序号。
权能表	RVM_CAPTBL_<PROC>，其中<PROC>是进程本身的名称。
线程	RVM_PROC_<PRO>_THD_<THD>，其中<PROC>是进程本身的名称，<THD>是线程的名称。当不同进程之间的线程名称相同时，通过进程名的不同加以区分。
迁移调用目标	RVM_PROC_<PROC>_INV_<INV>，其中<PROC>是进程本身的名称，<INV>是迁移调用目标的名称。当不同进程之间的迁移调用目标名称相同时，通过进程名的不同加以区分。
迁移调用入口	它没有 RVM 权能号，因为它是由迁移调用目标产生的。
接收信号端点	RVM_PROC_<PROC>_RECV_<RECV>，其中<PROC>是进程本身的名称，<RECV>是接收信号端点的名称。当不同进程之间的接收信号端点名称相同时，通过进程名的不同加以区分。
发送信号端点	它没有 RVM 权能号，因为它是由接收信号端点产生的。
向量信号端点	RVM_BOOT_VECT_<VECT>，其中<VECT>是向量的向量名。
内核功能调用	它没有 RVM 权能号，因为它是由初始内核功能调用权能产生的。

同时，装载这些内核对象的权能表也有它们的命名法则，如下表所示。

表 6-8 内核对象权能号宏命名原则

内核对象	命名规则
进程	RVM_PROC_<PROC>，其中<PROC>是进程本身的名称。
页表	RVM_PGTBL_<PROC>_N#num，其中<PROC>是进程本身的名称，#num 是序号。
权能表	RVM_CAPTBL_<PROC>，其中<PROC>是进程本身的名称。
线程	RVM_PROC_<PRO>_THD_<THD>，其中<PROC>是进程本身的名称，<THD>是线程的名称。当不同进程之间的线程名称相同时，通过进程名的不同加以区分。
迁移调用目标	RVM_PROC_<PROC>_INV_<INV>，其中<PROC>是进程本身的名称，<INV>是迁移调用目标的名称。当不同进程之间的迁移调用目标名称相同时，通过进程名的不同加以区分。
迁移调用入口	它没有 RVM 权能号，因为它是由迁移调用目标产生的。
接收信号端点	RVM_PROC_<PROC>_RECV_<RECV>，其中<PROC>是进程本身的名称，<RECV>是接收信号端点的名称。当不同进程之间的接收信号端点名称相同时，通过进程名的不同加以区分。
发送信号端点	它没有 RVM 权能号，因为它是由接收信号端点产生的。
向量信号端点	RVM_BOOT_VECT_<VECT>，其中<VECT>是向量的向量名。
内核功能调用	它没有 RVM 权能号，因为它是由初始内核功能调用权能产生的。

6.4.3 固定文件：rvm_boot.c

本文件负责用户态库启动时对内核对象的创建。各个创建流程以及其内部含有的函数将在下面一一介绍。这些函数都既没有返回值有没有参数。

其中，创建内核对象的函数会被 RVM_Boot_Kobj_Crt 统一调用，这些函数的列表如下。

表 6-9 rvm_boot.c 中创建内核对象的函数

名称	意义
RVM_Boot_Virtep_Crt	创建虚拟机使用的信号端点。这些信号端点会用来接收从用户态库的中断转发服务发来的信号。
RVM_Boot_Captbl_Crt	创建各个进程使用的权能表。这些权能表存放各个进程中能引用的权能。
RVM_Boot_Pgtbl_Crt	创建各个进程使用的页表。它们决定了各个进程的地址空间。
RVM_Boot_Proc_Crt	创建各个进程本身。
RVM_Boot_Inv_Crt	创建各个原生进程内部的迁移调用目标。
RVM_Boot_Recv_Crt	创建各个原生进程内部的接收信号端点。

在内核对象创建完成后，还要对它们进行初始化。这些初始化函数会由 RVM_Boot_Kobj_Init 统一调用。这些函数的列表如下所示。

表 6-10 rvm_boot.c 中初始化内核对象的函数

名称	意义
RVM_Boot_Virtcap_Init	创建虚拟机使用的信号端点。这些信号端点会用来接收从用户态库的中断转发服务发来的信号。
RVM_Boot_Captbl_Init	创建各个进程使用的权能表。这些权能表存放各个进程中能引用的权能。
RVM_Boot_Pgtbl_Init	创建各个进程使用的页表。它们决定了各个进程的地址空间。
RVM_Boot_Thd_Init	创建各个进程本身。
RVM_Boot_Inv_Init	创建各个原生进程内部的迁移调用目标。

除了这些函数之外，如果用户在系统中声明了虚拟机，那么本文件中还会存在虚拟机管理使用的数据。这些数据以结构体数组的形式保存在本文件中。

6.4.4 用户可编辑文件：rvm_user.c

本文件是用户可编辑文件，放置着两个用户可编辑的钩子。它们分别会在用户态库初始化自己的数据结构之前和之后被调用。

6.4.4.1 RVM_Boot_Pre_Init

本钩子用于在用户态库初始化其各数据结构之前初始化一些用户需要初始化的额外软件包或硬件。

表 6-11 RVM_Boot_Pre_Init 函数

原型	void RVM_Boot_Pre_Init(void)
意义	在用户态库初始化数据结构之前执行的初始化流程。
返回值	无。
参数	无。

6.4.4.2 RVM_Boot_Post_Init

本钩子用于在用户态库初始化其各数据结构之后初始化其他软件或硬件。

表 6-12 RVM_Boot_Post_Init 函数

原型	void RVM_Boot_Pre_Init(void)
意义	在用户态库初始化之后执行的初始化流程。
返回值	无。
参数	无。

6.4.5 固定文件：链接器脚本

这是用户态库工程链接时使用的链接器脚本。编译器会自动生成它；用户无需改动，除非用户需要添加自己的代码到工程中。

6.5 原生进程工程

原生进程工程负责原生进程源文件的组织和编译。每一个原生进程都对应一个工程；每个工程内部的文件都随着该工程的配置有所不同。

下面我们对这些工程的内容加以进一步讲解。

6.5.1 固定文件：进程固有文件

这些文件是每个原生进程都有的固有文件。不属于下面所提到的文件的文件都可以划分在这一组，它们提供通用的调用用户态库服务的基本接口。这些文件是编译器直接从用户态库的客户端源码中拷贝来的。对于这些文件，我们建议用户不要进行任何改动。

关于进程固有文件中提供的功能以及其各个接口的使用方式，请参考 RVM 的用户手册。

6.5.2 固定文件：进程汇编文件

进程汇编文件是以 proc_<name>_asm.S 或类似名称命名的汇编文件。在该汇编文件中包含了所有进程中的线程和迁移调用的入口点。这些入口点将被引用来运行这些线程和迁移调用入口。

6.5.3 固定文件：进程主头文件

进程主头文件是以 proc_<name>.h 命名的头文件。在该头文件中包含了所有在进程权能表中的、进程可引用的内核对象的本地权能号。这些本地权能号的宏定义的命名规则如下表所示。

