# RealView™ ARMulator® ISS

**1.3 版**

## 用户指南

ARM®


ARM DUI 0207ASC-00

# RealView ARMulator ISS
## 用户指南



### 版本信息

本书进行了以下更改。

**更改记录**

| 日期 | 发行号 | 更改 |
|---|---|---|
| 2002 年 8 月 | A | 1.3 版 |

### 所有权通告

# 目录
# RealView ARMulator ISS 用户指南

  

# 前言

此前言对 RealView™ ARMulator® 指令集模拟器 (RealView ARMulator ISS) 目标进行了介绍。它包含以下内容：

## 关于本书

本书提供了有关 RealView ARMulator ISS （即 ARM 处理器模拟器）的参考信息。

### 适用对象

本书专为使用远程调试接口 1.5.1 兼容调试器的所有开发人员而写。前提是假定您是一位有经验的软件开发人员，并且您熟悉 *ADS v1.2 Getting Started Guide* 或 *RealView Compilation Tools v1.2 Getting Started Guide* 中描述的 ARM 开发工具。

### 使用本书

本书由以下章节组成：

**第 1 章*简介***

阅读本章以简单了解本书内容以及 RealView ARMulator ISS 的概要描述。

**第 2 章 *ARMulator 基础知识***

阅读本章以简单了解 RealView ARMulator ISS，即 ARM 指令集模拟器。

**第 3 章*编写 ARMulator 模型***

阅读本章以帮助您记录您对 RealView ARMulator ISS 的扩展和修改。

**第 4 章 *ARMulator 参考***

本章提供帮助您使用 RealView ARMulator ISS 的更多详细内容。

## 印刷惯例

本书使用了以下印刷惯例：

*斜体* 突出显示重要注释，介绍特殊术语，表示内部交叉链接和引用。

**粗体** 突出显示界面要素，如菜单名称。必要时也用于强调说明列表中的重点以及 ARM 处理器信号名称。

等宽 表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

等宽 表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

*等宽斜体*

表示此处的命令和函数变量可用特定值代替。

**等宽粗体**

表示使用示例代码以外的语言关键字。

## 其它读物

本部分列出了 ARM Limited 和第三方发布的、可提供有关 ARM 系列处理器开发代码附加信息的相关读物。

ARM 将定期对其文档进行更新和更正。有关最新勘误表、附录以及 ARM 常见问题，请访问 http://www.arm.com。

### ARM 书刊

本书包含特定 RealView ARMulator ISS 的信息。有关将 RealView ARMulator ISS 与调试器配合使用的信息，请参阅调试器说明文档。

# 反馈

ARM Limited 乐于收到有关 RealView ARMulator ISS 及其文档的任何反馈。

## 关于 RealView ARMulator ISS 的反馈

如果您对 RealView ARMulator ISS 有任何疑问，请与供应商联系。为便于供应商提供快捷、有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用的版本的详情
- 您运行的平台的详情，如硬件平台、操作系统类型和版本
- 再现问题发生的独立样本代码
- 您期望发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 解释问题的示例输出
- 工具的版本字符串，包括版本号和日期。

## 关于本书的反馈

如果您对本书有任何问题，请发送电子邮件到 errata@arm.com，并提供：

- 文档标题
- 文档编号
- 您有意见的页码
- 您的意见的简单说明。

我们还欢迎您对新增和改进之处提出一般建议。

  

# 第 1 章
# 简介

本章介绍了 *RealView ARMulator 指令集模拟器* (RealView ARMulator ISS) 1.3 版提供的调试支持工具。本章包含以下内容：

## 1.1 RealView ARMulator ISS 概述

您可以使用远程调试接口 (RDI) 1.5.1 兼容调试器调试您的原型软件。调试器在主计算机上运行，并连接至运行原型软件的目标系统。

您的目标系统可以是以下任何一个：

- 软件模拟器， RealView ARMulator ISS，模拟 ARM 硬件
- ARM 评估或开发板
- 基于 ARM 的第三方开发板
- 您自己设计的基于 ARM 的硬件。

本文档只介绍 RealView ARMulator ISS。有关其它目标系统的详情，请参阅该目标的说明文档。

### 1.1.1 什么是 RealView ARMulator ISS?

RealView ARMulator ISS （后文称为 ARMulator）随 ARM 调试器提供，是一个独立产品。 ARMulator 在与调试器相同的主计算机上运行，并且包括与调试器通信的工具。

ARMulator 是一个*指令集模拟器* (ISS)。它与存储器系统和外围设备一起，可以模拟指令集和 ARM 处理器的结构。您还可以将其扩展至模拟其它外围设备和自定义存储器系统 （请参阅第 3 章*编写 ARMulator 模型* ）。

您可以将 ARMulator 用于软件开发，也可以将其作为以 ARM 为目标的软件的基准程序。它可以模拟指令集和计数周期。作为基准程序的精度会受到限制。请参阅第 2-2 页*精度*。

### 1.1.2 Semihosting

您可以使用主计算机上的 I/O 设备，而非目标系统提供的设备。这就是所谓的 *semihosting* （有关详情，请参阅 *RealView Compilation Tools v1.2 Compilers and Libraries Guide*）。

缺省情况下， ARM C 和 C++ 代码使用 semihosting 设备。

要从汇编代码访问 semihosting 设备，请使用 semihosting *软件中断* (SWI)。ARMulator 中止 semihosting SWI 并从主计算机请求服务。

  

# 第 2 章
# ARMulator 基础知识

本章描述 ARMulator，即提供 ARM 处理器软件模拟的程序集成。它包含以下内容：

## 2.1 关于 ARMulator

ARMulator 是一种指令集模拟器。它可以模拟指令集和各种 ARM 处理器结构。要在 ARMulator 上运行软件，请通过 RDI 1.5.1 兼容调试器进行访问。

ARMulator 适用于软件开发，并可作为以 ARM 为目标的软件的基准程序。它可以模拟指令集和计数周期 （请参阅第 2-14 页 *ARMulator 周期类型*）。作为基准程序的精度和周期计数精度会受到一定限制，请参阅*精度*。

ARMulator 可提供使全部 C 或 C++ 程序在模拟系统上运行所需的全部工具。有关 ARMulator 支持的 C 库 semihosting SWI 的信息，另请参阅 *RealView Compilation Tools v1.2 Compilers and Libraries Guide*。

### 2.1.1 精度

ARMulator 不能达到 100% 的精度，因为它并不是基于真实处理器的。一般而言，不是很复杂、非高速缓存的 ARM 处理器内核模型精度较高，而带高速缓存的那些内核模型可能与实际硬件的精度不完全相同。

ARMulator 适合用作系统设计的软件开发工具，但如果需要 100% 的精度，则必须使用硬件模型。

在以下情况下，您可以使用 ARMulator 作为基准程序：

- 您正在建模的内核没有高速缓存
- 您只需要近似比较。

ARMulator 不会在高速缓存内核上建立 Asynchronous Mode 模型。如果您在 CP15 中设置了控制位以便指定 Asynchronous Mode，ARMulator 将会发出警告：

```
Set to Asynch mode, WARNING this is not supported
```

您可以继续调试，但 ARMulator 将完全按照 Synchronous Mode 的方式运行。

  

## 2.2 ARMulator 组件

ARMulator 由一系列模块组成，可作为*动态链接库*（Windows 的 `.dll` 文件）或*共享对象*（Linux 或 Solaris 的 `.so` 文件或所有平台上的 `.sdi` 文件）来执行。

主模块是：

- ARM 处理器内核模型
- 处理器所用存储器的模型

这些部件中的每一个均有备选的预定义模块。您可以选择要使用的处理器和存储器模型组合。

其中一个预定义的存储器模型，`mapfile`，允许您详细指定模拟的存储器系统。`mapfile` 允许您指定窄内存和等待状态（请参阅第 2-28 页*使用映射文件进行存储器建模*）。

另外，还有其它您可以使用的预定义模块：

- 建立附加硬件模型，例如协处理器或外围设备
- 建立预安装的软件模型，例如 C 库、semihosting SWI 处理程序或操作系统
- 抽取调试或基准信息（请参阅第 2-5 页*跟踪器*和第 2-12 页 *Profiler*）。

您可以使用不同的预定义模块组合和不同的存储器映射（请参阅第 2-4 页*配置 ARMulator*）。

如果提供的模块不符合您的要求，您也可以写自己的模块或者编辑预定义模块的副本。例如：

- 建立不同外围设备、协处理器或操作系统的模型
- 建立不同存储器系统的模型
- 提供其它调试或基准信息。

一些模块的源代码已提供。您可以这些模块为示例，来写自己的模块（请参阅第 3 章*编写 ARMulator 模型*）。

### 2.2.1 配置 ARMulator

您可以从 RDI 1.5.1 兼容调试器配置一些 ARMulator 的详细资料。

要进行其它配置调整，您必须编辑 .ami 文件副本。随 ARMulator 提供以下 .ami 文件：

- bustypes.ami
- default.ami
- example1.ami
- peripherals.ami
- processors.ami
- vfp.ami。

这些文件位于：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

在此路径中：

- *platform* 指：
  - Windows：win_32-pentium
  - Linux：linux-pentium
  - Solaris：solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

如果要写自己的任何 ARMulator 模型，请先生成其它的 .ami 文件，以允许配置您的模型。有关如何操作的详情，请参阅第 4-64 页 *ARMulator 配置文件*。

调试器启动 ARMulator 后， ARMulator 会读取环境变量 ARMCONF 中定义的所有路径上的 .ami 文件。初始设置为 ARMulator 目录：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

以下小节依次描述每个预定义模块以及如何配置这些模块。

**注**

如果 .ami 文件中的配置设置与您在调试器中的设置发生冲突，则将优先采用调试器设置。

  

## 2.3 跟踪器

您可以使用跟踪器跟踪指令、存储器访问和事件。由配置文件 peripherals.ami 来控制跟踪内容 （请参阅第 4-64 页 *ARMulator 配置文件*）。

本节包括以下小节：

- *调试器跟踪支持*
- 第 2-6 页*解释跟踪文件输出*
- 第 2-10 页*配置跟踪器*。

### 2.3.1 调试器跟踪支持

如果您的调试器不直接支持跟踪功能，跟踪器将使用 RDI 记录级别变量的第 4 位来启用或禁用跟踪功能。记录级别变量为以下其中之一：

- $rdi_log
- RealView 调试器中的 @mdebug_rdi_log
- 调试器当量。

有关通过 ARMulator 使用跟踪功能的详情，请参阅调试器文档。

**注**

跟踪器由调试器开启或关闭，但由 .ami 文件控制。

### 2.3.2 解释跟踪文件输出

本节描述如何解释从跟踪器输出的信息。

#### 跟踪文件示例

以下示例显示了部分跟踪文件：

```
Date: Thu Aug  9 16:41:36 2001
Source: Armul
Options: Trace Instructions  (Disassemble)  Trace Memory Cycles

BNR4O___ A0000000 00000C1E
BNR8O___ 00008000 E28F8090 E898000F
BSR8O___ 00008008 E0800008 E0811008
BSR8O___ 00008010 E0822008 E0833008
BSR8O___ 00008018 E240B001 E242C001
MNR4O___ 00008000 E28F8090
IT 00008000 e28f8090 ADD      r8,pc,#0x90 ; #0x8098
MNR4O___ 00008004 E898000F
IT 00008004 e898000f LDMIA    r8,{r0-r3}
BNR4O___ A0000000 00000C1E
BNR8O___ 00008098 00007804 00007828
BSR8O___ 00008080 10844009 E3C44003
BSR8O___ 00008088 E2555004 24847004
BSR8O___ 00008090 8AFFFFFC EAFFFFF2
MNR8____ 00008098 00007804 00007828
BNR8O___ 000080A0 00007828 00007840
BSR8O___ 000080A8 E3A00840 E1A0F00E
BSR8O___ 000080B0 E92D400C E28F0014
BSR8O___ 000080B8 E5901000 E5900004
MNR8____ 000080A0 00007828 00007840
MNR4O___ 00008008 E0800008
IT 00008008 e0800008 ADD      r0,r0,r8
MNR4O___ 0000800C E0811008
IT 0000800C e0811008 ADD      r1,r1,r8
MNR4O___ 00008010 E0822008
```

以下类型的行可以出现在跟踪文件中：

- 第 2-7 页*跟踪存储器 （M 行）*
- 第 2-8 页*跟踪指令 （I 行）*
- 第 2-8 页*跟踪事件 （E 行）*
- 第 2-9 页*跟踪寄存器 （R 行）*
- 第 2-9 页*跟踪总线 （B 行）*。

### 跟踪存储器 （M 行）

M 行表示：

- 存储器访问，用于不带片上存储器的内核
- 片内存储器访问，用于带片内存储器的内核。

它们具有以下通用存储器访问格式：

```
M<type><rw><size>[O][L][S] <address> <data>
```

其中：

`<type>` 表示周期类型：
- `S` 连续
- `N` 不连续。

`<rw>` 表示读或写操作：
- `R` 读
- `W` 写。

`<size>` 表示存储器访问的大小：
- `4` 字 （32 位）
- `2` 半字 （16 位）
- `1` 字节 （8 位）。

`O` 表示操作码获取 （指令获取）。

`L` 表示锁定的访问 （`SWP` 指令）。

`S` 表示推测性的指令获取。

`D` 表示 ARM9TDMI™ 数据接口的 **DMORE** 信号是 HIGH。

`<address>` 按十六进制格式给出的地址，例如 `00008008`。

`<data>` 可显示为以下一种模式：
- *`value`* 给定读/写值，例如 `EB00000C`
- `(wait)` 表示 **nWAIT** 为 LOW （相对于插入等待状态而言）
- `(abort)` 表示 **ABORT** 为 HIGH （相对于中止访问而言）。

跟踪存储器行也可以有以下格式：

`MI` 用于空闲周期

`MC` 用于协处理器周期

`MIO` 用于 Harvard 体系结构处理器（如 ARM9TDMI™）指令总线上的空闲周期。

### 跟踪指令 （I 行）

跟踪指令 (I) 行的格式如下：

```
[ IT | IS ] <instr_addr> <opcode> [<disassembly>]
```

例如：

```
IT 00008044 e04ec00f SUB      r12,r14,pc
```

其中：

| | |
|---|---|
| IT | 表示指令被执行。 |
| IS | 表示指令被跳过 （几乎所有 ARM 指令都能有条件执行）。 |
| <instr_addr> | 按十六进制格式显示指令地址，例如 00008044。 |
| <opcode> | 按十六进制格式给出操作码，例如 e04ec00f。 |
| <disassembly> | 表示反汇编 （如果指令已被执行则大写），例如 SUB r12,r14,pc。此为可选项，可通过在 peripherals.ami 中设置 Disassemble=True 启用。 |

Thumb 代码中带链接的分支显示为两个条目，第一个标记为：

```
1st instr of BL pair.
```

### 跟踪事件 （E 行）

事件 (E) 行的格式如下：

```
E <word1> <word2> <event_number>
```

例如：

```
E 00000048 00000000 10005
```

其中：

| | |
|---|---|
| <word1> | 给出一对字的第一个，例如 pc 值。 |
| <word2> | 给出一对字的第二个，例如中断地址。 |
| <event_number> | 给出事件编号，例如 0x10005。这是 MMU Event_ITLBWalk。事件在第 4-29 页*事件* 中有描述。 |

### 跟踪寄存器 （R 行）

事件 (R) 行的格式如下：

```
R <register>=<newvalue>[,<anotherregister>=<newvalue>[...]]
```

例如：

```
R r14=20000060, cpsr=200000d3
```

其中：

`<register>` 当前指令的结果寄存器 , 保存指令执行结果。

`<newvalue>` 是 `<register>` 中的新内容。

### 跟踪总线 （B 行）

总线 (B) 行的格式与 M 行的格式相同。 B 行表示片外存储器访问。

### 2.3.3 配置跟踪器

ARMulator 外围设备配置文件 (peripherals.ami) 中有关于跟踪器的部分：

```
{ Default_Tracer=Tracer
;; Output options - can be plaintext to file, binary to file or to RDI log
;; window. (Checked in the order RDILog, File, BinFile.)
;VERBOSE=True
;RDILog=True
RDILog=False
File=armul.trc
BinFile=armul.trc
;; Tracer options - what to trace
TraceInstructions=True
TraceRegisters=False
OpcodeFetch=True
;;Normally True is useful, but sometimes it's too expensive.
TraceMemory=True
;TraceMemory=False
TraceIdle=True
TraceNonAccounted=False
TraceEvents=False
;;If there is a non-core bus, do we trace it (as well).
TraceBus=True
;; Flags - disassemble instructions; start up with tracing enabled;
Disassemble=True
TraceEIS=False
StartOn=False
}
```

其中：

`RDILog` 指示跟踪器输出至调试器的 RDI 日志窗口。

`File` 定义写跟踪信息的文件。另外，您还可以使用 `BinFile` 按二进制格式存储数据。

其它选项控制跟踪内容：

`TraceInstructions` 跟踪指令。

`TraceRegisters` 跟踪寄存器。

`OpcodeFetch` 跟踪指令获取存储器访问。

`TraceMemory` 跟踪存储器访问。

`TraceIdle` 跟踪空闲周期。

`TraceNonAccounted` 跟踪未统计的 RDI 存储器。即那些由调试器产生的访问。

  

| | |
|---|---|
| TraceEvents | 跟踪事件。有关更多信息，请参阅下面的*跟踪事件*。 |
| TraceBus | 可能是：<br>TRUE 总线 （已跟踪片外访问）<br>FALSE 内核 （未跟踪片外访问）。 |
| Disassemble | 反汇编指令。如果您启用反汇编，则模拟速度会较慢。 |
| TraceEIS | 如果设为 TRUE，则将输出格式与其它模拟器兼容。这可使工具对跟踪进行比较。 |
| StartOn | 指示 ARMulator 在执行一开始即进行跟踪。 |

### 其它跟踪控制

您还可以通过以下指令控制跟踪：

Range=*low address*,*high address*

只在指定的地址范围内执行跟踪。

Sample=*n* 只将每 *n* 个跟踪条目发送至跟踪文件。

### 跟踪事件

跟踪事件时，您可以选择要跟踪的事件，方法如下：

EventMask=*mask*,*value*

只跟踪那些编号被 *mask* 屏蔽后 （位与）与 *value* 相等的事件。

Event=*number* 只跟踪 *number*。（这相当于 EventMask=0xFFFFFFFF,*number*。）

例如，以下指令只跟踪 MMU/高速缓存事件：

```
EventMask=0xFFFF0000,0x00010000
```

有关更多信息，请参阅第 4-29 页*事件*。

  

