HPM6000 系列微控制器

片上 SRAM 使用指南

目录

版本：

日期	版本号	说明
2022-8-9	1.0	初版

1 简介

HPM6000 系列 MCU 是来自上海先楫半导体科技有限公司的高性能实时 RISC-V 微控制器，为工业自动化及边缘计算应用提供了极大的算力、高效的控制能力。上海先楫半导体目前已经发布了如 HPM6700/6400、HPM6300 等多个系列的高性能微控制器产品。

在 HPM6000 系列微控制器上，集成了大容量的 SRAM，可供用户存放代码，数据等，满足各类应用的需要。其中片上 SRAM 有多种分类，包括 RISC-V CPU 的指令和数据本地存储器 ILM，DLM，通用内存 AXI SRAM，AHB SRAM 等，还包括 AHB SRAM，APB SRAM 等。这些 SRAM 的最高访问频率不同，数据保持的条件也不同，恰当地使用他们，可以极大的提升用户应用的效率。

本文提供了与 HPM6000 系列微控制器的片上各类 SRAM 的介绍，以及它们各自的特点，并结合 Segger Embedded Studio 的 linker 文件介绍，提供了如何使用这些 SRAM 的建议。

2 片上 SRAM 介绍

HPM6000 系列高性能 MCU 均集成了大容量片上 SRAM，总结如下：

产品	ILM (KB)	DLM(KB)	AXI SRAM(KB)	AHB SRAM(KB)	APB SRAM(KB)
HPM6700	256+256	256+256	1024	32	8
HPM6400	256	256	1024+512	32	8
HPM6300	128	128	512	32	/

表1. HPM6000 系列片上 SRAM 总结

2.1 RISC-V CPU 的本地存储器 ILM 和 DLM

HPM6000 系列高性能微控制器的 RISC-V CPU 都包含有指令和本地存储器，分别称为 ILM（Instruction Local Memory，指令本地存储器）和 DLM（Data Local Memory，数据本地存储器)。如下图所示，RISC-V CPU 的 ILM 和 DLM 各自对应了内存映射表（Memory Map）中的 2 块地址映射区域。