表 6-13 进程可引用的内核对象权能号宏命名原则

内核对象	命名规则
进程	进程对象不可引用。
页表	页表对象不可引用。
权能表	权能表对象不可引用。
线程	线程对象不可引用。
迁移调用目标	迁移调用目标不可引用。
迁移调用入口	PROC_<PROC>_PORT_<PORT>，其中<PROC>是入口对应的调用目标所在进程的名称，<PORT>是调用入口对应的调用目标的名称。当不同进程之间的迁移调用目标名称相同时，通过进程名的不同加以区分，从而确定迁移调用入口对应的目标究竟为何。
接收信号端点	RECV_<RECV>。其中<RECV>是接收端点的名称。
发送信号端点	PROC_<PROC>_SEND_<SEND>，其中<PROC>是发送信号端点对应的接收信号端点所

内核对象	命名规则
	在进程的名称，<SEND>是发送信号端点对应的接收信号端点的名称。当不同进程之间的接收信号端点名称相同时，通过进程名的不同加以区分，从而确定发送信号端点对应的接收信号端点究竟为何。
向量信号端点	VECT_<VECT>，其中<VECT>是向量的向量名。
内核功能调用	KERN_<KERN>，其中<KERN>是向量的向量名。

此外，进程可以访问的所有内存块的基址和大小也被定义成宏，以方便用户应用程序进行引用。这些宏的命名规则如下。

在这些命名规则中，<NAME>都是内存块的名称。如果该内存块是匿名的，那么该部分将被内存块的十六进制起始地址替换，如某匿名私有代码内存块的起始地址为 0x1000，那么它对应的基址宏定义即为 CODE_0X1000_BASE。

表 6-14 进程可引用的内存块宏命名原则

内核对象	命名规则
私有代码内存基址	CODE_<NAME>_BASE
私有代码内存大小	CODE_<NAME>_SIZE
私有数据内存基址	DATA_<NAME>_BASE
私有数据内存大小	DATA_<NAME>_SIZE
私有设备内存基址	DEVICE_<NAME>_BASE
私有设备内存大小	DEVICE_<NAME>_SIZE
共享代码内存基址	SCODE_<NAME>_BASE
共享代码内存大小	SCODE_<NAME>_SIZE
共享数据内存基址	SDATA_<NAME>_BASE
共享数据内存大小	SDATA_<NAME>_SIZE
共享设备内存基址	SDEVICE_<NAME>_BASE
共享设备内存大小	SDEVICE_<NAME>_SIZE

6.5.4 用户可编辑文件：进程主源文件

进程主源文件是以 proc_<name>.c 命名的源文件，其中<name>为进程的名称。在该源文件中包含了所有的线程和迁移调用入口的钩子函数。用户需要填充这些钩子函数以使每个线程和迁移调用入口有代码可以运行。

其中，线程的钩子函数均命名为 Thd_<Name>，其中<Name>为线程名称；迁移调用目标的钩子函数均命名为 Inv_<Name>，其中<Name>为迁移调用目标名称。

6.6 虚拟机工程

虚拟机工程负责虚拟机源文件的组织和编译。每一个虚拟机都对应一个工程；每个工程内部的文件都随着该工程的配置有所不同。

下面我们对这些工程的内容加以进一步讲解。

6.6.1 固定文件：虚拟机固有文件

这些文件是每个虚拟机都有的固有文件。不属于下面所提到的文件的文件都可以划分在这一组，它们提供通用的调用虚拟机监视器服务的基本接口。这些文件是编译器直接从用户态库的客户端源码中拷贝来的。对于这些文件，我们建议用户不要进行任何改动。

关于虚拟机固有文件中提供的功能以及其各个接口的使用方式，请参考 RVM 的用户手册。

6.6.2 固定文件：虚拟机汇编文件

虚拟机汇编文件是以 proc_<name>_asm.S 或类似名称命名的汇编文件。在该汇编文件中包含了虚拟机中的两个线程[1]的入口点。这些入口点将被引用来运行这些线程。虚拟机中没有迁移调用目标，所以不存在迁移调用目标的入口点。

6.6.3 固定文件：虚拟机主头文件

虚拟机主头文件是以 proc_<name>.h 命名的头文件。在该头文件中包含了所有在虚拟机权能表中的、虚拟机可引用的内核对象的权能号，也包含了各个内存块的基址和长度。关于这些宏定义的信息请参看进程相关章节，因为虚拟机在这方面和进程一致。

6.6.4 用户可编辑文件：虚拟机主源文件

虚拟机主源文件是以 proc_<name>.c 命名的源文件，其中<name>为虚拟机的名称。在该源文件中包含了虚拟机中的两个线程的钩子函数，并且其内容已经被预填充。

如果用户在生成虚拟机时选择安装某个操作系统，那么本文件无需用户改动。如果用户选择了无虚拟机，那么用户程序将从 Thd_User 这一钩子中开始运行，用户需要自行填充该钩子的内容以启动裸机程序。

6.6.5 固定文件：客户操作系统文件

客户操作系统文件是用户选择安装操作系统时生成的文件。这些文件是直接从客户操作系统中拷贝的，并且会随着客户选择的操作系统的不同安装不同的操作系统文件。关于不同客户操作系统的使用方法，请参见各操作系统的使用手册。

---

[1] 分别为用户线程和中断线程

需要注意的是，在使用这些虚拟机时，用户必须遵循它们的操作系统的许可证。一个虚拟机内部的许可证并不会传染到虚拟机外部，这使得用户可以方便地运行任何许可证的软件。不过，AGPLv3 及类似许可证的软件因为具有跨网络、跨应用传染性，因此只要它和系统的其他部分发生任何通信，AGPLv3 许可证仍然会传染到整个系统。

6.7 本章参考文献

无

第 7 章　工程使用说明

7.1　原生进程使用说明

本编译器生成的进程是一个可以有多个线程在内部运行的、具备独立地址空间的保护域。这些原生进程内部可用的系统服务只有 RME 本身的系统调用。所有的进程内部的线程的优先级都比虚拟机要高；只有当所有原生进程内部的线程都阻塞时才轮得到虚拟机运行。

原生进程中一旦其中发生任何错误，比如访存错误、除以 0 错误等，都会导致整个系统的重启。这样设计的原因是，原生进程往往负责系统最底层的处理，一旦发生错误即意味着系统本身的崩溃。在原生进程之间采用隔离的唯一意义是防止错误扩散，以及能够最早发现错误并且加以处理，不会发生大规模系统错乱。

7.1.1　线程使用说明

原生进程中的线程的唯一目的是放置那些较为简单的、需要快速响应的硬实时驱动程序。对于较复杂的库和通讯协议，或者那些不可靠而可能随时重启的代码，推荐将它们置于虚拟机中而不是置于原生进程中。

每个线程在系统中应当代表一个相对独立的任务。要创建线程，需要给出其名称、栈大小、优先级和参数，然后编译器会生成代码以在系统启动时创建该线程。所有线程在系统启动后会自动开始运行，编译器不提供在用户态控制这些线程运行的方法，也不允许在运行时创建或者修改任何线程。这些线程也无法调用任何高级系统调用而仅有 RME 系统调用可用。