## 2.4 Profiler

Profiler 由调试器控制。有关详情，请参阅调试器文档。

除配置程序执行时间外， Profiler 还允许您使用配置机制对事件（如高速缓存错误）进行配置。

从调试器打开配置功能时，您应指定一个数字 *n*，以便控制配置频率。有关详情，请参阅第 2-13 页*配置 Profiler*。

Profiler 可以配置 C 和汇编语言函数。要配置汇编语言函数，您必须将函数用 `FUNCTION` 和 `ENDFUNC` 指令进行标记。有关详情，请参阅 *RealView Compilation Tools v1.2 Assembler Guide*。

  

### 2.4.1 配置 Profiler

ARMulator 外围设备配置文件 peripherals.ami 中有关于 Profiler 的部分：

```
{ Default_Profiler=Profiler
;VERBOSE=False
;; For example - to profile the PC value when cache misses happen, set:
;Type=Event
;Event=0x00010001
;EventWord=pc
Type=MICROSECOND

;;Alternatives for Type are
;;  Event, Cycle, Microsecond.
;;If type is Event then alternatives for EventWord are
;;  Word1,Word2,PC.
}
```

本部分中的每一行均为一个注释，因此 ARMulator 将执行其缺省配置。缺省设置是每隔 100 微秒进行一次配置取样。有关更多信息，请参阅调试器文档。

如果本部分未加注释，则会配置数据高速缓存出错。有关更多信息，请参阅第 4-29 页*事件*。

Type 条目控制配置间隔：

Type=Microsecond　指示 Profiler 每隔 *n* 微秒进行一次取样。此为缺省值。

Type=Cycle　指示 Profiler 每隔 *n* 个指令进行一次取样，并记录自上次取样后的内存周期数量。

Type=Event　指示 Profiler 每隔一定的相关事件即进行配置，请参阅第 4-29 页*事件*。 *n* 被忽略。

也允许 EventMask=*mask*,*value*。请参阅第 2-5 页*跟踪器*。

## 2.5 ARMulator 周期类型

除模拟 ARM 内核上的指令执行之外， ARMulator 还可以计算总线和处理器周期。从调试器您可以以 `$statistics` （或调试器当量）访问这些计数。

本节描述所计算的各种周期类型的含义。它包含以下部分：



### 2.5.1 RealView Debugger 中的周期计数器

ARMulator 可以按 RDI 中周期计数器调用所报告的那样增加周期计数器。有关详情，请参阅第 4-35 页*未知的 RDI 信息处理程序*。如果 RDI 周期计数器名称是 *X*，则符号 `@rdi_X` 即为周期计数器名称，其中 *X* 可将所有非字母数字字符转换为带下划线字符。这些计数器出现在寄存器选项卡中。

然而， RealView Debugger 跟踪不使用 `$statistics`， RealView Debugger 使用 `@cycle_count`。这由监测内核和高速缓存或存储器之间总线的存储器回调服务提供，并提供合理的时间定义。

**注**

尽管 `@cycle_count` 可能与周期计数器 （如 `Total`）不同，但这种差别并不重要。

内核内部和外部之间的总线比率意味着 `@cycle_count` 是其它时钟值的数倍。

### 2.5.2 未高速缓存的 von Neumann 内核

表 2-1 显示了未高速缓存的 von Neumann 内核的周期类型的含义。例如 ARM7TDMI®，即是一个未带高速缓存的 von Neumann 型内核。

**表 2-1 未带缓存的 von Neumann 架构内核的周期类型**

| 周期类型 | SEQ 信号 | nMREQ 信号 | 含义 |
|---|---|---|---|
| S_Cycles | 1 | 1 | 连续周期。有关详情，请参阅*连续周期*。 |
| N_Cycles | 0 | 1 | 非连续周期。 CPU 从与前述周期所用地址无关的地址请求转移或者转移到该地址。 |
| I_Cycles | 1 | 0 | 内部周期。 CPU 不需要请求转移，因为它正在执行内部函数。 |
| C_Cycles | 0 | 0 | 协处理器周期。 |
| Total | - | - | S_Cycles、 N_Cycles、 I_Cycles、 C_Cycles 和 Waits 的总和。 |
| IS | - | - | 合并的 I-S 周期。有关详情，请参阅*合并的 I-S 周期*。 |

#### 连续周期

CPU 可从以下地址请求转移或转移到以下地址：

- 与前述周期访问地址相同的地址
- 在前述周期访问地址一字之后的地址
- 在前述周期访问地址一个半字之后的地址 （仅限于 Thumb 指令获取）。

#### 合并的 I-S 周期

内存控制器可以在 I-Cycle 期间推测性地开始解码地址。如果 I_Cycle 后跟随一个 S_Cycle，则内存控制器可以比其它方式更早启动。此周期的计时功能取决于内存控制器的实施。

### 2.5.3 未带缓存的 Harvard 架构内核

表 2-2 显示了未带高速缓存的 Harvard 型内核的周期类型的含义。例如，ARM9TDMI 即是一个没有高速缓存的 Harvard 内核。

**表 2-2 未带高速缓存的 Harvard 架构内核的周期类型含义**

| 周期类型 | 指令总线 | 数据总线 | 含义 |
|---|---|---|---|
| 内核周期 | - | - | 内核时钟运行的总数。这包括由于互锁和指令需要多个周期而导致的管道中断。 |
| ID_Cycles | 活动 | 活动 | - |
| I_Cycles | 活动 | 空闲 | - |
| 空闲周期 | 空闲 | 空闲 | - |
| D_Cycles | 空闲 | 活动 | - |
| Total | - | - | 内核周期、 ID_Cycles、 I_Cycles、 Idle_Cycles、 D_Cycles 和 Waits 的总和。 |

### 2.5.4 带 MMU 或 PU 和 AMBA ASB 接口的高速缓存内核

表 2-3 显示了带 AMBA ASB 接口的高速缓存内核的总线周期类型的含义。有关这些内核的其它周期类型，请参阅第 2-17 页*带高速缓存内核的内部周期类型*。

例如， ARM920T 即是带 MMU 和高速缓存的内核。 ARM940T 是带 PU 和高速缓存的内核示例。

**表 2-3 带 AMBA ASB 接口的高速缓存内核的周期类型的含义**

| 周期类型 | 含义 |
|---|---|
| A_Cycles | 地址预测。无数据转移。在 $statistics 中作为 I_Cycles （或调试器当量）列出。 |
| S_Cycles | 从当前地址连续传输数据。 |

这些内核没有 N_Cycles。非连续访问使用 A_Cycle，后面跟随 S_Cycle。这与合并的 I-S 周期相同。

### 2.5.5 带 MMU 或 PU 和 AMBA AHB 接口的高速缓存内核

表 2-4 说明了在*先进高速总线* (AHB) 上使用的传输类型。例如，ARM946E-S 即是带 AHB 接口的高速缓存内核。有关这些内核的其它周期类型，请参阅*带高速缓存内核的内部周期类型*。

**表 2-4 AMBA AHB 接口上的周期类型**

| 周期类型 | 含义 |
|---|---|
| IDLE | 总线主控器不想使用总线。从属器必须以 **HRESP** 上的零等待状态 **OKAY** 进行响应。 |
| BUSY | 总线主控器位于分段传输中间，但不能进行下一个连续访问。从属器必须以 **HRESP** 上的零等待状态 **OKAY** 进行响应。 |
| NON-SEQ | 分段传输或单个访问的开始。地址与上次访问的地址没有关联。 |
| SEQ | 继续进行分段传输。地址等于前一个地址加上数据长度。 |

### 2.5.6 带高速缓存内核的内部周期类型

表 2-5 显示了高速缓存内核的内部周期类型的含义。

**表 2-5 高速缓存内核的内部周期类型**

| 周期类型 | 含义 |
|---|---|
| F_Cycles | 快速时钟 (**FLCK**) 周期。这些是访问高速缓存的内部内核周期。对于非高速缓存的访问，由于内核时钟转换为总线时钟，因此 F_Cycles 不会增加。 |
| Core Cycles | 内核周期是指内核时钟的跳转。每次时钟跳转 Core Cycles 均会增加，无论内核是运行在 **FCLK** （高速缓存访问）还是总线时钟 （**BCLK**，非高速缓存访问）下。 |
| True Idle Cycles | 空闲周期并不是 I-S 合并周期的一部分。 |

**注**

如果您希望计算执行时间，请使用外部总线周期计数 （请参阅第 2-16 页*带 MMU 或 PU 和 AMBA ASB 接口的高速缓存内核*或*带 MMU 或 PU 和 AMBA AHB 接口的高速缓存内核*）。您不能使用 F_Cycles 计算执行时间，因为对于未经高速缓存的访问， F_Cycles 不会增加。

### 2.5.7 StrongARM1

表 2-6 显示了 StrongARM1 周期类型的含义。

**表 2-6 StrongARM 特定周期类型**

| 周期类型 | 含义 |
|---|---|
| Core_Idle | 未从指令高速缓存获取指令。未从数据高速缓存获取数据。 |
| Core_IOnly | 从指令高速缓存获取指令。未从数据高速缓存获取数据。 |
| Core_DOnly | 未从指令高速缓存获取指令。从数据高速缓存获取数据。 |
| Core_ID | 从指令高速缓存获取指令。从数据高速缓存获取数据。 |

### 2.5.8 内核相关的详细统计数据

在 `default.ami` 文件中有以下一行：

```
Counters=False
```

您可以将其更改为：

```
Counters=True
```

这样，ARMulator 会计算其它统计数据，如高速缓存命中数和高速缓存未命中数，并显示在 `$statistics`（或调试器当量）中。这些统计数据是特定内核相关的。

## 2.6 页表模块

本节包括以下小节：



### 2.6.1 页表模块概述

页表模块可使您在带*存储器管理单元* (MMU) 或*保护单元* (PU) 的系统模型上运行代码，而无需为 MMU 或 PU 写初始化代码。

**注**

此模块允许您调试代码或执行近似的基准。对于真实系统，您必须写初始化代码才能设置 MMU 或 PU。您可以通过禁用页表模块在 ARMulator 上调试初始化代码。

在带 MMU 的 ARM v4 和 v5 处理器架构模型上，页表模块可以设置页表并初始化 MMU。在带 PU 的处理器上，页表模块可以设置 PU。要控制是否包括页表模型，请在 ARMulator 配置文件 `default.ami` 中查找 `PAGETAB` 变量，并根据需要进行修改 （另请参阅本文件中的 `Pagetables=$PAGETAB`）：

```
{PAGETAB=Default_Pagetables
}
```

或

```
{PAGETAB=No_Pagetables
}
```

`peripherals.ami` 的 `Pagetables` 部分可控制页表内容以及高速缓存和 MMU 或 PU 的配置。要找到 `Pagetables` 部分，请查找以下行：

```
{Default_Pagetables=PageTables
```

有关本节所述的标记、控制寄存器和页表的完整详情，请参阅 *ARM Architecture Reference Manual* 或您模拟的处理器的技术参考手册。

### 2.6.2 控制 MMU 或 PU 和高速缓存

第一个标记集可以启用或禁用高速缓存和 MMU 或 PU 的功能：

```
MMU=Yes
AlignFaults=No
Cache=Yes
WriteBuffer=Yes
Prog32=Yes
Data32=Yes
LateAbort=Yes
BigEnd=No
BranchPredict=Yes
ICache=Yes
HighExceptionVectors=No
FastBus=No
```

每个标记对应系统控制寄存器中的一个位，即 CP15 中的 c1。

一些标记只适用于特定处理器。例如：

- `BranchPredict` 只适用于 ARM810™
- `ICache` 适用于 StrongARM®-110 和 ARM940T™ 处理器，但不适用于 ARM720 处理器。

这些标记会被其它处理器模型忽略。

**注**

有关支持的处理器详情，请参阅调试器文档。

`FastBus` 标记被一些诸如 ARM940T 之类的内核使用。请参阅适用于您的内核的技术参考手册。如果您的系统使用 FastBus Mode，请设置 `FastBus=Yes` 以便作为基准。如果设置 `FastBus=No`，则由于 `MCCFG` 因素的影响， ARMulator 会假定存储器时钟比内核时钟慢。 ARMulator 不会建立非同步模式的模型。

MMU 标记用于在带 PU 的处理器中启用 PU。

### 2.6.3 控制寄存器 2 和 3

以下选项只适用于带 MMU 的处理器：

```
PageTableBase=0xA0000000
DAC=0x00000001
```

它们控制：

- 转换表基址寄存器 （系统控制寄存器 2）
- 域访问控制寄存器 （系统控制寄存器 3）。

转换表基址寄存器中的地址必须与 16KB 界限相匹配。

### 2.6.4 存储区域

其它页表配置部分定义了一组存储区域。每个区域均有自己的属性。

缺省情况下， peripherals.ami 包含两个区域的说明：

```
{ Region[0]
VirtualBase=0
PhysicalBase=0
Size=4GB
Cacheable=No
Bufferable=No
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}

{ Region[1]
VirtualBase=0
PhysicalBase=0
Size=128Mb
Cacheable=Yes
Bufferable=Yes
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}
```

您可以按相同的通用格式添加更多区域：

Region[*n*] 为区域命名，以 Region[0] 开始。 *n* 是一个整数。

VirtualBase 只适用于带 MMU 的处理器。它给出了处理器虚拟地址空间中的区域基址。此地址必须与 1MB 边界对齐。它由 MMU 映射至 PhysicalBase。

PhysicalBase 给出了区域的物理基址。对于带 MMU 的处理器，此地址必须与 1MB 边界对齐。

对于带 PU 的处理器，它所匹配的边界必须是区域大小的若干倍。

Size 指定此区域的大小。对于带 MMU 的处理器， Size 必须是一个以兆字节为单位的整数。对于带 PU 的处理器， Size 必须是 4KB 或 4KB 的二次方倍。

Cacheable 指定是否将区域标记为可高速缓存。如果是，则从该区域读取的内容会被高速缓存。

  

| | |
|---|---|
| Bufferable | 指定是否将区域标记为可缓冲。如果是，则写入区域的内容将使用写缓冲器。 |
| Updateable | 只适用于 ARM610™ 处理器。它控制转换表条目中的 U 位。 |
| Domain | 只适用于带 MMU 的处理器。它指定表条目的域字段。 |
| AccessPermissions | 指定对区域的访问控制。有关更多信息，请参阅处理器技术参考手册。 |
| Translate | 控制对此区域的访问是否引起转换错误。如果将区域设为 Translate=No，则无论何时处理器从该区域读取内容或写入内容，都会导致中止。 |

您必须确保定义的区域比目标硬件所能支持的少。必须至少定义一个区域。

### 2.6.5 页表模块和存储器管理单元

诸如 ARM720T™ 和 ARM920T™ 之类的处理器均有 MMU。

MMU 使用存储在存储器中的一组页表来定义存储区域。一旦重置，页表模块就会将顶级页表写出至转换表基址寄存器中指定的地址。该表与您在 `peripherals.ami` 的 `Pagetables` 部分中所定义的区域相对应。

例如，第 2-22 页*存储区域*中给出的缺省配置详情定义了以下页表：

- 整个地址空间， 4GB，被定义为一个单一区域。此区域不可高速缓存或缓冲。虚拟地址被直接映射至整个地址空间上相同的物理地址。
- 第一个 128MB 地址空间被定义为与第一个区域重叠的另一个区域。此区域可以高速缓存或缓冲。虚拟地址被直接映射至物理地址。

它们也可以按如下方式设置控制寄存器：

- 转换表基址寄存器，即寄存器 2，初始化后指向存储器中的页表 `0xA0000000`。
- 域访问控制寄存器，即寄存器 3，初始化后的值为 `0x00000001`。这可将对该区域的访问设为 *client*。
- 控制寄存器（即寄存器 1）的 M、 C 和 W 位被配置为启用 MMU、高速缓存和写缓冲器。如果处理器有单独的指令和数据高速缓存， I 位会配置启用指令高速缓存。

  

### 2.6.6 页表模块和保护单元

诸如 ARM740T™ 和 ARM940T™ 之类的处理器都有一个 PU。

PU 使用一组保护区域。每个保护区域的基址和大小均存储在 PU 的寄存器中。一旦重置，页表模块便会初始化 PU。

例如，上面给出的缺省配置详情会定义一个单一区域，即区域 0。此区域被标记为读/写、可高速缓存和可缓冲。它占据整个地址范围，从 0 至 4GB。

#### ARM740T PU

对于 ARM740T， PU 会按如下方式初始化：

- P、 C 和 W 位通过配置寄存器 （即寄存器 1）设置，可以启用保护单元、高速缓存和写缓冲器。
- 可高速缓存寄存器，即寄存器 2，被初始化为 1，标记区域 0 为可高速缓存区域。
- 写缓冲器控制寄存器，即寄存器 3，被初始化为 1，标记区域 0 为可缓冲区域。
- 保护寄存器，即寄存器 5，被初始化为 3，标记区域 0 为可读/写访问区域。该配置通过 `AccessPermissions` 行进行。
- 区域 0 的保护区域基址和大小寄存器被初始化为 `0x3F`，标记区域 0 的大小为 4GB 并标记区域为已启用。区域 0 的保护区域基址和大小寄存器是寄存器 6 的一部分。寄存器 6 实际上由八个寄存器组成，每一个都是一个区域的保护区域基址和大小寄存器。有关详情，请参阅处理器的技术参考手册。
- 区域 1 的保护区域基址和大小寄存器被初始化后，可以将区域 0 的大小设为 128MB 和已启用。

### ARM940T PU

对于 ARM940T， PU 会按如下方式初始化：

- P、 D、 W 和 I 位通过配置寄存器 （即寄存器 1）设置，可以启用 PU、写缓冲器、数据高速缓存和指令高速缓存。
- 两个可高速缓存寄存器，即寄存器 2，被初始化为 1，标记 I 和 D 高速缓存的区域 0 为可高速缓存区域。这在调试器中显示为 `0x0101`，其中：
  - — 低字节 （位 0..7）代表数据高速缓存的可高速缓存寄存器。
  - — 高字节 （位 8..15）代表指令高速缓存的可高速缓存寄存器。
- 写缓冲器控制寄存器，即寄存器 3，被初始化为 1，标记区域 0 为可缓冲区域。这只适用于数据高速缓存。指令高速缓存为只读。
- 两个保护寄存器，即寄存器 5，被初始化为 3，按如下方式标记指令和数据高速缓存的区域 0 为完全访问。这在调试器中显示为 `0x00030003`，其中：
  - — 低半字 （位 0..15）代表数据高速缓存保护寄存器
  - — 高半字 （位 16..31）代表指令高速缓存保护寄存器。

