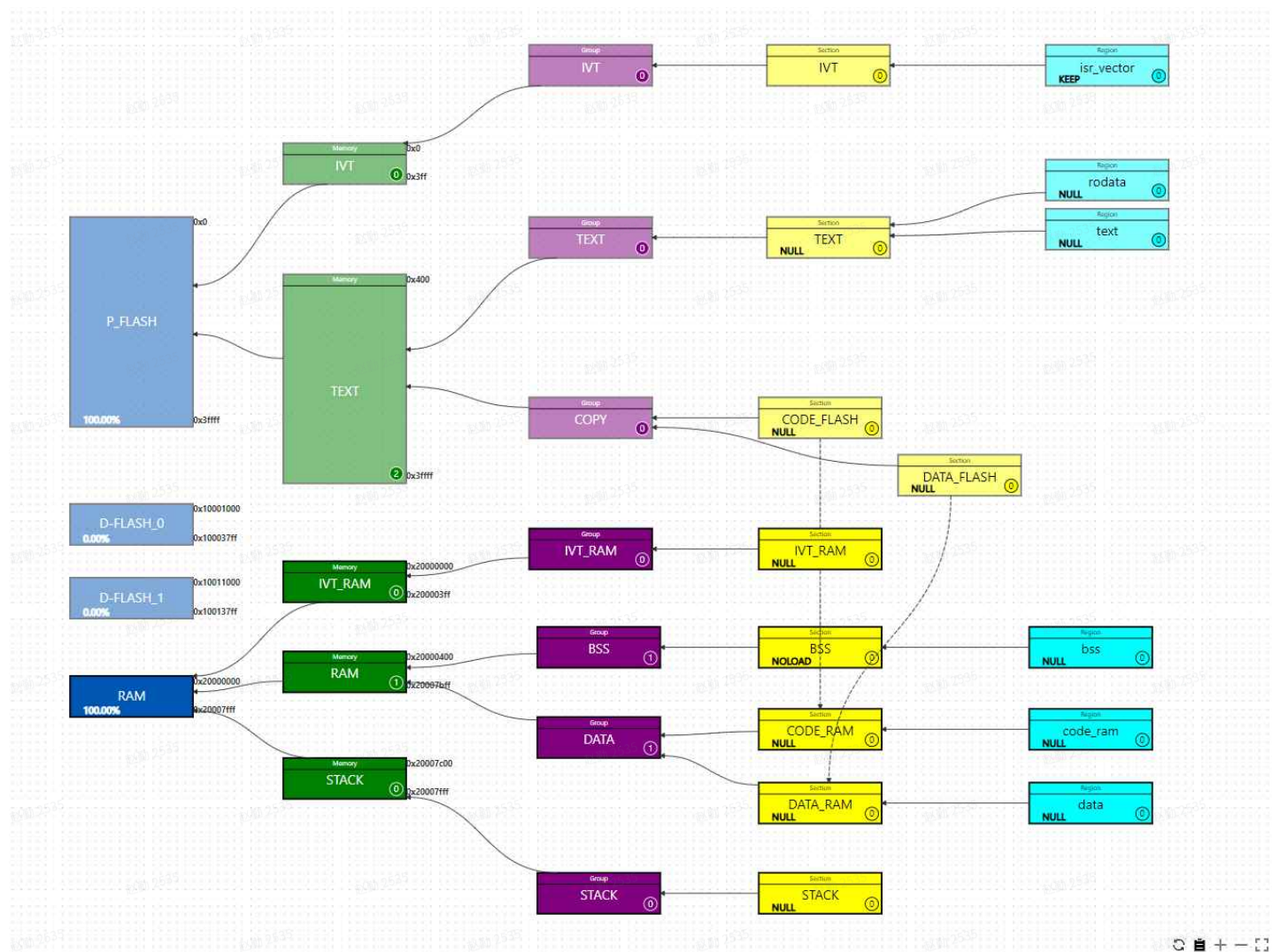


YT-LINK User Manual

介绍

YunTu YT-LINK 配置工具提供灵活强大的链接文件配置，精确控制每一字节的链接，所见即所得。



功能

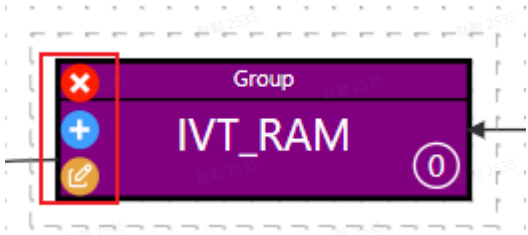
- 支持多种编译器，一次配置，无缝替换
 - GCC
 - IAR
 - KEIL (AC6)
 - GHS
- 根据Order和Name严格控制每个区间在内存上的布局
- 灵活的初始化策略，指定RAM区间初始化条件

- 不初始化（没有ECC的情况, 或者bootloader已经初始化）
- 正常初始化
- POR（Power On Reset）初始化
- 灵活的对齐模式
 - 开始地址对齐
 - 结束地址对齐
- 自动计算内存区间使用大小，开始地址和结束地址等
- 图像化拖拽界面配置，所见即所得
 - 可视化内存分配
 - 直观的内存COPY展示
- SYMBOL NAME自动产生，根据用户配置自动产生所有需要的SYMBOL
- 数据化的配置，方便MPU，Cache，Secure Boot的使用
- 默认配置，所有的芯片的都包含了默认YT-LINK配置，开箱即用

核心概念和使用指南

功能介绍

完整的一个YT-LINK配置需要包括以下部分，每个操作快在鼠标悬停在上面的时候会显示如下的操作符号，



	删除这个操作快，Block操作快不能删除
	添加子操作快，Region操作快作为最小元素，不能再添加子的操作快
	编辑操作快，Block操作快的基础信息是更加芯片固定的，不能删除