如果用户需要在运行时创建线程，或者需要较为复杂的系统服务。那么应当使用虚拟机，并且借助虚拟机内部的其他库来完成。通常而言，推荐的不同软件模块的功能放置位置如下表所示。

表 7-1　软件模块的推荐功能划分

软件模组	推荐放置位置
机电控制	原生进程
PID 算法	原生进程
音视频处理	虚拟机
无线通信	虚拟机
加热控制	原生进程
安全功能	原生进程
图形界面	虚拟机
键盘控制	虚拟机
现场总线通信	原生进程
文件系统	虚拟机

7.1.2 同步迁移调用使用说明

同步迁移调用是 RME 提供的唯一的同步通信方式，它允许一个进程内的线程暂时跨越保护域到另一个进程内部执行一段时间再返回。它的主要目的是方便客户进程调用服务器进程。比如 A 和 B 两个进程都依赖于 C 进程的服务，需要调用 C 进程中的函数，但是 C 进程并不信任 A 进程和 B 进程，而且 A 和 B 进程也不互相信任。

此时，可以在 C 进程中创建一个同步迁移调用目标，在 A 和 B 中使用同步迁移调用入口指向这个目标，然后在 A 和 B 中使用同步迁移入口时会自动跳转到 C 进程的这一目标处开始执行。

需要注意的是，每一个同步迁移调用目标在同一个时刻都只能有一个线程在使用。如果有一个迁移调用目标已经在执行，那么该调用目标必须等到执行完成后才可以给另一个线程执行。

要建立同步迁移调用目标，需要给出其名字和栈大小。然后，建立一个同步迁移调用入口即可，此时要填写目标所在的进程名和目标本身的名称。同步迁移调用是没有优先级的概念的，在执行时其优先级和调用它的线程的优先级相同，因为逻辑上讲只是一个线程的跨保护域执行。

同步迁移调用由两个系统调用负责，分别是入口端的 RVM_Inv_Act 和目标端的 RVM_Inv_Ret 当要进行同步迁移调用时，在调用端调用并传入 RVM_Inv_Act 要调用的入口的权能号[1]和参数。要退出同步迁移调用时，则必须调用 RVM_Inv_Ret 并传递返回值以返回给调用者。注意，同步迁移调用不能直接返回，要返回则必须调用 RVM_Inv_Ret 进行显式返回，否则会导致出错。关于这两个系统调用的确切信息请参见 RVM 本身的手册。

7.1.3 异步信号端点使用说明

异步信号端点是 RME 提供的唯一的异步通信方式。它包括发送信号端点和接收信号端点两种。某个线程可以先在接收信号端点上阻塞，然后由另一个线程调用发送信号端点将其解除阻塞。接收信号端点是计数的，发送多少个信号在接收信号端点上就会收到多少个信号并解除多少次阻塞。

异步信号端点相比同步迁移调用有更高的性能开销，但其使用灵活方便，并且不需要单独的栈区，对设计消息队列类传输非常友好。同时，接收端还支持四种不同的接收方法，支持阻塞和非阻塞，以及一次接收一个信号或全部信号。

要建立接收信号端点，仅需要给出其名字。然后，在建立发送端点时需要指明接收端点所在的进程及接收端点的名字。

异步信号端点由两个系统调用负责，分别是发送端的 RVM_Sig_Rcv 和接收端的 RVM_Sig_Snd。当要接收信号时，接收信号的线程调用 RVM_Sig_Rcv 并传入接收端点的权能号[2]，在该端点上发生阻塞或者不阻塞；发送信号端则可以调用 RVM_Sig_Snd 并传入发送端点的权能号[3]来唤醒潜在的阻塞线程。关于这两个系统调用的确切信息请参见 RVM 本身的手册。

---

[1] 由于权能号已经被定义为宏，因此只要传入那个宏定义即可。

[2] 由于权能号已经被定义为宏，因此只要传入那个宏定义即可。

[3] 由于权能号已经被定义为宏，因此只要传入那个宏定义即可。

7.1.4 向量信号端点使用说明

向量信号端点是 RME 提供来响应外部中断时使用的。它只包括一种端点，也即物理向量接收端点。它是 RME 提供的唯一一种接收中断的机制。这些信号端点和异步信号端点没有什么不同，唯一的区别是它们的发送是不受用户控制的，而是系统内核在收到中断时直接发送的。

要建立向量信号端点，仅需要给出其名字。这个名字必须在芯片 XML 中可以查找到，此中断向量信号端点的建立才能成功。需要注意的是，一个向量信号端点只能够被建立一次，也即如果一个向量信号端点已经在一个进程内被建立，不能在系统内的任何地方——无论同一进程内或其他进程内——建立同一个向量信号端点。

在向量信号端点上进行接收使用的函数和异步信号端点使用的函数是一样的，都是 RVM_Sig_Rcv。关于这个个系统调用的确切信息请参见 RVM 本身的手册。

7.1.5 从原生进程向虚拟机传递信号

要从原生进程发送通知给虚拟机，请参见 7.2.3。

7.2 虚拟机使用说明

本编译器生成的虚拟机是一个可以运行客户操作系统的、具备独立地址空间的保护域。这些虚拟机的优先级比所有原生进程都低，而且实时响应能力比原生进程相差很多[1]。它们在系统中存在的目的是运行那些功能多样但可靠性较差的任务。

在虚拟机中一旦其中发生任何错误，比如访存错误、除以 0 错误等，都会导致虚拟机本身的重启，但系统本身并不会重启。这样，虚拟机之间是完美隔离的，一个虚拟机的崩溃绝不会导致整个系统的崩溃[2]。这使得系统可以放心引入那些较为庞大的功能，比如不可靠的开源文件系统、协议栈或是多媒体功能等。一些易于造成故障的庞大的驱动程序，诸如显示驱动程序等等，也可以放置在虚拟机中运行。在使用虚拟机操作系统的接口时，可以完全将它们当做是传统 RTOS 的接口来进行编程，无需过多考虑它们的底层是如何实现的。所有的操作系统 API 都保证会像虚拟化之前那样运作。

在另一些场合，虚拟机的特性也会很有用处。比如，我们需要整合多个原本运行在裸机上的程序到同一颗微控制器中，这种场合下修改原有程序以使之适合新的操作系统接口本身是不合适的。我们希望能够在不修改绝大多数代码的情况下安全地复用这个工程。此时，可以在生成虚拟机时选择裸机[3]，此时用户只要将原有的 main 函数改名并放在 Thd_User 中调用即可。原有的代码除了涉及到微控制器初始化和中断向量注册[4]的部分一概不需要修改即可直接使用。