  显示的第一个寄存器值用于区域 0，第二个值用于区域 1，依次类推。
- 区域 0 和 1 的保护区域基址和大小寄存器初始化后，可以标记区域大小并将其标记为已启用。所有区域的保护区域基址和大小寄存器均是寄存器 6 的一部分。寄存器 6 实际上由十六个寄存器组成，每一个都是一个区域的保护区域基址和大小寄存器。有关详情，请参阅处理器的技术参考手册。
- 寄存器 7 是一个控制寄存器。无法预先知道是否可以从中读取数据。调试器启动时显示零值。它并不由页表模块进行写操作。
- 编程锁定寄存器，即寄存器 9，均被初始化为零。调试器中显示的第一个寄存器值用于数据锁定控制，第二个值用于指令锁定控制。
- 测试和调试寄存器，即寄存器 15，被初始化为零。只有 2 和 3 位在 ARMulator 中起作用。这些位控制高速缓存替换算法是随机还是循环。

## 2.7 缺省存储器模型

缺省存储器模型，即 flatmem，是一个零等待状态存储器系统模型。模拟的存储器大小并不固定。每次访问一个新的存储区域时，主机存储器均会被分配至 64KB 大小的块中。存储器大小受主计算机限制，但在理论上所有 4GB 地址空间均可用。缺省的存储器模型不会产生访问出错。

如果您没有在调试器中指定映射文件，则会使用缺省存储器模型。

缺省的存储器模型会把映射成存储器地址的外围设备适当地进行访问路径控制。路径依赖于在 `peripherals.ami` 或其它 `.ami` 文件中的配置信息。

## 2.8 使用映射文件进行存储器建模

本节包括以下小节：

- *使用映射文件进行存储器建模概述*
- *时钟频率*
- 第 2-29 页*选择映射文件存储器模型*
- 第 2-29 页*映射文件存储器模型如何计算等待状态*
- 第 2-29 页*配置映射存储器模型*。

### 2.8.1 使用映射文件进行存储器建模概述

`mapfile` 是一个可以自行配置的存储器模型。您可以在存储器映射文件的存储器系统中指定单个存储器块的大小、访问宽度、访问类型和访问速度 （请参阅第 4-60 页*映射文件*）。

ARMulator 可模拟所发生的每个存储器访问。它可以根据存储器访问类型计算等待状态。

调试器内部变量 `$memstats` 和 `$statistics` （或这些调试器变量当量）可以给出每个周期类型访问、访问的存储区域以及访问每个区域所花费时间等各方面的详情 （有关检索调试器内部变量详情的信息，请参阅调试器文档）。

**注**

RealView Debugger 不支持 `$memstats`。

### 2.8.2 时钟频率

您可以从调试器配置 `mapfile` 所用的时钟频率。有关详情，请参阅调试器文档。

时钟频率用于确定要添加至每次存储器访问的等待状态数，以及计算一定数量的周期所需的时间。

如果您没有指定时钟速率，则使用 20MHz。如果您指定了速率，但没有指定单位，则假定单位为 Hz。您可以指定 Hz、 kHz 或 MHz。

### 2.8.3 选择映射文件存储器模型

如果您在调试器中指定了存储器映射文件，则在启用存储器映射时，系统会自动使用文件中定义的映射存储器模型。有关如何指定映射文件的详情，请参阅调试器文档。

### 2.8.4 映射文件存储器模型如何计算等待状态

对于不同存储区域的非连续/连续读/写操作，存储器映射文件会以纳秒为单位指定访问时间。通过插入等待状态，映射存储器模型可确保 ARM 处理器的每次访问都至少占用如此长时间。

插入的等待状态数是使总访问时间大于存储器映射文件中指定的纳秒数所需的最小数。设计系统时应考虑这一点。

例如，在时钟速率为 33MHz （30ns 为一个周期）时，在存储器映射文件中指定费时 70ns 的一次访问会产生两个要插入的等待状态，从而使整个访问延长至 90ns。

如果访问时间是 60ns （只快 14%），则模型只会插入一个等待状态 （快 33%）。

处理器时钟速率和存储器映射文件之间的不匹配有时可导致更快的处理器速率反而得到较差性能。例如， 100MHz 处理器 （10ns 周期）需要插入五个等待状态才能访问 60ns 存储器 （总访问时间为 60ns）。 110MHz 时，映射存储器模型必须插入六个等待状态才能访问 （总访问时间为 63ns）。因此 100MHz 的处理器系统反而比 110MHz 处理器的要快。（这不适用于带高速缓存处理器下的情况，此时 110MHz 处理器速度将会较快。）

**注**

为了精确模拟真实硬件，存储器映射文件中指定的访问时间必须包括传输延迟和内存控制器解码时间以及存储器设备的访问时间。例如，对于 70ns RAM，如果存在 10ns 的传输延迟，则将映射文件配置为 80ns。

### 2.8.5 配置映射存储器模型

您可以配置映射存储器模型，以便建立几种不同类型的内存控制器模型，这可通过编辑 peripherals.ami 文件中的条目实现：

```
{ Default_Mapfile=Mapfile
AMBABusCounts=False
;SpotISCyles=True|False
SpotISCyles=True
;ISTiming=Late|Early|Speculative
ISTiming=Late
}
```

### 计算 AMBA™ 解码周期

您可以配置模型，以便为处理器的每次非连续访问插入一个额外的解码周期。这会建立解码周期模型，可在一些 AMBA 总线系统上看到。可以通过在 peripherals.ami 中设置 AMBABusCounts=True 来启用此设置。

### 合并的 I-S 周期

所有 ARM 处理器，特别是高速缓存处理器，均可以作为一对空闲和连续周期来执行非连续访问，这便是*合并的 I-S 周期*。缺省情况下，模型会将这些周期作为非连续访问处理，方法是在 S 周期内插入等待状态，为非连续访问延长时间。

您可以通过在 peripherals.ami 中设置 SpotISCycles=False 来禁用此设置。但是，这可能导致夸大的性能指标，特别是在建立带高速缓存的 ARM 处理器模型时。

模型可以使用以下三种方法中的任何一种来模拟合并的 I-S 周期：

**Speculative** 这种方法建立一个存储控制器可以在空闲周期里预测解码的系统模型。控制器可以使用 I 和 S 周期执行访问。这可以导致少一个等待状态。

**Early** 在 ARM 声明下一个周期将是 S 周期 （即 I 周期的一半）时开始解码。这有时可能导致少一个等待状态。（是否存在更少的等待状态取决于该存储区域的周期时间和非连续访问时间。）

这是缺省设置。您可以通过在 peripherals.ami 中设置 ISTiming=Spec 或 ISTiming=Late 来更改此设置。

**Late** 直至 S 周期才开始解码。实际上， I 周期之后的所有 S 周期均被视为 N 周期。

有关合并的 I-S 周期的详情，请参阅第 2-14 页 *ARMulator 周期类型*。

  

## 2.9 Semihosting

Semihosting 是指代码在 ARM 的目标设备上运行 , 可以利用运行着调试器的主机的一种机制。可以利用的主机资源包括键盘输入、屏幕输出和磁盘 I/O 等。

有关详情，请参阅 *RealView Compilation Tools v1.2 Compilers and Libraries Guide*。

### 2.9.1 Semihosting 配置

semihosting SWI 处理程序配置由 peripherals.ami 中的一部分控制。它包含以下项目：

**注**

如果您的调试器设置了 AngelSWIARM 和 AngelSWIThumb SWI，则它们会覆盖 peripherals.ami 中的设置。

```
{Default_Semihost=Semihost
; Demon is only needed for validation.
DEMON=False
ANGEL=TRUE
AngelSWIARM=0x123456
AngelSWIThumb=0xab
; And the default memory map
HeapBase=0x00000000
HeapLimit=0x07000000
StackBase=0x08000000
StackLimit=0x07000000
}
```

## 2.10 外围设备模型

ARMulator 包括几个外围设备模型。本部分介绍关于这些模型的基本用户信息。

本节包括以下小节：



### 2.10.1 配置 ARMulator 使用外围设备模型

可以通过更改 `default.ami` 文件副本中的相关条目来启用或禁用每个外围设备模型，例如：

```
{ WatchDog=No_watchdog
}
```

可以更改为：

```
{ Watchdog=Default_WatchDog
}
```

其它外围设备模型也可用相同方法进行控制，即在外围设备名称前添加 `No_` 和 `Default_` 前缀。

### 2.10.2 配置外围设备详细资料

可在 `peripherals.ami` 中找到配置外围设备模型的详细资料。有关如何修改 `.ami` 文件的信息，请参阅第 2-4 页*配置 ARMulator*。

### 2.10.3 中断控制器

中断控制器是参考中断控制器的一个实施工具 （请参阅第 4-74 页*中断控制器*）。

中断控制器模型的配置由 peripherals.ami 中的一部分控制。它包含以下项目：

```
{ Default_Intctrl=Intctrl
WAITS=0
Range:Base=0x0a000000
}
```

Range:Base 指定中断控制器寄存器映射的存储器区域。有关中断控制器寄存器的详情，请参阅第 4-74 页*中断控制器*。

WAITS 指定访问中断控制器时影响到处理器的等待状态数。最大值是 30。

### 2.10.4 计时器

计时器是参考计时器的一个实施工具。它提供两个计数器 - 计时器。有关详情，请参阅第 4-76 页*计时器*。

计时器模型配置由 peripherals.ami 中的一部分控制。它包含以下项目：

```
{Default_Timer=Timer
WAITS=0
Range:Base=0x0a800000
;Frequency of clock to controller.
CLK=20000000
;; Interrupt controller source bits - 4 and 5 as standard
IntOne=4
IntTwo=5
}
```

Range:Base 指定计时器寄存器映射的存储器区域。有关中断控制器寄存器的详情，请参阅第 4-76 页*计时器*。

CLK 用于指定外围设备的时钟频率。这通常与处理器时钟频率相同。

IntOne 为计时器 1 中断指定到中断控制器的中断线连接。IntTwo 为计时器 2 中断指定到中断控制器的中断线连接。

WAITS 指定访问计时器时影响到处理器的等待状态数。最大值是 30。

  

### 2.10.5 看门狗

使用看门狗可以防止因程序出错而锁定系统。如果程序在预定时间前访问看门狗失败，看门狗会暂停 ARMulator 并回控调试器。

**注**

这是看门狗计时器的一般模型。它可以帮助用户建立系统环境的模型。它并不建立 ARM 提供的任何实际硬件的模型。

看门狗配置由 peripherals.ami 中的一部分控制。它包含以下项目：

```
{Default_WatchDog=WatchDog
WAITS=0
Range:Base=0xb0000000
KeyValue=0x12345678
WatchPeriod=0x80000
IRQPeriod=3000
IntNumber=16
StartOnReset=True
RunAfterBark=True
}
```

Range:Base 指看门狗寄存器映射的存储器区域。

这是双计时器看门狗。

如果 StartOnReset 设为 True，则第一个计时器会在重置时启动。如果 StartOnReset 设为 False，则第一个计时器只在程序将配置的密钥值写入 KeyValue 寄存器时才启动。这可从 Range:Base 行 (0xB0000000) 中给定的地址中找到。

第一个计时器可在 WatchPeriod 内存周期后生成 **IRQ**，并启动第二个计时器。如果程序尚未将配置的密钥值写入 KeyValue 寄存器，则第二个计时器会在 IRQPeriod 内存周期后出现超时。将 IRQPeriod 配置为合适的值，以便允许程序对 **IRQ** 作出反应。

如果 RunAfterBark 设为 True，则看门狗会在第二个计时器超时时暂停 ARMulator。您可以继续执行或调试。

如果 RunAfterBark 设为 False，则看门狗会暂停 ARMulator 并回控调试器。

IntNumber 指定看门狗所连接的中断行号。

WAITS 指定访问看门狗时影响到处理器的等待状态数。最大值是 30。

### 2.10.6 堆栈跟踪器

堆栈跟踪器会检查每个指令后的堆栈指针 (r13) 的内容。它会记录最低值并据此计算出堆栈的最大大小。启用堆栈跟踪后， ARMulator 的运行速度会更慢。

StackUse 模型会连续监控堆栈指针并报告 $statistics （或调试器当量）中所用的堆栈数量。它必须通过堆栈位置进行配置。

缺省情况下，禁用堆栈跟踪器。要启用堆栈跟踪器，请编辑 default.ami 副本：

1. 请查找行：
   ```
   { StackUse=No_StackUse
   ```
2. 将其更改为：
   ```
   { StackUse=Default_StackUse
   ```

初始化之前，堆栈指针可以包含堆栈限制以外的值。您必须配置堆栈限制，以便堆栈跟踪器可以忽略这些预初始化值。此配置位于 peripherals.ami 中：

```
{ Default_StackUse=StackUse
StackBase=0x80000000
StackLimit=0x70000000
}
```

StackBase 是堆栈顶部的地址。StackLimit 是堆栈的下限。更改这些值不会改变堆栈在存储器中的位置。要改变堆栈位置，您必须重新配置调试监控器模型。

### 2.10.7 Tube

tube 是存储器映射的寄存器。如果您将可印刷字符写在 tube 上，则字符会出现在控制台上。它允许您检查写入操作是否发生在存储器中的指定位置。

您可以更改 Tube 被映射的地址。这由 peripherals.ami 中的一个条目控制：

```
{Default_Tube=Tube
Range:Base=0x0d800020
}
```

这是缺省地址。

# 第 3 章
# 编写 ARMulator 模型

本章意在帮助您编写模型以添加至 ARMulator。它包含以下部分：

## 3.1 ARMulator 扩展套件

您也可以在不更改现有模型的情况下将其它模型添加至 ARMulator。每个模型均是独立和完备的，通过定义的接口与 ARMulator 进行通信。这些接口的定义在第 4 章 *ARMulator 参考* 中提供。

本文档其余部分中指定的路径 *ExtensionKit_path* 是指：

*install_directory*/RVARMulator/ExtensionKit

对于 Windows，请用 \ 替换 /。

### 3.1.1 文件位置

ARMulator 扩展套件包括某些模型的源代码。您可以复制这些模型并修改其副本。

#### 源文件位置

供您进行复制的模型源代码位于：

*ExtensionKit_path*/1.3/Final/*platform*/armulext

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

#### 标头文件的位置

标头文件位于：

*ExtensionKit_path*/1.3/release/*platform*/armulif

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

### makefile 的位置

makefile 位于：

*ExtensionKit_path*/1.3/Final/linux-pentium/armulext/*model*.b/*target*

在该路径中：

- *model* 是模型名称。例如， millisec.b 是执行简单毫秒计时器功能的外设模型 millisec 的目录。
- *target* 是：
  - Windows 的 intelrel
  - Linux 的 linux86
  - Solaris 的 gccsolaris。
- 对于 Windows，请用 \ 替换 /。

使用这些文件作为编写用户自己模型的例子。为了帮助您选择合适的模型进行检查，本章包含这些模型的列表并附有各自功能的简单介绍 （请参阅第 3-4 页 *附带的模型*）。

### 3.1.2 附带的模型

ARMulator 附带用于以下模型组的源代码：

- *基本模型*
- 第 3-5 页*外围设备模型*。

#### 基本模型

`tracer.c` 跟踪器模块可以跟踪指令执行和来自 ARMulator 内部的事件 （请参阅第 4-58 页*跟踪器*）。您可以将自己的跟踪代码链接至跟踪器模块。

`profiler.c` profiler 模块可以提供配置功能 （请参阅第 2-12 页 *Profiler*）。这包括基本指令抽样和更多高级使用 （例如配置高速缓存故障）。这可以通过提供处理来自 RDI 1.5.1 可兼容调试器的配置请求的 `UnkRDIInfoHandler` 来实现 （参见第 4-35 页*未知的 RDI 信息处理程序*）。

`pagetab.c` 重置时，此模块将设置 ARMulator 内的高速缓存、 PU 或 MMU 及相关页表 （请参阅第 2-19 页*页表模块*）。

`stackuse.c` 如果启用，此模型即会跟踪堆栈大小。堆栈用法在 ARMulator 存储器统计数据表中报告。您可以在 `peripherals.ami` 文件中设置堆栈的上限和下限 （请参阅第 2-36 页*堆栈跟踪器*）。

`nothing.c` 此模型不起作用。您可以在 `peripherals.ami` 文件中使用它来禁用模型 （请参阅第 3-12 页*配置 ARMulator 以禁用模型*）。

`semihost.c` 此模型提供 *RealView Compilation Tools v1.2 Compilers and Libraries Guide* 中说明的 semihosting SWI。

`dcc.c` 这是 *Debug Communications Channel* (DCC) 的模型。

`mapfile.c` 此模型允许您指定存储器系统的特征。有关详情，请参阅第 4-60 页*映射文件*。

`flatmem.c` `flatmem` 可模拟零等待状态的存储器系统。有关详情，请参阅第 2-27 页*缺省存储器模型*。

### 外围设备模型

| | |
|---|---|
| intc.c | 请参阅第 2-33 页*中断控制器*。intc 是 *Reference Peripherals Specification* (RPS) 中说明的中断控制器外围设备的模型。 |
| timer.c | 请参阅第 2-34 页*计时器*。timer 是 RPS 计时器外围设备的模型。提供两个计时器。timer 必须用于连接中断控制器，但不必使用 intc。 |
| millisec.c | 简单的毫秒计时器。 |
| watchdog.c | 看门狗。请参阅第 2-35 页*看门狗*。watchdog 是普通看门狗模型。它并不模拟特定的看门狗硬件，只提供一般的看门狗功能。 |
| tube.c | Tube 请参阅第 2-36 页 *Tube*。tube 是个简单的调试辅助设备。它允许您检查写操作是否发生在存储器中的指定位置。 |

