UNIVERSITY OF MICROCHIP
UOFM
MASTERS 2007
第八届中国技术精英年会

11026 C30

MPLAB® C30的高级功能

前提条件/目标

- 您应该熟悉Microchip的16位架构并已具备MPLAB® C30的一些基础知识

- 我们将提供一些高级信息，帮助您了解该工具，从而可使用该架构的高级功能

课程目标

- 今天您将了解到...
  - 混合使用C和汇编语言
  - 多种存储区类型的分配
  - 访问闪存中的数据
  - CodeGuard™安全支持

日程安排

- 混合使用C和汇编语言
  - 概述
  - 从C程序中调用汇编函数
  - 行内汇编
- 存储模型
- 程序空间可视性
- **CodeGuard™安全支持**

混合使用C和汇编语言

使用汇编语言的原因：

- 架构要求：
  - 精确的时序
  - 生成特定代码序列
  - 生成编译器不支持的指令

混合使用C和汇编语言

- 编写完整的汇编函数
  - 从C程序中调用汇编程序或从汇编程序中调用C程序
  - 关键主题：
    - 调用约定和寄存器用法
    - 堆栈用法
- 编写行内汇编
  - 如何引用C变量

混合使用C和汇编语言

如何选择

汇编函数

- 长序列
- 调用成本低
- 限制对C数据的引用
- 允许控制流

行内汇编

- 短序列
- 调用成本太高
- 引用C数据
- 无控制流

外部汇编函数

- **MPLAB® ASM30参考手册：DS51317F_CN**
  - 汇编语法
- **dsPIC30F/33F程序员参考手册：DS70157B_CN**
  - 汇编语言指令集
- **MPLAB C30用户指南：DS51284F_CN**
  - 第4章，调用约定

外部汇编函数

- 汇编文件的基本形式：

```
.section my_code, code
.global _myfunction
; myfunction is externally visible
; and starts here!
_myfunction:
 clr w0
 ; and so on
```

外部汇编函数

- 如何从C程序调用外部汇编函数？
- 首先，用**extern**声明函数

```
extern void myfunction(void);
```

- 随后，将其作为普通函数进行调用

```
void main(void) {
    myfunction();
    /* and so on */
}
```

调用约定

- **参数存放到W0至W7**
  - 参数被存放到第一个*正确对齐*的寄存器
  - 从最左边的参数开始存放
  - 如果有足够的寄存器空间来保存整个对象
- **其他参数被压入堆栈**
  - 首先压入最右边未分配的参数

调用约定

- 值返回到**W0**
  - 如果需要，也会返回到W1至W3
- 被调用函数可使用**W0-W7**，而不保存其内容
  - 返回调用函数后，这些寄存器无需保持原值
  - 必须保存W8-W15的内容
- 无未使用的寄存器
- **ISR——保存所有使用的寄存器!**

寄存器对齐

- 什么是*正确对齐*?

Type	registers required	alignment
char	1	none
short	1	none
int	1	none
data pointer	1	none
long	2	even
float	2	even
long long	4	divisible by 4
long double	4	divisible by 4
structure	1 per 2 bytes	none

行内汇编

- 有两种形式
- *简单形式:*

```
asm("assembly text");
```

- *复杂形式:*

```
asm( "template" :
   "format"(variable),... :
   "format"(variable),... :
   "clobbers");
```

行内汇编

- 复杂形式这么复杂!为什么还要使用?
  - 如果汇编指令使用了任何寄存器
  - 如果需要访问任何C变量
- 若不理解原因，将导致无法预料的结果
  - 环境变化时，程序可能无法正常执行

行内汇编

- 格式字符串的用途：
  - 标识所需操作数的类型
  - 约束变量使之保持正确的格式
  - 例如，“r” – 寄存器或
    “m” – 存储器地址

```
asm( “template” :
   “format”(variable),... :
   “format”(variable),... :
   “clobbers”);
```

行内汇编

- 格式字符串还可包含其他信息，如：
  - 只读操作数
  - 只写操作数
- 首先列出输出（写）操作数：

```
asm( "template" :
   "format"(variable),... :// outputs
   "format"(variable),... :// inputs
   "clobbers");
```

行内汇编

- 什么是**clobbers**?
  - 一些指令将隐式地修改寄存器中存储的值
    - 编译器需要知道这种操作何时发生

```
asm( "template" :
   "format"(variable),... :
   "format"(variable),... :
   "clobbers");
```

行内汇编

- 什么是模板？
  - 通常是汇编程序文本
  - 特殊符号**%0,%1,...%n**可指代参数
    - 这些参数可以是修改后的**%d1**

```
asm( “template” :
   “format”(variable),... :
   “format”(variable),... :
   “clobbers”);
```

行内汇编

- 示例：

```
asm ( “add %1, #%2, %0” :
      “=r”(result)      :
      “r”(input), “i”(CONSTANT) );

asm ( “mul.su %1, #%2, %0 :
      “=r”(result)        :
      “r”(input), “i”(CONSTANT) );
```

行内汇编

```
#define CONSTANT 10
int add(int input) {
  int result;
  asm ( “add %1, #%2, %0” :
        “=r”(result)      :
        “r”(input), “i”(CONSTANT) );
  return result;
}
```

行内汇编

```
#define CONSTANT 10
long mulsu(int input) {
  long result;
  asm ( “mul.su %1, #%2, %0” :
        “=r”(result)          :
        “r”(input), “i”(CONSTANT) );
  return result;
}
```

