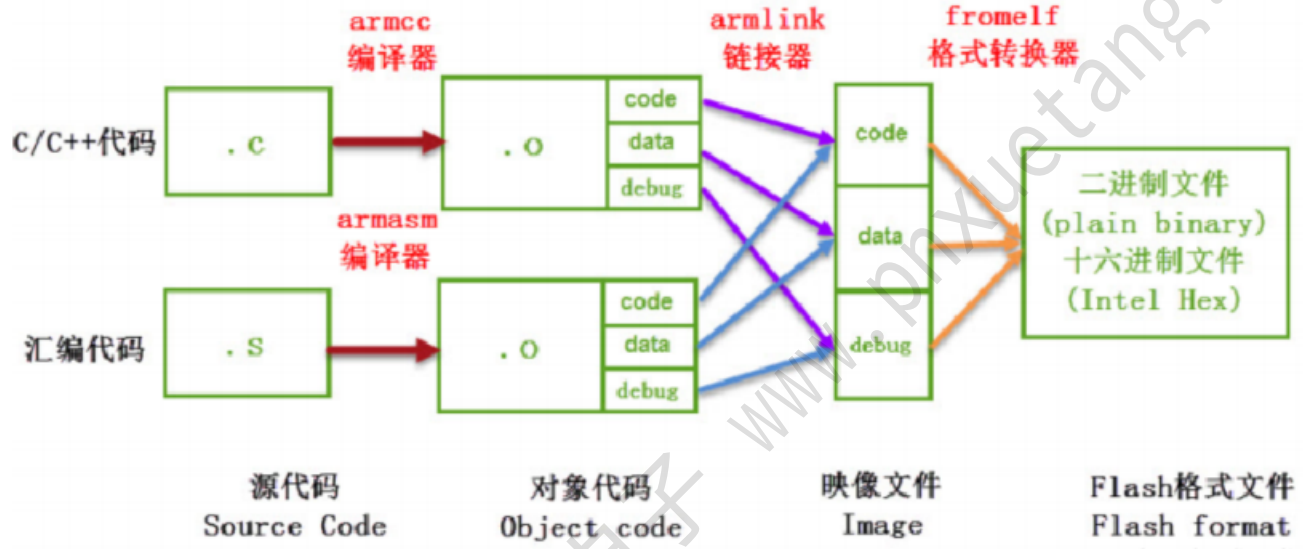
嵌入式C语言之-

程序编译与运行简介

讲师：叶大鹏

编译过程简介

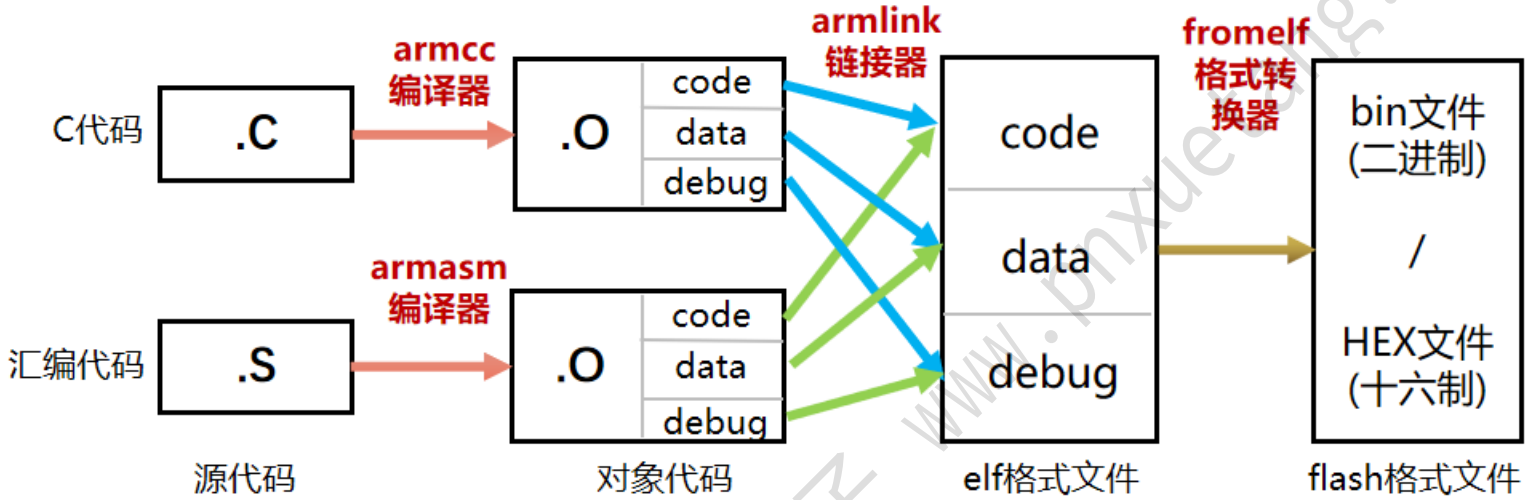


(1) 编译，MDK 软件使用的编译器是 armcc 和 armasm，它们根据每个 c/c++ 和汇编源文件编译成对应的以“.o”为后缀名的对象文件 (Object Code，也称目标文件)，其内容主要是从源文件编译得到的机器码，包含了代码、数据以及调试使用的信息；

(2) 链接，链接器 armlink 把各个.o 文件及库文件链接成一个映像文件“.axf”或“.elf”；

(3) 格式转换，一般来说 Windows 或 Linux 系统使用链接器直接生成可执行映像文件 elf 后，内核根据该文件的信息加载后，就可以运行程序了，但在单片机平台上，需要把该文件的内容加载到芯片上，所以还需要对链接器生成的 elf 映像文件利用格式转换器 fromelf 转换成“.bin”或“.hex”文件，交给下载器下载到芯片的 FLASH 或 ROM 中。

编译过程简介



(1) **编译**，MDK 软件使用的编译器是 armcc 和 armasm，它们根据每个 c/c++ 和汇编源文件编译成对应的以“.o”为后缀名的对象文件 (Object Code，也称目标文件)，其内容主要是从源文件编译得到的机器码，包含了代码、数据以及调试使用的信息；

(2) **链接**，链接器 armlink 把各个.o 文件及库文件链接成一个映像文件“.axf”，它是elf格式文件；

(3) **格式转换**，一般来说 Windows 或 Linux 系统使用链接器直接生成可执行映像文件 elf 后，内核根据该文件的信息加载后，就可以运行程序了，但在单片机平台上，需要把该文件的内容加载到芯片上，所以还需要对链接器生成的 elf 映像文件利用格式转换器 fromelf 转换成“.bin”或“.hex”文件，交给下载器下载到芯片的 FLASH 或 ROM 中。

具体工程中的编译过程

```
Rebuild started: Project: share mem