## 3.2 写新外围设备模型

本节包括以下小节：

- *使用样本模型作为模板*
- *返回值*
- 第 3-7 页*初始化、最终化和状态宏*
- 第 3-7 页*注册模型*。

### 3.2.1 使用样本模型作为模板

要写新模型，最好的步骤是先复制一个现成的模型，然后再编辑其副本。要执行此操作：

1. 选择一个最接近您需要的模型。例如可能是计时器。
2. 复制源文件 （此处指 `timer.c`）并使用新名称 （例如 *`mymodel`*`.c`）。
3. 复制 make 子目录 （此处指 `timer.b`）并使用相应的新名称 （此处为 *`mymodel`*`.b`）。
4. 检查模型的 `Makefile` （请参阅第 3-2 页*文件位置*）。

   将 `Makefile` 载入文本编辑器并将所有的 `timer` 实例更改为 *`mymodel`*。

现在您可以编辑 *`MyModel`* 了。

### 3.2.2 返回值

模型必须返回以下存储器访问状态中的其中一个：

| | |
|---|---|
| `PERIP_OK` | 如果模型可以响应请求。 |
| `PERIP_BUSY` | 如果存储器访问需要等待状态。模型不能将此状态返回至调试器。 |
| `PERIP_DABORT` | 如果外围设备在总线上置起 **DABORT** 信号。 |
| `PERIP_NODECODE` | 如果模型连同属于它的地址一起被调用，但这对它没有意义。存储器模型将调用视为存储器访问处理。 |

### 3.2.3 初始化、最终化和状态宏

为了帮助您编写新的 ARMulator 模型， minperip.h 中提供了以下宏：

- BEGIN_INIT()
- END_INIT()
- BEGIN_EXIT()
- END_EXIT()
- BEGIN_STATE_DECL()
- END_STATE_DECL()。

使用以下内容为模型定义初始化函数：

```
BEGIN_INIT(your_model)
{
    /*
      * （此处输入您的初始化代码）
    */
}
END_INIT(your_model)
```

使用以下内容为模型定义最终化函数：

```
BEGIN_EXIT(your_model)
{
    /*
      * （此处输入您的最终化代码）
    */
}
END_EXIT(your_model)
```

BEGIN_INIT() 宏可定义一种结构，以便存放模型使用的所有私有数据，END_EXIT() 宏可以释放这些数据。使用以下内容声明数据结构：

```
BEGIN_STATE_DECL(your_model)
     /*
       * （此处输入您的私有数据）
     */
END_STATE_DECL(your_model)
```

### 3.2.4 注册模型

您的模型必须通过调用 registerPeripFunc() 来自行注册。这可使 ARMulator 调用您的模型并访问属于模型的存储器位置。请参阅第 4-45 页 *ARMul_BusRegisterPeripFunc*。

  

## 3.3 构建新模型

ARMulator 预期在可执行路径中查找模型：

*install_directory*/RVARMulator/ARMualtor/1.3/release/*platform*

当您构建新模型时，构建的文件放入以下构建路径：

*ExtensionKit_path*/1.3/release/*platform*/armulext/*mymodel*.b/*target*

在这些路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- *mymodel* 是您为新模型设定的名称。
- *target* 是：
  - Windows 的 intelrel
  - Linux 的 linux86
  - Solaris 的 gccsolaris。
- 对于 Windows，请用 \ 替换 /。

### 3.3.1 如何构建新模型

要构建新模型：

1. 将系统的当前目录更改为 build_path。
2. 使用安装在系统中的 make 实用程序构建模型。它可以是以下各项之一：
   - Windows 的 nmake
   - Linux 或 Solaris 的 make。

   文件放入 build_path。

   **注**

   为了说明清楚，在本节中，可执行和构建路径称为 exec_path 和 build_path。有关完整路径名，请参阅第 3-8 页*构建新模型*。
3. 将已构建文件从 build_path 移至 exec_path。视您的系统而定，文件名可以是：
   - 在 Windows 中为 *mymodel*.dll
   - 在 Linux 或 Solaris 中为 *mymodel*.so。

## 3.4 配置 ARMulator 以使用新模型

ARMulator 可以通过读取 .ami 和 .dsc 配置文件决定要使用的模型。请参阅第 4-64 页 *ARMulator 配置文件*。

在 ARMulator 使用新模型之前，您必须为模型添加 .dsc 文件，同时其参考信息必须添加至配置文件 default.ami 和 peripherals.ami。

以下小节说明了执行步骤：

- *添加 .dsc 文件*
- 第 3-11 页*编辑 default.ami 和 peripherals.ami*。

### 3.4.1 添加 .dsc 文件

创建名为 *MyModel*.dsc 的文件并将其放入：

*install_directory*/RVARMulator/ARMualtor/1.3/release/*platform*

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

*MyModel*.dsc 必须包含以下内容：

```
;; ARMulator configuration file type 3
{ Peripherals
  {MyModel
    MODEL_DLLfilename=MyModel
  }
  {
    No_MyModel=Nothing
  }
}
```

模型的名称视系统而定 （请参阅第 3-9 页*如何构建新模型*）：

- *MyModel*.dll
- *MyModel*.so.

  

Nothing 是一个不起作用的预定义模型。No_*MyModel*=Nothing 行允许在 .ami 文件中使用 No_*MyModel*。这使用户可配置 ARMulator 以排除模型 （请参阅第 3-12 页*配置 ARMulator 以禁用模型*）。

如有必要，您可以将其它配置详细信息包含在 *MyModel*.dsc 文件中。例如，请根据以下路径参阅附带的 .dsc 文件：

*install_directory*/RVARMulator/ARMualtor/1.3/release/*platform*

### 3.4.2 编辑 default.ami 和 peripherals.ami

本说明假定您的模型基于计时器：

1. 将 default.ami 文件载入文本编辑器，并找到以下行：

   ```
   {Timer=Default_Timer
   }
   ```

2. 将参考信息添加至您的模型：

   ```
   {Timer=Default_Timer
   }

   {MyModel=Default_MyModel
   }
   ```

3. 保存您编辑过的 default.ami 文件。

4. 将 peripherals.ami 文件载入文本编辑器，然后查找 “计时器”部分：

   ```
   { Default_Timer=Timer
   .
   .
   .
   }
   ```

5. 以此作为实例，为您的模型添加配置部分。根据您的模型与计时器的差别程度，您可能可以很容易地编辑计时器部分的副本。

6. 保存您编辑过的 peripherals.ami 文件。

## 3.5 配置 ARMulator 以禁用模型

您可以通过更改 peripherals.ami 中模型的条目来禁用此模型。例如，要禁用 Tube 模型：

1. 在 peripherals.ami 中找到以下行：

```
{Default_Tube=Tube
Range:Base=0x0d800020
}
```

2. 将其更改为：

```
{Default_Tube=No_Tube
Range:Base=0x0d800020
}
```

表示使用 nothing.c 模型覆盖 tube.c 模型。nothing 将忽略所有配置详细信息，例如 Range:Base。

# 第 4 章
# ARMulator 参考

本章介绍有关 ARMulator 的参考信息。包含以下几个部分：

## 4.1 ARMulator 模型

ARMulator 由一系列模拟基于 ARM 的硬件的模型组成。这些模型可以使您在硬件工作完成之前开始评估、开发和调试软件的工作。

### 4.1.1 通过 ToolConf 配置模型

您可以通过 ToolConf 配置 ARMulator 模型。ToolConf 是一个包含标签和数值的数据库，这些标签和数值由 ARMulator 在初始化期间从配置文件 （.dsc 和 .ami 文件）中读取 （请参阅第 4-69 页 *ToolConf*）。

系统提供了用于从此数据库查找数值的一系列函数。全套函数在以下位置定义：

*install_directory*/RVARMulator/ARMulator/1.3 /release/clx/toolconf.h

对于 Windows，请用 \ 替换 /。

所有函数均采用一个称为 toolconf 的不透明句柄。

  

## 4.2 与内核进行通信

在初始化期间，所有模型均会收到一个 RDI_ModuleDesc * 型的 mdesc 结构的指针。它们会将此结构作为字段 （称之为 coredesc）复制到其自身的状态。这作为第一个参数传送至大多数 *ARMulif*（ARMulator 接口）函数。 ARMulator 将导出这些函数，以便启用模型来通过此句柄访问 ARMulator 状态。

以下函数提供对 ARM 寄存器的读写访问：

- 第 4-5 页 *ARMulif_GetReg*
- 第 4-5 页 *ARMulif_SetReg*
- 第 4-6 页 *ARMulif_GetPC 和 ARMulif_GetR15*
- 第 4-6 页 *ARMulif_SetPC 和 ARMulif_SetR15*
- 第 4-7 页 *ARMulif_GetCPSR*
- 第 4-7 页 *ARMulif_SetCPSR*
- 第 4-8 页 *ARMulif_GetSPSR*
- 第 4-8 页 *ARMulif_SetSPSR*。

在调用内核的 ARMulif 中调用函数时，模型必须将指针传送至其 coredesc 结构。

以下函数可提供对 CPSR 中特定位或字段的便利的访问：

- 第 4-9 页 *ARMulif_ThumbBit*
- 第 4-9 页 *ARMulif_GetMode*。

以下函数可调用协处理器的读写方法：

- 第 4-10 页 *ARMulif_CPRead*
- 第 4-11 页 *ARMulif_CPWrite*。

**注**

从模型的某些组成部分访问状态的一些组成部分会不恰当。例如，您不能通过存储器访问函数设置 ARM 寄存器的内容，因为存储器访问函数可以在模拟指令期间被调用。相反，有时需要通过 SWI 处理程序函数设置 ARM 寄存器的内容。

### 4.2.1 模式编号

以下一些函数采用 **unsigned** 模式参数指定处理器模式。模式编号在 armdefs.h 中定义，列示如下：

- USER32MODE
- FIQ32MODE
- IRQ32MODE
- SVC32MODE
- ABORT32MODE
- UNDEF32MODE
- SYSTEM32MODE

此外还定义了特殊值 CURRENTMODE。这使 ARMulif_GetReg() 可以完成一些操作，例如返回当前模式的寄存器。

### 4.2.2 ARMulif_GetReg

此函数可读取指定处理器模式下的寄存器。

#### 语法

```
ARMword ARMulif_GetReg(RDI_ModuleDesc *mdesc, ARMword mode, unsigned reg)
```

其中：

*mdesc* 是内核的句柄。

*mode* 是处理器模式。模式的值在 armdefs.h 中定义 （请参阅第 4-4 页*模式编号*）。

*reg* 是要读取的寄存器。寄存器 r0 至 r14、PC 或 CPSR 的有效值是 0 至 14。

#### 返回

此函数可返回特定模式的指定寄存器的值。

### 4.2.3 ARMulif_SetReg

此函数可写入指定处理器模式的寄存器。

#### 语法

```
void ARMulif_SetReg(RDI_ModuleDesc *mdesc, ARMword mode,
                    unsigned reg, ARMword value)
```

其中：

*mdesc* 是内核的句柄。

*mode* 是处理器模式。模式编号在 armdefs.h 中定义 （请参阅第 4-4 页*模式编号*）。

*reg* 是要写入的寄存器。寄存器 r0 至 r14、PC 或 CPSR 的有效值是 0 至 14。

*value* 是要写入特定处理器模式下的寄存器 reg 的值。

#### 用法

您可使用此函数写入任何通用寄存器 r0 至 r14、 PC 或 CPSR。

### 4.2.4 ARMulif_GetPC 和 ARMulif_GetR15

此函数可读取 pc。ARMulif_GetPC 和 ARMulif_GetR15 是同义词。

#### 语法

```
ARMword ARMulif_GetPC(RDI_ModuleDesc *mdesc)
ARMword ARMulif_GetR15(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

#### 返回

此函数可返回 pc 的值。

### 4.2.5 ARMulif_SetPC 和 ARMulif_SetR15

此函数可将值写入 pc。ARMulif_SetPC 和 ARMulif_SetR15 是同义词。

#### 语法

```
void ARMulif_SetPC(RDI_ModuleDesc *mdesc, ARMword value)
void ARMulif_SetR15(RDI_ModuleDesc *mdesc, ARMword value)
```

其中：

*mdesc* 是内核的句柄。

*value* 是要写入 pc 的值。

### 4.2.6 ARMulif_GetCPSR

此函数可读取 CPSR。

#### 语法

```
ARMword ARMulif_GetCPSR(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

#### 返回

此函数可返回 CPSR 的值。

### 4.2.7 ARMulif_SetCPSR

此函数可将值写入 CPSR。

#### 语法

```
void ARMulif_SetCPSR(RDI_ModuleDesc *mdesc, ARMword value)
```

其中：

*mdesc* 是内核的句柄。

*value* 是要写入 CPSR 的值。

### 4.2.8 ARMulif_GetSPSR

此函数可返回指定处理器模式下的 SPSR 当前内容。

#### 语法

```
ARMword ARMulif_GetSPSR(RDI_ModuleDesc *mdesc, ARMword mode)
```

其中：

*mdesc* 是内核的句柄。

*mode* 是要读取的 SPSR 处理器模式。

#### 用户模式

如果 *mode* 是 USER32MODE，则 ARMulif_GetSPSR 可返回 CPSR 的当前内容。

### 4.2.9 ARMulif_SetSPSR

此函数可将值写入指定处理器模式下的 SPSR。

#### 语法

```
void ARMulif_SetSPSR(RDI_ModuleDesc *mdesc, ARMword mode, ARMword value)
```

其中：

*mdesc* 是内核的句柄。

*mode* 是要写入的 SPSR 的处理器模式。

*value* 是要写入指定模式的 SPSR 的值。

#### 用户模式

如果 *mode* 为 USER32MODE，则 ARMulif_SetSPSR 不起作用。

### 4.2.10 ARMulif_ThumbBit

如果内核处于 Thumb 状态，则此函数返回 1；如果内核处于 ARM 状态，则此函数返回 0。

#### 语法

```
unsigned ARMulif_ThumbBit(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

### 4.2.11 ARMulif_GetMode

此函数可读取当前处理器模式。

#### 语法

```
unsigned ARMulif_GetMode(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

### 4.2.12 ARMulif_CPRead

此函数可调用协处理器的 read 方法。

#### 语法

```
int ARMulif_CPRead(RDI_ModuleDesc *mdesc, unsigned cpnum,
                   unsigned reg, ARMword *data)
```

其中：

*mdesc* 是内核的句柄。

*cpnum* 是协处理器的编号。

*reg* 是从中进行读取的协处理器寄存器编号，由 LDC 或 STC 指令中的 CR*n* 编入索引。

*data* 是从协处理器寄存器读取的数据的指针。被传输的字数及顺序与协处理器有关。

#### 返回

函数必须返回：

- ARMul_DONE，如果可以读取寄存器的话
- ARMul_CANT，如果无法读取寄存器的话。

### 4.2.13 ARMulif_CPWrite

此函数可调用协处理器的 write 方法。它还可以截取调用以写入 FPE 模拟的寄存器。

#### 语法

```
int ARMulif_CPWrite(RDI_ModuleDesc *mdesc, unsigned cpnum,
                    unsigned reg, ARMword *data)
```

其中：

*mdesc* 是内核的句柄。

*cpnum* 是协处理器的编号。

*reg* 是从中进行读取的协处理器寄存器编号，由 LDC 或 STC 指令中的 CR*n* 编入索引。

*data* 是从协处理器寄存器读取的数据的指针。传输的字数及顺序与协处理器有关。

#### 返回

函数必须返回：

- ARMul_DONE，如果可以写寄存器的话
- ARMul_CANT，如果无法写寄存器的话。

## 4.3 基本模型接口

本部分包括以下小节：

- 第 4-13 页*私有状态数据结构的声明*
- 第 4-14 页*模型初始化*
- 第 4-14 页*模型最终化*。

对于每个模型，您均必须写入一个初始化函数。对于其它功能，您必须记录回调。

对于以下选取内容，`minperip.h` 中提供了宏：

- 第 4-13 页*私有状态数据结构的声明*
- 第 4-14 页*模型初始化*
- 第 4-14 页*模型最终化*。

另请参阅第 3-7 页*初始化、最终化和状态宏*。

### 4.3.1 私有状态数据结构的声明

每个模型均必须将其状态保存在私有数据结构中。初始化和最终化宏可由 ARMulif 提供。这些宏需要使用此数据结构中的某些字段。

要声明状态数据结构，请按照以下说明使用 BEGIN_STATE_DECL 和 END_STATE_DECL 宏：

```
/*
 * Create a YourModelState data structure
 */
BEGIN_STATE_DECL(YourModel)
/*
 * Your private data here
 */
END_STATE_DECL(YourModel)
```

这可声明以下结构：

```
typedef struct YourModelState
```

此结构包含：

- 预定义的数据字段：
  - toolconf config
  - const struct RDI_HostosInterface *hostif
  - RDI_ModuleDesc coredesc;
  - RDI_ModuleDesc agentdesc
- 您输入的私有数据位于宏之间。

### 4.3.2 模型初始化

BEGIN_INIT() 和 END_INIT() 宏形成模型的初始化函数的启动和完成。初始化函数在以下时间被调用：

- ARMulator 初始化期间
- 无论何时从调试器下载新映像时

初始化函数中包含以下局部变量：

- **bool** coldboot

  在 ARMulator 进行初始化时为 TRUE，在从调试器下载新映像时为 FALSE。

- YourModelState *state

  私有状态数据结构的指针。初始化宏将为其分配和清除存储器，并且预定义数据字段会被初始化。

在初始化函数中，您的模型必须：

- 初始化任何私有数据
- 安装任何回调。

### 4.3.3 模型最终化

BEGIN_EXIT() 和 END_EXIT() 宏可形成模型的最终化函数的启动和完成。关闭 ARMulator 时将会调用最终化函数。

最终化函数中包含以下局部变量：

YourModelState *state

您的模型必须卸载最终化函数中的所有回调。

END_EXIT() 宏将会释放分配给状态的存储器。

## 4.4 协处理器模型接口

协处理器模型接口在 armul_copro.h 中定义。基本协处理器函数有：

- 第 4-16 页 *ARMulif_InstallCoprocessorV5*
- 第 4-17 页 *LDC*
- 第 4-18 页 *STC*
- 第 4-19 页 *MRC*
- 第 4-20 页 *MCR*
- 第 4-19 页 *MRC*
- 第 4-20 页 *MCR*
- 第 4-21 页 *MCRR*
- 第 4-22 页 *MRRC*
- 第 4-23 页 *CDP*。