---

[1] 通常而言，慢 3-5 倍。

[2] 除非系统的主要功能由该虚拟机执行。

[3] 也即不预安装操作系统代码。

[4] 涉及到特权操作的部分需要重写。最常见的特权操作是微控制器初始化和中断向量注册。

7.2.1 虚拟机中特权操作的特殊处理

如果虚拟机中存在一些涉及特权操作的部分，由于虚拟机整个运行在用户态，因此这部分代码可能需要重新设计。取决于性质的不同，重新设计的思路有三种，分别如下所述。

7.2.1.1 直接删除特权代码或将它们移动到内核

在裸机程序中总有一部分代码是负责初始化 CPU 的。这部分代码可以直接删除，因为 CPU 已经在系统内核启动时初始化过了。

如果有一些代码需要永久性初始化某个外设，那么可以使用内核工程提供的启动时初始化钩子进行一次性初始化。

7.2.1.2 使用超级调用代替特权操作

通常 RTOS 中都包含一些开关中断、锁调度器的函数。这些函数在底层很有可能调用特权态操作。通常而言，编译器预安装好的操作系统源代码中已经通过超级调用实现了这些功能，此时只要直接使用虚拟机中客户操作系统提供的 API 即可。

如果使用裸机程序，开关中断的操作需要直接调用对应的超级调用进行。关于这些超级调用的信息，请参见 RVM 的手册。

7.2.1.3 将特权操作操作封装在内核功能调用中

有一些特权操作涉及 CPU 的特殊功能，并且 RVM 也没有提供相应的超级调用来使用这些功能。此时，需要将这些特权操作封装在内核功能调用中使用。在内核工程中有内核调用钩子，用户可以分配系统没有使用的内核调用号来使用这些内核功能。

关于内核功能，请参见 RME 和 RVM 的使用手册。

7.2.2 虚拟机中断源使用说明

虚拟机要能够对外部中断做出响应，就必须将其物理中断源和虚拟中断源建立一对一或一对多的映射。系统中每一个物理中断源都会被分配一个中断号，每一个虚拟中断源也同样会被分配一个虚拟中断号。当物理中断源收到响应时，所有登记在该物理中断源下的虚拟中断源都会产生中断；在这之后，所有注册在这些虚拟中断源上的虚拟中断向量均会按照虚拟机的优先级从大到小加以调用。

要在物理中断源和虚拟中断源之间建立映射，需要使用 RVM_Hyp_Reg_Phys，并且传入需要注册的物理中断源和虚拟中断源的中断号。关于哪个物理中断源对应哪个物理中断，需要参看对应架构上的参考手册。同一个物理中断源只能在一个虚拟机中被注册一次到一个虚拟中断向量；如果有重复注册的情况则会被驳回。

要使得虚拟中断源能够响应该虚拟中断，还应该给它注册中断处理函数。在不注册中断处理函数的情况下，即便虚拟中断源被激活，也不会发生任何响应。这个中断处理函数的性质和裸机程序的中断向量是别无二致的。要给某虚拟中断源注册中断处理函数，使用 RVM_Virt_Reg_Vect 并传入虚拟中断号和虚拟中断向量的函数指针。如果要解除这个注册，使用 RVM_Virt_Reg_Vect 并传入虚拟中断号和空指针即可。

要解除物理中断源和虚拟中断源之间的映射，需要使用 RVM_Hyp_Del_Vect 并传入虚拟中断源的中断号，此时该虚拟中断源与物理中断源的对应被解除。

关于上述函数的具体信息，请参考 RVM 的技术手册。

7.2.3 虚拟机事件源使用说明

在某些状况下，虚拟机之间以及虚拟机和原生进程也需要通信。关于虚拟机发送、原生进程接收的情况，和原生进程之间进行异步通信的方法是完全一致的，当虚拟机中的发送端点被激活，原生进程中的接收端点就会收到信号并解除线程的阻塞。

当虚拟机之间进行通信或原生进程需要发送信号到虚拟机时，需要使用一种称为虚拟机事件源的资源。这些事件源和物理中断源是很相似的，其区别在于它们的编号是独立与物理中断源的，并且不响应任何外部刺激。

要在事件源和虚拟中断源之间建立映射，需要使用 RVM_Hyp_Reg_Evt，并且传入需要注册的事件源和虚拟中断源的中断号。同一个事件源只能在一个虚拟机中被注册一次到一个虚拟中断向量；如果有重复注册的情况则会被驳回。和使用物理中断源时的情况类似，虚拟中断源需要注册处理函数才能响应。

要解除事件源和虚拟中断源之间的映射，也同样使用 RVM_Hyp_Del_Vect 并传入虚拟中断源的中断号，此时该虚拟中断源与事件源的对应被解除。

任何原生进程都可以通过调用 RVM_Proc_Send_Evt 向任意的事件源发送事件。一旦事件源被激活，所有注册于该事件源的虚拟中断源都会收到中断。

虚拟机要向事件源发送事件则需要针对每个事件源的单独的发送权限。每个虚拟机都可以通过 RVM_Hyp_Add_Evt 自行授权，并可以通过 RVM_Hyp_Del_Evt 撤销对自己的授权。自行授权是信息安全的，因为我们能够确定虚拟机初始化时其代码段未经修改，而在初始化结束后我们调用下面会介绍的 RVM_Hyp_Lock_Vect 锁定所有的授权。确定虚拟机有对某事件源的发送权限后，可以调用 RVM_Hyp_Send_Evt 向事件源发送事件。

有时候我们不希望虚拟机在完成初始化后还能修改这些映射或权限。对于某些信任度较低的虚拟机，我们需要在完成初始化后锁定这些映射，这样一旦虚拟机被攻破，也不会造成拒绝服务攻击。为此，RVM 提供了 RVM_Hyp_Lock_Vect 来锁定虚拟机的映射和权限。一旦锁定，这些信息就不可修改，除非虚拟机因为故障或看门狗超时重启。对于那些需要额外灵活性的、较受信任的虚拟机，不锁定也是可以的。

关于上述函数的具体信息，请参考 RVM 的技术手册。

7.2.4 虚拟机看门狗使用说明

为了提高软件的功能安全性，RVM 提供了虚拟机看门狗功能。要使用此功能，在 GSC 的项目中看门狗超时时间必须不为 0。当虚拟机启动时，看门狗是默认关闭的。当虚拟机完成其所有初始化后，调用 RVM_Hyp_Feed_Wdog 即启动看门狗。投喂看门狗也使用同样的 RVM_Hyp_Feed_Wdog。看门狗会从其超时时间开始倒数，一旦倒数至 0，虚拟机会被重启，而投喂则会重置该计时器。看门狗计算的超时时间是虚拟机实际运行的时间，如果虚拟机未被调度，则看门狗不会倒数。

7.2.5 从虚拟机向原生进程传递信号

要从虚拟机向原生进程发送通知，使用的也是异步信号端点。虽然在虚拟机内部不能存在接收端点，但是可以存在向其他原生进程发送的发送端点。要向其他原生进程通信，只要在 GSC 中声明这些端点在虚拟机中调用 RVM_Sig_Snd 向它们发送即可。

7.3 本章参考文献

无

第 8 章 移植 GSC 到新架构

8.1 移植概述

移植 GSC 到新架构指使 GSC 支持该新架构上的工程的编译。GSC 使用 C++14 编写，在移植时仅需要重写和架构有关的部分。在移植时要重写的类是固定的。只要继承这些类并修改相关文件即可。