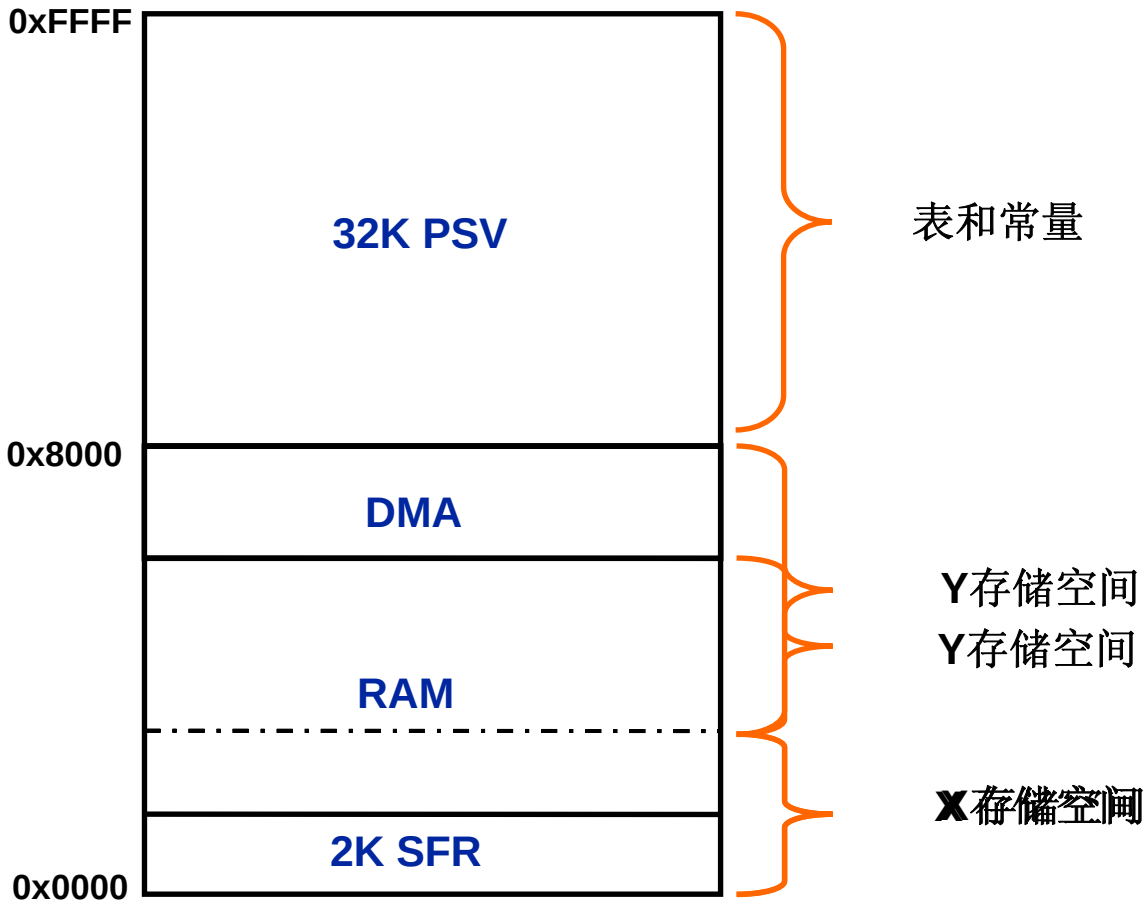
日程安排

- 混合使用C语言和汇编语言
- 存储模型
  - 16位架构概述
  - 应用
- 程序空间可视性
- **CodeGuard™安全支持**

存储器架构

- 哈佛架构
  - 数据RAM
    - **16** 位宽
    - **16** 位地址
  - 程序闪存
    - **24** 位宽
    - **23** 位地址

数据存储空间映射



程序空间存储器

- 存储可执行指令
- 器件配置熔丝
- **EEPROM数据**
- 也可存储数据
  - 通过程序空间可视性（PSV）窗口访问
  - 通过专门的读指令访问
- 执行期间可再编程

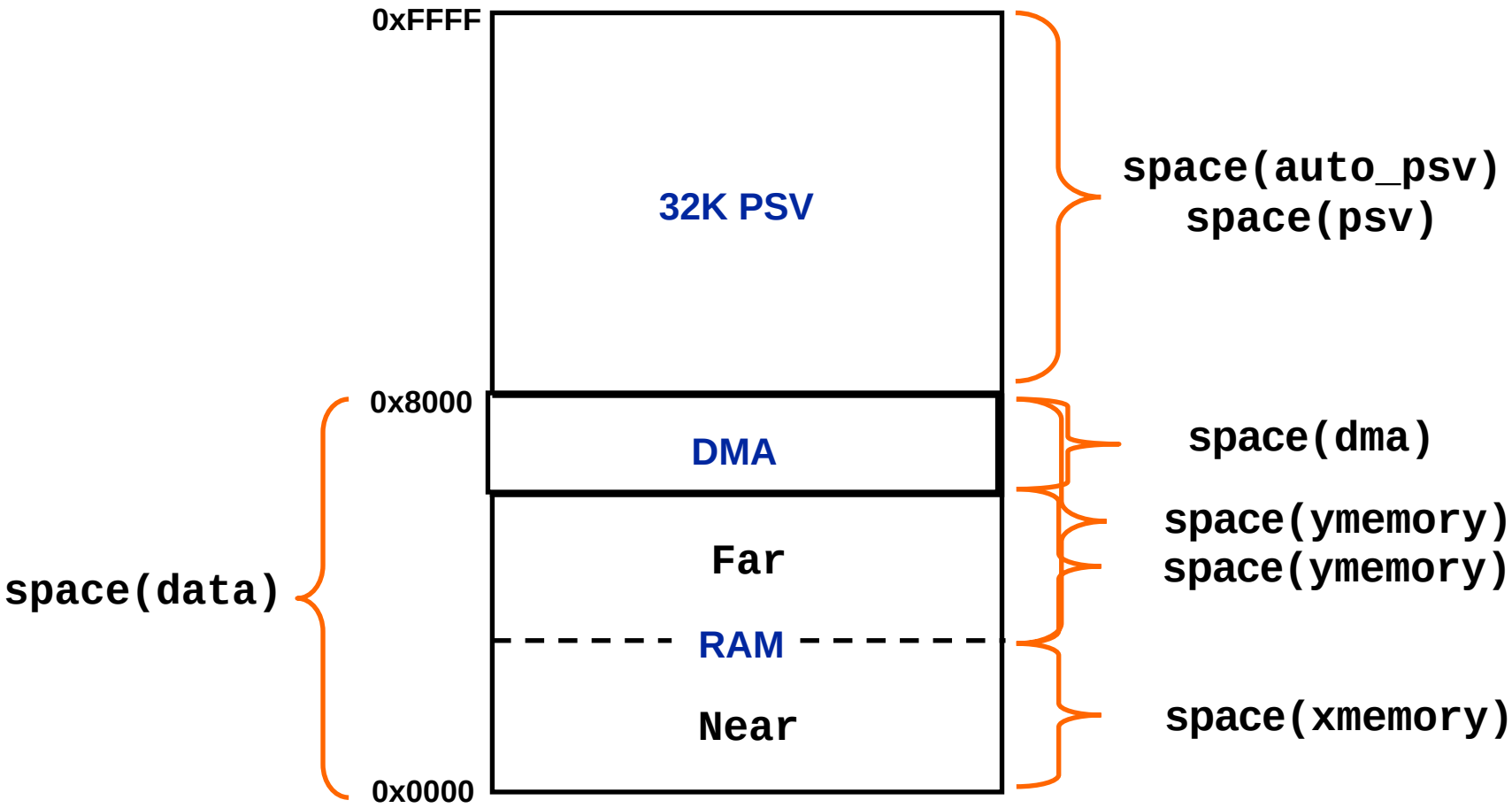
应用

- 为什么要关注存储器布局？
  - 不同的数据存储空间需求：
    - **DSP使用X存储空间或Y存储空间**
    - **DMA使用DMA存储空间**
  - 一些指令序列执行效率更高!
    - 使**ALU存取near存储空间**
    - 相对转移/调用速度更快