**注意**

某些协处理器具有只写寄存器。写入这些寄存器的值必须是特定值。如果有不正确的值写入模型中的此类寄存器，则结果不可预知，并且可能不遵循硬件中出现的情况。

此外，系统还提供两个函数，使 RDI 1.5.1 可兼容调试器可以通过*远程调试接口* (RDI) 读写协处理器寄存器。这两个函数是：

- 第 4-24 页 *read*
- 第 4-25 页 *write*。

如果协处理器无法处理一个或更多此类函数，则必须保持 ARMul_CPInterface 结构中的条目不变。

### 4.4.1 ARMulif_InstallCoprocessorV5

使用此函数记录一个协处理器处理程序。

此函数在 armul_copro.h 中有原型。

#### 语法

```
unsigned ARMulif_InstallCoprocessorV5(RDI_ModuleDesc *mdesc, unsigned number,
                                      struct ARMul_CoprocessorV5 *cpv5, void *handle)
```

其中：

*mdesc* 是内核的句柄。

*number* 是协处理器编号。

*cpv5* 是协处理器接口结构的指针。

*handle* 是传送到每个协处理器函数的专有数据的指针。

#### 返回

此函数可返回以下其中一个值：

- ARMulErr_NoError，如果没有错误的话
- 一个 ARMul_Error 值。

有关错误代码的完整列表，请参阅 armerrs.h 和 errors.h。错误必须通过 Hostif_RaiseError() 传送以进行格式化 （请参阅第 4-53 页 *Hostif_RaiseError*）。

  

### 4.4.2 LDC

识别协处理器的 LDC 指令时将调用此函数。

#### 语法

**unsigned** LDC(**void** **handle*, **int** *type*, ARMword *instr*, ARMword **data*)

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_TRANSFER | 表示 ARM 即将执行加载。 |
| ARMul_CP_DATA | 表示有效数据包含在 *data* 中。 |

*instr* 当前操作码。

*data* 是从存储器加载至协处理器的数据的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_INC，以从内核请求更多数据 （仅响应 ARMul_CP_FIRST、ARMul_CP_BUSY 或 ARMul_CP_DATA）。
- ARMul_CP_DONE，以表示协处理器操作已完成 （仅响应 ARMul_CP_DATA）。
- ARMul_CP_BUSY，以表示协处理器正忙 （仅响应 ARMul_CP_FIRST 或 ARMul_CP_BUSY）。
- ARMul_CP_CANT，以表示指令不受支持，或者无法访问指定的寄存器 （仅响应 ARMul_CP_FIRST 或 ARMul_CP_BUSY）。
- ARMUL_CP_LAST，以表示下一次加载是顺序中的最后一次。这仅需用于 ARM9。

### 4.4.3 STC

识别协处理器的 STC 指令时会调用此函数。

#### 语法

**unsigned** STC(**void** **handle*, **int** *type*, ARMword *instr*, ARMword **data*)

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将中断，因此协处理器必须放弃当前指令。系统通常会稍后重试此指令。此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_DATA | 表示协处理器必须返回 **data* 中的有效数据。 |

*instr* 是当前操作码。

*data* 是保存至存储器的数据位置的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_INC，以表示有更多数据传送至内核 （仅响应 ARMul_CP_FIRST、ARMul_CP_BUSY 或 ARMul_CP_DATA）。
- ARMul_CP_DONE，以表示协处理器操作已完成 （仅响应 ARMul_CP_DATA）。
- ARMul_CP_BUSY，以表示协处理器正忙 （仅响应 ARMul_CP_FIRST 或 ARMul_CP_BUSY）。
- ARMul_CP_CANT，以表示指令不受支持，或者无法访问指定的寄存器 （仅响应 ARMul_CP_FIRST 或 ARMul_CP_BUSY）。
- ARMUL_CP_LAST，以表示下一次保存是顺序中的最后一次。这仅需用于 ARM9。

### 4.4.4 MRC

识别协处理器的 MRC 指令时会调用此函数。如果请求的协处理器寄存器不存在或无法写入其中，则函数必须返回 ARMul_CP_CANT。

#### 语法

**unsigned** MRC(**void** **handle*, **int** *type*, ARMword *instr*, ARMword **data*)

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_DATA | 表示有效数据包含在 **data* 中。 |

*instr* 是当前操作码。

*data* 是从协处理器转移至内核的数据位置的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成，并且有效数据已返回 **data*
- ARMul_CP_BUSY，表示协处理器正忙
- ARMul_CP_CANT，表示指令不受支持或无法访问指定的寄存器。

### 4.4.5 MCR

识别协处理器的 MCR 指令时将调用此函数。如果请求的协处理器寄存器不存在或无法写入，则函数必须返回 ARMul_CP_CANT。

#### 语法

**unsigned** MCR(**void** **handle*, **int** *type*, ARMword *instr*, ARMword **data*)

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_DATA | 表示有效数据包含在 *data* 中。 |

*instr* 是当前操作码。

*data* 是转移至协处理器的数据的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_BUSY，表示协处理器正忙
- ARMul_CP_CANT，表示指令不受支持或无法访问指定的寄存器。

### 4.4.6 MCRR

识别协处理器的 MCRR 指令时将调用此函数。

如果出现以下情况，函数必须返回 ARMul_CP_CANT：

- 请求的协处理器寄存器不存在
- 请求的协处理器寄存器无法写入
- 协处理器的 ARM 架构是 v4T 或更早设计。

#### 语法

```
unsigned MCRR(void *handle, int type, ARMword instr, ARMword *data)
```

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_DATA | 表示有效数据包含在 *data* 中。 |

*instr* 是当前操作码。

*data* 是转移至协处理器的数据的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_BUSY，表示协处理器正忙
- ARMul_CP_CANT，表示指令不受支持或无法访问指定的寄存器。

### 4.4.7 MRRC

识别协处理器的 MRRC 指令时将调用此函数。

如果出现以下情况，函数必须返回 ARMul_CP_CANT：

- 请求的协处理器寄存器不存在
- 请求的协处理器无法读取
- 协处理器的 ARM 架构是 v4T 或更早设计。

#### 语法

```
unsigned MRRC(void *handle, int type, ARMword instr, ARMword *data)
```

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |
| ARMul_CP_DATA | 表示有效数据包含在 *data* 中。 |

*instr* 是当前操作码。

*data* 是从协处理器转移的数据的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_BUSY，表示协处理器正忙
- ARMul_CP_CANT，表示指令不受支持或无法访问指定的寄存器。

### 4.4.8 CDP

识别协处理器的 CDP 指令时将调用此函数。如果请求的协处理器操作不受支持，则函数必须返回 ARMul_CP_CANT。

#### 语法

**unsigned** CDP(**void** **handle*, **int** *type*, ARMword *instr*, ARMword **data*)

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*type* 是协处理器访问的类型。它可以是以下各项之一：

| | |
|---|---|
| ARMul_CP_FIRST | 表示首次调用该指令的协处理器模型。 |
| ARMul_CP_BUSY | 表示这是第一个调用忙等待之后的随后的调用。 |
| ARMul_CP_INTERRUPT | 警告协处理器 ARM 即将处理中断，因此协处理器必须放弃当前指令。系统通常会在稍后重试指令，此时 *type* 将被重设为 ARMul_CP_FIRST。 |

*instr* 是当前操作码。

*data* 未使用。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_BUSY，表示协处理器正忙
- ARMul_CP_CANT，表示指令不受支持。

### 4.4.9 read

此函数使 RDI 1.5.1 可兼容调试器可以通过 RDI 读协处理器寄存器。此函数可读编号为 *reg* 的协处理器寄存器，并将其值转移至由 *value* 寻址的位置。

如果请求的协处理器寄存器不存在或无法读取寄存器，则函数必须返回 ARMul_CP_CANT。

#### 语法

```
unsigned read(void *handle, int reg, ARMword instr, ARMword *value)
```

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*reg* 是要被读取的协处理器寄存器的记录编号。

*instr* 未使用。

*value* 是要从协处理器读取的数据位置的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_CANT，表示寄存器不受支持。

#### 用法

此函数由调试器通过 RDI 调用。

### 4.4.10 write

此函数使 RDI 1.5.1 可兼容调试器可以通过 RDI 写入协处理器寄存器。

函数可将 *value* 寻址的位置的值写入编号为 *reg* 的协处理器寄存器。

如果请求的协处理器不存在或无法写入寄存器，则函数必须返回 ARMul_CP_CANT。

#### 语法

```
unsigned write(void *handle, int reg, ARMword instr, ARMword *value)
```

其中：

*handle* 是 ARMulif_InstallCoprocessorV5 的句柄。

*reg* 是要被写入的协处理器寄存器的记录编号。

*instr* 未使用。

*value* 是要写入协处理器的数据位置的指针。

#### 返回

此函数必须返回以下其中一个值：

- ARMul_CP_DONE，表示协处理器操作已完成
- ARMul_CP_CANT，表示寄存器不受支持。

#### 用法

此函数由调试器通过 RDI 调用。

## 4.5 异常

以下函数可启用模型以便设置或清除信号：

- *ARMulif_SetSignal*
- 第 4-27 页 *ARMulif_GetProperty*。

### 4.5.1 ARMulif_SetSignal

ARMulif_SetSignal 函数用于设置信号状态或属性。

#### 语法

```
void ARMulif_SetSignal(RDI_ModuleDesc *mdesc, ARMSignalType sigType,
                       SignalState sigState)
```

其中：

*mdesc* 是内核的句柄。

*sigtype* 是要设置的信号。*sigtype* 可以是以下其中之一：

RDIPropID_ARMSignal_IRQ
: 表明中断。

RDIPropID_ARMSignal_FIQ
: 表明快速中断。

RDIPropID_ARMSignal_RESET
: 表明重设的信号。内核将会重设，并在取消表明重设信号之前不会重新启动。

RDIPropID_ARMSignal_BigEnd
: 设置此信号以进行大端操作，或清除此信号以进行小端操作。

RDIPropID_ARMSignal_HighException
: 设置异常向量的基址。

RDIPropID_ARMSignal_BranchPredictEnable
: （仅适用于 ARM10）

RDIPropID_ARMSignal_LDRSetTBITDisable
: （仅适用于 ARM10）

RDIPropID_ARMSignal_WaitForInterrupt
: （仅适用于 ARM10 和 XScale）

RDIPropID_ARMSignal_DebugState
: 进入或退出调试状态。

  

RDIPropID_ARMulProp_CycleDelta

等待内核指定数量的周期。

RDIPropID_ARMulProp_Accuracy

选择建模精度，以百分比表示的范围为 0% 至 100%。目前仅影响 ARM10 模型。小于 50% 的设置将关闭互锁建模。互锁建模关闭时， ARMulator 的运行速度更快，但周期计数的精确度会降低。

*sigstate* 对于信号，您必须为 *sigstate* 指定以下其中一个值：

FALSE 信号关闭

TRUE 信号打开。

对于属性，您必须为 *sigstate* 指定一个整数值。

**注**

有关使用中断控制器时的信号中断说明，请参阅第 4-74 页*中断控制器*。

### 4.5.2 ARMulif_GetProperty

ARMulif_GetProperty 函数用于读取属性和信号的值。

#### 语法

```
void ARMulif_GetProperty(RDI_ModuleDesc *mdesc, ARMSignalType id,
                         ARMword *value)
```

其中：

*mdesc* 是内核的句柄。

*id* 是要读取的信号或属性。*id* 可以是以下任何一个：

RDIPropID_ARMSignal_IRQ

TRUE，如果表明中断信号的话。

RDIPropID_ARMSignal_FIQ

TRUE，如果表明快速中断信号的话。

RDIPropID_ARMSignal_RESET

TRUE，如果表明重设信号的话。

RDIPropID_ARMSignal_BigEnd

TRUE，如果表明大端信号的话。

RDIPropID_ARMSignal_HighException

TRUE，如果向量表处于 0xFFFF0000 的话。

  

RDIPropID_ARMSignal_BranchPredictEnable

（仅适用于 ARM10）

RDIPropID_ARMSignal_LDRSetTBITDisable

（仅适用于 ARM10）

RDIPropID_ARMSignal_WaitForInterrupt

（仅适用于 ARM10 和 XScale）

RDIPropID_ARMulProp_CycleCount

计算初始化后所执行的周期数量。

RDIPropID_ARMulProp_RDILog

RDI 记录层的当前设置。通常在禁用记录时为零，在启用记录时非零。

RDIPropID_ARMSignal_ProcessorProperties

与所模拟的处理器相关的属性字。这是属性的位域，在 armdefs.h 中定义。

*value* 是写入属性的块的指针。允许属性超过 32 位。但是，所有列出的属性实际上最多只有 32 位宽。

  

## 4.6 事件

ARMulator 具有传播和处理事件的机制。这些事件包括一个事件编号和一个字对。编号用于标识事件。详细信息视事件而定。

核心 ARMulator 可生成一些事件实例，在 armdefs.h 中定义。这些事件可分为以下几组：

- 来自 ARM 处理器内核的事件，在第 4-30 页表 4-2 中列示。
- 来自 MMU 和高速缓存 （不是在 StrongARM-110 上）的事件，在表 4-1 中列示
- 来自预取单元的事件 （只适用于基于 ARM8 的处理器），在第 4-30 页表 4-3 中列示
- 配置更改事件，在第 4-31 页表 4-5 中列示。

如果启用跟踪并打开跟踪事件，则这些事件可以记录在跟踪文件中。其它模块可以提供按同样方式处理的新事件类型。用户定义的事件必须具有介于 UserEvent_Base (0x100000) 和 UserEvent_Top (0x1FFFFF) 之间的值。

您可以通过安装事件处理程序来捕获事件 （请参阅第 4-37 页*事件处理程序*）。您可以通过调用 ARMulif_RaiseEvent() 来生成事件 （请参阅第 4-32 页 *ARMulif_RaiseEvent*）。

**表 4-1 来自 MMU 和高速缓存 （不在 StrongARM-110 上）的事件**

| 事件名 | 字 1 | 字 2 | 事件编号 |
|---|---|---|---|
| MMUEvent_DLineFetch | 丢失地址 | 受影响的地址 | 0x10001 |
| MMUEvent_ILineFetch | 丢失地址 | 受影响的地址 | 0x10002 |
| MMUEvent_WBStall | 写物理地址 | 写缓冲内的字数 | 0x10003 |
| MMUEvent_DTLBWalk | 丢失地址 | 受影响的地址 | 0x10004 |
| MMUEvent_ITLBWalk | 丢失地址 | 受影响的地址 | 0x10005 |
| MMUEvent_LineWB | 丢失地址 | 受影响的地址 | 0x10006 |
| MMUEvent_DCacheStall | 导致停止的地址 | 获取的地址 | 0x10007 |
| MMUEvent_ICacheStall | 导致停止的地址 | 获取的地址 | 0x10008 |

**表 4-2 来自 ARM 处理器内核的事件**

| 事件名 | 字 1 | 字 2 | 事件编号 |
|---|---|---|---|
| CoreEvent_Reset | - | - | 0x1 |
| CoreEvent_UndefinedInstr | pc 值 | 指令 | 0x2 |
| CoreEvent_SWI | pc 值 | SWI 编号 | 0x3 |
| CoreEvent_PrefetchAbort | pc 值 | - | 0x4 |
| CoreEvent_DataAbort | pc 值 | 中止的地址 | 0x5 |
| CoreEvent_AddrExceptn | pc 值 | 中止的地址 | 0x6 |
| CoreEvent_IRQ | pc 值 | - | 0x7 |
| CoreEvent_FIQ | pc 值 | - | 0x8 |
| CoreEvent_Breakpoint | pc 值 | RDI_PointHandle | 0x9 |
| CoreEvent_Watchpoint | pc 值 | 观察地址 | 0xA |
| CoreEvent_IRQSpotted | pc 值 | - | 0x17 |
| CoreEvent_FIQSpotted | pc 值 | - | 0x18 |
| CoreEvent_ModeChange | pc 值 | 新模式 | 0x19 |
| CoreEvent_Dependency | pc 值 | 互锁寄存器位掩码 | 0x20 |

**表 4-3 来自预取单元的事件 （只适用于 ARM810）**

| 事件名 | 字 1 | 字 2 | 事件编号 |
|---|---|---|---|
| PUEvent_Full | 下一个 pc 值 | - | 0x20001 |
| PUEvent_Mispredict | 分支的地址 | - | 0x20002 |
| PUEvent_Empty | 下一个 pc 值 | - | 0x20003 |

**表 4-4 调试事件**

| 事件名 | 字 1 | 字 2 | 事件编号 |
|---|---|---|---|
| DebugEvent_InToDebug | - | - | 0x40001 |
| DebugEvent_OutOfDebug | - | - | 0x40002 |
| DebugEvent_DebuggerChangedPC | pc | - | 0x40003 |

**表 4-5 配置事件**

| 事件名 | 字 1 | 字 2 | 事件编号 |
|---|---|---|---|
| ConfigEvent_AllLoaded | - | - | 0x50001 |
| ConfigEvent_Reset | - | - | 0x50002 |
| ConfigEvent_VectorsLoaded | - | - | 0x50003 |
| ConfigEvent_EndiannessChanged | 1 （大端）<br>或<br>2 （小端） | - | 0x50005 |

### 4.6.1 ARMulif_RaiseEvent

此函数可调用事件。事件被传送至用户提供的事件处理程序。

#### 语法

```
void ARMulif_RaiseEvent(RDI_ModuleDesc *mdesc, ARMword event,
                        ARMword data1, ARMword data2)
```

其中：

*mdesc* 是内核的句柄。

*event* 是在第 4-29 页表 4-1、第 4-30 页表 4-2、第 4-30 页表 4-3 或第 4-31 页表 4-4 中定义的其中一个事件编号。

*data1* 是事件的第一个字。

*data2* 是事件的第二个字。

## 4.7 处理程序

ARMulator 可用于在某些状态值更改时回调模型。您可以通过安装相关的 *event handler* 来进行此操作。

如果您要将其用于自己的模型中，则必须提供事件处理程序的执行说明。有关信息，请参阅 ARM 作为实例提供的模型中的执行说明。

您可以将事件处理程序用于避免在每次访问时检查状态值。例如，外围设备模型预期将显示 ARM 内核，以及 ARM 处理器**大端**信号值的以正确字节顺序排列的数据。外围设备模型可以连接至 `EventHandler()`，以便在信号更改时得到通知（参见第 4-37 页*事件处理程序*）。

  