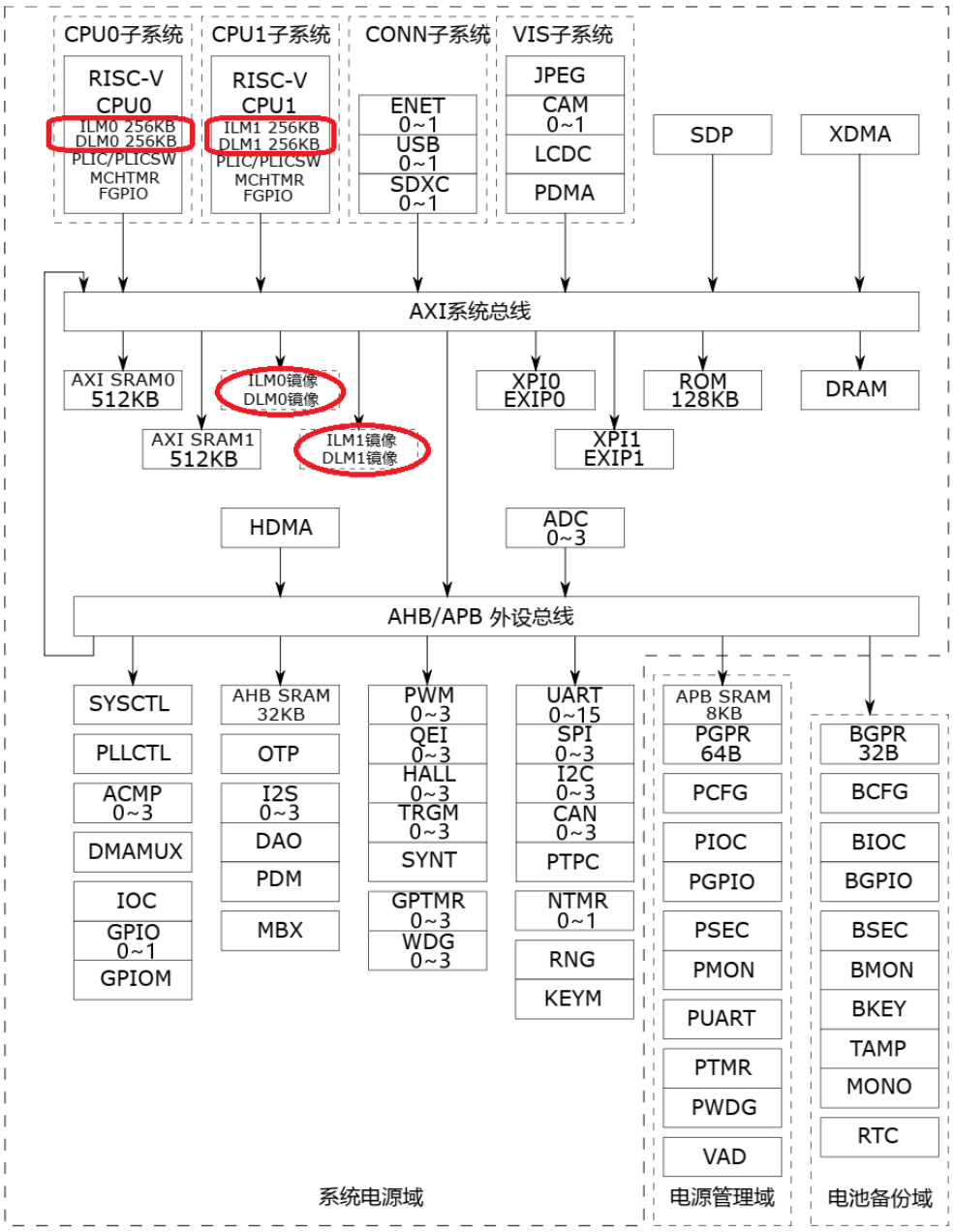


图1. HPM6700 系列 ILM/DLM 示意图

ILM 的映射有：

- ILM，以 HPM6700 系列为例，容量 256 KB，地址范围为 0x00000000 ~ 0x0003FFFF。该地址区域只有 RISC-V CPU 可以通过自身的 ILM 接口访问，RISC-V CPU 从 ILM 取指时，可以实现零等待周期，即 ILM 的访问时钟频率，与 RISC-V CPU 主频一致，并且 RISC-V CPU 的一级高速指令缓存 I-Cache-L1 对 ILM 无效。
- CPUx_ILM_SLV，以 HPM6700 系列的 CPU0 ILM 为例，容量 256 KB，地址范围为 0x01000000 ~ 0x0103FFFF。该地址区域也称为 ILM0 镜像，该地址区域可以供总线上所有的主设备访问，如 DMA 等，包括 RISC-V CPU 自身，也可以通过 CPUx_ILM_SLV 访问 ILM。如图所示，CPUx_ILM_SLV 是 AXI 系统总线的从接口，其访问时钟频率于总线时钟频率一致。RISC-V CPU 访问 CPUx_ILM_SLV 时，一级高速指令缓存 I-Cache-L1 是有效的。

DLM 的映射有：

- DLM，以 HPM6700 系列为例，容量 256 KB，地址范围为 0x00080000 ~ 0x000BFFFF。该地址区域只有 RISC-V CPU 可以通过自身的 DLM 接口访问，RISC-V CPU 从 DLM 读写数据时，可以实现零等待周期，即 DLM 的访问时钟频率，与 RISC-V CPU 主频一致，并且 RISC-V CPU 的一级高速指令缓存 D-Cache-L1 对 DLM 无效。
- CPUx_DLM_SLV，以 HPM6700 系列的 CPU0 DLM 为例，容量 256 KB，地址范围为 0x01040000 ~ 0x0107FFFF。该地址区域也称为 ILM0 镜像，该地址区域可以供总线上所有的主设备访问，如 DMA 等，包括 RISC-V CPU 自身，也可以通过 CPUx_DLM_SLV 访问 ILM。如图所示，CPUx_DLM_SLV 是 AXI 系统总线的从接口，其访问时钟频率于总线时钟频率一致。RISC-V CPU 访问 CPUx_DM_SLV 时，一级高速指令缓存 D-Cache-L1 是有效的。

2.1.1 HPM6700 系列

HPM6700 系列微控制器上，指令/数据本地存储器的 2 块地址映射 xLM 和 CPUx_xLM_SLV 虽然地址不同，但访问的是同一块物理内存，RISC-V CPU 可

以通过 xLM 访问自身的指令/数据本地存储器，而其他总线主设备，比如 DMA，需要通过 CPUx_xLM_SLV 来访问 CPUx 的指令/数据本地存储器。注意，RISC-V CPU 本身，也可以通过 CPUx_xLM_SLV 访问自己的指令/数据本地存储器。

HPM6700 系列支持双核 RISC-V CPU，CPU 从 xLM 地址映射总是访问到自身指令/数据本地存储器，而从 CPUx_xLM_SLV 可以访问到自身或者另一个 CPU 的指令/数据本地存储器。

以 HPM6700 系列 RISC-V CPU0 为例，从 0x00000000 和 0x01000000 读取到的，是CPU0指令本地存储器ILM0的首地址。从0x01180000读到的，是 CPU1 指令本地存储器 ILM1 的首地址。

以 RISC-V CPU1 为例，从 0x00000000 和 0x01180000 读取到的，是 CPU1 指令本地存储器 ILM1 的首地址。从 0x01000000 读到的，是 CPU0 指令本地存储器 ILM0 的首地址。

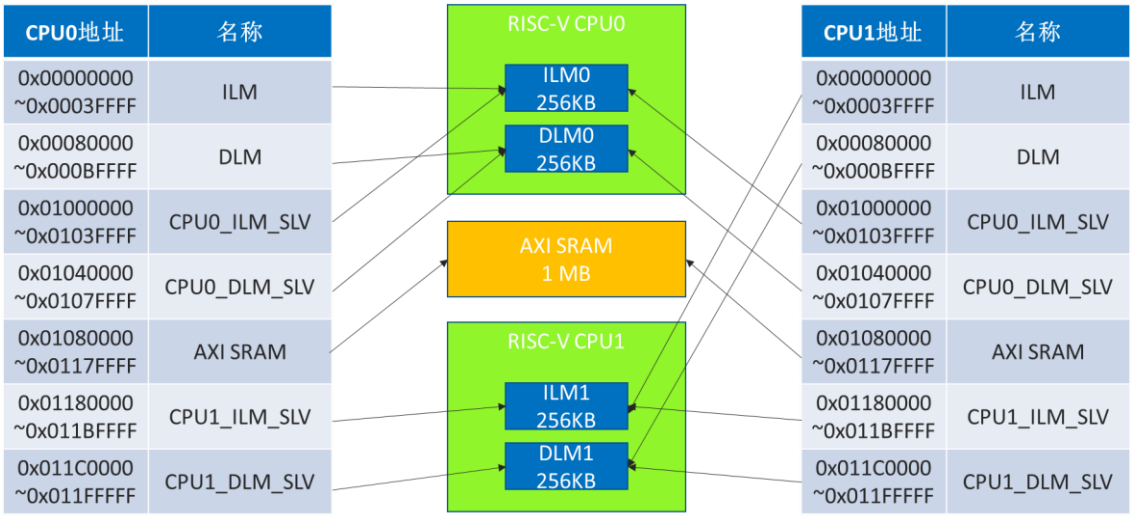


图2. HPM6700 双核地址空间访问示意图

2.1.2 HPM6300 系列

HPM6300 系列为单核 RISC-V CPU，RISC-V CPU 和其他总线主设备都可以通过 xLM 地址映射访问 CPU 的指令/数据本地存储器，其中 RISC-V CPU 仍然可以通过自身的 xLM 接口对 xLM 实现零等待周期访问，即 xLM 的读写时钟频率与 CPU 的时钟同频。而其他总线主设备，需要以 AXI 系统总线的时钟频率访问 xLM。

同时，xLM 的镜像，CPUx_xLM_SLV 映射仍然有效，包括 RISC-V CPU 在内的所有总线主设备，访问 CPUx_xLM_SLV 与访问 xLM 效果相同，访问的是同一块物理内存。注意 RISC-V CPU 通过 CPUx_xLM_SLV 访问自身的 xLM 时，会和其他总线主设备一样，读写时钟频率与 AXI 系统总线时钟频率一致，不再支持零周期等待，但是 CPU 的高速一级缓存会生效。