应用

- 易于约束存储空间的使用!
- C全局存储模型
  - 通过命令行选项或IDE单选按钮
- 单个存储设置
  - 通过C或汇编语言属性

数据存储空间映射



默认存储模型

- 数据分配

  可扩展数据类型变量： near数据

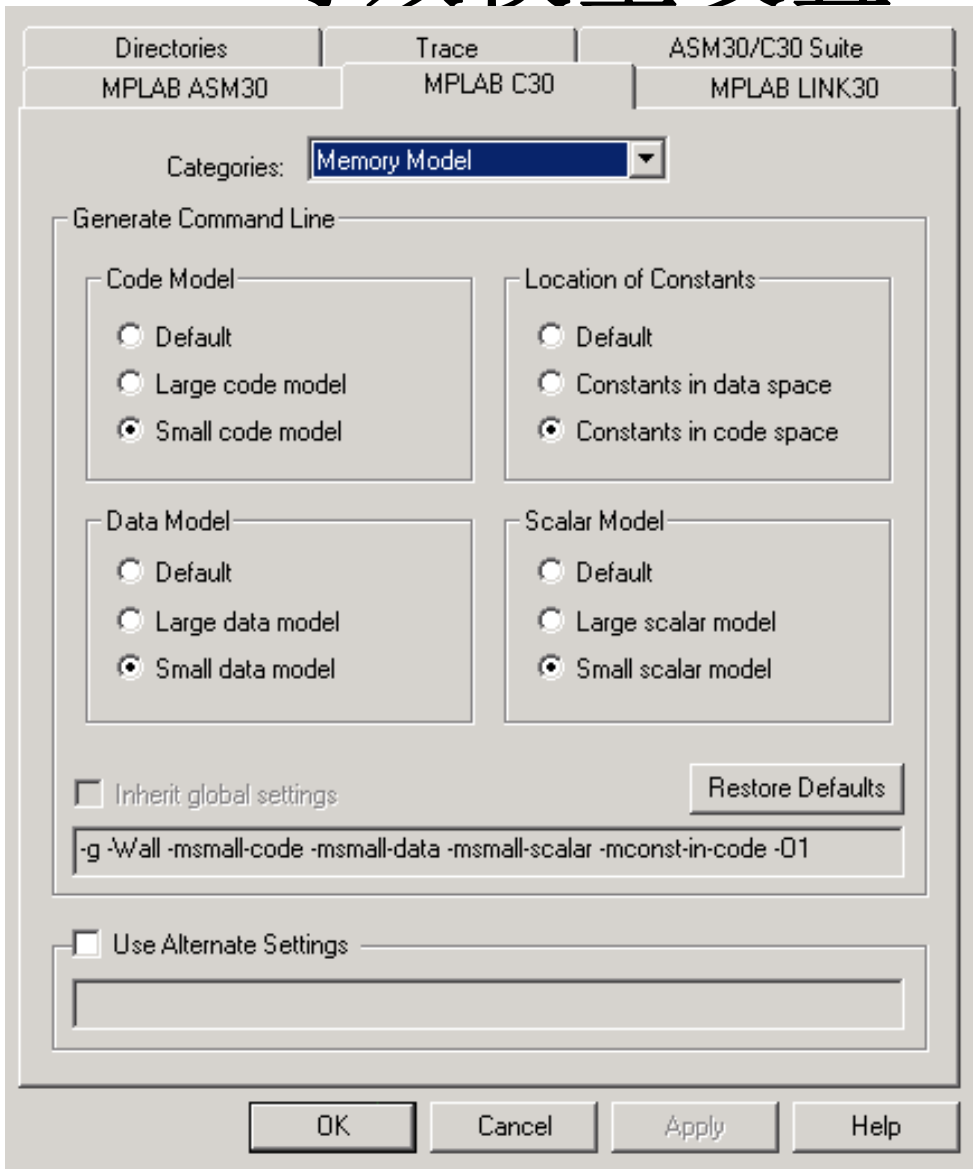
  构造数据类型变量： near数据

  常量： 自动PSV

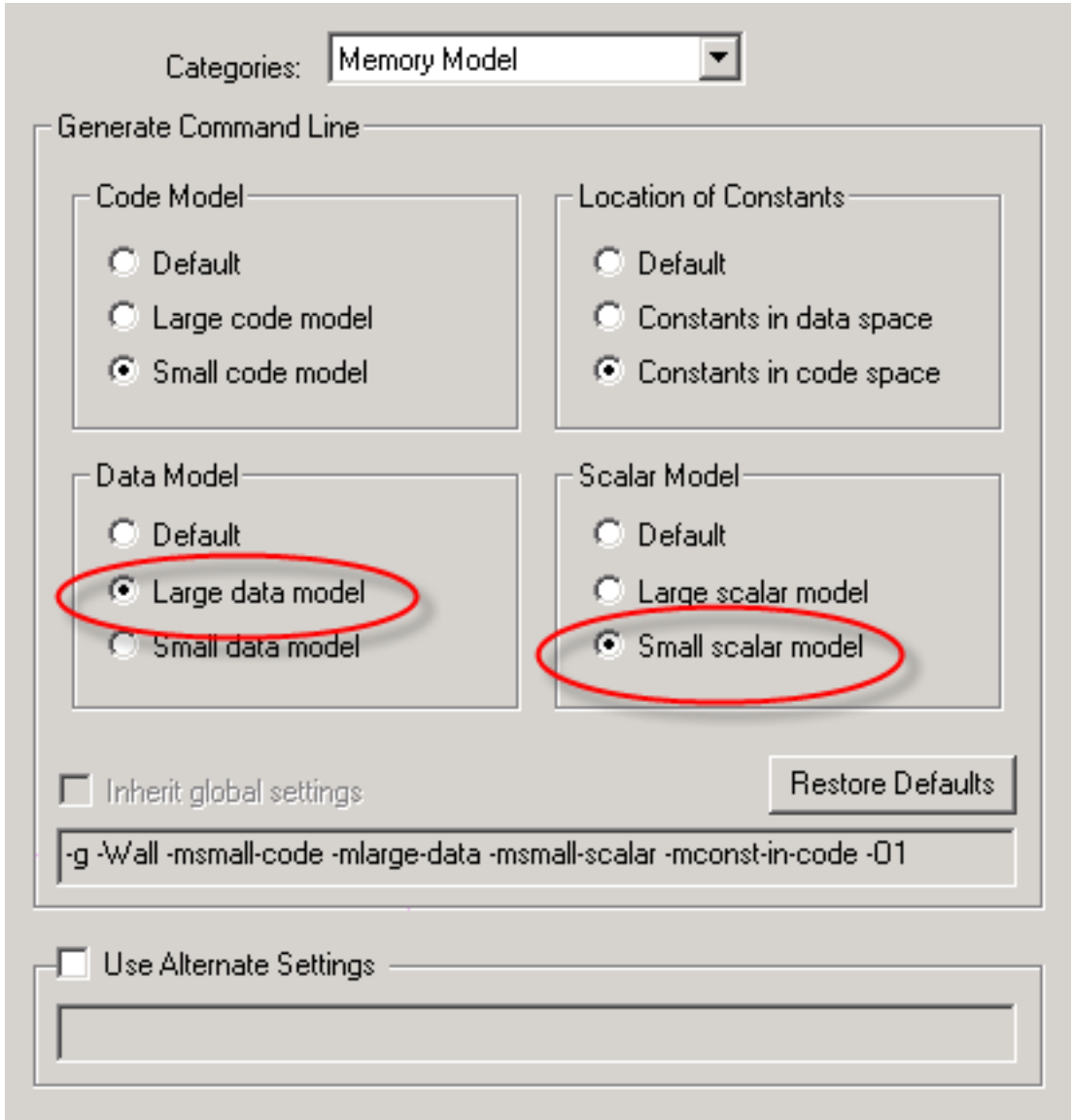
- 函数

  默认为小代码模型

等效模型设置



更好的模型设置？



单个存储设置

- 在全局设置中，不能将变量存放到**X**或**Y**存储空间!
- 在声明中添加属性:

```
int my_data[256]
    __attribute__((space(xmemory)));
int more_data[1024]
    __attribute__((space(dma)));
```