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'pointer'
assembling startup_ARMCM4.s...
compiling system_ARMCM4.c...
compiling main.c...
linking...
Program Size: Code=1396 RO-data=2436 RW-data=16 ZI-data=4096
FromELF: creating hex file...
".\Objects\share mem.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

1 2 3 4 5

构建工程的提示输出主要分5个部分：

1.提示信息的第一部分说明构建过程调用的编译器。图中的编译器名字是“V5.06”，后面附带了该编译器所在的文件夹。在电脑上打开该路径，可看到该编译器包含图编译工具 中的各个编译工具，如armar、armasm、armcc、armlink 及 fromelf，后面四个工具已在图 MDK 编译过程 中已讲解，而armar 是用于把.o 文件打包成 lib 文件的。

具体工程中的编译过程

```
Rebuild started: Project: share mem
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'pointer'
assembling startup_ARMCM4.s...
compiling system_ARMCM4.c...
compiling main.c...
linking...
Program Size: Code=1396 RO-data=2436 RW-data=16 ZI-data=4096
FromELF: creating hex file...
".\Objects\share mem.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

2

构建工程的提示输出主要分5个部分：

2.使用 armasm 编译汇编文件。图中列出了编译 startup 启动文件时的提示，编译后每个汇编源文件都对应有一个独立的.o 文件.