比如，RISC-V CPU 和 DMA，从 0x00000000 读取到的，是 CPU0 指令本地存储器 ILM0 的首地址。从 0x01000000 读取到的，也是 CPU0 指令本地存储器 ILM0 的首地址。注意 RISC-V CPU 从 0x01000000 取值后，如果高速缓存打开，其数据会被存入缓存。

2.2 AXI 总线 SRAM

HPM6000 系列高性能 MCU 支持通用的片上 SRAM，称为 AXI SRAM，可以用来存放数据或者代码。

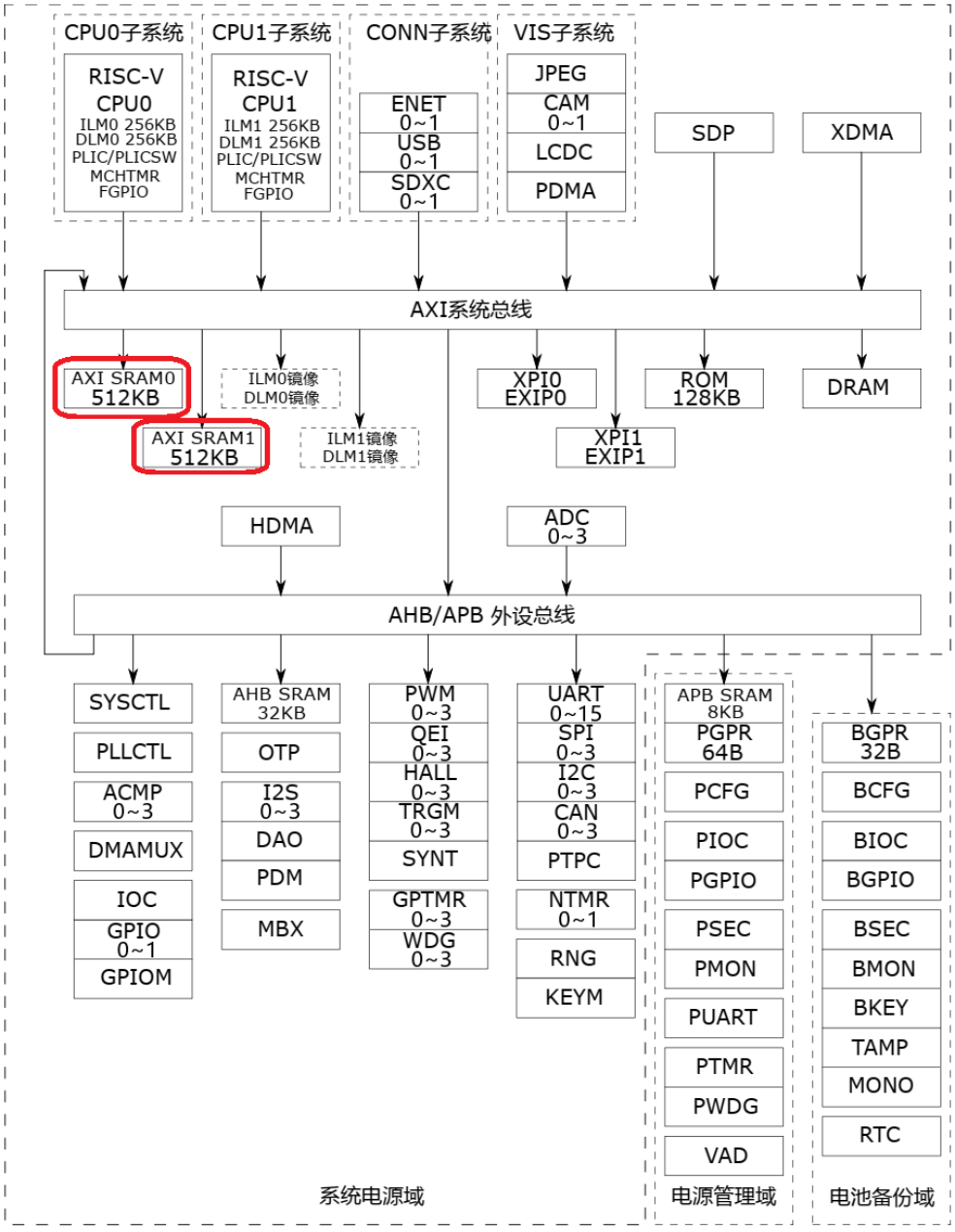


图3. HPM6700 系列 AXI SRAM 示意图

上图以 HPM6700 系列 MCU 为例，展示了 AXI SRAM 在系统中的位置。顾名思义，AXI SRAM 连接到 AXI 系统总线的从接口，它的读写时钟频率就是 AXI 系统总线的时钟频率。所有的 AXI 系统总线主设备都可以访问 AXI SRAM。

注意：系统内存映射表上，AXI SRAM 的地址映射与 RISC-V CPU 的 ILM/DLM 镜像映射 CPUx_xLM_SLV 在地址上是连续的。因此可以把 AXI SRAM 和 RISC-V CPU 的本地存储器当作一块联系的大内存使用。详情请参考后文的 SRAM 使用建议。

2.3 AHB 总线 SRAM

HPM6000 系列高性能 MCU 包含挂载在外设总线 AHB 的片上 SRAM，称为 AHB SRAM，如下图所示，AHB SRAM 连接到 AHB 外设总线的从设备接口。

AHB SRAM 的读写时钟频率为 AHB 外设总线的时钟频率。注意，包括 RISC-V CPU 在内的总线主设备需要通过 AXI 系统总线，来访问 AHB 外设总线下的各个外设寄存器。而外设总线 AHB 上的主设备，比如 HDMA，可以直接通过 AHB 总线访问外设寄存器，以及 AHB SRAM。因此 AHB SRAM 比其他类型的片上 SRAM 更适合用作寄存器和 SRAM 之间的数据搬运。基于同样的理由，HDMA 也比 XDMA 更适合用作 AHB SRAM 和寄存器之间的数据搬运。