架构有关的部分均位于 GSC 的以架构名命名的文件夹内，在源文件夹内存放的是 C++源文件，在头文件夹内存放的是 C++头文件，例如，ARMv7-M 架构的源文件存放位置为 Source/A7M，头文件的存放位置为 Include/A7M。在开发时，这个架构也可以作为移植参考。

本章的读者是 GSC 的开发人员。GSC 的使用者可以直接略过本章。在本章中，我们先介绍 GSC 的工作流程和原理，再介绍其移植。

8.2 GSC 的主要工作流程

和语言编译器一样，GSC 也包括了前端、中间优化和后端三个部分。其中，前端包括 XML 分析和架构选择，中间优化包括了内存分配和内核对象分配，后端则包括了到所有工具链的输出。下面我们将介绍其各个阶段的工作原理，及其在代码中对应的位置。

8.2.1 XML 分析阶段和架构选择阶段

这一阶段主要分析 XML 文件，并根据 XML 文件的描述选择相应的架构。这一部分的主要步骤及其对应的函数如下。

表 8-1 XML 分析阶段和架构选择阶段

编号	文件	主要函数	操作
1	rme_mcu.cpp	Main::Main	读取命令行参数，并判断它们是否合法。
2	rme_mcu.cpp rme_proj.cpp rme_proc.cpp rme_comp.cpp rme_raw.cpp	Main::Parse Proj::Proj RME::RME Comp::Comp Raw::Raw	解析工程 XML 文件中的内核部分。
3	rme_mcu.cpp rme_proj.cpp rme_proc.cpp rme_comp.cpp	Main::Parse Proj::Proj RVM::RVM Comp::Comp	解析工程 XML 文件中的用户态库部分。
4	rme_mcu.cpp rme_proj.cpp rme_proc.cpp	Main::Parse Proj::Proj Proc::Proc	解析工程 XML 文件中的原生进程和虚拟机部分，及其中的所有内核对象。

编号	文件	主要函数	操作
	rme_mem.cpp	Virt::Virt	
	rme_kobj.cpp	Mem::Mem	
	rme_captbl.cpp	Kobj::Kobj	
	rme_inv.cpp	Inv::Inv	
	rme_kern.cpp	Kern::Kern	
	rme_pgtbl.cpp	Pgtbl::Pgtbl	
	rme_port.cpp	Port::Port	
	rme_proc.cpp	Proc::Proc	
	rme_recv.cpp	Recv::Recv	
	rme_send.cpp	Send::Send	
	rme_thd.cpp	Thd::Thd	
	rme_vect.cpp	Vect::Vect	
5	rme_chip.cpp rme_mem.cpp rme_vect.cpp	Chip::Chip Option::Option Mem::Mem Vect::Vect	解析芯片 XML 文件。
6	rme_mcu.cpp	Main::Parse	根据架构选择合适的后端进行初始化。
7	rme_mcu.cpp	Main::Check_Chip	检查芯片的各个设置项都在允许范围内。

8.2.2 内存分配阶段和页表分配阶段

这一阶段主要将各段内存按照架构的要求进行对齐、分配那些未分配的共享内存和进程私有内存。在这之后，我们还要调用架构提供的页表分配函数完成页表的分配。在下面的表格中，进程指原生进程和虚拟机，代码段指私有代码段和共享代码段。

表 8-2 内存分配阶段和页表分配阶段

编号	文件	主要函数	操作
1	rme_proj.cpp	Plat::Align_Mem	进行架构所需的内存对齐。
2	rme_mcu.cpp	Main::Alloc_Mem Main::Add_Extmem	添加外部存储器到芯片的存储器列表中并检查是否有相互覆盖的段落。如果没有则继续。
3	rme_mcu.cpp	Main::Alloc_Mem Main::Alloc_Code	检查指定了起始地址的代码内存段的合法性，并分配未指定起始地址的代码内存段。
4	rme_mcu.cpp	Main::Alloc_Mem Main::Alloc_Data	检查指定了起始地址的数据内存段的合法性，并分配未指定起始地址的数据内存段。

编号	文件	主要函数	操作
5	rme_mcu.cpp	Main::Alloc_Mem Main::Add_Shmem	根据各进程的引用添加共享内存段到各个进程的地址空间。
6	rme_mcu.cpp	Main::Alloc_Mem Main::Check_Addrspace	检查各进程的地址空间没有相互覆盖的内存段。
7	rme_mcu.cpp	Main::Alloc_Mem Main::Check_Code	检查各进程的主私有代码段不相互重叠。
8	rme_mcu.cpp	Main::Alloc_Mem Main::Check_Device	检查各进程的各设备段的合法性。
9	rme_proj.cpp	Plat::Alloc_Pgtbl	分配各个进程的页表。

8.2.3 内核对象权能号及其宏定义分配阶段

这一阶段主要统计各种内核对象的个数，并为它们分配一个全局权能号和一个本地权能号。只有那些需要创建、占据内核内存的内核对象才会拥有全局权能号，只有那些能够在原生进程或虚拟机中引用的权能才会具有本地权能号。这些权能号的宏定义也在此分配。

全局权能号是一个从 0 开始的、每类内核对象分开计算的线性权能号，它的增长没有上限。虽然在 32 位系统中，权能表大小的上限是 128，但全局权能号仍然有可能超过这个值。编译器的处理方式是将这些权能都放在扩展的权能表中，每当有 128 个权能，就新开一个扩展权能表放置这些权能。

本地权能号则是直接放置在进程权能表中的权能号。任何一个进程的本地权能号的数量都不能超过权能表上限的大小，也即在 32 位系统中，我们最多在进程中可以引用 128 个内核对象。但是，这并不意味着内核对象的数目不能超过 128，因为有很多内核对象是不能在进程中加以引用的。

表 8-3 内核对象权能号及其宏定义分配阶段

编号	文件	主要函数	操作
1	rme_mcu.cpp	Main::Alloc_Cap	检查一些通用参数的合法性，比如内核优先级数量是否是处理器字长的整数倍等等。
2	rme_mcu.cpp rme_kobj.cpp rme_proc.cpp	Main::Alloc_Cap Kobj::Check_Kobj	检查各个进程是否有重名现象，以及检查各个进程的名字是否是合法的 C 标识符。
3	rme_mcu.cpp rme_proc.cpp rme_kobj.cpp	Main::Alloc_Cap Proc::Check_Kobj Kobj::Check_Kobj	检查各个进程内部的各类内核对象是否有重名现象，以及检查这些内核对象的名字是否是合法的 C 标识符。
4	rme_mcu.cpp rme_vect.cpp	Main::Alloc_Cap Vect::Check_Vect	检查各个进程内部的中断向量端点的名称是否是合法的 C 标识符，以及它们是否是系统中对

编号	文件	主要函数	操作
			该中断向量唯一的引用。一个中断向量在整个工程中只允许被引用一次。
5	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Cap Proc::Alloc_Loc	分配各个能够在本地被引用的内核对象的本地权能号。
6	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Cap Proc::Alloc_RVM	分配各个要被创建的内核对象的全局权能号。
7	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Cap Proc::Alloc_Macro	分配本地权能号和全局权能号的宏定义。
8	rme_mcu.cpp	Main::Alloc_Mem Main::Link_Cap	检查是否每个迁移调用入口都对应一个存在的迁移调用目标。
9	rme_mcu.cpp	Main::Alloc_Mem Main::Link_Cap	检查是否每个发送信号端点都对应一个接收信号端点。
10	rme_mcu.cpp	Main::Alloc_Mem Main::Link_Cap	检查是否每个中断向量端点都是合法的，也即它们是否在芯片 XML 中被声明。