用户也可以用窗口右下角的交互按钮来控制YT-LINK的显示和配置。



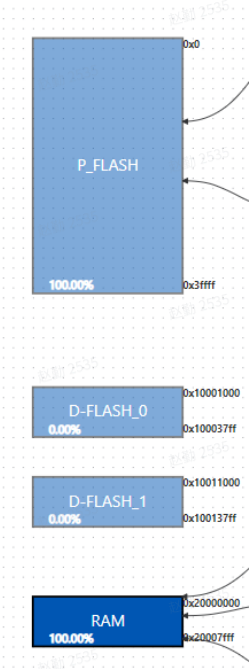
	恢复系统默认的芯片初始化值
	显示整个YT-Link的配置树
	放大
	缩小
	自适应窗口

Order

绝大多数操作快都要Order的属性，Order决定了操作快的排序优先级，Order越小，优先级越高，同样Order的情况下按照Name来排序。

Block操作快

每个芯片都有固定的基础Block，例如我们常见的内部FLASH和RAM，当然也可以增加一些额外的Block如外部的FLASH（Nor/Nand Flash)或者RAM(SRAM，DDR）等。



- Block操作快会根据子模块的使用大小，动态显示整体的使用率
- Block操作快会列出开始的物理地址和结束的物理地址
- Block操作快可以通过添加按钮来添加子的Memory操作快

Memory操作快

每个Block操作快下可以有多个Memory操作快，每个Memory需要指定一个具体的大小空间，所有Memory的和不能超过父Block操作快的大小。Memory可以指定从上开始计算还是从下开始计算，支持多种初始化策略，如不初始化，POR（Power on reset) 下初始化，正常初始化等。Order属性决定了排列顺序。

×

Edit Memory

Name

RAM

Boundary ⓘ

UPPER

Order ⓘ

1

Memory Size

0x7800

Init Policy ⓘ

NORMAL

Save

- Name: 同一个Block下操作快的Name必须是唯一的
- Boundary
 - UPPER: 从上（小地址）开始计算
 - LOWER: 从下（大地址）开始计算
- Order: 排序优先级，Order相同，按照Name来排序
- Memory Size: Memory操作快的大小
- Init Policy:
 - NULL，不初始化
 - NORMAL，正常初始化
 - POR_ONLY，PowerOnReset的时候才初始化

左图的配置，由于IVT_RAM排在第一个位置（方便地址对齐，不浪费RAM），我们讲IVT_RAM的Order设置为0，RAM的Order设置为1，由于STACK是的Boundary是LOWER，他的Order不会影响UPPER区域的排序。Memory操作快可以添加Group操作快

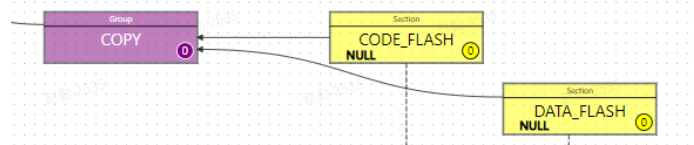
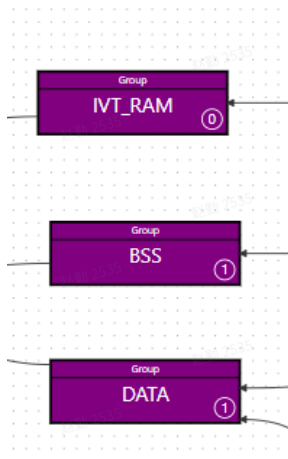
Group操作快

Group操作快是一个虚拟的概念，可以方便Section操作快的管理和分类。Group操作快下面可以有多个Section操作快。

Edit Group

Name

Order ⓘ



Section操作快

Section操作快是链接文件中重要的组成部分，Section支持灵活的配置，如对齐，固定大小，复制等功能（data区域的copy）。

Edit Section

Name

Order ⓘ

Alignment[Byte]

End Alignment[Byte]

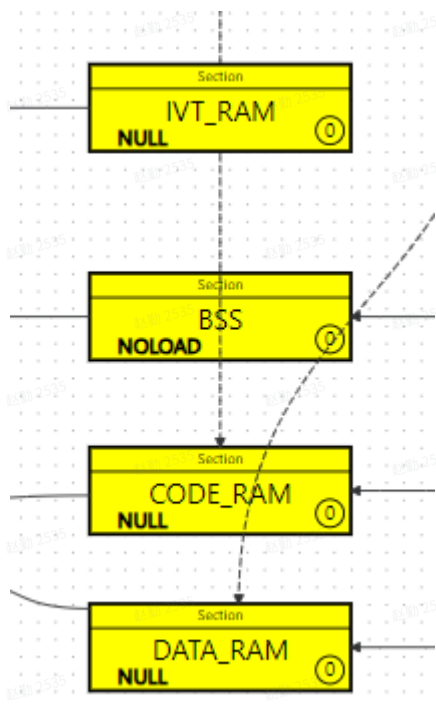
Section Size

Flag

CopyFrom

Clear Section ☐

- Name: 同一个Group操作快下的Name必须是唯一的
- Order: 排序优先级，Order相同，按照Name来排序



- Alignment: 必须是2的倍数，Section开始地址的对齐要求
- End Alignment: 必须是2的倍数，Section结束地址的对齐要求
- Section Size: 当Section操作快没有子的Region操作快时，可以指定固定的大小来撑开一个区间，stack，或者heap通常会用到这个功能。
- Flag: 对Section的一些特殊属性定义，这里每个编译器会有差别
 - GCC: 支持NOLOAD,
 - 其他不支持
- CopyFrom: 只有RAM块才有这个属性，当添加了CopyFrom代表这个RAM区间的内容会在初始化的时候从Flash复制到RAM里，被Copy的Section操作快的子Region操作快这个时候没有意义，以发起Copy的Section为准。被copy的section会有一个箭头指向到发起copy的section
- Clear Section: 是否清空这个Section，BSS默认要开启这个属性。