C目标存储空间属性

- **`__attribute__((space(`*`area`*`)));`**

  其中***area***为:
  - **data** - 一般数据空间
  - **auto_psv** - 由编译器管理的PSV
  - **psv** - 由用户管理的PSV
  - **xmemory** - 数据存储空间 （X）
  - **ymemory** - 数据存储空间 （Y）
  - **dma** - DMA存储空间
  - **eedata** - EEDATA存储空间
  - **prog** - 程序闪存

汇编目标存储空间属性

- **`.section` *`name, area`***

  其中***`area`***为:

  - **`data`** - 已初始化的数据存储空间
  - **`bss`** - 清零后的数据存储空间
  - **`psv`** - 由用户管理的PSV
  - **`xmemory`** - 数据存储空间（X）
  - **`ymemory`** - 数据存储空间（Y）
  - **`dma`** - DMA存储空间
  - **`eedata`** - EEDATA存储空间
  - **`code`** - 程序闪存

单个汇编设置

```
.section *,bss,xmemory
.global _my_data
_my_data:
.space 512

.section *,bss,dma
.global _more_data
_more_data:
.space 2048
```

其他属性

- **aligned()** - 起始对齐边界
- **reverse()** - 结束对齐边界
- **near** - near数据（前8K）
- **far** - far数据（任意位置）
- **address()** - 起始地址
- **persistent** - 热复位时未初始化
- **section** - 赋予一个利于分组的特定段名（常用PSV变量）

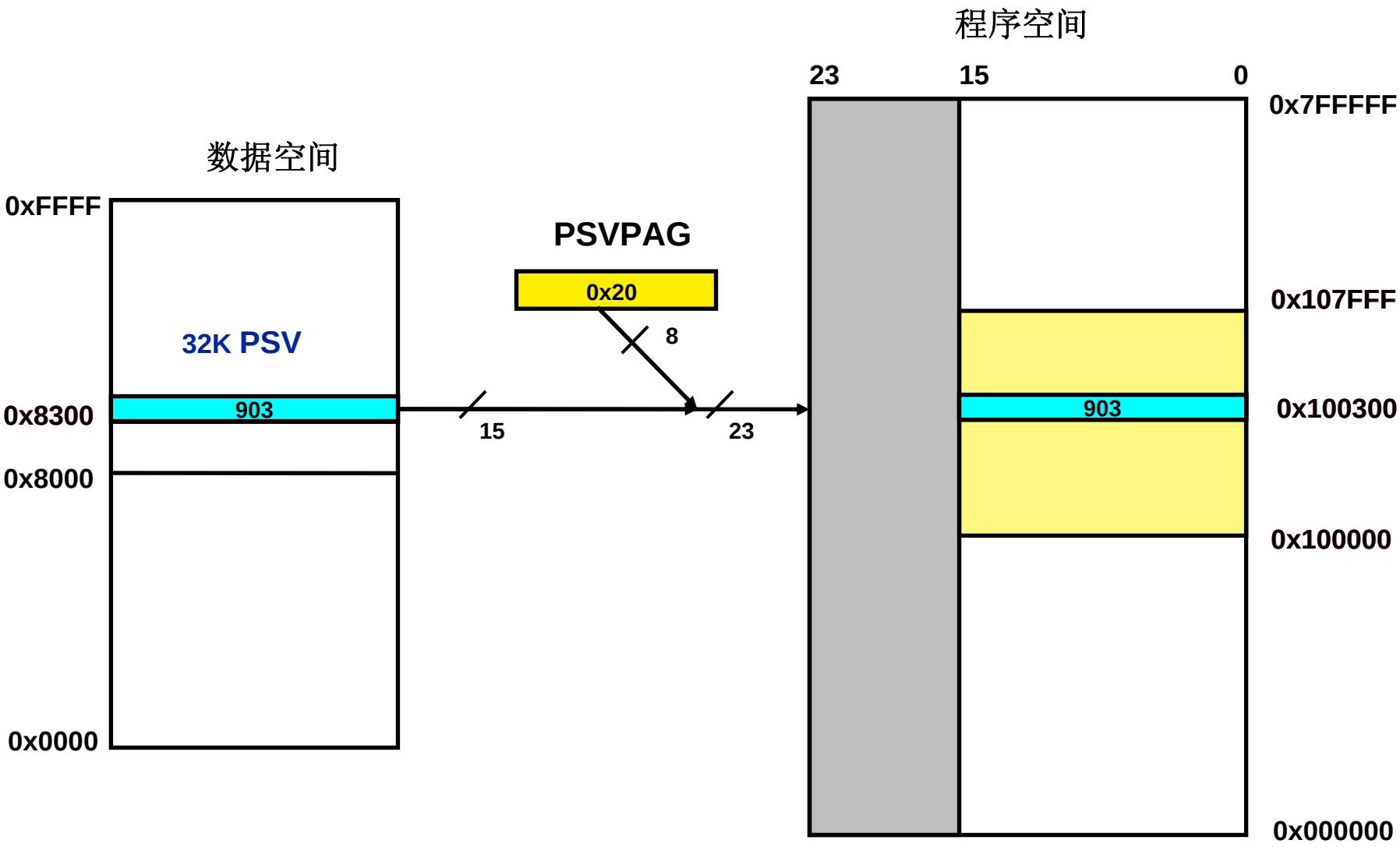
日程安排

- 混合使用C和汇编语言
- 存储模型
- 程序空间可视性
  - 概述
  - 三种操作模式
- **CodeGuard™安全支持**

程序空间可视性

- **PSV**在程序闪存中提供了一个**32K**的数据窗口
- 当使能此窗口时，此窗口将映射到数据空间的**0x8000**至**0xFFFF**
- 编译器支持**3**种**PSV**窗口使用模式
  - 支持由用户管理的PSV
  - 自动PSV模式
  - 由编译器管理的PSV

程序空间可视性工作原理



由用户管理的PSV

- 此编译器工具箱不起任何作用!
- 您必须:
  - 将数据存放到程序闪存
    - `space(psv)`
  - 使能PSV窗口
    - `CORCONbits.PSV = 1;`
  - 配置PSV页
    - `PSVPAG = ???;`