因此，当使用 HDMA 用作通讯接口，如 UART，SPI 的数据收发时，推荐使用 AHB SRAM 作为数据的缓冲区。

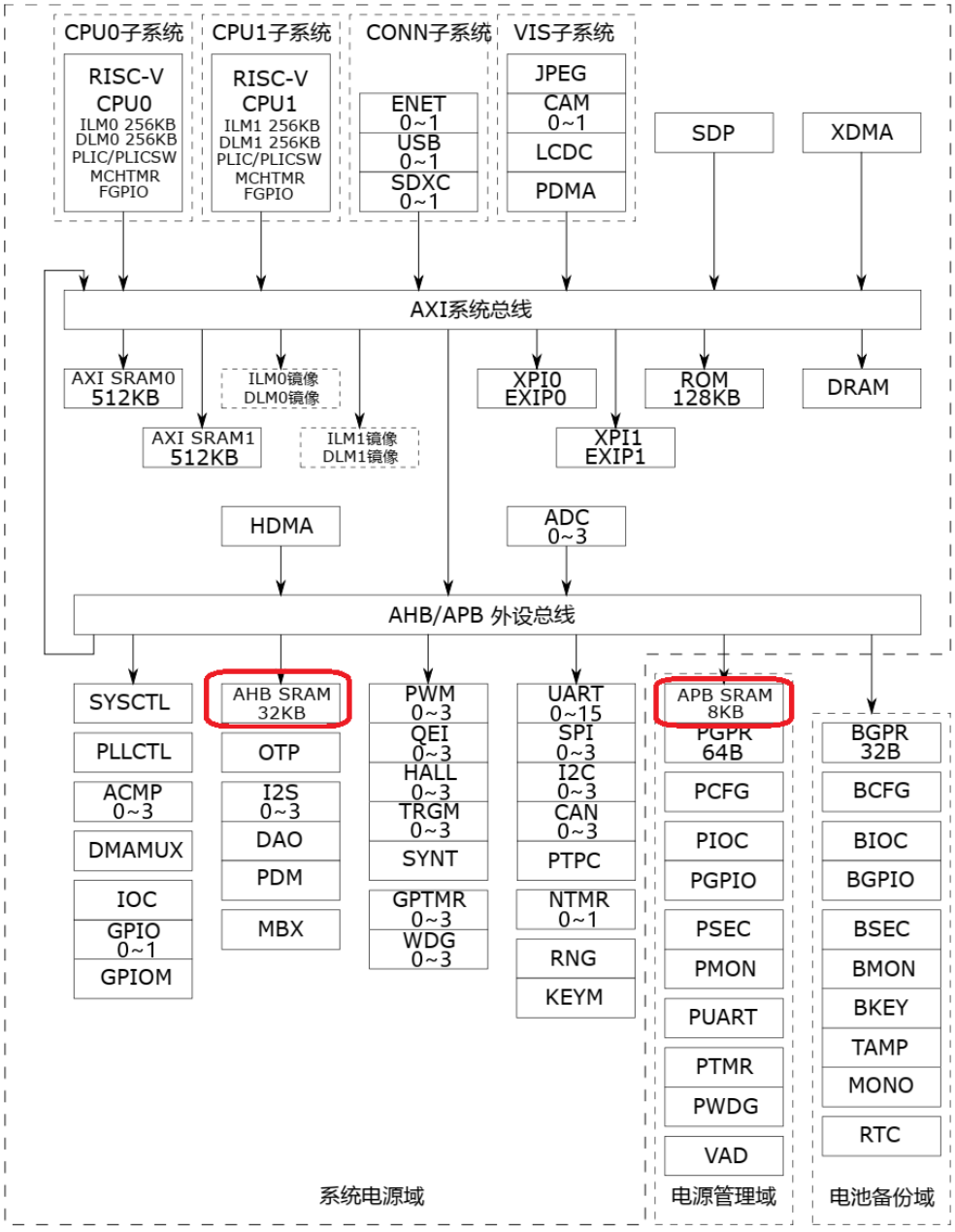


图4. HPM6700 系列 AHB SRAM 和 APB SRAM 示意图

2.4 APB 总线 SRAM

HPM6000 系列高性能 MCU 的部分型号支持 APB SRAM。上图以 HPM6700 系列 MCU 为例，展示了 APB SRAM 在系统中的位置。

APB SRAM 位于电源管理域，由 VPMC 引脚供电。当 MCU 处于低功耗模式时，有可能通过关闭系统电源域的电源，使得 VDD_SOC 掉电。此时，APB SRAM 可以作为片上的备份 SRAM，保存必要的数据。

注意，对于不支持 APB SRAM 的型号，可以使用电源管理域通用寄存器 PGPR，保存一定的备份数据。

在电池模式（VBAT Only Mode）下，VPMC 也掉电，HPM6000 系列 MCU 仅 VBAT 保持供电。此时，电池备份域通用寄存器 BGPR，可以用来保存一定的备份数据。

3 Linker 文件介绍

本章节以 hpm_sdk 提供的基于外部串行 NOR Flash 在线执行的 linker 文件(flash_xip.icf)为例，介绍了 Segger Embedded Studio 集成开发环境的 linker 文件的格式和一些关键字段。用户可以以此为基础，通过修改 linker 文件来分配代码和数据的位置。

本章节的 linker 示例文件 flash_xip.icf 来自 hpm_sdk 0.13.0 解压包的以下位置：

sdk_env_v0.13.0\hpm_sdk\soc\HPM6750\toolchains\segger\

3.1 Linker 文件的 Regions

在 linker 文件的起始，定义了 HPM6750 微控制器，映射到内存寻址空间的各个区域（Region），这些 Region 基本对应了微控制器片上的各块物理存储器，如下所示：

```
/* Regions */
define region NOR_CFG_OPTION = [ from 0x80000400 size 0x1000 ];
define region BOOT_HEADER = [ from 0x80001000 size 0x3000 ];
define region XPI0 = [from 0x80003000 size _flash_size - 0x3000];
/*XPI0 */
define region ILM_SLV   = [from 0x1000000 size 256k]; /* ILM slave */
define region DLM   = [from 0x80000 size 256k];    /* DLM */
define region AXI_SRAM  = [from 0x1080000 size 768k];
define region SDRAM  = [from 0x40000000 size _extram_size];
define region NONCACHEABLE_RAM = [from 0x1140000 size 256k];
```