每个section有如下的symbol name, **`${section_name}_start`**, **`${section_name}_end`**, 分别对应这个section实际的Block起始地址和结束地址。

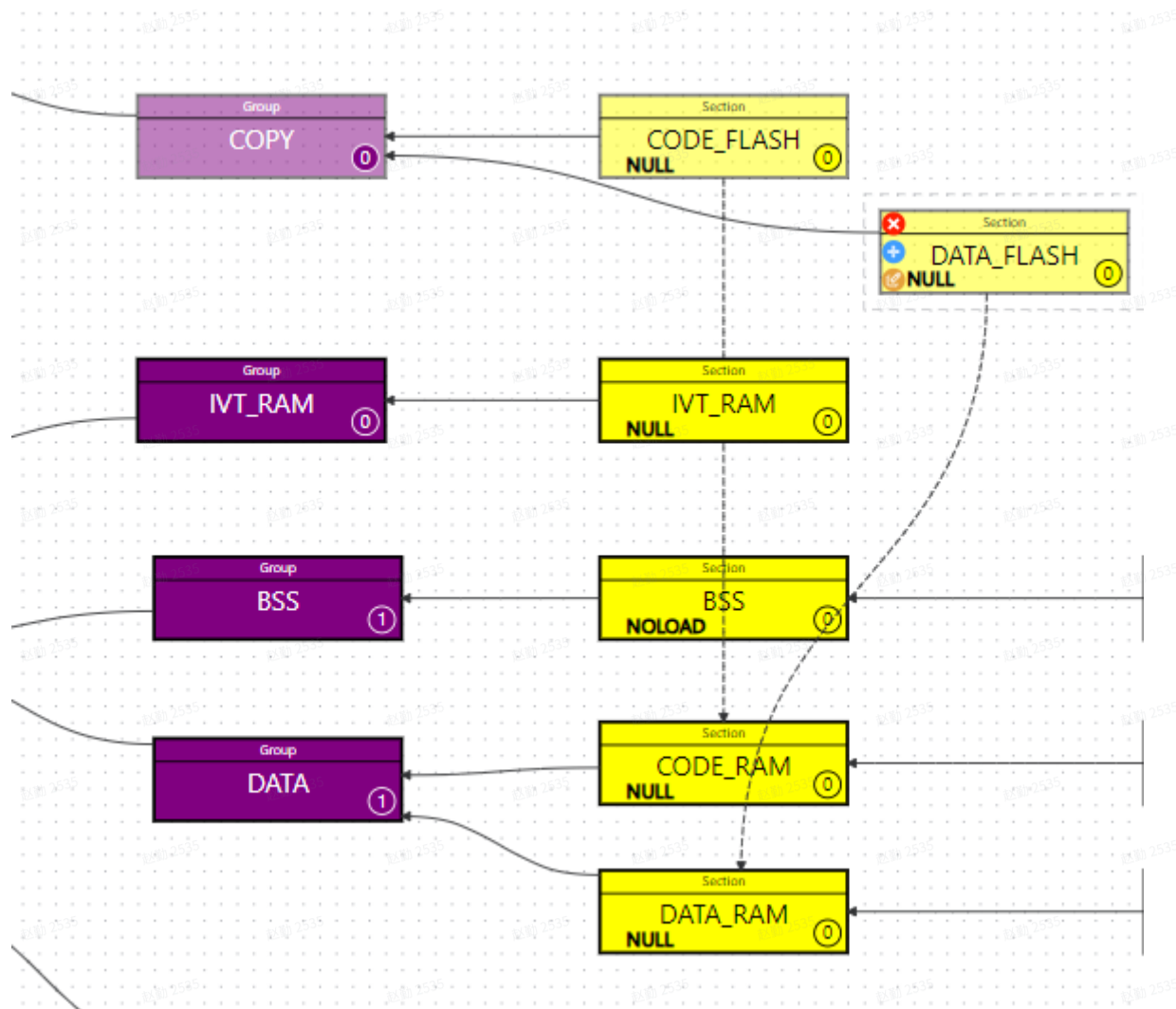
如上图：你可以在你的C代码或者汇编代码访问，**`IVT_RAM_start`**，**`IVT_RAM_end`** 等symbol名字。