### 4.7.1 异常处理程序

只要 ARM 处理器出现异常，系统均会调用此事件处理程序。

#### 语法

```
typedef unsigned GenericCallbackFunc(void *handle, void *data)
```

其中：

*handle* 是传送至 ARMulif_InstallExceptionHandler 的句柄。

*data* 必须造型为 (ARMul_Event *) 并包含：

((ARMul_Event *)data)->event
: 是导致异常的核心事件 （请参阅第 4-30 页表 4-2）。

((ARMul_Event *)data)->data1
: 是异常的硬件向量地址。

((ARMul_Event *)data)->data2
: 是导致异常的指令。

#### 用法

作为实例，可由操作系统模型用于截取和模拟 SWI。如果安装的处理程序返回非零值，则 ARM 不会产生异常 （异常被忽略）。

**注**

如果处理器处于 Thumb 状态，则会提供效果等同的 ARM 指令。

要安装异常处理程序，可使用：

```
int ARMulif_InstallExceptionHandler(RDI_ModuleDesc *mdesc,
                                    GenericCallbackFunc *func, void *handle)
```

要删除异常处理程序，可使用：

```
int ARMulif_RemoveExceptionHandler(RDI_ModuleDesc *mdesc,
                                   GenericCallbackFunc *func, void *handle)
```

### 4.7.2 未知的 RDI 信息处理程序

如果 ARMulator 自己无法处理 RDI_InfoProc 请求，则会调用未知的 RDI 信息函数。它将返回 RDIError 值。此函数可由扩展 ARMulator 和调试器之间的 RDI 接口的模型使用。例如配置器模块 （在 profiler.c 中）可提供 RDIProfile 信息调用。

#### 语法

```
typedef int RDI_InfoProc(void *handle, unsigned type,
                         ARMword *arg1, ARMword *arg2)
```

其中：

*handle* 是传送至 ARMulif_InstallUnkRDIInfoHandler 的句柄。

*type* 是 RDI_InfoProc 子代码。它们在 rdi_info.h 中定义。有关实例，请参阅下面的内容。

*arg1/arg2* 是从 ARMulator 传送至处理程序的变量。

#### 用法

如果返回一个值而不是 RDIError_UnimplementedMessage， ARMulator 可停止调用 RDI_InfoProc() 函数。

以下代码是可以指定为 *type* 的 RDI_InfoProc 子代码实例：

RDIInfo_Target

可使模型声明如何扩展目标功能。例如， profiler.c 可截取此调用以设置 RDITarget_CanProfile 标记。

RDIInfo_SetLog

可环绕传送，以便模型在打开和关闭记录信息之间切换。例如，tracer.c 可使用此调用从 RDI 记录层值 （$rdi_log、或 RealView Debugger 中的 @mdebug_rdi_log、或调试器的相当值）的位 4 打开和关闭跟踪功能。

RDIRequestCyclesDesc

可使模型扩展由 $statistics 中的调试器 （或调试器中的相当值）提供的计数器列表。模型调用 ARMul_AddCounterDesc() （请参阅第 4-41 页*通用函数*）以依次声明每个计数器。同时模型也可以诱捕 RDICycles RDI 信息调用，这一点很重要。

RDICycles　通过捕获 RDIRequestCyclesDesc 来声明统计计数器的模型同时还必须对 RDICycles 作出响应，其方式是按其被声明时的相同顺序，为每个计数器依次调用 ARMul_AddCounterValue()（请参阅第 4-41 页*通用函数*）。

以上 RDI 信息调用已由 ARMulator 处理，并仅传送用于信息或以便模型可以将信息添加至回复。模型必须始终响应这些具有 RDIError_UnimplementedMessage 的消息，以便消息即使在模型已作出响应的情况下也可以被传递。

要安装处理程序，可使用：

```
int ARMulif_InstallUnkRDIInfoHandler(RDI_ModuleDesc *mdesc,
                                     RDI_InfoProc *func, void *handle)
```

要删除处理程序，可使用：

```
int ARMulif_RemoveUnkRDIInfoHandler(RDI_ModuleDesc *mdesc,
                                    RDI_InfoProc *func, void *handle)
```

### 实例

随 ARMulator 提供的 semihost.c 模型使用 UnkRDIInfoUpcall() 与调试器进行交互：

RDIErrorP　可将在 ARMulator 下运行的程序所生成的错误返回至调试器。

RDISet_Cmdline　查找调试器为程序设置的命令行。

RDIVector_Catch　截取硬件向量。

### 4.7.3 事件处理程序

此处理程序可捕获 ARMulator 事件 （请参阅第 4-29 页*事件*）。

#### 语法

```
typedef unsigned GenericCallbackFunc(void *handle, void *data)
```

其中：

*handle* 是传送至 ARMulif_InstallEventHandler 的句柄。

*data* 必须传递为 (ARMul_Event *) 并包含：

((ARMul_Event *)data)->event

是在第 4-29 页表 4-1、第 4-30 页表 4-2 和第 4-30 页表 4-3 中定义的其中一个事件编号。

((ARMul_Event *)data)->addr1

是事件的第一个字。

((ARMul_Event *)data)->addr2

是事件的第二个字。

#### 用法

要安装处理程序，可使用：

```
void *ARMulif_InstallEventHandler(RDI_ModuleDesc *mdesc, uint32 events,
                                  GenericCallbackFunc *func, void *handle)
```

为 *events* 指定以下一个或更多个函数：

- CoreEventSel
- MMUEventSel
- PUEventSel
- DebugEventSel
- TraceEventSel
- ConfigEventSel。

要删除处理程序，可使用：

```
int ARMulif_RemoveEventHandler(RDI_ModuleDesc *mdesc, void *node)
```

#### 处理程序安装实例

```
ARMulif_InstallEventHandler(mdesc, CoreEventSel | ConfigEventSel, func, handle)
```

## 4.8 存储器访问函数

存储器系统可由外围设备模型使用一系列读写存储器的函数进行探测。这些函数均可在总线上不加插额外周期的情况下访问存储器。如果您的模型在总线上加插周期，则必须将其自身作为存储器模型安装，可能安装在内核和实际存储器模型之间。

**注**

无法判断这些调用是否会导致数据出错。

### 4.8.1 从指定的地址读取

以下函数可在特定的地址返回字、半字或字节。每个函数均可在总线上不加插额外周期的情况下访问存储器。

#### 语法

```
ARMword ARMulif_ReadWord(RDIModuleDesc *mdesc, ARMword address)

ARMword ARMulif_ReadHalfword(RDIModuleDesc *mdesc, ARMword address)

ARMword ARMulif_ReadByte(RDIModuleDesc *mdesc, ARMword address)
```

其中：

*mdesc* 是内核的句柄。

*address* 是从其中读取字、半字或字节的模拟存储器中的地址。

#### 返回

此函数可返回适当的字、半字或字节。

### 4.8.2 写入指定的地址

以下函数可在指定的地址写入特定的字、半字或字节。每个函数均可在总线上不加插额外周期的情况下访问存储器。

#### 语法

**void** ARMulif_WriteWord(RDIModuleDesc **mdesc*, ARMword *address*, ARMword *data*)

**void** ARMulif_WriteHalfword(RDIModuleDesc **mdesc*, ARMword *address*, ARMword *data*)

**void** ARMulif_WriteByte(RDIModuleDesc **mdesc*, ARMword *address*, ARMword *data*)

其中：

*mdesc* 是内核的句柄。

*address* 是要写入其中的模拟存储器中地址。

*data* 是要写的字或字节。

## 4.9 事件调度函数

以下函数可使您调度或删除事件：

- *ARMulif_ScheduleTimedFunction*
- *ARMulif_DescheduleTimedFunction*。

### 4.9.1 ARMulif_ScheduleTimedFunction

此函数可使用存储器系统周期调度事件。它可以在将来指定的周期号上调用函数。

#### 语法

```
void *ARMulif_ScheduleTimedFunction(RDI_ModuleDesc *mdesc,
                                    ARMul_TimedCallback *tcb)
```

其中：

*mdesc* 是内核的句柄。

*tcb* 是供您从基于指定存储器周期的事件发生时删除函数使用的句柄。

**注**

只能在指定周期之后的第一个指令边界上调用此函数。

### 4.9.2 ARMulif_DescheduleTimedFunction

ARMul_DescheduleTimedFunction() 可删除基于预定的存储器周期的事件。

#### 语法

```
unsigned ARMulif_DescheduleTimedFunction(RDI_ModuleDesc *mdesc, void *tcb);
```

其中：

*mdesc* 是内核的句柄。

*tcb* 是 ARMulif_ScheduleTimedFunction 在首次安装事件时提供的句柄。

## 4.10 通用函数

以下是通用的 ARMulator 函数。它们包括访问处理器属性、添加计数器说明和数值、停止 ARMulator 以及执行代码的函数：

### 4.10.1 ARMul_AddCounterDesc

ARMul_AddCounterDesc() 函数可将新计数器添加至 $statistics （或调试器中的相当值）。

#### 语法

```
int ARMul_AddCounterDesc(void *handle, ARMword *arg1, ARMword *arg2,
                         const char *name)
```

其中：

*handle* 不再使用。

*arg1*/*arg2* 是传送至 UnkRDIInfoUpcall() 的变量。

*name* 是为统计计数器命名的字符串。字符串必须小于 32 字符长。

#### 返回

此函数可返回以下其中之一：

- RDIError_BufferFull
- RDIError_UnimplementedMessage。

#### 用法

当 ARMulator 收到来自调试器的 RDIRequestCycleDesc() 调用时，如果它希望提供任何统计数据计数器，则会使用 UnkRDIInfoUpcall() （请参阅第 4-35 页*未知的 RDI 信息处理程序*）依次询问每个模型。每个模型均可通过调用具有传送至 UnkRDIInfoUpcall() 的变量的 ARMul_AddCounterDesc() 进行响应。

所有统计数据计数器必须是 32 或 64 位字并可单调递增。也就是说，统计值必须随时间递增。这是调试器计算 $statistics_inc 时所采用方法的要求。

### 4.10.2 ARMul_AddCounterValue

此函数可为模型提供设备，以便其供应调试器的统计数据进行显示。

#### 语法

```
int ARMul_AddCounterValue(void *handle, ARMword *arg1, ARMword *arg2, bool is64,
                          const ARMword *counter)
```

其中：

*handle* 不再使用。

*arg1*/*arg2* 是传送至 UnkRDIInfoUpcall() 的变量。

*is64* 表示计数器是组成 64 位计数器的一个 32 位字对 （最低有效字在前），还是单个 32 位值。这可使用模型提供全 64 位计数器。

*counter* 是计数器当前值的指针。

#### 返回

此功能始终返回 RDIError_UnimplementedMessage。

#### 用法

您的模型必须调用此函数，或从其 UnkRDIInfoUpcall() 处理程序中调用 ARMul_AddCounterValue64。除了计数器的字顺序外， ARMul_AddCounterValue64 与 ARMul_AddCounterValue 完全相同。

### 4.10.3 ARMul_AddCounterValue64

此函数可为模型提供设备，以便其供应调试器的统计数据进行显示。

#### 语法

```
int ARMul_AddCounterValue64(void *handle, ARMword *arg1, ARMword *arg2,
                            const uint64 counterval)
```

其中：

*handle* 不再使用。

*arg1*/*arg2* 是传送至 UnkRDIInfoUpcall() 的变量。

*counterval* 计数器的当前值。

#### 返回

此功能始终返回 RDIError_UnimplementedMessage。

#### 用法

您的模型必须调用此函数，或从其 UnkRDIInfoUpcall() 处理程序中调用 ARMul_AddCounterValue。除了根据主机系统 endian 配置不同外，此函数与 ARMul_AddCounterValue 完全相同。

### 4.10.4 ARMul_BusRegisterPeripFunc

外围设备模型必须调用此函数来向 ARMulator 进行注册。这可使 ARMulator 在任何时候都可以通过访问属于外部的存储器地址来调用此外设模型。

#### 语法

```
int ARMul_BusRegisterPeripFunc(enum BusRegAct act,
                               ARMul_BusPeripAccessRegistration *breg);
```

其中：

*act* 是所需的操作。*act* 必须为以下其中一个值：`insert` 或 `remove`。

*breg* 是包含信息的 ARMulator 结构。您可以通过调用 `ARMulif_ReadBusRange` 来获得此结构 （请参阅第 4-51 页 *ARMulif_ReadBusRange*）。

*breg* 是 `ARMul_BusPeripAccessRegistration` 类型的一种结构 （有关详情，请参阅 *ARMul_BusPeripAccessRegistration*）。

#### ARMul_BusPeripAccessRegistration

文件 `armul_bus.h` 中声明了这种结构和类型，该文件位于：

*ExtensionKit_path*`/1.3/Final/`*platform*`/armulif`

在该路径中：

- *platform* 是：
  - Windows 的 `win_32-pentium`
  - Linux 的 `linux-pentium`
  - Solaris 的 `solaris-sparc`。
- 对于 Windows，请用 \ 替换 /。

声明如下：

```
typedef struct ARMul_BusPeripAccessRegistration {
    ARMul_BusPeripAccessFunc *access_func;
    void *access_handle;
    uint32 capabilities; /* See PeripAccessCapability_* below */
    struct ARMul_Bus *bus;
    /* 0=> normal peripheral, earlier in list than anything it
     * overlaps with. */
    unsigned priority;
     /* 0..100%
      * A higher number will be placed earlier in the list than
      * anything that it doesn't overlap with and has a lower access_frequency.
      */
```

```
    unsigned access_frequency;
    unsigned addr_size; /* Number of elements in range[] */
    AddressRange range[1];
} ARMul_BusPeripAccessRegistration;
```

其中：

*access_func* 指定地址范围中存储器访问需调用的函数的指针。

*access_handle* *access_func* 的目标数据的指针。

*capabilities* 请参阅 *PeripAccessCapability*。

*bus* 是 ARMulif_QueryBus 返回的内容。请不要更改该内容。

*priority* 可使用此字段为外围设备分配优先级。零是最高的优先级。如果外围设备具有重叠的地址范围，则首先访问优先级最高的外围设备。只有当优先级较高的外围设备在未处理调用的情况下返回时才能访问优先级较低的外围设备。

*access_frequency* 可使用此字段通知 ARMulator 您希望经常访问的外围设备。这样 ARMulator 可以更有效地访问外围设备。指定百分比为 0% 至 100% 的范围内的频率。

*addr_size* 此内容有待继续扩展。1 表示 32 位地址。这是当前唯一受支持的地址大小。

*range* 外围设备所占用的地址范围。

### PeripAccessCapability

此参数定义了外围设备的功能。它是各种单独功能值的总和 （请参阅第 4-47 页表 4-6）。

例如：

- 值 0x20020 表示此外围设备可以处理字数据访问 （但不能处理字节、半字或双字数据访问），另外可识别 **Endian** 信号。此值预定义为 PeripAccessCapability_Minimum。
- 值 0x20038 表示此外围设备可以处理字节、半字或字数据访问 （但不能处理双字数据访问），另外可识别 **Endian** 信号。此值预定义为 PeripAccessCapability_Typical。

  

**表 4-6 外围设备访问功能**

| 功能 | 预定义名称 | 值 |
|---|---|---|
| 字节 | PeripAccessCapability_Byte | 0x8 |
| 半字 | PeripAccessCapability_HWord | 0x10 |
| 字 | PeripAccessCapability_Word | 0x20 |
| 双字 | PeripAccessCapability_DWord | 0x40 |
| 外围设备接受空闲周期 | PeripAccessCapability_Idles | 0x10000 （**未指定长度**） |
| 外围设备识别 **Endian** 信号 | PeripAccessCapability_Endian | 0x20000 （**未指定长度**） |
| 外围设备识别字节道 | PeripAccessCapability_Bytelane | 0x40000 （**未指定长度**） |

### 4.10.5 ARMulif_CoreCycles

此函数返回主流水线已步进的次数 （需要内核模块的支持）。

**注**

对 ARM9 模型来说是门控时钟。此时钟可以通过存储器或互锁设备来安装。

#### 语法

```
ARMTime ARMulif_CoreCycles(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

#### 返回

返回主流水线已步进的次数 （需要内核模块的支持）。

### 4.10.6 ARMulif_CPUCycles

此函数返回以 CPU 速度单位表示的时间。

**注**

仅在支持基于 ARM10 和 XScale 的模型。

#### 语法

```
ARMTime ARMulif_CpuCycles(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

#### 返回

返回以 CPU 速度单位表示的时间 （需要内核模块的支持）。

### 4.10.7 ARMulif_EndCondition

此函数返回传送至 ARMulif_StopExecution 的 *reason*。

#### 语法

**unsigned** ARMulif_EndCondition(RDI_ModuleDesc **mdesc*)

其中：

*mdesc* 是内核的句柄。

### 4.10.8 ARMulif_GetCoreClockFreq

此函数返回单位为赫兹的 CPUSPEED 。

#### 语法

ARMTime ARMulif_GetCoreClockFreq(RDI_ModuleDesc **mdesc*)

其中：

*mdesc* 是内核的句柄。

### 4.10.9 ARMulif_InstallHourglass

使用此函数安装从 ARMulator 到用户模型的沙漏计时回调。

#### 语法

```
void *ARMulif_InstallHourglass(RDI_ModuleDesc *mdesc,
                               armul_Hourglass *newHourglass, void *handle);
```

其中：

*mdesc* 是内核的句柄。

*newHourglass* armul_Hourglass 类型的函数。您可以使用以下路径，在文件 armul_types.h 中找到 armul_Hourglass 的原型：

*ExtensionKit_path*/1.3/Final/*platform*/armulif

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

*handle* 函数 *newHourglass* 所需的数据的指针。

#### 用法

安装沙漏计时后，每次执行指令后 ARMulator 均将回调您的模型。

#### 返回

此函数将返回一个句柄，以便您的模型用于删除沙漏回调。

  

### 4.10.10 ARMulif_ReadBusRange

您必须提供一个 *breg* 结构以注册外围设备。调用此函数以初始化该结构中的字段。

#### 语法

```
int ARMulif_ReadBusRange(struct RDI_ModuleDesc *mdesc,
                         struct RDI_HostosInterface const *hostif,
                         toolconf config,
                         struct ARMul_BusPeripAccessRegistration *breg,
                         uint32 default_base, uint32 default_size,
                         char const *default_bus_name);
```

其中：

*mdesc* 是内核的句柄。

*hostif* 是主机接口的句柄。

*config* 是传送至 BEGIN_INIT 中模型的配置。

*breg* 是包含信息的 ARMulator 结构。在 registerPeripFunc() 中需用到此结构 （请参阅第 4-45 页 *ARMul_BusRegisterPeripFunc*）。