各个 Region 解释如下：

NOR_CFG_OPTION 区域，用来存放外部串行 NOR flash 的配置信息，在 MCU 启动时 BOOT ROM 会根据此配置信息，来初始化 XPI0 端口，用于后续进一步与 NOR Flash 通讯。在当前 linker 文件，将 NOR_CFG_OPTION 区域指定在地址 0x80000400，即串行总线控制器 XPI0 存储空间映射的 0x400 偏移位置，对应与 XPI0 连接的串行 NOR Flash 地址偏移 0x400。

BOOT_HEADER 区域，用来存放用户代码镜像的固件容器头，在 MCU 启动时 BOOT ROM 会根据固件容器头，进一步执行启动流程。在当前 linker 文件，将 BOOT_HEADER 区域指定在地址 0x80001000，即串行总线控制器 XPI0 存储空间映射的 0x1000 偏移位置，对应与 XPI0 连接的串行 NOR Flash 地址偏移 0x1000。

有关 HPM6700 系列微控制器的 NOR_CFG_OPTION 和 BOOT_HEADER 的详细信息，请常考《HPM6700/6400 系列微控制器用户手册》的“BOOTROM”章节。

XPI0 区域，即用户代码镜像的存储位置。在当前 linker 文件，将 XPI0 区域指定在地址 0x80003000，即串行总线控制器 XPI0 存储空间映射的 0x3000 偏移位置，对应与 XPI0 连接的串行 NOR Flash 地址偏移 0x3000。显然，此偏移之前的区域，保留给了 NOR_CFG_OPTION 区域和 BOOT_HEADER 区域。

ILM_SLV 区域，即 HPM6750 RISC-V CPU0 的指令本地存储器 ILM 镜像，在当前 linker 文件，按照内存映射表，将 ILM_SLV 区域指定在地址 0x1000000。

DLM 区域，即 HPM6750 RISC-V CPU0 的 DLM，数据本地存储器，在当前 linker 文件，按照内存映射表，将 DLM 区域指定在地址 0x80000。

AXI_SRAM 区域，即 HPM6750 的通用内存 AXI_SRAM，在当前 linker 文件，按照内存映射表，将 AXI SRAM 区域指定在地址 0x1080000。注意，此区域的大小定义为 768 KB，与实际 AXI SRAM 1 MB 容量有 256KB 的差距。此部分保留给了 NONCACHEABLE_RAM 区域。

SDRAM 区域，即 HPM6750 DRAM 控制器存储空间，在当前 linker 文

件，按照内存映射表，将 SRAM 区域指定在地址 0x40000000。

NONCACHEABLE_RAM 区域，在当前 linker 文件，将此区域指定在地址 0x1140000，对应内存映射表上通用内存 AXI_SRAM 的一部分。在 hpm_sdk 的 MCU 启动代码部分，会根据 NONCACHEABLE_RAM 区域的起始地址和大小，配置 RISC-V CPU 的缓存控制器，使得 CPU 的 cache 对此区域不生效。通常，建议用户将 RISC-V CPU 和其他总线主设备共享的存储区域，配置为 CPU 缓存无效，避免 CPU 和其他总线主设备交替访问同一块内存导致 cache 内数据和实际内存数据不一致。例如，USB，以太网等收发数据缓冲区，应当分配到 NONCACHEABLE_RAM 区域。

3.2 Linker 文件的 Blocks

在 linker 文件中，同样定义了各个 blocks，如下图所示：

```
/* Blocks */
define block vectors  { section .isr_vector, section .vector_table };
define block ctors    { section .ctors,     section .ctors.*, block
with    alphabetical order { init_array } };
define block dtors    { section .dtors,     section .dtors.*, block with
reverse alphabetical order { fini_array } };
define block eh_frame  { section .eh_frame,  section .eh_frame.* };
define block tbss  { section .tbss,      section .tbss.*  };
define block tdata { section .tdata,     section .tdata.* };
define block tls   { block tbss, block tdata };
define block tdata_load  { copy of block tdata };
define block heap  with size = __HEAPSIZE__,  alignment = 8, /* fill
=0x00, */ readwrite access { };
define block stack with size = __STACKSIZE__, alignment = 8, /* fill
=0xCD, */ readwrite access { };

define block framebuffer with alignment = 8 { section .framebuffer };
define block safe_stack with size = 512, readwrite access {};
```

其中部分 blocks 由编译器自动使用，在此不详述每个 block 的含义，其中与用户程序开发密切相关的 blocks 如下：

vectors：RISC-V CPU 的中断向量表，hpm sdk 默认打开了 RISC-V CPU 的中断向量扩展和中断嵌套扩展，每一个外部中断都在中断向量表中占有一个

32 位长的入口。block vectors 包含的就是中断向量表。

heap：block heap 包含了程序的堆，堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

stack：block stack 包含了程序的栈，是用户存放程序临时创建的局部变量。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出(FIFO)特点，所以栈也用来保存/恢复调用现场。比如 RISC-V CPU 响应中断时，也会将当前 CPU 内现场数据保存到 stack。

3.3 Linker 文件的 Sections

在 linker 文件中，定义了各个 section，指定了它们的初始化方式，并将不同的 section，分配到了对应不同存储器的 region 中。

比如，不需要初始化的 section .noncachable：

```
do not initialize                               { section .noncacheable };
```

以及提示了需要初始化的部分 sections：

```
initialize by copy with packing=auto  { section .noncacheable.init };
initialize by copy with packing=none  { section .data, section .data.*,
section .*.data, section .*.data.* };    // Static data sections
initialize by copy with packing=auto  { section .sdata,
  section .sdata.* };
initialize by copy with packing=auto  { section .fast, section .fast.*,
section .*.fast, section .*.fast.* };    // "RAM Code" sections
```

section .data 为程序的数据段，会以复制的形式进行初始化。

section .fast 为复制到内部 RAM 执行的代码，也在初始化阶段复制到 RAM 中。

3.4 Linker 存储区域分配

Linker 文件会对各个 block 和 section 进行存储器区域的分配，将它们定位到各个 region，比如程序的入口位置，分配到 XPI0（即连接的片外串行 NOR Flash）的起始位置：

```
place at start of XPI0 with fixed order { symbol _start};
```