KEIL限制

Copy From限制

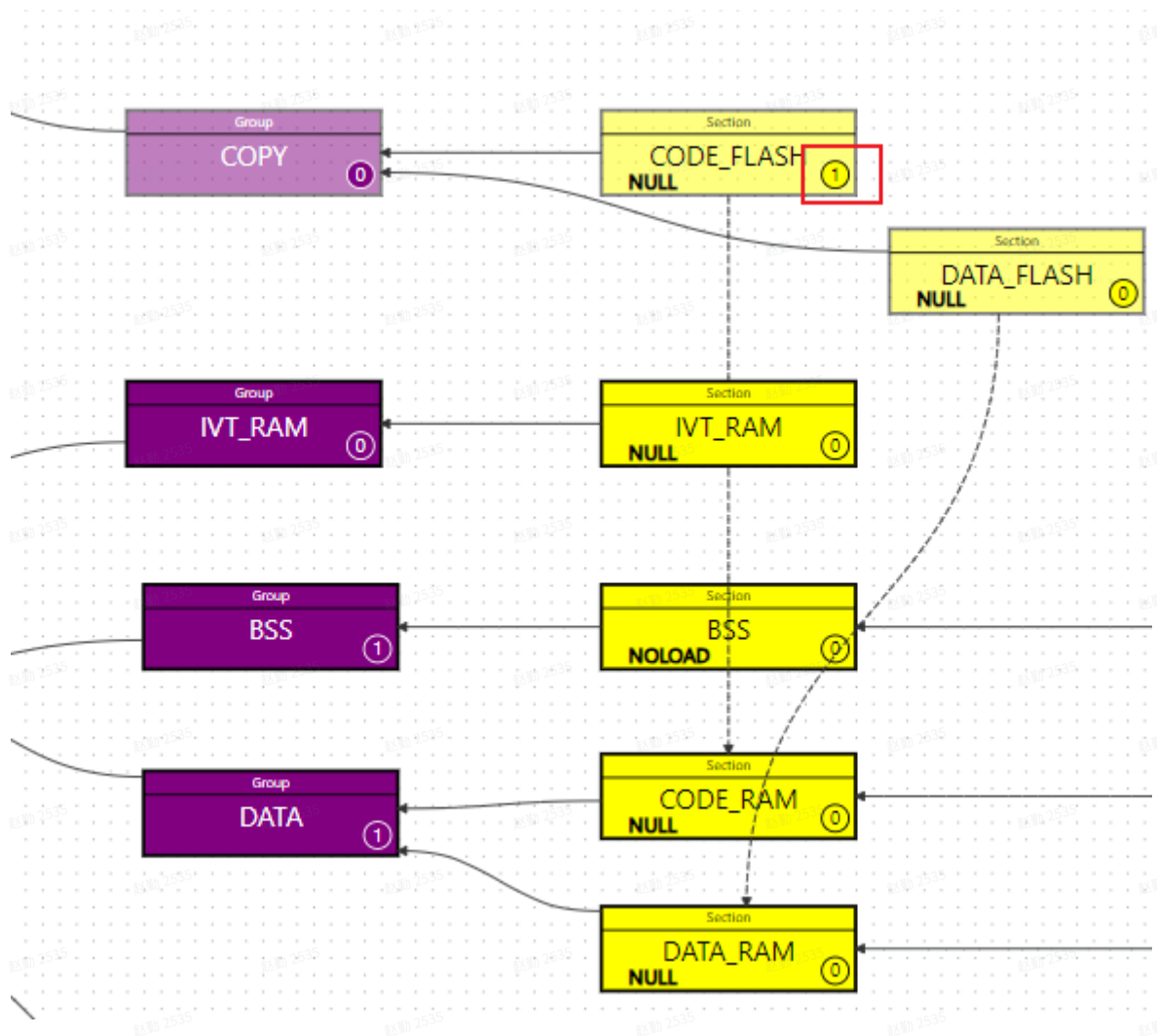
由于Keil链接文件的语法特性，要求RegionName在使用前必须已经被定义(具体可以参考KEIL-Arm® Compiler for Embedded Reference Guide, 4.7.6 章节)，所以在有多个CopyFrom Section存在的情况下，如果A的Section优先级高于B的Section的优先级，那么被A Copy的Section的优先级也要高于被B Copy的优先级。

可以工作的情况如下图：



CODE_RAM 的优先级高于DATA_RAM (但他的Order意义, 但是字母C大于字母D), 同时被COPY的的CODE_FLASH的优先级也是高于DATA_FLASH.

不可以工作的情况如下图:



CODE_RAM 的优先级高于DATA_RAM (但他的Order意义, 但是字母C大于字母D), 但是被COPY的CODE_FLASH的优先级小于DATA_FLASH (DATA_FLASH的Order小于CODE_FLASH的Order) . 这个情况下会提示如下错误:

✖ ymlink: The section is not enough to be arranged, see YT-LINK UM KEIL Limitation for more information.

SymbolName限制

💡 由于KEIL的限制, $\text{\$}\{\text{section_name}\}_\text{start}=\text{large}\$\$\{\text{section_name}\}_\text{start}\$\$Base$, $\text{\$}\{\text{section_name}\}_\text{end}=\text{large}\$\$\{\text{section_name}\}_\text{end}\$\$Limit$

在KEIL中, 你可以访问 $\text{large}\$\$IVT_RAM_start\$\$Base$, $\text{large}\$\$IVT_RAM_end\$\$Limit$ 等 symbol名字。

Region操作快

Region操作对应的是section具体里面包含的section name, 如.data,.txt,.bss等

Edit Region ×

Name

Order

Alignment[Byte]

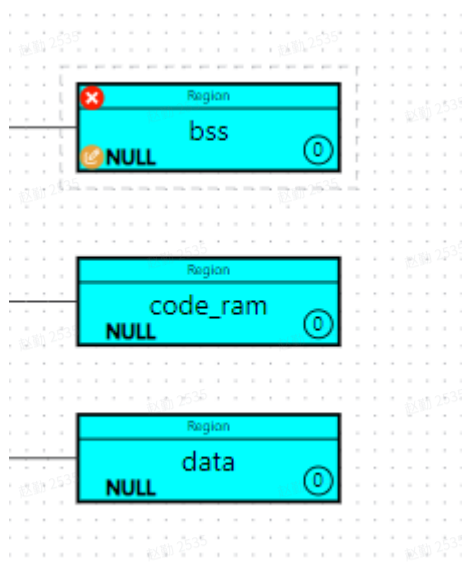
Flag

FileNames

Alt Symbol Start Alt Symbol End

Add Wildcard ☐

上图会生成code_ram的区域，可以使用`attribute((section (".code_ram")))`把对应的代码放到RAM区间。



- Name: 同一个Section操作快下的Name必须是唯一的,所有的name会自动加上"."作为前缀，不需要再在name里加"."
- Order: 排序优先级，Order相同，按照Name来排序
- Alignment: 必须是2的倍数，Region开始地址的对齐要求
- Flag: 对Region的一些特殊属性定义，这里每个编译器会有差别
 - KEEP: 都支持，KEIL通过--keep链接属性来支持
- FileNames: 把.o或者.a 放入某个region, 具体可以相应的例子和demo
- Alt Symbol Start: 生成一个额外的region开始Symbol
- Alt Symbol End: 生成一个额外的region结束Symbol
- Wildcard: 勾选这个选择后，region的名字会变为，.name*, 不同编译器可能有细微差异，方便把.text.mane 也放到text Region区间内