8.2.4 内核对象内存分配阶段

这一阶段主要统计各个内核对象所需的内存大小，并对这些内核对象的存储位置进行静态分配。在这一阶段，整个系统的内存布局细节将被完全决定。

表 8-4 内核对象内存分配阶段

编号	文件	主要函数	操作
1	rme_mcu.cpp rme_proj.cpp	Main::Alloc_Obj Proj::Kobj_Alloc	进行初次内核对象地址分配，此时主权能表大小仍然未知，因此未参与分配。
2	rme_mcu.cpp rme_proj.cpp	Main::Alloc_Obj Proj::Kobj_Alloc	进行二次内核对象地址分配，此时主权能表大小已知，得到最终的内核内存布局。
3	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Obj RME::Alloc_Kmem	分配内核本身的内存布局。分配是从内核数据段的末尾开始的。首先分配的是中断标志区域[1]。然后分配的是事件标志区域[2]。然后分配内核栈区和内核内存，剩下的部分为内核本身的全局变量所用。
4	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Obj RVM::Alloc_Mem	分配用户态库的内存布局。分配是从用户态库数据段的末尾开始的。首先分配的是启动守护进程的栈，然

[1] 该区域大小在任何架构上现在都固定为 512 字节

[2] 该区域大小在任何架构上现在都固定为 512 字节

编号	文件	主要函数	操作
			后是安全守护进程的栈。如果用到了虚拟机，那么还要分配超级调用守护进程的栈、中断分配守护进程的栈和定时器处理进程的栈。
5	rme_mcu.cpp rme_proc.cpp	Main::Alloc_Obj Proc::Alloc_Mem	分配各个进程的内存布局。这分为两方面，一方面是其线程入口和迁移调用目标入口在代码段上分配，另一方面是其线程的栈和迁移调用的栈在数据段上的分配。线程入口和迁移调用入口会在代码段的开头开始分配[1]；其栈则会在数据段的结尾分配，先分配那些线程的栈，再分配那些迁移调用的栈。如果这个进程是虚拟机，那么还要分配其虚拟向量标志位空间[2]、虚拟调用参数空间[3]和虚拟机寄存器空间[4]。

8.2.5 输出生成阶段

这一阶段将产生真正的输出文件，包括所用到的全部头文件、源文件、链接器脚本和 IDE 工程文件。在这一阶段，所有的文件拷贝和编辑功能将被完成，生成可被 IDE 加载的工程。

表 8-5 输出生成阶段

编号	文件	主要函数	操作
1	rme_mcu.cpp rme_genrme.cpp	Main::Gen_RME RME_Gen::Folder	生成内核工程的各个文件夹。
2	rme_mcu.cpp rme_genrme.cpp	Main::Gen_RME RME_Gen::Conf_Hdr	生成内核工程的配置头文件。
3	rme_mcu.cpp rme_genrme.cpp	Main::Gen_RME RME_Gen::Boot_Hdr	生成内核工程的启动头文件。
4	rme_mcu.cpp rme_genrme.cpp	Main::Gen_RME RME_Gen::Boot_Src	生成内核工程的启动源文件。
5	rme_mcu.cpp rme_genrme.cpp	Main::Gen_RME RME_Gen::User_Src	生成内核工程的用户源文件。
6	rme_mcu.cpp	Main::Gen_RME	生成内核工程的架构相关文件。

[1] 并且每一项都总是占据 8 个机器字的空间

[2] 该区域大小在任何架构上现在都固定为 256 字节

[3] 该区域大小在任何架构上现在都固定为 256 字节

[4] 该区域大小在任何架构上现在都固定为 512 字节

编号	文件	主要函数	操作
	rme_genrme.cpp	RME_Gen::Plat_Gen	
7	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::Folder	生成用户态库工程的各个文件夹。
8	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::Conf_Hdr	生成用户态库工程的配置头文件。
9	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::Boot_Hdr	生成用户态库工程的启动头文件。
10	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::Boot_Src	生成用户态库工程的启动源文件。
11	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::User_Src	生成用户态库工程的用户源文件。
12	rme_mcu.cpp rme_genrvm.cpp	Main::Gen_RVM RVM_Gen::Plat_Gen	生成用户态库工程的架构相关文件。
13	rme_mcu.cpp rme_genproc.cpp	Main::Gen_Proc Proc_Gen::Folder	生成进程工程的各个文件夹。
14	rme_mcu.cpp rme_genproc.cpp	Main::Gen_Proc Proc_Gen::Proc_Hdr	生成进程工程的主源文件。
15	rme_mcu.cpp rme_genproc.cpp	Main::Gen_Proc Proc_Gen::Proc_Src	生成进程工程的主头文件。
16	rme_mcu.cpp rme_genproc.cpp	Main::Gen_Proc RVM_Gen::Plat_Gen	生成进程工程的架构相关文件。
17	rme_mcu.cpp rme_genproj.cpp	Main::Gen_Proj Proj_Gen::Folder	生成工作空间的各个文件夹。
18	rme_mcu.cpp rme_genproj.cpp	Main::Gen_Proj Proj_Gen::Plat_Gen	生成工作空间的架构相关文件。

8.2.6 报告生成阶段

这一阶段将生成最终的报告。最终报告中包含了这一次编译所生成的工程的基本情况，如详细内存使用状况、内核对象的数目等。至此，编译器的整个编译流程结束。

表 8-6 输出生成阶段

编号	文件	主要函数	操作

编号	文件	主要函数	操作
1	rme_mcu.cpp	Main::Gen_Report Stats::Kobj_Stats	生成和内核对象数量有关的统计数据。
2	rme_mcu.cpp	Main::Gen_Report Stats::Kmem_Stats	生成和内核对象内存消耗有关的统计数据。
3	rme_mcu.cpp	Main::Gen_Report Stats::User_Stats	生成和用户程序内存消耗有关的统计数据。
4	rme_mcu.cpp	Main::Gen_Report Stats::Plat_Stats	生成和平台内存使用率有关的统计数据。
5	rme_mcu.cpp	Main::Gen_Report	根据上面各步骤生成的数据，打印报告本身。

8.3 次要 GSC 组件的工作原理

GSC 中还有一些辅助上述主要工作流程的次要组件。关于这些次要组件的简要说明如下。

8.3.1 错误检查原理

错误检查与报告在 GSC 中是使用异常实现的。当发现任何错误时，GSC 都会将该错误立即报告给用户并且停止输出流程。所有的错误报告都必须使用 try-catch 语句块实现。GSC 的每个函数都有可能抛出异常，因此在主函数 main 处有专门处理异常并打印的语句。