中断向量表，为了便于快速访问，以复制的形式指定到片内 ILM0 的镜像地址：

```
place at start of ILM_SLV with fixed order { block vectors };
initialize by copy { block vectors };
```

以及各个存放到 XPI0 连接的片外串行 NOR Flash 的 block：

```
place in XPI0 with minimum size order     {
               block tdata_load,   // Thread-local-storage load image
               block ctors,        // Constructors block
               block dtors,        // Destructors block
               block eh_frame,     // Exception frames placed directly
into flash overriding default placement (sections writable)
               readonly,
              // Catch-all for readonly data (e.g. .rodata, .srodata)
               readexec
              // Catch-all for (readonly) executable code (e.g. .text)
                                          };
```

其中包括常量等只读数据 readonly，以及代码 readexec。

还有复制到片内 ILM0 的镜像执行的 section .fast，一些对执行性能有要求的代码，可以分配到此 section:

```
place in ILM_SLV                          {
                section .fast, section .fast.*, // "ramfunc" section
                                          };
```

以及指定到 DLM 区域的数据段 readwrite，和不需要初始化的数据段 zeroinit，如.bss：

```
place in DLM with auto order              {
         block tls,  // Thread-local-storage block
         readwrite,  // Catch-all for initialized/uninitialized data
sections (e.g. .data, .noinit)
```

```
        zeroinit    // Catch-all for zero-initialized data sections
(e.g. .bss)
                                        };
```

framebuffer，通常用作通讯接口的收发数据缓冲，分配到通用内存 AXI SRAM 中。heap 堆也分配到通用内存 AXI SRAM，stack 栈则分配到通用内存 AXI SRAM 的末尾部分。DLM 的末尾指定给 safe stack：

```
place in AXI_SRAM           { block framebuffer };
place in NONCACHEABLE_RAM   { section .noncacheable,
section .noncacheable.init, section .noncacheable.bss };
place in AXI_SRAM           { block heap };  // Heap reserved block
place at end of AXI_SRAM    { block stack }; // Stack reserved block at
the end
place at end of DLM         { block safe_stack }; // Safe stack
reserved block at the end
```

4 SRAM 使用建议

HPM6000 系列支持大容量，多种类型的片上 SRAM，可以用作不同的用途，本章节给出了一些代码、数据存储的建议，供用户参考。

本章节也会给出一些修改 linker 文件的例子，以 Segger Embedded Studio 集成开发环境为例，示例代码基于 hpm_sdk 0.13.0。

4.1 代码位置建议

HPM6000 系列 MCU 通常通过 XPI 接口连接外部串行 NOR Flash，用作代码和数据的存储。外部 NOR Flash 的访问带宽，按照由实际使用 NOR Flash 种类和特性决定。通常情况下，外部存储器的访问速度相对高性能 CPU 来说，是偏慢的。

为了弥补外部存储器慢速的缺点，HPM6000 系列高性能微控制器支持各 32KB 的指令和数据缓存，在大多数情况下，RISC-V CPU 可以从缓存中读取到需要的指令或者数据。只有在缓存不命中（cache miss）的情况下，CPU 需要访问外部存储器。此时 CPU 需要等待外部存储器返回数据，这有可能影响程序执行的效率，甚至降低系统相应的实时性。

HPM6000 的 RISC-V CPU 指令本地存储器（ILM）的访问频率与 CPU 相同，支持零等待周期访问。因此，CPU 执行存放在 ILM 中的代码，性能最优，并且可以规避 cache 随机替换导致 cache miss 造成的随机性访问延时。

由于 ILM 的容量有限，有可能不够存放用户的全部代码，因此建议用户考虑把执行性能要求最高，响应要求最严格的代码放置到 ILM 里。

比如：RISC-V CPU 的中断向量表，每当 RISC-V 响应中断时，总是从中断向量表中，载入中断的入口。

以下例子采用 hpm sdk 的 flash xip 工程，以其 linker 文件 flash_xip 工程的 linker 文件作为模板（flash_xip.icf），展示了如何将中断向量表分配到 ILM 中。

首先，定义一块 ILM 区域，取 HPM6700 CPU0 的 ILM 后 128KB。

```
define region ILM = [from 0x20000 size 128k];
```

注意：在原版 flash xip 的 linker 文件中，定义了 ILM_SLV 区域，此区域对应了 ILM 的镜像 CPUx_ILM_SLV，由于 ILM 和 ILM 镜像实际对应一块相同的物理内存，因此，为避免冲突，应修改 ILM_SLV 区域范围，避免与新定义的 ILM 区域重叠。这里，将 ILM_SLV 修改为只保留前 128KB。

```
define region ILM_SLV   = [from 0x1000000 size 128k]; /* ILM slave */
```

将中断向量表从 ILM_SLV 重新分配到 ILM：

```
place at start of ILM with fixed order { block vectors };
```

除了中断向量表外，用户可以把其他函数，分配到 ILM 中，如以下例子：

```
define block myilmfunc                          { section .myilmfunc };
initialize by copy with packing=auto            {section .myilmfunc};
place in ILM                                  { block myilmfunc};
```

定义目标分配到 ILM 的函数，如下：

```
__attribute__ ((section(".myilmfunc"))) uint32_t myfunctionilm
(uint32_t a, uint32_t b);
uint32_t myfunctionilm (uint32_t a, uint32_t b)
{
  return (a+b);
}
```

由于 ILM 容量有限，将代码分配到 AXI SRAM 执行，是一个次优的选项。以下例子展示了如何将某个函数，分配到 AXI SRAM 中：

```
define block myaxisramfunc                  { section .myaxisramfunc };
```

```
initialize by copy with packing=auto     {section .myaxisramfunc};
place in AXI_SRAM         { block framebuffer, block myaxisramfunc };
```

定义目标分配到 ILM 的函数，如下：

```
__attribute__ ((section(".myaxisramfunc"))) uint32_t myfunctionaxisram
(uint32_t a, uint32_t b);
uint32_t myfunctionaxisram (uint32_t a, uint32_t b)
{
  return (a*a+b*b);
}
```