每个Region有如下的symbol name `${region_name}_region_start`, `${region_name}_region_end`, 也支持增加额外的开始和结束symbol name, 开始的 **alt symbol start** = `${region_name}_region_start`, 结束的 **alt symbol end** = `${region_name}_region_end`.

KEIL限制

Name限制

如果这个section是用来放变量的, 那么要根据情况在name里加入 `bss.${name}` 或者 `data.${name}` 的前缀。

The section attribute specifies that a variable must be placed in a particular data section. Normally, armclang places the data it generates in sections like `.data` and `.bss`. However, you might require additional data sections or you might want a variable to appear in a special section,

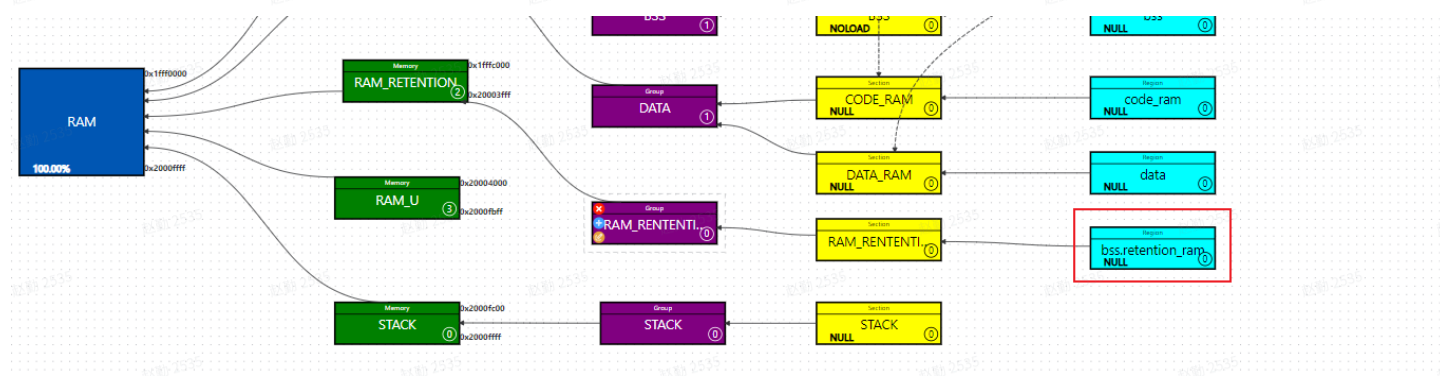
for example, to map to special hardware.

If you use the section attribute, read-only variables are placed in RO data sections, writable variables are placed in RW data sections.

To place ZI data in a named section, the section must start with the `.bss.` prefix. Non-ZI data cannot be placed in a section name with the `.bss.` Prefix.

From Keil attribute((section("name"))) variable attribute

例如 `uint32_t a` 要放到 `retention_ram` region, 那么region名字必须要为 `(bss.retention_ram)`



同时代码也要如此

```
attribute((section(".bss.retention_ram")))
```

```
uint32_t retentionRamData[RETENTION_RAM_SIZE_IN_WORDS/100];
```

SymbolName限制

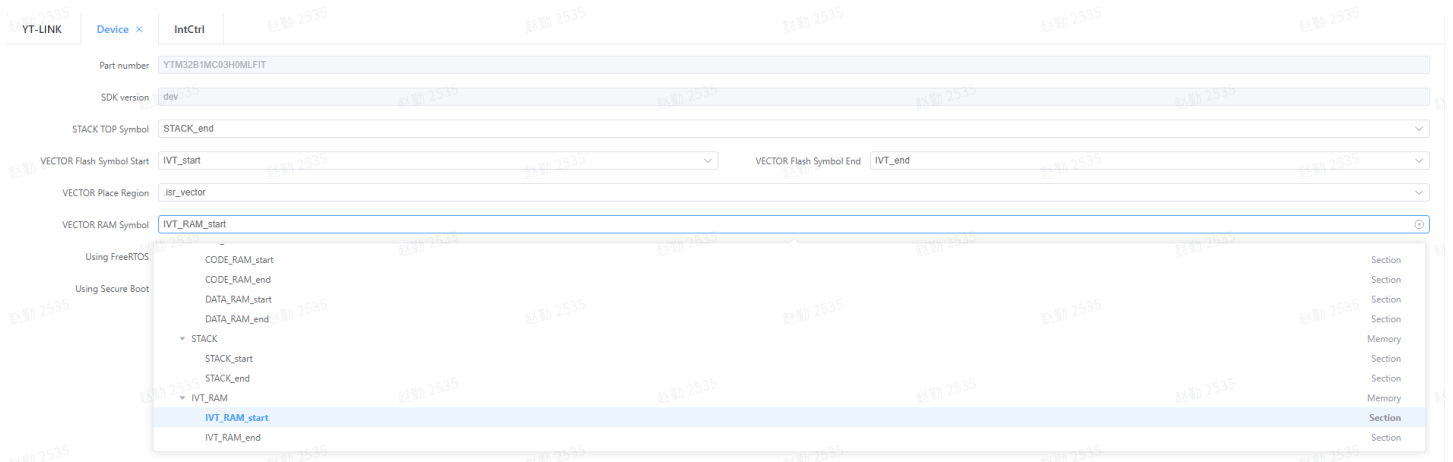


由于KEIL的限制，

$\{\text{section_name}\}_{\text{region_start}} = \text{large} \{\text{section_name}\}_{\text{region_start}} \text{Base}$,
 $\{\text{section_name}\}_{\text{region_end}} = \text{large} \{\text{section_name}\}_{\text{region_end}} \text{Limit}$