由用户管理的PSV示例

```
int data[256]
    __attribute__((space(psv)));

main() {
  CORCONbits.PSV = 1; // enable PSV
  PSVPAG = __builtin_psvpage(&data);
  // now safe to access data[]
  if (data[26] == 3) {
  }
}
```

自动PSV

- 几乎所有工作均由此编译器工具箱完成!
  - 支持一个32K的PSV页
- 您必须:
  - 将数据存放到程序闪存
    - **`space(auto_psv)`** 或
    - 声明为**`const`**
  - 编译器工具箱使能PSV并设置页

自动PSV示例

```
int data[256]
    __attribute__((space(auto_psv)));

main() {
  // now safe to access data[]
  if (data[26] == 3) {
  }
}
```

管理的PSV

- 几乎所有工作均由编译器工具箱完成！
  - 支持许多32K PSV页
- 您必须：
  - 将数据存放到程序闪存
    - **space(psv)** 或
    - **space(prog)**
  - 将声明标识为受管理状态
  - 编译器工具箱使能PSV

受管PSV示例

```
__psv__ int data[256]
    __attribute__((space(psv)));

main() {
  // now safe to access data[]
  if (data[26] == 3) {
  }
}
```

受管PSV详细信息

- 添加了两种新的类型限定符：

  `__psv__` - 对象不能跨越PSV页

  `__prog__` - 对象可以跨越PSV页

- 将限定符应用到对象时，编译器将在访问前先对**PSV**页进行设置
- 在指针声明中，可修改所指向的对象（如同`const`）

管理的PSV详细信息

- 带有指针的函数不会接受一个受管理的**PSV指针！**
  - 它们是不同的
- 我们的库当前不接受受管理的**PSV指针**
  - **`printf()`** 将不会打印受管理的PSV字符串

受管理的PSV示例

- 受管理的PSV对象

```
__psv__ int foo
  __attribute__((space(psv)));
```

- 指向受管理PSV中的对象的指针

```
__psv__ int *foo_p = &foo;
```

  - 指针存放在数据RAM中

受管理的PSV

- 小结：
  - 依然受32K数据项的限制
  - 指针较大，可容纳页信息
  - 访问速度较慢（必须设置页）
  - 可能需要修改中断服务程序
  - 支持Beta版

日程安排

- 混合使用C和汇编语言
- 存储模型
- 程序空间可视性
- **CodeGuard™安全支持**
  - 引导段和安全段
  - 执行控制
  - 安全模型

CodeGuard™安全

- 什么是**CodeGuard**安全？
  - 一种硬件功能，能够...
    - 将存储器划分为2个或3个段
    - 控制段之间的可见性和执行
- 有何用途？
  - 允许多方共享同一芯片上的资源

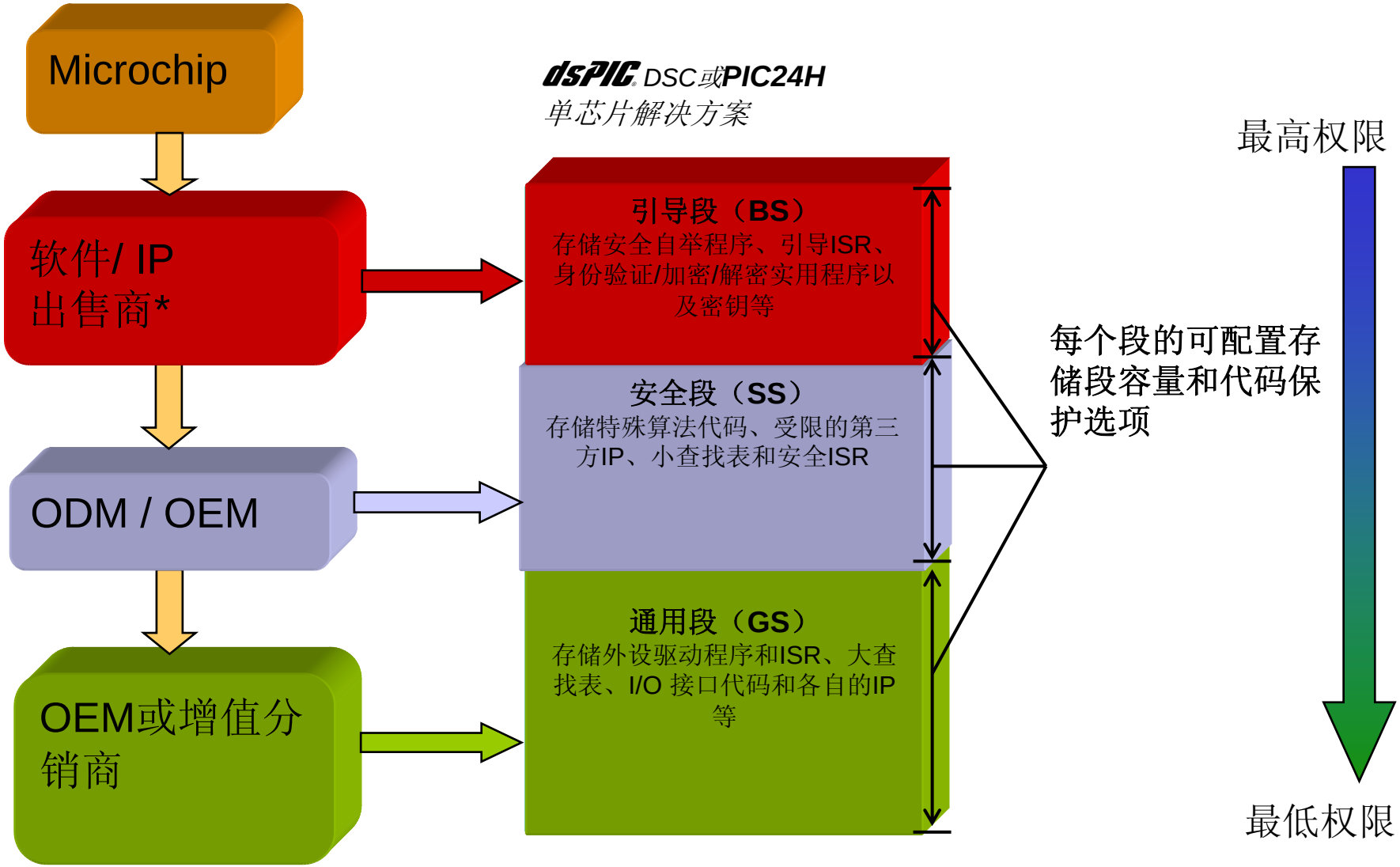
CodeGuard™安全

- 哪些器件系列具有**CodeGuard**安全功能？
  - **dsPIC33F、PIC24H**和一些**dsPIC30F**器件
  - 但语言扩展可用于任何器件！