具体工程中的编译过程

```
Rebuild started: Project: share mem
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'pointer'
assembling startup_ARMCM4.s...
compiling system_ARMCM4.c...
compiling main.c...
linking...
Program Size: Code=1396 RO-data=2436 RW-data=16 ZI-data=4096
FromELF: creating hex file...
".\Objects\share mem.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

3

构建工程的提示输出主要分5个部分：

3. 使用 armcc 编译 c/c++ 文件。图中列出了工程中所有的 c/c++ 文件的提示，同样地，编译后每个 c/c++ 源文件都对应有一个独立的.o 文件。

具体工程中的编译过程

```
Rebuild started: Project: share mem
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'pointer'
assembling startup_ARMCM4.s...
compiling system_ARMCM4.c...
compiling main.c...
linking...
Program Size: Code=1396 RO-data=2436 RW-data=16 ZI-data=4096
FromELF: creating hex file...
".\Objects\share mem.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

4

构建工程的提示输出主要分5个部分：

4.使用 armlink 链接对象文件，根据程序的调用把各个.o 文件的内容链接起来，最后生成程序的axf 映像文件，并附带程序各个域大小的说明，包括 Code、RO-data、RW-data 及 ZI-data 的大小。

具体工程中的编译过程

```
Rebuild started: Project: share mem
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'pointer'
assembling startup_ARMCM4.s...
compiling system_ARMCM4.c...
compiling main.c...
linking...
Program Size: Code=1396 RO-data=2436 RW-data=16 ZI-data=4096
FromELF: creating hex file...
".\Objects\share mem.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

5

构建工程的提示输出主要分5个部分：

5.使用 fromelf 生成下载格式文件，它根据 axf 映像文件转化成 hex 文件，并列出编译过程出现的错误 (Error) 和警告 (Warning) 数量。

具体工程中的编译过程

第5步中，在工程的编译提示输出信息中有一个语句“Program Size：Code=xx RO-data=xx RW-data=xx ZI-data=xx”，它说明了程序各个域的大小，编译后，应用程序中所有具有同一性质的数据(包括代码) 被归到一个域，程序在存储或运行的时候，不同的域会呈现不同的状态，这些域的意义如下：

程序组件	所属类别