其他模块依赖

Device模块需要选择对应的LINK的section或者region来生成正确的启动文件，如下图

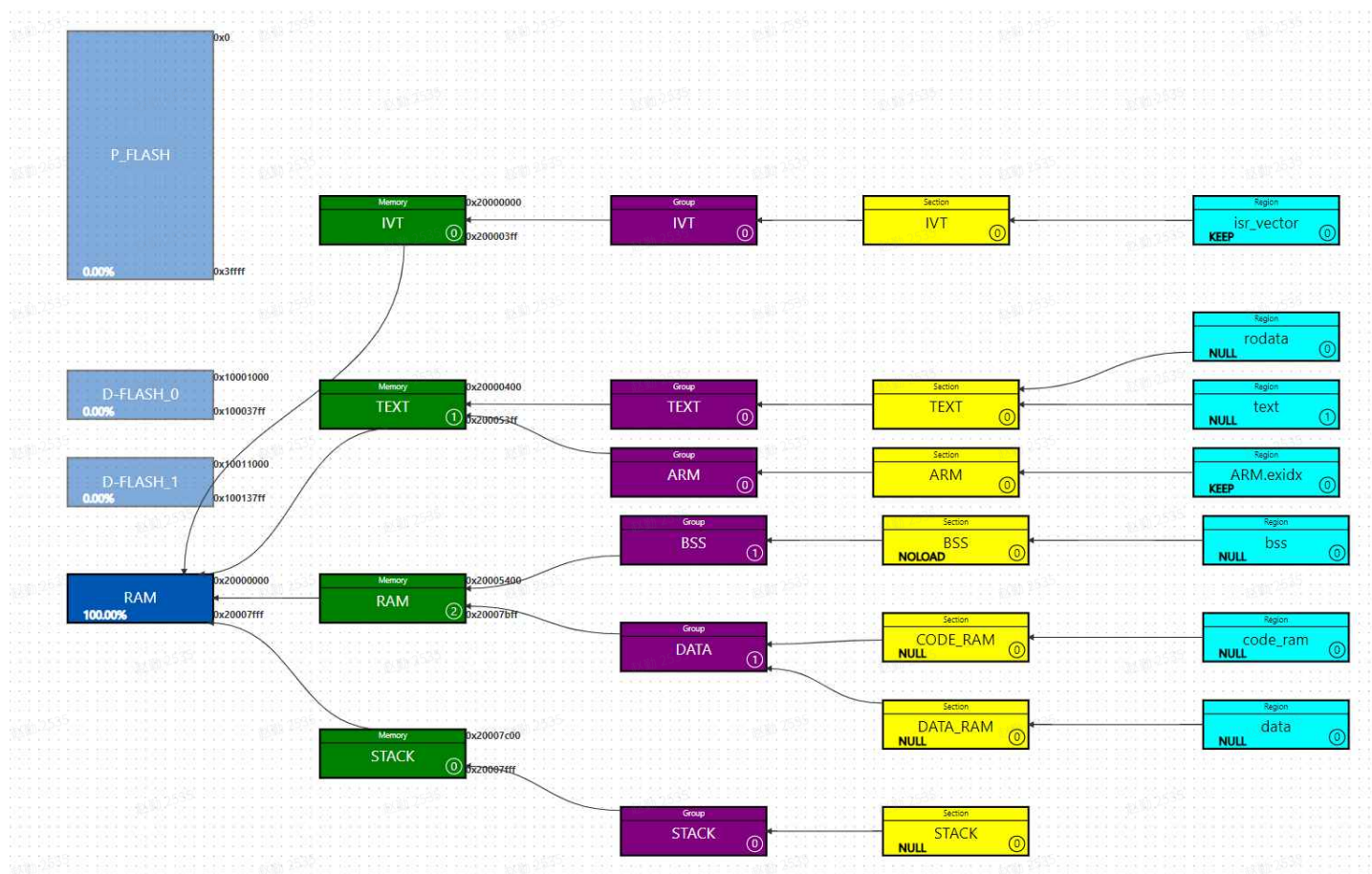


VECTOR RAM SYMBOL 需要选择一个RAM Section来存放VECTOR_TABLE。

Example

在RAM上运行

以YTM32B1MC03为例，YTM32B1MC03的RAM空间为0x2000_0000 ~ 0x2000_7FFF共32KB，想让程序在RAM上运行可按如下方式配置YT-LINK