8.3.2 文件读写原理

文件读写在 GSC 中由 rme_doc.cpp 和 rme_fsys.cpp 两个文件组成。rme_doc.cpp 主要负责文件的以段或行为单位的格式化读写，而 rme_fsys.cpp 则负责与底层文件系统的交互。GSC 支持两个文件系统，分别是本机文件系统和打包文件系统；本机文件系统在开发时使用，而打包文件系统则在发布时使用。

8.3.3 虚拟机源码自动安装原理

虚拟机源码自动安装由 rme_guest.cpp 及其关联文件组成，其基本原理是在生成虚拟机时向工程的文件列表中自动添加文件。每种架构和每种 IDE 对虚拟机的要求都可以不同。

8.4 GSC 移植的主要流程

在移植 GSC 到新架构时，首先要进行一系列检查工作，然后对 Plat、RME_Gen、RVM_Gen、Proc_Gen、Proj_Gen 四个类进行继承即可。

8.4.1 移植前的检查工作

8.4.1.1 架构本身的特性

本编译器要求架构必须是 32 位及以上的、具有内存保护单元的微控制器，且内置程序存储器或者可以外扩能直接执行的程序存储器。绝大多数微控制器均内置了 Flash 作为其程序存储器，部分未内置程序存储器的单片机能够外扩并行或串行 Nor Flash。

8.4.1.2 集成开发环境的特性

此微控制器的集成开发环境工程文件格式必须是文本格式或者可编辑的二进制格式，而且工具链的链接器脚本必须允许用户手动设置。

8.4.1.3 RME 内核

RME 内核本身必须在该架构上移植完成。关于具体的移植方法请查看 RME 的相关文档。

8.4.1.4 RVM 用户态库

RVM 用户态库也必须在该架构上移植完成。关于具体的移植方法请查看 RVM 的相关文档。

8.4.2 对 Plat 类进行继承

Plat 类负责基本的内核对象大小信息计算、内存对齐和页表分配。

8.4.2.1 子类的构造函数

在继承时，子类的构造函数应当有两个参数，分别是 std::unique_ptr<class Proj>& Proj 和 std::unique_ptr<class Chip>& Chip。此外，在继承后，应当对 Plat 的构造函数传入四个参数。这四个参数的相关信息如下。

表 8-1 Plat 的构造函数

原型	Plat::Plat(ptr_t Word_Bits, ptr_t Init_Num_Ord, ptr_t Thd_Size, ptr_t Inv_Size)
意义	初始化各个架构均具备的必要参数。
返回值	无。
参数	ptr_t Word_Bits 处理器的位数。32 位处理器就填写 32，,64 位处理器就填写 64，依此类推。
	ptr_t Init_Num_Ord 该架构在启动时顶层页表的数量级数。这个值必须和内核启动代码中创建顶层页表时所用的数量级数一致。
	ptr_t Thd_Size 线程内核对象的原始大小，单位为字节。
	ptr_t Thd_Size

	迁移调用内核对象的原始大小，单位为字节。

在构造函数中，除了要对子类内部的信息进行架构相关的必要初始化之外，还要将工程 XML 和芯片 XML 中的架构特有选项进行解析，并且填充进子类内部的数据结构，以方便最终生成工程。

8.4.2.2 子类应实现的虚函数：Raw_Pgtbl_Size

本函数负责计算页表内核对象的原始大小，其单位为字节。

表 8-2 Raw_Pgtbl_Size 的原型

原型	ptr_t Raw_Pgtbl_Size(ptr_t Num_Order, ptr_t Is_Top)
意义	根据数量级数和顶层标志计算页表内核对象的初始大小。
返回值	无。
参数	ptr_t Num_Order 页表的数量级数。
	ptr_t Is_Top 顶层标志，如果为 0 则不是顶层页表，为 1 则是顶层页表。

8.4.2.3 子类应实现的虚函数：Align_Mem

本函数负责对齐工程 XML 文件中提到的各段内存的起始地址及大小。具体的对齐方式随着架构的不同而不同。

表 8-3 Align_Mem 的原型

原型	void Align_Mem(std::unique_ptr<class Proj>& Proj)
意义	根据架构需求对齐工程中的各内存段。
返回值	无。
参数	std::unique_ptr<class Proj>& Proj 指向整个工程的指针。工程中涉及到的所有内存段信息均可在此对齐或检查。

8.4.2.4 子类应实现的虚函数：Alloc_Pgtbl

本函数负责根据进程的地址空间布局分配它们的页表。具体的分配方式也是根据各个架构的不同而不同的。分配过后，各个 Proc 对象的 Pgtbl 都应当被赋值，而各个 Pgtbl 对象则被组织成一棵树，其中母页表中含有指向子页表的指针。具体如何实现该函数的内部逻辑请参见 ARMv7-M 的例子。

表 8-4 Alloc_Pgtbl 的原型

原型	void Alloc_Pgtbl(std::unique_ptr<class Proj>& Proj, std::unique_ptr<class Chip>& Chip)
意义	生成各个进程的页表。
返回值	无。
参数	std::unique_ptr<class Proj>& Proj 指向整个工程的指针。可从该指针遍历各个进程。
	std::unique_ptr<class Chip>& Chip 指向芯片信息的指针。可从该指针中提取任何芯片信息。

8.4.3 对 RME_Gen 类进行继承

RME_Gen 类负责生成内核本身的工程。

8.4.3.1 子类的构造函数

它的构造函数不应被显式声明。

8.4.3.2 子类应实现的虚函数：Plat_Gen

该函数没有参数，也没有返回值。它生成最终的内核本身的工程，其中包括修改芯片配置头文件、选择合适于工具链的汇编代码、生成链接器脚本和生成 IDE 工程文件。

8.4.4 对 RVM_Gen 类进行继承

RVM_Gen 类负责生成用户态库本身的工程。

8.4.4.1 子类的构造函数

它的构造函数不应被显式声明。

8.4.4.2 子类应实现的虚函数：Plat_Gen

该函数没有参数，也没有返回值。它生成最终的用户态库本身的工程，其中包括修改芯片配置头文件、选择合适于工具链的汇编代码、生成链接器脚本和生成 IDE 工程文件。

8.4.5 对 Proc_Gen 类进行继承

Proc_Gen 类负责生成各个原生进程和虚拟机的工程。

8.4.5.1 子类的构造函数

它的构造函数不应被显式声明。

8.4.5.2 子类应实现的虚函数：Plat_Gen

该函数没有参数，也没有返回值。它生成最终的各个原生进程和虚拟机本身的工程，其中包括修改芯片配置头文件[1]、选择合适于工具链的汇编代码、生成各个线程入口的汇编函数桩、生成链接器脚本和生成 IDE 工程文件。

8.4.6 移植 Proj_Gen 类

Proj_Gen 类负责生成包含各个工程的整个工作空间。

8.4.6.1 子类的构造函数

它的构造函数不应被显式声明。

8.4.6.2 子类应实现的虚函数：Plat_Gen

该函数没有参数，也没有返回值。它只有一个随着架构和 IDE 不同而不同的步骤，也即生成包含所有工程的工作空间。

8.4.7 修改 rme_mcu.c 中的 Main::Parse 函数

Main::Parse 类函数负责解析工程 XML 文件和芯片 XML 文件，并且根据架构选择合适的后端进行初始化。当新的架构被添加时，需要在这里加上相应的判断语句以选择该新架构。