有关此结构的详情，请参阅以下路径中的 armulbus.h：

*ExtensionKit_path*/1.3/Final/*platform*/armulif

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

*default_base* 用于外围设备的缺省基址。如果 *config* 中未包含用于外围设备的基址，则会使用此地址。

*default_size* 用于外围设备的缺省存储器区域大小。如果 *config* 中未包含用于外围设备的存储器大小，则会使用此大小。

*default_bus_name*

是字符串的指针。如果用于此外围设备的配置参数中未发现总线名称，则会使用此字符串 （例如在 .dsc 或 .ami 文件中）。

### 4.10.11 ARMulif_RemoveHourglass

使用此函数删除沙漏回调。

#### 语法

```
int ARMulif_RemoveHourglass(RDI_ModuleDesc *mdesc, void *node);
```

其中：

*mdesc* 是内核的句柄。

*node* 是 ARMulif_InstallHourglass 返回的句柄。

### 4.10.12 ARMulif_StopExecution

此函数在当前指令结束时停止模拟器的执行，并给出原因代码。

#### 语法

```
void ARMulif_StopExecution(RDI_ModuleDesc *mdesc, unsigned reason)
```

其中：

*mdesc* 是内核的句柄。

*reason* 是一个 RDIError 错误值。调试器将解释 *reason* 并产生一定的消息。可能的错误有：

RDIError_NoError
: 程序自然停止。

RDIError_BreakpointReached
: 停止条件为断点。

RDIError_WatchPointReached
: 停止条件为观察点。

RDIError_UserInterrupt
: 用户中断执行。

### 4.10.13 ARMulif_Time

此函数返回自系统重置起执行的存储器周期数。

#### 语法

```
ARMTime ARMulif_CoreCycles(RDI_ModuleDesc *mdesc)
```

其中：

*mdesc* 是内核的句柄。

#### 返回

此函数返回自系统重置起执行的周期总数。

### 4.10.14 Hostif_RaiseError

一些初始化和安装函数可以返回 ARMul_Error 类型的错误。这些错误必须通过 Hostif_RaiseError() 传送。此函数与 printf 函数相似，用于格式化与 ARMul_Error 错误代码相关的错误消息。

Hostif_RaiseError 仅打印错误消息。调用此函数后，模型肯定会返回适当的错误，例如 RDIError_UnableToInitialise。

Hostif_RaiseError 只能在初始化过程中使用。

#### 语法

```
void Hostif_RaiseError(const struct RDI_HostosInterface *hostif,
                       const char *format, ...)
```

其中：

*hostif* 是主机接口的句柄。

*format* 是要格式化的错误消息的错误代码。

*...* 是 printf 样式的各种类型说明符。

## 4.11 访问调试器

本部分介绍了可用于访问调试器的输入、输出和 RDI 函数。

一些函数用于在主机调试器中显示消息。在调试器中，它们将消息显示在相关的窗口：

- 第 4-54 页 *Hostif_ConsolePrint*
- 第 4-55 页 *Hostif_ConsoleRead*
- 第 4-55 页 *Hostif_ConsoleReadC*
- 第 4-56 页 *Hostif_ConsoleWrite*
- 第 4-56 页 *Hostif_DebugPause*
- 第 4-56 页 *Hostif_DebugPrint*
- 第 4-57 页 *Hostif_PrettyPrint*
- 第 4-57 页 *Hostif_WriteC*。

所有这些函数都将下面的值作为第一个参数：

```
const struct RDI_HostosInterface *hostif
```

此值可用于模型的状态数据结构中，在 `BEGIN_STATE_DECL()` 和 `END_STATE_DECL()` 宏之间定义 （请参阅第 4-12 页*基本模型接口*）。

### 4.11.1 Hostif_ConsolePrint

此函数将格式字符串中指定的文本打印至 ARMulator 控制台。在 GUI 调试器中，该文本显示在相应的 I/O 窗口。

#### 语法

```
void Hostif_ConsolePrint(const struct RDI_HostosInterface *hostif,
                         const char *format, ...)
```

其中：

*hostif* 是主机接口的句柄。

*format* 是 printf 样式的格式化输出字符串的指针。

*...* 是与 *format* 相关的各个参数数量。

**注**

使用 `Hostif_PrettyPrint()` 显示启动信息。

### 4.11.2 Hostif_ConsoleRead

此函数从 ARMulator 控制台读取字符串。到达新行或缓冲器的末尾时停止读取。

#### 语法

```
char *Hostif_ConsoleRead(const struct RDI_HostosInterface *hostif,
                         char *buffer, int len)
```

其中：

*hostif* 是主机接口的句柄。

*buffer* 是存放字符串的缓冲器的指针。

*len* 是缓冲器的最大长度。

#### 返回

此函数返回缓冲器指针，或者在出错时或文件末尾返回 NULL 。

缓冲器最多可容纳 *len*-1 个字符，以零终结。如果读取到新行，则包含在零前面的字符串中。

### 4.11.3 Hostif_ConsoleReadC

此函数从 ARMulator 控制台读取字符。

#### 语法

```
int Hostif_ConsoleReadC(const struct
                        RDI_HostosInterface *hostif)
```

其中：

*hostif* 是主机接口的句柄。

#### 返回

此函数返回读取的字符的 ASCII 值或 EOF。

### 4.11.4 Hostif_ConsoleWrite

此函数将字符串写入 ARMulator 控制台。

#### 语法

```
int Hostif_ConsoleWrite(const struct RDI_HostosInterface *hostif,
                        const char *buffer, int len)
```

其中：

*hostif* 是主机接口的句柄。

*buffer* 存放以零终结的字符串的缓冲器指针。

*len* 是缓冲器的长度。

#### 返回

此函数返回实际写入的字符数。如果不发生错误，则为 *len*。

### 4.11.5 Hostif_DebugPause

此函数等待用户按任意键。

#### 语法

```
void Hostif_DebugPause(const struct RDI_HostosInterface *hostif)
```

其中：

*hostif* 是主机接口的句柄。

### 4.11.6 Hostif_DebugPrint

此函数将消息显示在 GUI 调试器的记录窗口或命令行调试器的控制台中。

#### 语法

```
void Hostif_DebugPrint(const struct RDI_HostosInterface *hostif,
                       const char *format, ...)
```

其中：

*hostif* 是主机接口的句柄。

*format* 是 printf 样式的格式化输出字符串的指针。

*...* 是与 *format* 相关的各个参数数量。

  

### 4.11.7 Hostif_PrettyPrint

此函数以与 Hostif_ConsolePrint() 相同的方式打印字符，另外还可执行行中断的检测以避免自动换行。可使用此函数显示启动信息。

#### 语法

```
void Hostif_PrettyPrint(const struct RDI_HostosInterface *hostif,
                        struct hashblk * /*toolconf*/ config,
                        const char *format, ...)
```

其中：

*hostif* 是主机接口的句柄。

*config* 模型的 toolconf 配置数据库的指针。此值可用于模型的状态数据结构中，在 BEGIN_STATE_DECL() 和 END_STATE_DECL() 宏之间定义（请参阅第 4-12 页*基本模型接口*）。

*format* 是 printf 样式的格式化输出字符串的指针。

*...* 是与 *format* 相关的各个参数数量。

### 4.11.8 Hostif_WriteC

此函数将字符写入 ARMulator 控制台。

#### 语法

```
void Hostif_ConsoleWriteC(const struct
                          RDI_HostosInterface *hostif, int c)
```

其中：

*hostif* 是主机接口的句柄。

*c* 要写入的字符。*c* 将转换为无符号的字符。

## 4.12 跟踪器

本部分介绍了跟踪器模块 tracer.c 提供的函数。

**注**

这些函数并未导出。如果您想在模型中使用这些函数，则必须将模型与 tracer.c 一起构建。

可通过编译 EXTERNAL_DISPATCH 定义的 tracer.c 来更改这些函数的缺省实施情况。

tracer.h 中说明了 Trace_State 和 Trace_Packet 的格式。

### 4.12.1 Tracer_Open

初始化跟踪器时会调用此函数。

#### 语法

```
unsigned Tracer_Open(Trace_State *ts)
```

#### 用法

tracer.c 中的实施步骤将从此函数打开输出文件并写入标题。

### 4.12.2 Tracer_Dispatch

对于每个指令、事件或存储器访问，系统在所有跟踪事件上均会调用此函数。

#### 语法

```
void Tracer_Dispatch(Trace_State *ts, Trace_Packet *packet)
```

#### 用法

在 tracer.c 中，此函数将信息包写入跟踪文件。

### 4.12.3 Tracer_Close

此函数在跟踪结束时调用。

#### 语法

```
void Tracer_Close(Trace_State *ts)
```

#### 用法

文件 tracer.c 使用此函数关闭跟踪文件。

### 4.12.4 Tracer_Flush

此函数在跟踪被禁用时调用。

#### 语法

```
extern void Tracer_Flush(Trace_State *ts)
```

#### 用法

文件 tracer.c 使用此函数将输出刷新至跟踪文件。

## 4.13 映射文件

模拟系统中的存储器类型和速度可在映射文件中详细说明。映射文件定义了附加存储器的区域数量，并且还对每个区域定义了以下内容：

- 该区域映射时的地址范围
- 以字节表示的数据总线宽度
- 存储器区域的访问时间。

调试器接受任何名字的映射文件。有关在调试会话中如何使用特定映射文件的详情，请参阅调试器说明文档。

为计算每个可能的存储器访问类型的等待状态数， ARMulator 使用映射文件中指定的访问次数和调试器的时钟频率 （请参阅调试器说明文档）。

另请参阅第 2-28 页*使用映射文件进行存储器建模*。

**注**

存储器映射文件说明了 peripherals.ami 中定义的存储器区域的特征 （请参阅第 4-64 页 *ARMulator 配置文件*）。.map 文件必须定义至少与在 peripherals.ami 中为堆和堆栈指定的区域大小相同并且处于相同位置的 rw 区域。否则，执行过程中可能会发生数据出错。

### 4.13.1 映射文件的格式

每行的格式为：

*start size name width access*{*} *read-times write-times*

其中：

*start* 是十六进制存储器区域的起始地址，例如 80000。

*size* 是十六进制存储器区域的大小，例如 4000。

*name* 是可用于在显示存储器访问统计数据时识别存储器区域的单个字。您可以使用任何名称。为提高存储器访问统计数据的可读性，请使用 SRAM、 DRAM 或 EPROM 等描述性名称。

*width* 以字节表示的数据总线宽度 （即 1 表示 8 位总线， 2 表示 16 位总线， 4 表示 32 位总线）。

  

*access* 说明可在该存储器区域上执行的访问类型：

r 只读。

w 只写。

rw 既可读也可写。

- 无法访问。任何访问均会导致数据或预取出错。

您可以在 *access* 上添加星号 (*)，说明它是基于 Thumb 的系统，该系统使用 32 位数据总线的存储器，但具有 16 位锁存器锁住了上部的 16 位数据，以便您直接从该锁存器中获取随后的 16 位循序存取。

*read-times*

以纳秒表示非连续的和连续的读取时间。先输入非连续的读取访问时间，然后输入一个斜线 (/)，再在斜线后输入连续的读取访问时间。如果省略斜线并仅使用一个数字，则表示非连续访问时间与连续访问时间相同。

**注**

为建立精确的真实设备模型，您可能需在对存储器芯片设定的读写时间上添加信号传输延迟 （20 到 30 纳秒）。

*write-times*

说明非连续和连续的写入时间。其格式与为读取时间设定的格式相同。

以下实例假定缺省的时钟速度为 20MHz。

### 实例 1

```
0 80000000 RAM 4 rw 135/85 135/85
```

此实例描述了一个系统，此系统具有从 0 到 0x7FFFFFFF 的 RAM 单个连续区域，并且该区域具有 32 位数据总线、读写访问功能、135 纳秒的非连续访问时间和 85 纳秒的连续访问时间。

### 实例 2

此实例描述了一个典型的嵌入式系统，具有 32KB 芯片上的存储器、16 位 ROM 和 32KB 的外部 DRAM：

```
00000000 8000 SRAM  4 rw   1/1     1/1
00008000 8000 ROM   2 r  100/100 100/100
00010000 8000 DRAM  2 rw 150/100 150/100
7FFF8000 8000 Stack 2 rw 150/100 150/100
```

存储器区域有：

- 从 0 到 0x7FFF 的快速区域，具有 32 位数据总线。此区域标记为 SRAM。
- 从 0x8000 到 0xFFFF 的速度较慢的区域，具有 16 位数据总线。此区域标记为 ROM 并包含映像代码。该区域为只读。
- 从 0x10000 到 0x17FFF 的 RAM 区域，用于映像数据。
- 从 0x7FFF8000 到 0x7FFFFFFF 的 RAM 区域，用于堆栈数据。堆栈指针被初始化为 0x80000000。

在最终的硬件中，两个截然不同的外部 DRAM 区域组合在一起。但这并不影响模拟的精确度。

为表示快速 （无等待状态）存储器，为 SRAM 区域设定的访问时间为 1 纳秒。实际上这表示每次访问需要 1 个时钟周期，因为 ARMulator 使此访问时间最大限度地靠近时钟周期。但是，将其指定为 1 纳秒可使相同的映射文件用于多个具有不同时钟速度的模拟。

**注**

为保证模拟的精确度，请确保存储器映像中描述了您模拟的映像可能访问的所有存储器区域。

---

为确保已描述您认为映像可能访问的所有存储器区域，您可以定义覆盖整个地址范围的单个存储器区域，作为映射文件的最后一行。例如，您可以在上述描述中添加以下行：

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

您可以使用 print $memory_statistics 命令 （或用于调试器的与 $memory_statistics 效果等同的命令）检测所需的存储器区域以外是否发生任何读或写操作。

**注**

虚拟存储器区域必须为映射文件的*最后*一个条目。

---

### 读取存储器统计数据

要读取存储器统计数据，请使用 print 命令，例如：

```
print $memory_statistics
```

print $memstats 是 print $memory_statistics 的缩写。

对于调试器，请使用与 $memstats 或 $memory_statistics 效果等同的命令。

**注**

$memstats 在 RealView Debugger 中不受支持。

实例 4-1 显示了给定的报告格式。

**实例 4-1**

```
address  name    W acc R(N/S)  W(N/S)    reads(N/S)   writes(N/S)  time (ns)
00000000 Dummy   4 -     1/1     1/1        0/0          0/0      0
7FFF8000 Stack   2 rw  150/100 150/100   9290/10590    4542/11688  8538300
00010000 DRAM    2 rw  150/100 150/100  18817/18      11031/140    8915800
00008000 ROM     2 r   100/100 100/100  48638/176292      0/0      44817000
00000000 SRAM    4 rw    1/1     1/1        0/0          0/0      0
```

实例 4-1 中的报告说明：

- ROM 访问对于此应用程序来说至关重要。可考虑使用速度更快的 ROM、使用可分段传输的 ROM 或增大 ROM 的宽度 （32 位）。
- 0x0 处无需使用 SRAM。可考虑将堆栈或其它数据定位在 0x0 处。

## 4.14 ARMulator 配置文件

本节包括以下小节：



ARMulator 配置文件 （.ami 文件）是 ToolConf 文件。请参阅第 4-69 页 *ToolConf*。

视您的系统而定，这些文件位于以下路径：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

在该路径中：

- *platform* 是：
  - Windows 的 win_32-pentium
  - Linux 的 linux-pentium
  - Solaris 的 solaris-sparc。
- 对于 Windows，请用 \ 替换 /。

您可以复制 .ami 文件并编辑它们。为您的新 .ami 文件设置合适的目录，并将其路径添加至 ARMCONF 环境变量。在 ARMCONF 中，请确保您的目录出现在 ARMulator 目录的前面：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

在缺省情况下，此目录具有以下 .ami 文件：

- bustypes.ami
- default.ami
- example1.ami
- peripherals.ami
- processors.ami
- vfp.ami。

ARMulator 将加载它在环境变量 ARMCONF 中任何指定路径上发现的所有 .ami 文件。最初被设定为指向：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

如果两个文件中指定的配置不同，则使用第一个指定值。如果 ARMCONF 中有多个目录， ARMulator 将按 .ami 文件出现在列表中的顺序从目录加载这些文件。ARMulator 按无法预见的顺序从每个目录加载 .ami 文件。

### 4.14.1 预定义标签

读取 .ami 文件之前， ARMulator 将根据您指定给调试器的设置自行创建若干个标签。这些标签在表 4-7 中列示。 .ami 文件中的预处理指令将使用这些标签控制配置。

**表 4-7 ARMulator 预定义的标签**

| 标签 | 说明 |
|---|---|
| CPUSpeed | 设置为在 GUI 调试器的配置窗口或命令行调试器的 -clock 命令行选项中设定的速度。例如 CPUSpeed=30MHz。 |
| FCLK | 设置为与 CPUSpeed 相同的值 （如果该值不为零）。如果 CPUSpeed 为零则不会设置。 |
| MCLK | 为非高速缓存内核设置为与 FCLK 相同的值。<br>为高速缓存内核设置为 FCLK/MCCFG。 |
| ByteSex | 如果从调试器指定了 bytesex，则设置为 L 或 B 。否则不设置。 |
| FPE | 从调试器设置为 True 或 False 。 |

### 4.14.2 处理器

处理器区域是子 ToolConf 数据库 （请参阅第 4-69 页 *ToolConf*）。它具有 ARMulator 支持的处理器完整列表。此列表是您的调试器中处理器列表和命令行调试器的 -processor 选项所接受的变量列表的基础。

您可以在此列表中添加各种处理器，以达到在定义中包含特定存储器模型等目的。例如，请查看位于以下路径的 example1.ami 文件：

*install_directory*/RVARMulator/ARMulator/1.3/release/*platform*

Default 将指定使用的处理器 （如果未指定其它任何处理器）。Processors 区域中的其它每个条目都是处理器的名称。

第 4-66 页的实例 4-2 声明了两种处理器，即 TRACED_ARM10 和 PROFILED_ARM7。在此实例中， MCCFG 是处理器上的时钟频率与外部总线上的时钟频率之比。