注意不用对 RAM memory 进行初始化

Edit Memory

Name

RAM

Boundary

UPPER

Order

2

Memory Size

0x2800

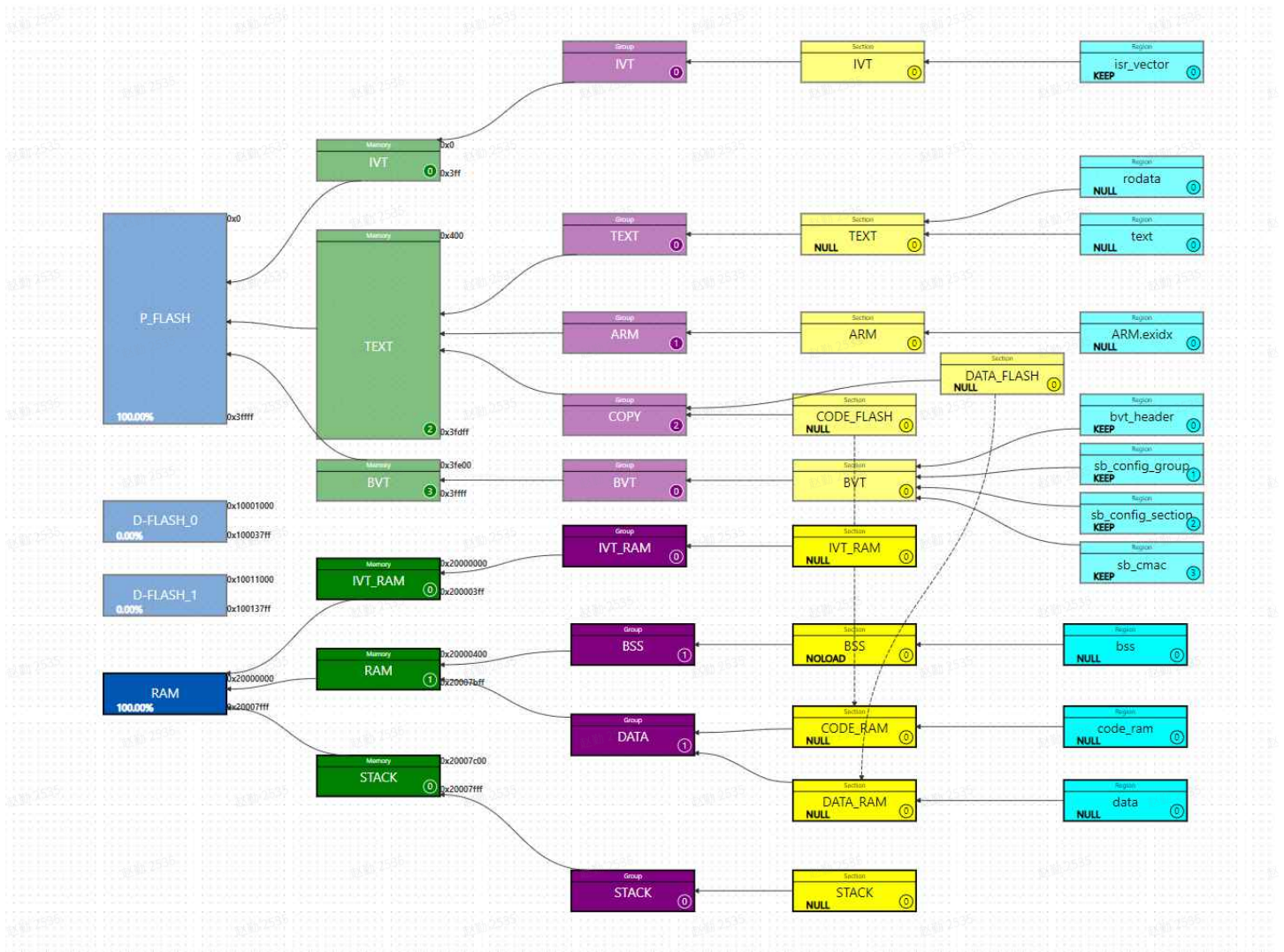
Init Policy

NULL

Save

Secure Boot 配置

以 YTM32B1MC03 为例，YTM32B1MC03 的 Secure Boot 的配置区域(BVT) 可以放在 0x0003_FE00 ~ 0x0003_FFFF，可参考如下配置



开辟一段空间存放Bootloader和APP的交互信息

当Bootloader更新APP时，需要在RAM存放一些临时数据比如更新请求、完成标志，然后产生功能复位。所以这里的需求就是功能复位以后，能够保持一部分RAM中的数据不变。

1. 创建一个Memory例如BOOT_RAM，**Init Policy**选择**POR_ONLY**。保证BOOT_RAM部分Memory仅在上电复位（POR）时初始化一次ECC，当功能复位时不初始化ECC，以便达到功能复位保留RAM内容的效果。

Edit Memory



Name

BOOT_RAM

Boundary ⓘ

UPPER



Order ⓘ

1



Memory Size

8

Init Policy ⓘ

POR_ONLY



Save

2. 创建一个Group例如BOOT_RAM。

Edit Group



Name

BOOT_RAM

Order ⓘ

0



Save

3. 创建一个Section例如BOOT_RAM，**Flag**选择**NOLOAD**。因为该段的数据由Bootloader和APP维护更新，不需要从Text段加载。

Edit Section



Name

Order ⓘ

Alignment[Byte]

End Alignment[Byte]

Section Size

Flag

CopyFrom

Clear Section ☐

4. 创建一个Region例如boot_bss。

Edit Region



Name

Order ⓘ

Alignment[Byte]