机器代码指令	Code
常量	RO-data
初值非 0 的全局变量	RW-data
初值为 0 的全局变量	ZI-data
局部变量	ZI-data 栈空间
使用 malloc 动态分配的空间	ZI-data 堆空间

具体工程中的编译过程

- **Code**：即代码域，它指的是编译器生成的机器指令，这些内容被存储到 ROM 区。

- **RO-data**：Read Only data，即只读数据域，它指程序中用到的只读数据，这些数据被存储在ROM区，因而程序不能修改其内容。例如 C 语言中 const 关键字定义的变量就是典型的RO-data。

- **RW-data**：Read Write data，即可读写数据域，它指初始化为“非 0 值”的可读写数据，程序刚运行时，这些数据具有非 0 的初始值，且运行的时候它们会常驻在 RAM 区，因而应用程序可以修改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“非 0 值”给该变量进行初始化。
- **ZI-data**：Zero Initialie data，即 0 初始化数据，它指初始化为“0 值”的可读写数据域，它与 RW-data 的区别是程序刚运行时这些数据初始值全都为 0，而后续运行过程与 RW-data 的性质一样，它们也常驻在 RAM 区，因而应用程序可以更改其内容。例如 C 语言中使用定义的全局变量，且定义时赋予“0 值”给该变量进行初始化 (若定义该变量时没有赋予初始值，编译器会把它当 ZI-data 来对待，初始化为 0)；

- **ZI-data 的栈空间 (Stack) 及堆空间 (Heap)**：在 C 语言中，函数内部定义的局部变量属于栈空间，进入函数的时候从向栈空间申请内存给局部变量，退出时释放局部变量，归还内存空间。而使用 malloc 动态分配的变量属于堆空间。在程序中的栈空间和堆空间都是属于ZI-data 区域的，这些空间都会被初始值化为 0 值。编译器给出的 ZI-data 占用的空间值中包含了堆栈的大小 (经实际测试，若程序中完全没有使用 malloc 动态申请堆空间，编译器会优化，不把堆空间计算在内)。

具体工程中的编译过程

- 实际占用单片机FLASH（ROM）存储空间大小：

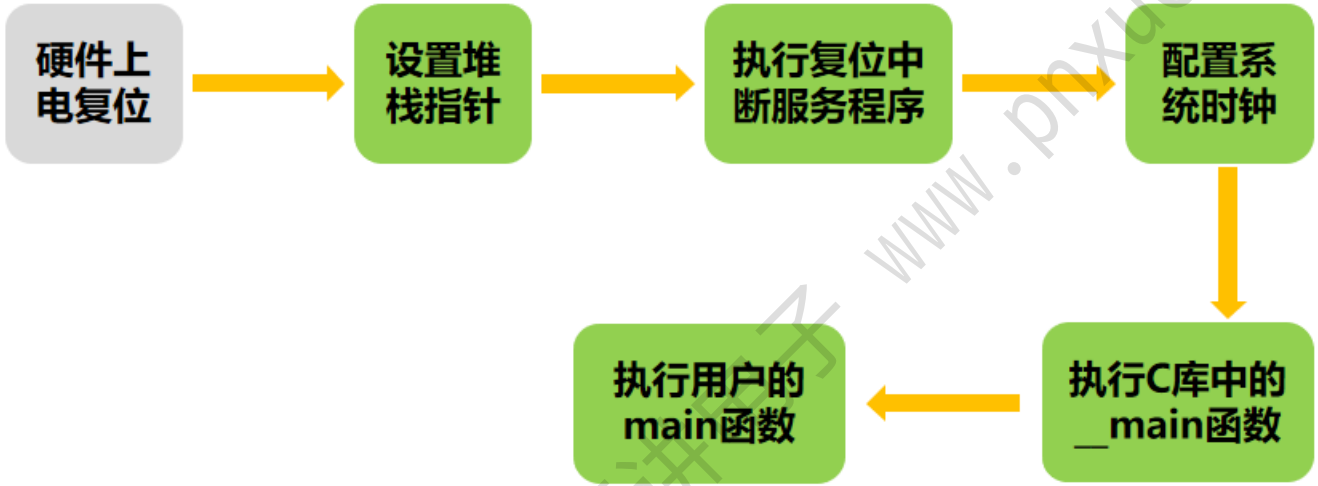
Code + RO-data + RW-data ，对于例程，大小为 3848 字节。

，并不是生成代码工程生成的hex文件大小，因为hex中还有额外的辅助下载信息

编译生成文件

Output 目录下的文件	
*.lib	库文件
*.dep	整个工程的依赖文件
*.d	描述了对应.o 的依赖的文件
*.crf	交叉引用文件，包含了浏览信息(定义、引用及标识符)
*.o	可重定位的对象文件(目标文件)
*.bin	二进制格式的映像文件，是纯粹的 FLASH 映像，不含任何额外信息
*.hex	Intel Hex 格式的映像文件，可理解为带存储地址描述格式的 bin 文件
*.elf	由 GCC 编译生成的文件，功能跟 axf 文件一样，该文件不可重定位
*.axf	由 ARMCC 编译生成的可执行对象文件，可用于调试，该文件不可重定位
*.sct	链接器控制文件(分散加载)
*.scr	链接器产生的分散加载文件
*.lnp	MDK 生成的链接输入文件，用于调用链接器时的命令输入
*.htm	链接器生成的静态调用图文件
*.build_log.htm	构建工程的日志记录文件
Listing 目录下的文件	
*.lst	C 及汇编编译器产生的列表文件
*.map	链接器生成的列表文件，

软件启动流程简介



由于启动代码涉及到汇编指令、堆和栈的内容，比较复杂，所以在 函数 的课程章节再详细讲解。

THANK YOU!