**实例 4-2 toolconf 文件中的处理器**

```
{Processors

  {TRACED_ARM10=ARM10200E

    ;CPUSPEED=400MHz

    ;Memory clock divisor.
    ;(The AHB runs this many times slower than the core.)
    MCCFG=4

    {Flatmem
      {Peripherals
        {Tracer=Default_Tracer
          ;; Output options - can be plaintext to file, binary to file or to
          ;; RDI log window. (Checked in the order RDILog, File, BinFile.)
          RDILog=False
          File=armul.trc
          BinFile=armul.trc
          ;; Tracer options - what to trace
          TraceInstructions=True
          TraceRegisters=False
          TraceMemory=True
          TraceEvents=False
          ;; Flags - disassemble instructions; start up with tracing enabled.
          Disassemble=True
          StartOn=True
        }
      }
    }

  ;End TRACED_ARM10
  }

  {PROFILED_ARM7=ARM720T
    {Flatmem
      {Peripherals
        {Profiler=Default_Profiler
        }
      }
    }
  }

;End Processors
}
```

### 查找选定处理器的配置

ARMulator 使用以下算法查找选定处理器的配置：

1. 将当前区域设置为 Processors。
2. 在当前区域中找到选定的处理器。
3. 如果标签具有子标签，则该子标签是所需的配置。

### 添加不同的处理器模型

假设您已创建了一个名为 MyASIC 的存储器模型，用于与 ARM7TDMI 处理器内核结合以组建一个名为 ARM7TASIC 的新微控制器。为了能从调试器选择此存储器模型，请添加在 example1.ami 上建模的 .ami 文件。

## 4.14.3 更改可综合处理器的高速缓存或 TCM 的大小

要更改可综合处理器的高速缓存或 TCM 的大小，请复制 processors.ami 文件，然后将副本放入适当的目录中 （请参阅第 4-64 页 *ARMulator 配置文件*）并进行编辑。

例如，要将 ARM946E-S 的两个高速缓存均更改为 8KB：

1. 在 processors.ami 文件的副本中找到以下几行：

```
{ARM946E-S=ARM946E-S-REV1
}
```

2. 插入几行，使此段变为：

```
{ARM946E-S=ARM946E-S-REV1
ICache_Lines=256
DCache_Lines=256
}
```

此操作将覆盖 armulate.dsc 中相应的行。

**注意**

如果执行此更改，从 ARM946E-S 继承属性的任何内核也会受到影响，这些内核包括 ARM946E-S-ETM-(L)、 ARM946E-S-ETM-(M) 或 ARM946E-S-ETM-(S) 等。

未从 ARM946E-S 继承属性的内核不会受到影响，这些内核包括 ARM946E-S-REV0 或 ARM946E-S-REV1 等。

如果您想更改处理器的高速缓存或 TCM 大小，并且该处理器的 processors.ami 中还没有程序段，则您可以添加一个程序段。例如，要将 ARM926EJ-S 的指令 RAM 大小从 64KB 更改为 32KB：

1. 打开 processors.ami 文件的副本。
2. 插入一个新程序段：

```
{ARM926EJ-S=Processors_Common_ARMULATE
IRamSize=0x8000
}
```

此操作将覆盖 armulate.dsc 中相应的行。

文件中未指定的任何详细信息将保持不变，与在 armulate.dsc 中指定的内容相同。

## 4.15 ToolConf

本节包括以下小节：



### 4.15.1 Toolconf 概述

ToolConf 是 ARMulator 中的模块。 ToolConf 文件是树结构的数据库，由标签和值对组成。标签和值都是字符串，并且通常不区分大小写。 ToolConf 文件是 .ami 或 .dsc 类型的文件。

您可以从 ToolConf 数据库查找与标签相关的值，或者添加或更改值。

如果标签被多次赋值，则使用第一个值。

### 4.15.2 文件格式

以下是典型的 ToolConf 数据库行：

```
TagA=ValueA
TagA=NewValue
Othertag
Othertag=Othervalue
;; Lines starting with ; (semicolon) are comments.
; Tag=Value
```

第一行在 ToolConf 中创建一个名为 TagA 的标签，该标签的值为 ValueA。

第二行不起作用，因为 TagA 已具有一个值。

第三行创建一个名为 Othertag 的标签，该标签不具有值。

第四行将值 Othervalue 赋给 Othertag。

数据库行的开头、标签内、值内、标签或值与 “=”号之间不允许有空格。

按照惯例，普通注释以两个分号开头。以一个分号开头的行通常是被注释禁止行。您可以注释禁止一行来禁用它，或者不注释一个注释禁止行来启用它。

注释必须自成一行。

  

### 文件标题

如果您添加任何 ToolConf 文件，文件的第一行必须是：

```
;; ARMulator configuration file type 3
```

ARMulator 将忽略所有不以此标题开头的 .ami 或 .dsc 文件。

### 树结构

每个标签可具有其它与之相关的 ToolConf 数据库，称为它的子库。系统在子库中查找标签时，如果在子库中未找到标签，则会继续在父库中搜索，如果仍然找不到，则在父库的父库中查找，以此类推，直到找到标签为止。

这意味着子库仅包含其值与父库中相同标签的值不同的标签。

如果为同一个父库多次指定子数据库，则这些子数据库会合并起来。

### 指定子数据库

在 ToolConf 数据库中指定子库的方法有两种。

以下一种方法更适合于指定大型子库：

```
{ TagP=ValueP
TagC1=ValueC1
TagC2=ValueC2
}
```

这将创建一个名为 TagP 的标签，此标签具有值 ValueP 和一个子数据库。子库中的两个标签已被赋值。

另一种方法适合于指定小型子库：

```
TagP:TagC=ValueC
```

这将创建一个名为 TagP 的标签，此标签不具有 value。TagP 具有一个子库，此子库中创建了值为 ValueC 的标签 TagC。它相当于：

```
{ TagP
TagC=ValueC
}
```

### 条件表达式

支持完整的 #if...#elif...#else...#endif 的语法。您可以使用此语法跳过 ToolConf 数据库的区域。表达式使用来自文件的标签，例如 C 预处理器序列：

```
#define Control True

#if defined(Control) && Control==True
#define controlIsTrue Yes
#endif
```

映射至 ToolConf 序列：

```
Control=True

#if Control && Control=True
ControlIsTrue=Yes
#endif
```

系统根据该点的配置内容从左至右评估条件。表 4-8 显示了可用于 ToolConf 条件表达式的运算符。

**表 4-8 用于 ToolConf 预处理器表达式的运算符**

| 运算符 | 实例 | 说明 |
|---|---|---|
| *none* | Tag | 测试是否存在标签定义 |
| == | Tag==Value | 不区分大小写的字符串同等测试 |
| != | Tag!=Value | 不区分大小写的字符串不同等测试 |
| (...) | (Tag==Value) | 分组 |
| && | TagA==ValueA && TagB==ValueB | 布尔值 AND |
| \|\| | TagA==ValueA \|\| TagB==ValueB | 布尔值 OR |
| ! | !(Tag==Value) | 布尔值 NOT |

### 文件包含

您可以使用 #include 指令将一个 ToolConf 文件包含在另一个此类文件中。如果此指令所处的区域是在条件表达式的控制下跳过的区域，则此指令将被忽视。

### 4.15.3 ToolConf 数据库中的布尔标记

表 4-9 显示了布尔标记的所有允许的值。这些字符串不区分大小写。

**表 4-9 布尔值**

| True | False |
|---|---|
| True | False |
| On | Off |
| High | Low |
| Hi | Lo |
| 1 | 0 |
| T | F |

### 4.15.4 ToolConf 数据库中的 SI 单位

您可以使用 SI (Système Internationale) 单位指定某些值，例如：

```
ClockSpeed=10MHz
MemorySize=2Gb
```

定标因数由单位的前缀设定。ARMulator 只接受表示千、兆、千兆的 k、M、或 G 前缀。这些前缀分别对应于 $10^3$、 $10^6$ 和 $10^9$ 或者 $2^{10}$、 $2^{20}$ 和 $2^{30}$ 定标。ARMulator 将根据上下文决定使用哪一种定标。

### 4.15.5 ToolConf_Lookup

此函数用于在 `.ami` 或 `.dsc` 文件的指定标签中执行搜索。如果发现标记，则会返回与其相关的值。否则将返回 `NULL` 。

#### 语法

```
const char *ToolConf_Lookup(toolconf hashv, tag_t tag)
```

其中：

*hashv* 是要在其中执行搜索的数据库。

*tag* 要在数据库中搜索的标签。此标签视情况而定。

#### 返回

此函数将返回：

- 标签值的 **const** 指针，如果搜索成功的话
- NULL，如果搜索失败的话

#### 实例

```
const char *option = ToolConf_Lookup(db, ARMulCnf_Size);

/* ARMulCnf_Size is defined in armcnf.h */
```

### 4.15.6 ToolConf_Cmp

此函数用于对两个 ToolConf 数据库标签值执行不区分大小写的比较。

#### 语法

```
int ToolConf_Cmp(const char *s1, const char *s2)
```

其中：

*s1* 是要比较的第一个字符串值的指针。

*s2* 是要比较的第二个字符串值的指针。

#### 返回

此函数将返回：

- 1，如果字符串相同的话
- 0，如果字符串不相同的话。

#### 实例

```
if (ToolConf_Cmp(option, "8192"))
```

## 4.16 外围设备参考

这里将详细说明两种参考外围设备：

- *中断控制器*
- 第 4-76 页*计时器*。

### 4.16.1 中断控制器

中断控制器的基址 IntBase 是可配置的 （请参阅第 2-33 页*中断控制器*）。

表 4-10 显示单个寄存器的位置。

**表 4-10 中断控制器存储器映射**

| 地址 | 读 | 写 |
|---|---|---|
| IntBase | IRQStatus | 保留 |
| IntBase + 004 | IRQRawStatus | 保留 |
| IntBase + 008 | IRQEnable | IRQEnableSet |
| IntBase + 00C | 保留 | IRQEnableClear |
| IntBase + 010 | 保留 | IRQSoft |
| IntBase + 100 | FIQStatus | 保留 |
| IntBase + 104 | FIQRawStatus | 保留 |
| IntBase + 108 | FIQEnable | FIQEnableSet |
| IntBase + 10C | 保留 | FIQEnableClear |

### 定义的中断控制器位

FIQ 中断控制器是一位宽。它位于 0 位上。

表 4-11 详细说明了与 IRQ 中断控制器寄存器中位 1 到位 5 相关的中断源。您可以将位 0 用于复制的 FIQ 输入。

**表 4-11 中断源**

| 位 | 中断源 |
|---|---|
| 0 | FIQ 源 |
| 1 | 计划的中断 |
| 2 | 通信通道 Rx |
| 3 | 通信通道 Tx |
| 4 | 计时器 1 |
| 5 | 计时器 2 |

**注**

计时器 1 和计时器 2 可配置为使用 IRQ 控制器寄存器中的不同位，请参阅第 2-34 页*计时器*。

  

### 4.16.2 计时器

计时器的基址 TimerBase 是可配置的 （请参阅第 2-34 页*计时器*）。

有关单个寄存器的位置信息，请参阅表 4-12 。

**表 4-12 计时器存储器映射**

| 地址 | 读 | 写 |
|---|---|---|
| TimerBase | Timer1Load | Timer1Load |
| TimerBase + 04 | Timer1Value | 保留 |
| TimerBase + 08 | Timer1Control | Timer1Control |
| TimerBase + 0C | 保留 | Timer1Clear |
| TimerBase + 10 | 保留 | 保留 |
| TimerBase + 20 | Timer2Load | Timer2Load |
| TimerBase + 24 | Timer2Value | 保留 |
| TimerBase + 28 | Timer2Control | Timer2Control |
| TimerBase + 2C | 保留 | Timer2Clear |
| TimerBase + 30 | 保留 | 保留 |

#### 计时器加载寄存器

将一个值写入其中一个寄存器中，以设置相应计时计数器的初始值。您必须将最高的 16 位写为零 。

如果计时器处于周期模式，当计数器达到零时，此值也会重新加载计时计数器。

如果您从此寄存器读取数据，下面的 16 位将返回您写入的值。最高的 16 位未经过定义。

#### 计时器值寄存器

计时器值寄存器为只读。下面的 16 位提供了计时计数器的当前值。最高的 16 位未经过定义。

#### 计时器清除寄存器

计时器清除寄存器为只写。写入其中一个寄存器将清除相应计时器所产生的中断。

### 计时器控制寄存器

有关计时器寄存器位的详情，请参阅表 4-14 和表 4-13。仅使用位 7、6、3 和 2。您必须将其它位均写成零。

**表 4-13 使用位 2 和位 3 的时钟预定标**

| 位 3 | 位 2 | 时钟的划分依据 | 预定标步骤 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 16 | 4 |
| 1 | 0 | 256 | 8 |
| 1 | 1 | 未定义 | - |

计数器是往下计数的。它计算按照 16 或 256 划分的 **BCLK** 周期或 **BCLK** 周期。位 2 和位 3 定义适用于时钟的预定标。

**表 4-14 使用位 6 和位 7 的计时器启用和模式控制**

| | 0 | 1 |
|---|---|---|
| 位 7 | 计时器已禁用 | 计时器已启用 |
| 位 6 | 自由运行模式 | 周期模式 |

在自由运行模式中，计时计数器将在达到零时溢出，然后继续从 `0xFFFF` 开始向下计数。

在周期模式中，当计数器达到零时，计时器将发生中断。然后从加载寄存器重新加载值，并从该值继续向下计数。

  

# 词汇表

本词汇表中的项目按拼音顺序列出，前面是英文字母。

**ADP** *请参阅* Angel 调试协议。

**AMBA** *请参阅* 先进微控制器总线体系结构。

**Angel** Angel 是一个程序，您可用其开发和调试在基于 ARM 的硬件上运行的应用程序。 Angel 可以调试在 ARM 状态或 Thumb 状态中运行的应用程序。

**Angel 调试协议 (ADP)**

Angel 使用名为 Angel 调试协议 (ADP) 的调试协议在主机系统和目标系统之间通信。 ADP 支持多个通道并提供了一个纠错通信协议。

**ARMulator** *请参阅* RealView ARMulator ISS。

**CPSR** 当前程序状态寄存器。

*请参阅* 程序状态寄存器。

**MMU** *请参阅* 存储器管理单元。

**Multi-ICE** 用于嵌入式系统的、基于 JTAG 的多处理器调试工具。 Multi-ICE 是 ARM 的注册商标。

**PSR** *请参阅* 程序状态寄存器。

**PU** *请参阅* 保护单元。

**RDI** *请参阅* 远程调试接口。

**RealView ARMulator ISS**
RealView ARMulator 指令集模拟器 (RealView ARMulator ISS)。模拟指令集和各种 ARM 处理器体系结构的模块集。

**Semihosting** 一种机制，其中目标将以应用程序代码提出的 I/O 请求传送至主机系统，而不是试图支持 I/O 自身。

**SPSR** 保存程序状态寄存器。

*请参阅* 程序状态寄存器。

**SWI** *请参阅* 软件中断。

**半字** 16 位单元长度的数据。除非另有说明，否则其内容将被当作无符号整数。

**保存程序状态寄存器 (SPSR)**
*请参阅* 程序状态寄存器。

**保护单元** 控制高速缓存和存储块访问许可的硬件。

**程序状态寄存器** *程序状态寄存器* (PSR) 包含有关当前程序以及当前处理器的某些信息。因此通常也称为*处理器状态寄存器*。

有时也将其称为*当前 PSR* (CPSR)，以强调与*保存 PSR* (SPSR) 之间的区别。调用当前函数时， SPSR 会保留 PSR 已有的值，并在返回控制时恢复这些值。

**处理器** 在目标上运行的实际或仿真处理器。处理器始终至少有一个执行语句。

**处理器状态寄存器** *请参阅* 程序状态寄存器。

**存储器管理单元 (MMU)**
控制高速缓存和存储块访问许可，以及将虚拟地址转换成物理地址的硬件。

**大端** 字的最低有效字节比最高有效字节位于更高的存储器地址。

*请参阅* 小端。

**断点** 映像中的位置。如果执行到此位置，则调试器将暂停映像的执行。

*请参阅* 观察点。

**跟踪** 在日志文件中记录诊断信息，以显示映像部件的执行频率和顺序。所记录的文本字符串是您在定义断点或观察点时指定的那些字符串。

*请参阅* 断点和观察点。

**观察点** 监控映像的位置。如果此处存储的值发生改变，调试器会暂停执行映像。

*请参阅* 断点。

**函数** C++ 方法或自由函数。

**目标** 运行目标应用程序的目标处理器 （真实或模拟的）。

任何调试会话中的基本对象。调试系统的基础。将运行目标软件的环境。它实质上是真实或模拟处理器的集合。

**配置报告** 执行被调试的程序期间累加的统计资料，用于测量性能或确定关键代码区域。

*Call-graph profiling* 提供了极为详细的信息，但会大大降低执行速度。
*Flat profiling* 提供较为简单的统计资料，对执行速度影响不大。

对于这两种类型的配置报告，您均可以指定统计资料两次收集操作之间的时间间隔。

**软件中断 (SWI)** 导致处理器调用程序员特定子程序的指令。 ARM 使用此指令处理 semihosting。

**双字** 信息的 64 位单元。除非另有说明，否则其内容将被当作无符号整数。

**调试器** 监控和控制另一个应用程序运行的应用程序。通常用于查找应用程序流中的错误。

**先进微控制器总线体系结构 (AMBA)**
用于高性能 32 位和 16 位嵌入式微控制器的片上通信标准。

**小端** 字的最低有效字节比最高有效字节位于更低地址的存储器。

*请参阅* 大端。

**协处理器** 用于某些操作的附加处理器。通常用于浮点数学计算、信号处理或存储器管理。

**映像** 一种可执行代码文件，可以载入目标上的存储器并由目标上的处理器执行。

**源文件** 作为映像建立过程一部分处理的文件。源文件与映像相关联。

**远程调试接口 (RDI)** 远程调试接口 (RDI) 是调试器与调试代理之间的一种 ARM 标准程序接口。RDI 为调试器提供了一种与以下设备进行通信的统一方式：

- 在主机上运行的调试代理 （例如 ARMulator）
- 在基于 ARM 的硬件上运行的调试监控器 （例如 Angel）（通过通信链接与该监控器进行通信）
- 通过硬件调试支持对 ARM 处理器进行控制的调试代理 （例如 Multi-ICE）。

**主机** 向其它计算机提供数据和其它服务的计算机。

**字** 32 位长度单元的数据。除非另有说明，否则其内容将被当作无符号整数。

# 索引

本索引按拼音排序，开始是字母和符号。给出的参考是指页码。

## 字母

## 符号



## B



## C



## D



## F



## G



## H

## J



## L



## M



## N



## S



## X



## Y



## Z