Flag

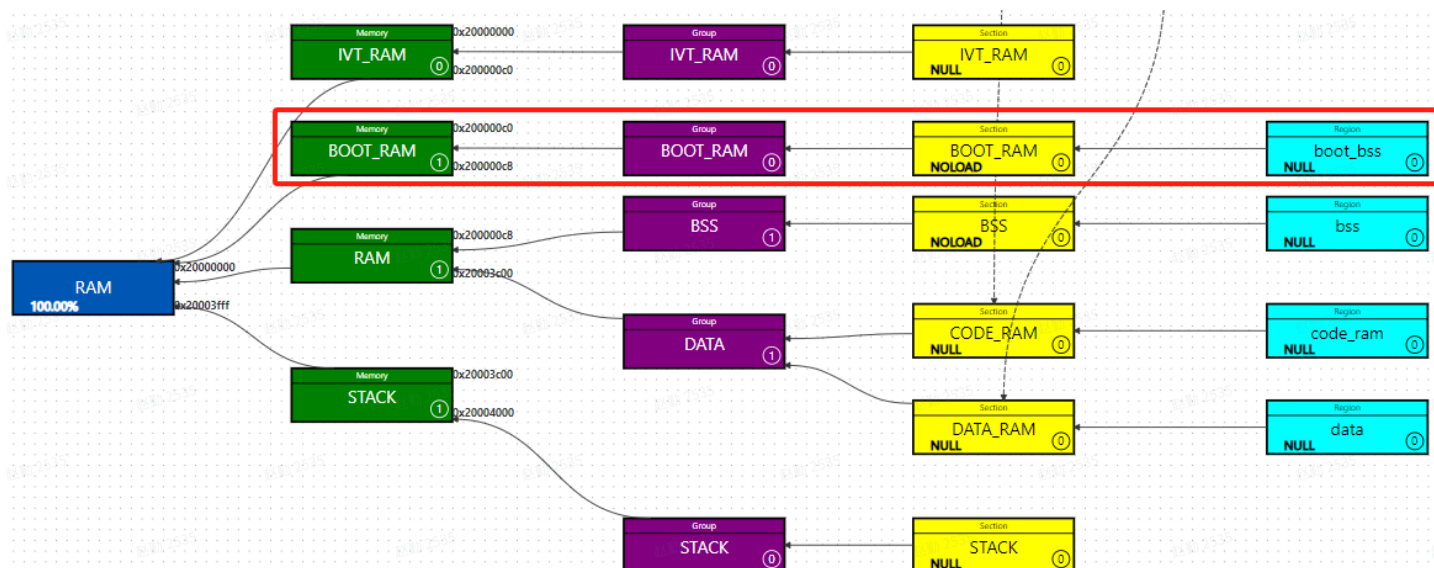
Filenames ⓘ

Alt Symbol Start

Alt Symbol End

Add Wildcard ⓘ ☐

5. 创建完的LINK分布如下图。



6. 在代码中使用使用 `__attribute__((section(".section_name")))` 关键字去修饰声明的变量例如 `BootInfo`。

```
1 __attribute__((section(".boot_bss"))) volatile uint32_t BootInfo[20];
```

该变量在map文件中位置如下

```
1  *(.boot_bss)
2  .boot_bss      0x200000c0      0x8 CMakeFiles/demo.elf.dir/app/main.c.o
3                  0x200000c0      BootInfo
4                  0x200000c8      boot_bss_region_end = .
5                  0x200000c8      BOOT_RAM_end = .
```

利用Region的FileNames把一个库直接放到RAM里运行

有时候我们一些已经编译好的库可以通过FileNames的方式直接放到RAM里，启动文件会自动把整个库的目标区域复制到RAM区域，下面以GCC为例，把libGENERATED_SDK_TARGET.a这个库的text段全部放到RAM里。

1. 修改默认text region 的配置，去掉wildcard勾选，用EXCLUDE_FILE语法，告诉text region 包含除了libGENERATED_SDK_TARGET.a库的其他所有text.*

Edit Region

Name

Order

Alignment[Byte]

Flag

FileNames

Alt Symbol Start

Alt Symbol End

Remove Dot ☐

Add Wildcard ☐

Save

2. 修改data region的配置，把libGENERATED_SDK_TARGET.a的所有text放到data区域

Edit Region

Name

Order

Alignment[Byte]

Flag

FileNames

Alt Symbol Start

Alt Symbol End

Remove Dot ☐

Add Wildcard ☒

Save

这样我们这个库的所有text（函数）就被放到RAM里面运行了，下面是链接文件的部分截图：

```

0x200200b4      0x4 libGENERATED_CONFIG_TARGET.a(clock_config.c.o)
0x200200b4      clock_config0PeripheralClockConfig
.data.g_clockManConfigsArr
0x200200b8      0x4 libGENERATED_CONFIG_TARGET.a(clock_config.c.o)
0x200200b8      g_clockManConfigsArr
.data.linflexd_lin_config0
0x200200bc      0x18 libGENERATED_CONFIG_TARGET.a(linflexd_lin_config.c.o)
0x200200bc      linflexd_lin_config0
.data.SystemCoreClock
0x200200d4      0x4 libGENERATED_SDK_TARGET.a(system_YTM32B1HA0.c.o)
0x200200d4      SystemCoreClock
libGENERATED_SDK_TARGET.a(.text*)
.text.CLOCK_SYS_ConfigureSystemClock
0x200200d8      0x34 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
.text.CLOCK_DRV_GetPllFreq
0x2002010c      0x54 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x2002010c      CLOCK_DRV_GetPllFreq
.text.CLOCK_DRV_GetFreq
0x20020160      0x160 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x20020160      CLOCK_DRV_GetFreq
.text.CLOCK_SYS_Init
0x200202c0      0x18 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x200202c0      CLOCK_SYS_Init
.text.CLOCK_SYS_GetFreq
0x200202d8      0x8 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x200202d8      CLOCK_SYS_GetFreq
.text.CLOCK_DRV_Init
0x200202e0      0x544 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x200202e0      CLOCK_DRV_Init
.text.CLOCK_SYS_UpdateConfiguration
0x20020824      0x100 libGENERATED_SDK_TARGET.a(clock_YTM32B1Hx.c.o)
0x20020824      CLOCK_SYS_UpdateConfiguration
.text.INT_SYS_InstallHandler
0x20020924      0x8c libGENERATED_SDK_TARGET.a(interrupt_manager.c.o)
0x20020924      INT_SYS_InstallHandler

```

MPU CACHE 与 YT-LINK 联动

 彭建棚 后续有时间可以完善一下这个例子