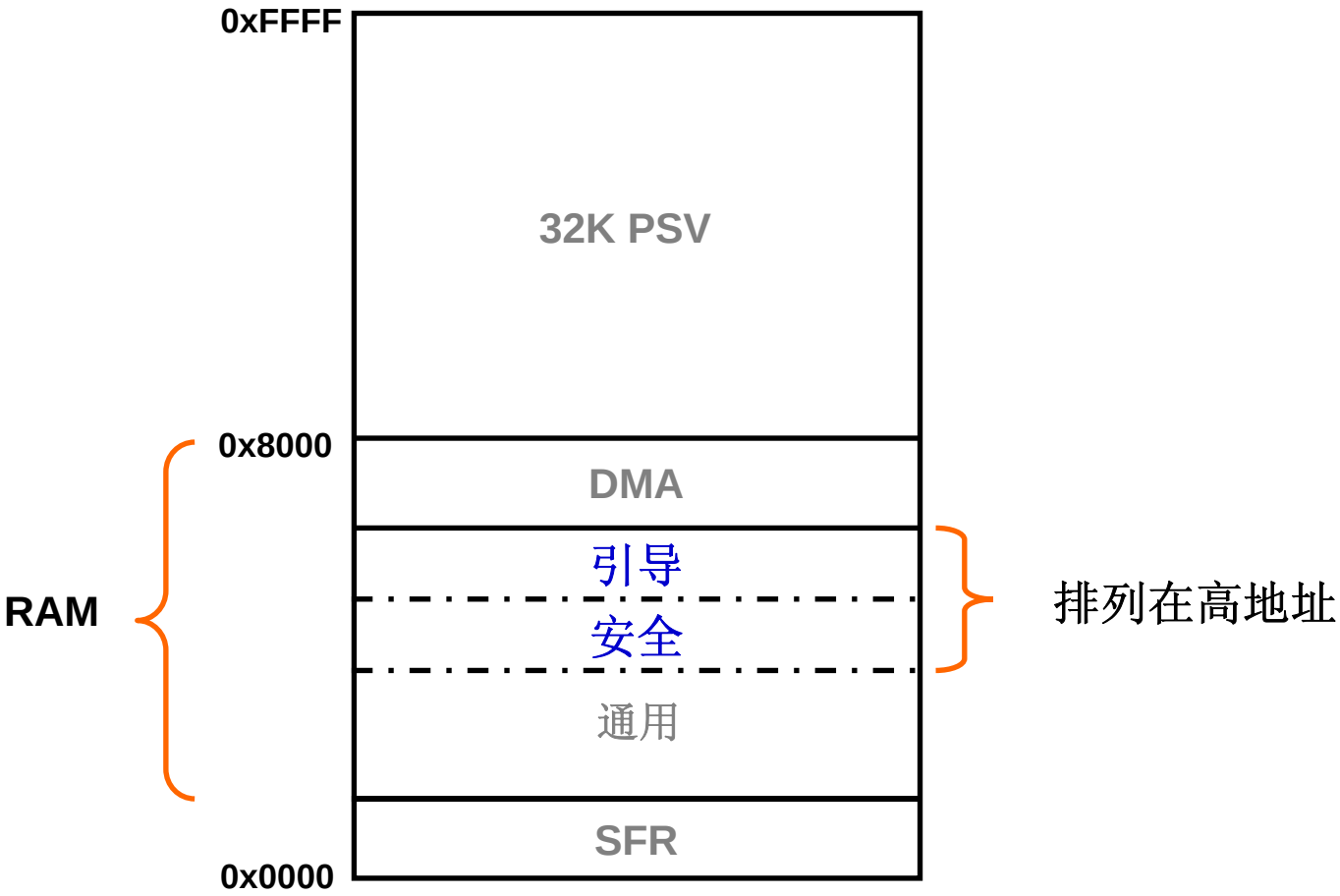
引导段和安全段

- 存储器可划分为：
  - 1或2个特殊段
  - 加一个通用段
- 段容量可以是小容量、中等容量或大容量（根据器件而不同）
- 每个段可独立链接

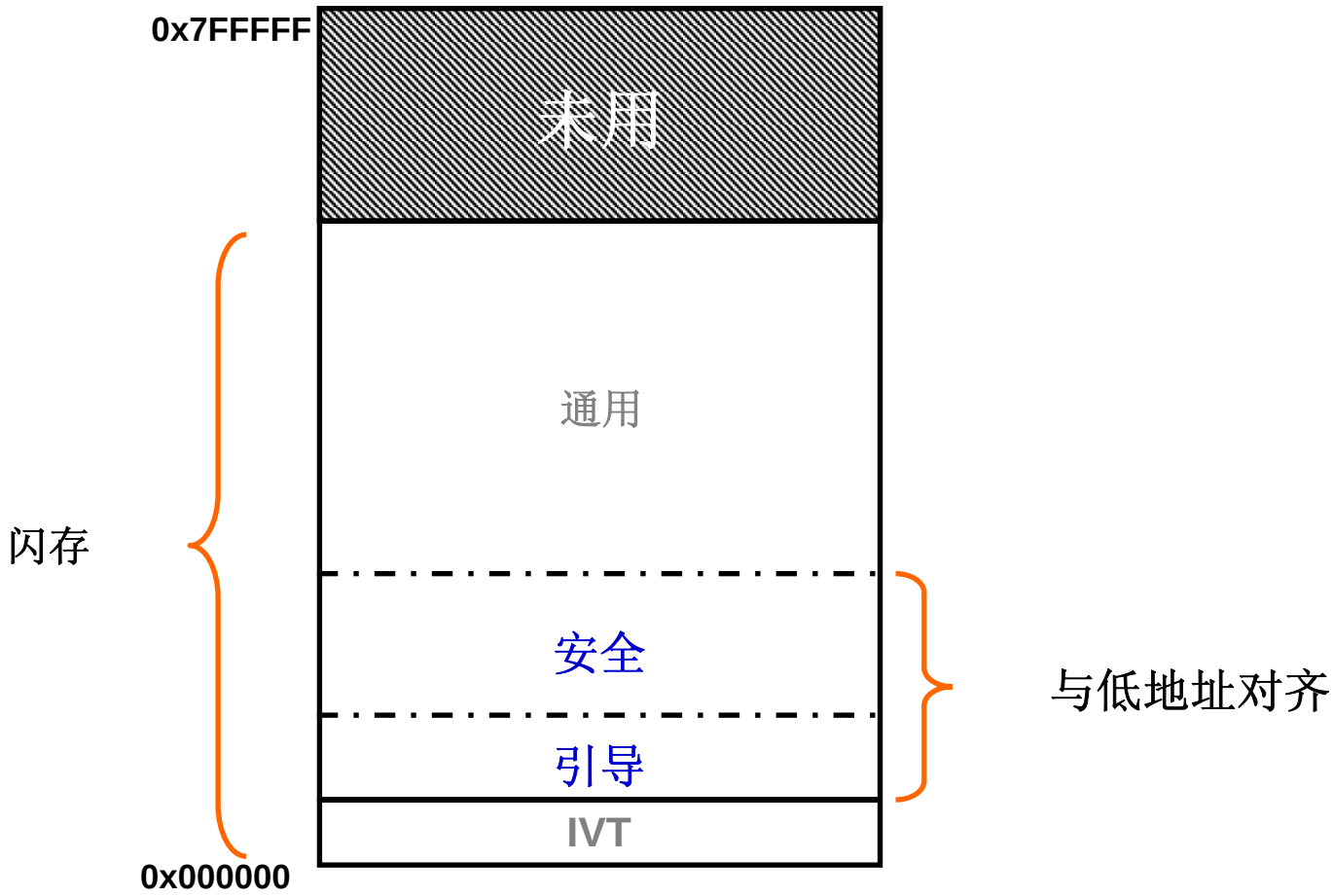
引导段和安全段



数据空间中的段



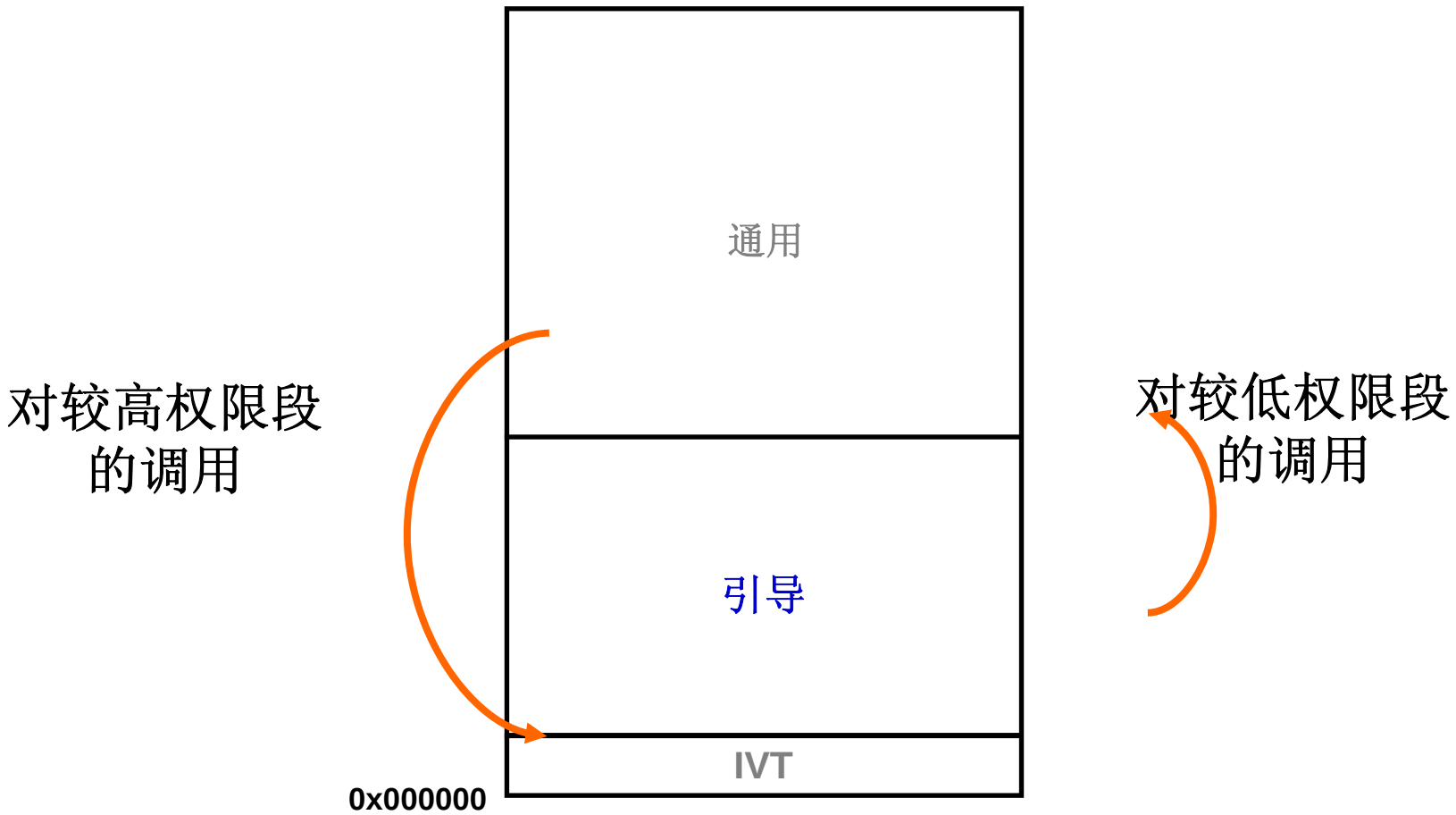
程序空间中的段



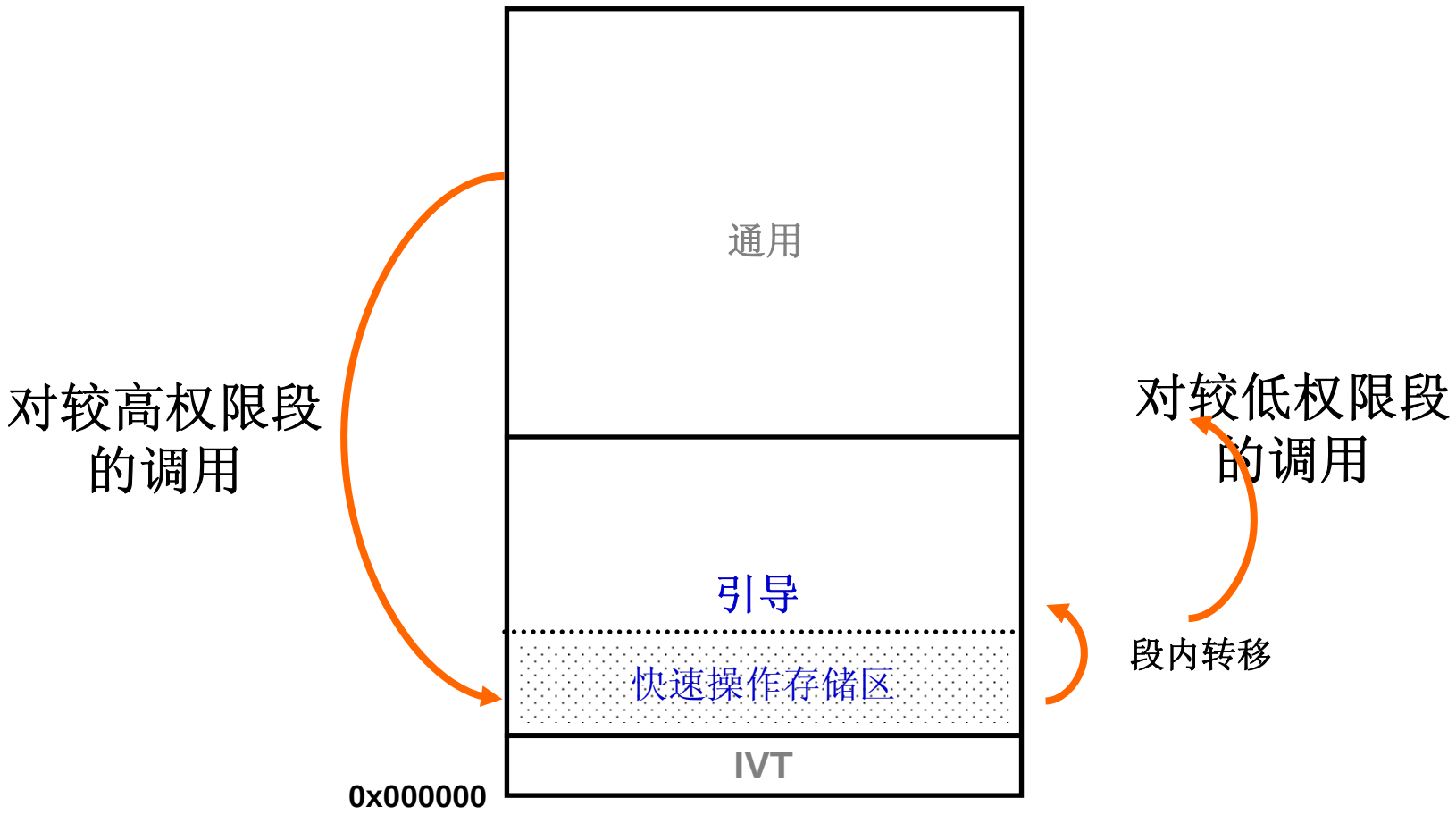
执行控制

- 高权限的段始终可调用较低权限的段
- 标准安全性
  - 允许调用较高权限的段
- 高安全性
  - 必须通过快速操作存储区来实现对更高权限段的调用

执行控制：标准安全性



执行控制：高安全性



快速操作存储区

- 具有高安全性：
  - 只有前32个存储单元可由具有较低权限的段访问
  - 工具创建快速操作存储区并自动管理引用
  - 被实现为跳转表

快速操作存储区

- 通过访问入口槽（**slot**）编号传送控制信号是单独链接的程序段的关键
- 在任何器件上使用这些结构
  - 即使硬件中无CodeGuard™安全功能

快速操作存储区示例：C

- 如何调用访问入口槽（**slot**）？
- 首先，声明函数

```
extern void __attribute__((boot(4)))
   myfunction(void);
```

- 随后，将其作为普通函数进行调用!

```
void main(void) {
    myfunction();
    /* and so on */
}
```

C示例（续）

- 如何定义访问入口槽（**slot**）？
- 使用**boot**或**secure**属性

```
void __attribute__((boot(4)))
  entry4(void)
 {
   /* insert code here */
 }
```

快速操作存储区示例：汇编语言

- 如何调用访问入口槽（**slot**）？
- 使用引导或安全操作码

```
call  boot(4)
rcall secure(2)
bra   cc,boot(8)

mov   #boot(5),w0 ; 16-bit address
call  w0
```

汇编语言示例（续）

- 如何定义访问入口槽（**slot**）？
- 使用引导或安全属性

```
        .section *,code,boot(4)
        .global _entry4
_entry4:
        ; do something
        return
```

引导和安全中断

- 在引导或安全段中执行时，所有中断均通过快速操作存储区来跳转
- 所有中断源使用一个访问入口槽（**slot**）（**16**）

中断示例：C

- 如何定义引导中断处理程序？

```
void __attribute__((interrupt,boot))
  my_boot_isr(void) {
  /* insert code here */
}
```

中断示例：汇编语言

- 如何定义引导中断处理程序？

```
.section *,code,boot(isr)
    .global _my_boot_isr
_my_boot_isr:
    ; do something
    retfie
```

安全模型

- 段容量和选项被编码为**3**个配置字：
  - FBS：引导段
  - FSS：安全段
  - FGS：通用段

- 这些设置共同构成“安全模型”

安全模型示例：C

- 在源代码中定义

```
#include <p33Fxxxx.h>

_FBS(BSS_SMALL_FLASH_HIGH &
  BRWP_WRPROTECT_ON);
_FSS(SSS_MEDIUM_FLASH_STD);
_FGS(GWRP_OFF);
```

- 或使用**IDE**
  - Build Options（编译选项）：LINK30：Code Guard

安全模型示例： 汇编语言

- 在源代码中定义

```
.include “p33Fxxxx.inc”

config _FBS, BSS_SMALL_FLASH_HIGH &
  BRWP_WRPROTECT_ON
config _FSS, SSS_MEDIUM_FLASH_STD
config _FGS, GWRP_OFF
```

- 或使用**IDE**
  - Build Options（编译选项）：LINK30：Code Guard

用户定义的安全模型

- 对于硬件中不具备**CodeGuard™**安全功能的器件
- 使用链接器命令选项

```
--boot flash_size=128
--boot ram_size=64

--secure flash_size=256
--secure ram_size=64:flash_size=256
```

课程总结

- 混合使用C和汇编语言
- 存储模型
- 程序空间可视性
- **CodeGuard™安全**

更多资源

- Microchip网站
  - 学生版工具套件
  - C30 README文件
  - 16位参考资料
  - 开发板