8.5 本章参考文献

无

[1] 此处该步骤可能存在也可能不存在

第 9 章　虚拟机客户操作系统实现规范

9.1　客户操作系统规范概述

GSC 编译器和 RVM 用户态库支持虚拟化其他小型 RTOS，但它们必须符合一定的接口规范。本章节将讲解那些小型 RTOS 中常见的操作，并对转换这些操作到 RVM 超级调用提供帮助。本章所指的小型 RTOS 是指那些不具备独立保护域的 RTOS，也即所有线程均运行于一个地址空间的 RTOS。

本章的预定读者是那些需要将 RTOS 移植到 RVM 的用户。RTOS 本身的用户可以直接跳过本章。

9.2　RTOS 移植的主要流程

小型 RTOS 一般由以下八个部分组成：软硬件初始化代码，调度器中断向量，定时器中断向量，其它中断向量，开关中断 API，锁调度器 API，线程间通信 API 以及其它功能 API。这八个部分都需要被移植，下面我们将针对这八个部分进行一一说明。

9.2.1　软硬件初始化代码

软硬件初始化代码在 RTOS 启动时被调用。硬件初始化代码一般会将处理器设置到正确的工作频率，初始化内存控制器、定时器和中断管理组件等。软件初始化代码则将 RTOS 内部的数据结构初始化。

初始化处理器的硬件初始化代码在移植到 RVM 时可以完全删除，因为这部分初始化工作已经由 RME 代替了。如果需要初始化额外的外设[1]，那么需要用户手动添加这些外设所在的内存段到虚拟机地址空间中。

9.2.2　调度器中断向量

调度器中断向量是小型 RTOS 的核心。它一般的流程如下：首先将所有寄存器压栈，然后运行调度器选择最高优先级的线程，切换当前栈指针到该选定线程的栈，并从该栈中弹出所有寄存器。在有 FPU 存在的情况下，可能还要判断 FPU 是否被使用，如果 FPU 被使用，那么在压栈弹栈的流程中还要增加对 FPU 寄存器的保存和恢复。

在这个流程中，寄存器压栈和弹栈的操作要进行重写。原来的代码一般是使用汇编编写的；在这里我们需要使用 C 语言重写它。在压栈时，从寄存器组中取出堆栈指针的值，然后将其递减，并且在其指向的位置填入各个原先需要压栈的寄存器的值。在弹栈时按照相反方向进行操作即可。

由于虚拟机和 RVM 之间使用结构体 struct RVM_Param_Area 的 struct RVM_Param User 成员来传递由用户线程发起的超级调用的参数和返回值，这个成员的内容物也要被压栈弹栈，以确保在线程之间不会因为数据冒险而引起混乱。具体的做法是，在完成常规寄存器压栈之后，将此成员中的数据项依次入栈，在弹栈时则先将此成员中的数据项弹出，再进行常规寄存器的弹栈。

涉及到线程栈初始化和第一个线程的第一次启动的相关代码也可能要进行一些修改以适应到 RVM 的移植。关于本流程的具体处理请参见 RMP 在 ARMv7-M 上的 RVM 移植。

---

[1] 一般而言这种状况并不存在，因为 RTOS 一般不初始化具体外设

9.2.3 定时器中断向量

定时器中断向量负责小型 RTOS 的计时。该功能通常而言非常简单，均为直接调用一个计时器循环函数，若该函数中检测到现有线程的时间片已经用尽，则进行相应的处理，然后使用软中断触发调度器中断向量进行调度。

这个中断向量不需要进行任何重写，只需要使用函数 RVM_Virt_Reg_Timer 登记原有中断向量函数到虚拟机的时钟中断向量即可。关于此函数的具体信息请参看 RVM 的使用手册。

9.2.4 其他中断向量

其他中断向量负责 RTOS 对外界刺激的响应。这些中断向量通常而言也非常简单，一般仅负责调用可在中断向量中调用的那些线程间通信函数，然后就会退出。

这些函数中的涉及硬件操作的部分应当删除，因为这部分已经由 RME 代替了。其他部分则无需修改，只需使用函数 RVM_Virt_Reg_Vect 将其注册到某个中断号即可。关于此函数的具体信息请参看 RVM 的使用手册。

9.2.5 开关中断 API

开关中断之函数分别使用 RVM_Hyp_Ena_Int 和 RVM_Hyp_Dis_Int 超级调用进行原位替换即可。其他部分无需修改。这两个函数会完全禁止虚拟机对任何中断的响应，也即虚拟机监视器将不会对虚拟机发送任何中断。关于这两个函数的具体信息请参看 RVM 的使用手册。

9.2.6 锁调度器 API

RTOS 在锁调度器时，通常会调用一个部分关中断函数，该函数使得那些可能调用线程间通信函数的中断被屏蔽，但并非所有中断均被屏蔽。这个部分关中断函数可以用 RVM_Virt_Mask_Int 代替，其作用是使得虚拟机仍可以从虚拟机监视器接收中断，但虚拟机会暂时将这些中断悬起而不做响应。由于该函数不是超级调用而仅仅是虚拟机内部的中断屏蔽，因此效率比 RVM_Hyp_Dis_Int 高很多[1]。等到解除调度器锁时，可以使用函数 RVM_Virt_Unmask_Int 解除屏蔽，此时虚拟机会开始一一处理那些悬起的中断。

9.2.7 线程间通信 API

线程间通信 API 可以分为两类。一类是只能在中断向量中调用的线程间通信 API，它们往往有特殊的后缀标识这一点。另一类是只能在线程中调用的线程间通信 API，它们往往没有后缀标识。这两类 API 均不需要改动即可使用。

9.2.8 其他功能 API

[1] 虽然在这里使用后者也是没有任何问题的

其他功能 API 负责内存分配等等方面，这些 API 也不需要修改即可使用。

9.3 本章参考文献

无

第 10 章 附录

10.1 架构相关部分的错误码说明

RME 可 GSC 会针对每个架构输出不同的错误码，这些错误码均以 A 开头。关于这些错误码，需要查看具体架构的相关手册以阐明其作用。

10.2 GSC 的已知问题和制约

GSC 存在一些已知的限制和制约。这些限制和制约将在下面一一说明。

10.2.1 无多核支持

GSC 暂时不支持多核微控制器。这是因为微控制器和微处理器的产品逻辑完全不同。微控制器多核的最大意义在于实时性和可靠性：不同的核心的功能在系统设计时静态分配，一旦一个核发生任何软硬件问题，哪怕甚至是硬件 BUG 导致的永久死锁或核心损坏，其他核心可以不受影响继续运行。为此，多核微控制器在的各核心差异往往很大，某些 CPU 支持 FPU，另一些则不支持；某些 CPU 支持 MPU，另一些则不支持。将 RME 和 RVM 移植到这些平台上是有可能的，但其易用性会受到很多损失。如果你对这样的应用有兴趣，请联系我们。

10.3 本章参考文献

无