除了修改 Segger Embedded Studio 的 linker 文件，自定义 block 并分配到 ILM 或者 AXI SRAM 之外，用户也可以利用 hpm_sdk 内部预先定义的宏

```
#define ATTR_RAMFUNC ATTR_PLACE_AT(".fast")
```

来指定函数在 RAM 中运行：

```
ATTR_RAMFUNC uint32_t myramfunction(uint32_t a, uint32_t b);

uint32_t myramfunction(uint32_t a, uint32_t b)
{
    return (a*a + b);
}
```

可以看到，该宏将函数指定到了 section .fast，可以通过编辑 linker 文件，指定该区域的位置：

```
place in ILM_SLV                              {
   section .fast, section .fast.*, // "ramfunc" section
                                              };
```

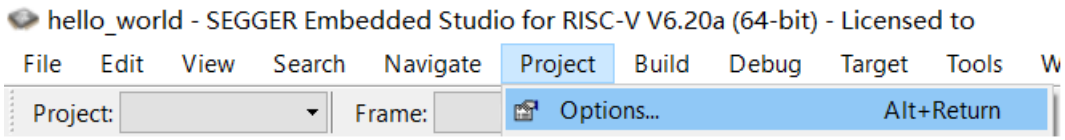
4.2 代码批量复制到片上 SRAM 运行

使用 Segger Embedded Studio 调试 HPM6000 系列微控制器的用户可以采用以下方法，批量地把执行性能要求高的代码复制到片上 SRAM 运行。

在 Segger Embedded Stduio 中，已预先定义了 section .fast，section .fast 中的代码会预先从 Flash 中复制到 RAM 的指定位置。

```
initialize by copy with packing=auto  { section .fast, section .fast.*,
section .*.fast, section .*.fast.* }; // "RAM Code" sections
```

用户可以通过菜单 Project -> Options，打开当前工程选项：



选择 Code->Section，将 Code Section Name，由.text 修改成.fast，如下图所示：

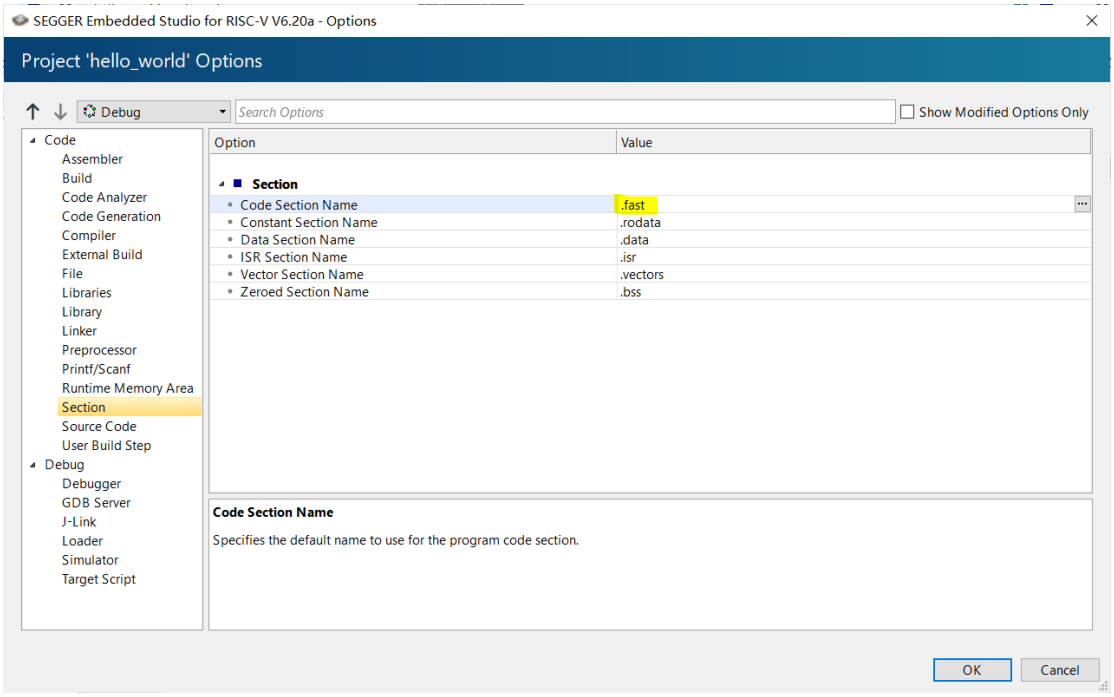


图5. 修改工程的 Code Section Name

注意，项目的启动代码必须保留在 Flash 中，因为所有 RAM 执行的代码需要由启动代码复制到 RAM。因此，用户需要修改包含启动代码的文件夹，在 hpm_sdk 中，启动代码位于 toolchains 文件夹。用户可以右键点击 toolchains 文件夹，Code->Section，将 Code Section Name，由.fast 修改为.text。如下图所示：