  - 硅片

问答

- 提问？
- 更多问题？
  - 请“咨询专家”
  - 访问在线论坛!

商标

Microchip的名称和徽标组合、**Microchip**徽标、**Accuron**、**dsPIC**、**KeeLoq**、**KeeLoq**徽标、**microID**、**MPLAB**、**PIC**、**PICmicro**、**PICSTART**、**PRO MATE**、**rfPIC**和**SmartShunt**均为**Microchip Technology Incorporated**在美国和其他国家或地区的注册商标。

AmpLab、**FilterLab**、**Linear Active Thermistor**、**Migratable Memory**、**MXDEV**、**MXLAB**、**SEEVAL**、**SmartSensor**和**The Embedded Control Solutions Company**均为**Microchip Technology Incorporated**在美国的注册商标。

Analog-for-the-Digital Age、**Application Maestro**、**CodeGuard**、**dsPICDEM**、**dsPICDEM.net**、**dsPICworks**、**dsSPEAK**、**ECAN**、**ECONOMONITOR**、**FanSense**、**FlexROM**、**fuzzyLAB**、**In-Circuit Serial Programming**、**ICSP**、**ICEPIC**、**Mindi**、**MiWi**、**MPASM**、**MPLAB Certified**徽标、**MPLIB**、**MPLINK**、**PICkit**、**PICDEM**、**PICDEM.net**、**PICLAB**、**PICtail**、**PowerCal**、**PowerInfo**、**PowerMate**、**PowerTool**、**REAL ICE**、**rfLAB**、**Select Mode**、**Smart Serial**、**SmartTel**、**Total Endurance**、**UNI/O**、**WiperLock**和**ZENA**均为**Microchip Technology Incorporated**在美国和其他国家或地区的商标。

SQTP是**Microchip Technology Incorporated**在美国的服务标记。

在此提及的所有其他商标均为各持有公司所有。