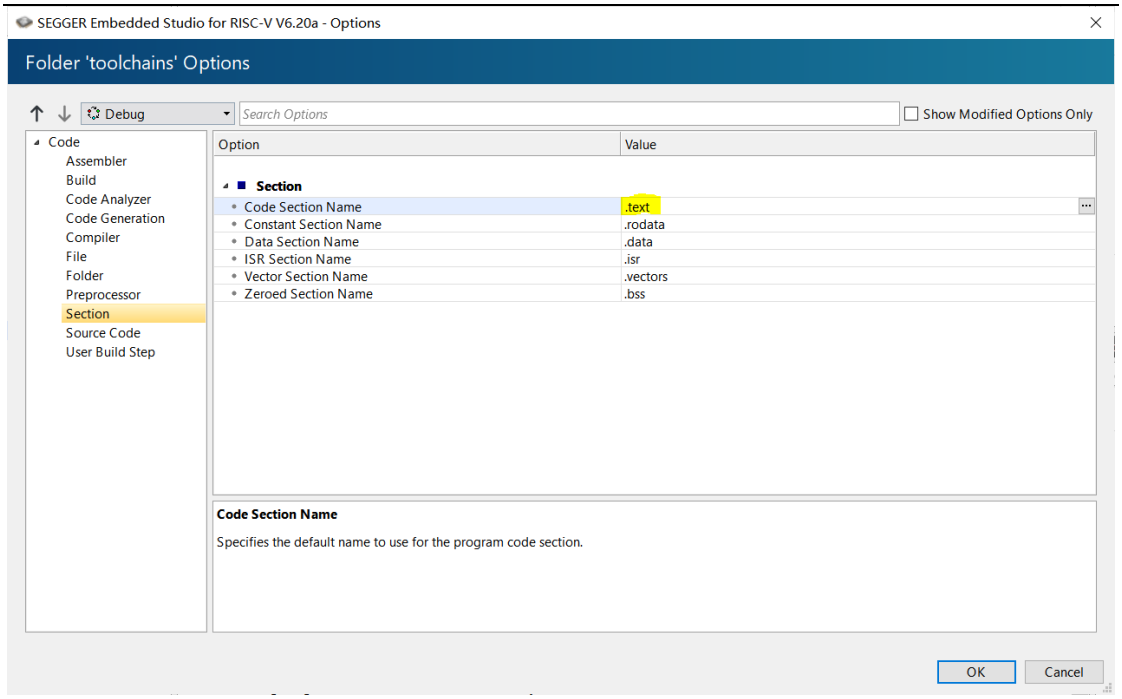


图6. 将启动代码保留在.text 区段

同理，用户可以通过右键点击文件夹或者右键点击单个文件，并选择 Options，修改 Code->Section，将特定文件夹或者特定源文件的代码，指定到 RAM 执行。

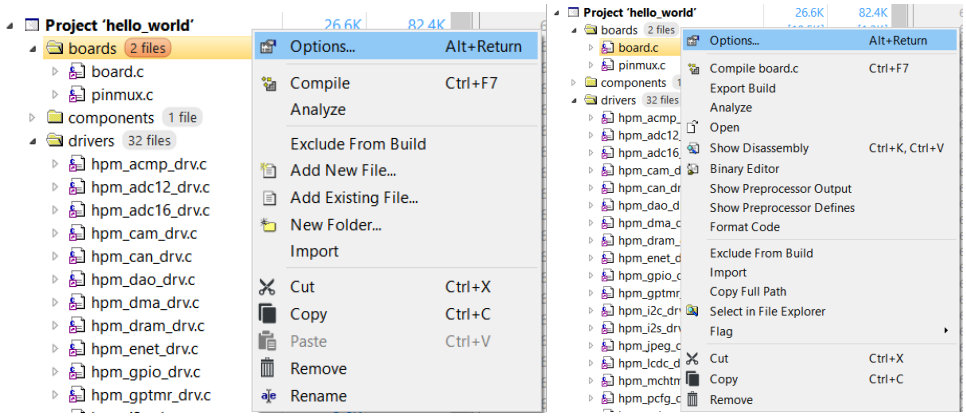


图7. 配置文件夹或者文件的选项

4.3 数据位置建议

HPM6000 的 RISC-V CPU 数据本地存储器（DLM）的访问频率与 CPU 相同，支持零等待周期访问。因此，对于 CPU 访问频率最高的数据，存放在 DLM 中可能达到性能最优。

比如程序的 stack，通常在函数调用，CPU 中断时，都会大量执行保存现

场到stack和从stack中恢复现场的操作。因此，建议将stack分配到DLM中。以下例子展示了如何编辑 linker 文件，分配 stack 到 DLM 中。

```
define region DLM    = [from 0x80000 size 256k];    /* DLM */
place at end of DLM with fixed order { block stack, block safe_stack };
```

也可以把 CPU 可能频繁读写的数据定义到 DLM 中：

```
define block dataindlm                          { section .dataindlm };
initialize by copy with packing=auto            {section .dataindlm };
place in DLM          {block dataindlm};
```

在数据声明的时候：

```
__attribute__ ((section(".dataindlm"))) uint32_t mydataindlm[1024]={0};
```

4.4 SRAM 容量最大化使用建议

在 HPM6000 系列高性能微控制器，允许用户把 RISC-V CPU 的指令/数据本地存储器和片上通用内存 AXI SRAM，当作一块地址连续的整体内存使用。用户可以参考相应产品的用户手册，查阅内存映射表可以发现，ILM 和 DLM 的镜像，与 AXI SRAM 的地址是连续的。

用户可以修改 linker 文件，把所有的片上 SRAM 作为一个整体来使用，下面以 HPM6700 系列为例：

```
//define region ILM_SLV    = [from 0x1000000 size 256k]; /* ILM slave */
//define region DLM    = [from 0x80000 size 256k];    /* DLM */
define region AXI_SRAM  = [from 0x1000000 size 2048k];
define region SDRAM  = [from 0x40000000 size _extram_size];
define region NONCACHEABLE_RAM = [from 0x0 size 0];
```

由于所有的 ILM 和 DLM 都合并入 AXI SRAM 使用，所以应注释掉原有的相关区域定义，并把 AXI_SRAM 区域大小定为 2048KB，暂时假设应用不使用 NONCACHEABLE 区域，将其大小定义为 0。

把 vector table 等区域的位置都修改到新的 AXI_SRAM 中：

```
place at start of AXI_SRAM with fixed order { block vectors };
```

以及原有的分配到 RAM 执行的函数：

```
place in AXI_SRAM                               {
    section .fast, section .fast.*,            // "ramfunc" section
                                               };
```

以及 frambuffer：

```
place in AXI_SRAM                          { block framebuffer };
```

还有堆栈：

```
place in AXI_SRAM                          { block heap };
place at end of AXI_SRAM with fixed order  { block stack, block
safe_stack};
```

5 总结

本文主要介绍了 HPM6000 系列高性能微控制器片上 SRAM 的分类。HPM6000 系列片上 SRAM 包括可以支持 RISC-V CPU 零等待周期访问的 ILM 和 DLM，以及通用 SRAM，包括 AXI SRAM，AHB SRAM 等待。

用户可以根据实际的应用需求，把代码和数据分配到不同的片上 SRAM，以实现性能优化。

本文同时给出了一些通过修改 Segger Embedded Studio 的 linker 文件，把特定数据或者函数，分配到不同的片上 SRAM 的例子。