3

让板子跑起来

本章开始就需要读者亲自动手调试了。这章的终极目的是让我们的实验板正常运行起来，也就是说要为实验板编写一个有效的启动代码。当然，读者完全可以使用开发板自带的许多实验例程使板子正常运行起来，但本章的目的却不仅如此。笔者希望带领读者探讨更多问题的来龙去脉，如开发板结构原理，程序的编译、链接、调试，开发工具的原理及使用方法等，让读者知其然，知其所以然，这样在以后遇到相关问题时不至于太纠结，最后本章的重点会落在实验板的启动代码编写上。总之，本章的内容会包括：

- ADS 集成开发环境，包括程序的编译、链接与调试等内容；
- 实验环境的搭建；
- 启动代码的编写；
- 几个测试代码。

本章开始会涉及一些编译器、连接器等开发工具的原理，这部分东西多且杂，部分读者可能在理解时会出现一定的困难，Take it easy!另外，对于启动代码的编写，主要是使用 ARM 汇编语言，且需针对特定的开发环境进行，因此，读者先要对第二章的基础知识有很好的理解。

3.1 ARM 开发工具

3.1.1 ADS 简介

开发工具的选择对程序编写起着关键的作用,用户程序从最初的源码编辑到最终可执行机器代码，这都需要各种开发工具发挥它们的威力。好的开发工具往往能加快开发进度，节省开发成本。目前市场上有多款针对 ARM 处理器的开发工具，它们基本是以集成开发环境（IDE）的形式提供

给用户的，集成开发环境通常都集成了编辑、编译、汇编、链接、调试、工程管理及函数库等模块。基于 IDE 环境进行程序开发，能使编辑、编译、汇编、链接、调试等开发工作都在一台 PC 机上完成，大大提高了用户的开发效率。

目前能支持 ARM 处理器程序开发的主要集成开发环境有：ADS、RealView MDK、IAR、Keil 等。ADS 全称为 ARM Developer Suite，是 ARM 公司为方便用户在 ARM 芯片上进行应用软件开发而推出的一整套集成开发工具，本书也将基于它进行开发和讲解。ADS 由 ARM SDT 发展而来，ARM SDT 即 ARM Software Development Kit，也是 ARM 公司早期推出的一套针对 ARM 芯片的集成开发环境，后来该集成开发环境被 ADS 取代。

ADS 包含了一系列的开发工具，以及相关的使用说明支持文档、使用范例等，能使用户在此基础上开发 ARM 处理器相关的程序。用户程序最终能变成可执行文件需要经过一系列的阶段，对于一个常见的 C/C++工程来说，首先需要使用 C/C++编译器将工程中的 C/C++源程序编译为 ELF 格式（Executable and Linking Format，执行时链接格式文件）的可重定位目标代码。同样，如果工程中含有 ARM 汇编源文件，则需要使用 ARM 汇编器将各汇编文件转化为 ELF 目标代码。接下来，连接器会将工程中的所有 ELF 目标代码进行链接和定位，连接器能够根据用户的定位信息将代码区、数据区、堆栈区放在指定位置，最终将目标模块链接成一个单一的、绝对定位的 ELF 目标文件。同时，连接器可以产生一些链接结果信息，比如数据段大小、程序段大小、符号的绝对定位地址等，若需要调试，连接器还可以产生包含调试信息的 AXF 文件，这个文件可以在调试工具 AXD 中用来对程序进行调试。最后，若需要将编译好后的文件下载到 Flash 中运行，可能涉及目标文件格式转换的问题，如将 ELF 转换为普通二进制文件等。这里整个过程被笔者描述得很纠结，看了后面的内容，相信读者能有更好的理解。

ADS 主要由以下四大部分组成：GUI 开发环境（Code Warrior IDE 和 AXD）、ADS 命令行开发工具、实用辅助开发软件和其他支持软件组成。下面来简单说说这四个部分的主要组成或功能，让读者有个简单的了解。

➢ GUI 开发环境

GUI 开发环境包括了两个部分：CodeWarrior IDE 和 AXD。它们是两个互相独立的部分，CodeWarrior IDE 主要用于管理工程文件，实现源文件的编辑、编译连接等，它通过调用后续会讲到的命令行工具来完成这些工作，CodeWarrior IDE 为我们提供了用户交互界面来配置这些命令行工作的属性，使用户对程序的编译、连接过程控制更简单；AXD 主要用于程序调试，可用于 C/C++程序或 ARM 汇编程序的调试，它支持软件和硬件调试，在 3.2 节中将详细涉及 AXD 的相关知识。

➢ ADS 命令行开发工具

ADS 环境下集成了许多命令行开发工具，命令行工具为 GUI 环境下程序的编译连接提供了支持；用户也可以基于命令行来直接使用这些工具，但是这对于开发人员的专业知识要求较高，在本书中也不采用这种方式，而是在 GUI 界面下直接开发。这里主要介绍以下几种命令行工具。

● armcc

armcc 是 ARM 的 C 编译器，具有优化功能，兼容于 ANSI C 标准。armcc 主要用于将基于 ANSI

C 的源程序编译成 32 位 ARM 指令代码。此外，ADS 下还有其他几个命令行工具与 armcc 功能相似，主要包括：tcc，用于将基于 ANSI C 的源程序编译成 16 位 ARM 指令代码；armcpp，用于将基于 ANSI C++或 EC++的源程序编译成 32 位 ARM 指令代码；tcpp，用于将基于 ANSI C++或 EC++的源程序编译成 16 位 ARM 指令代码。为了配合这些编译器的工作，也为了方便用户程序的编写，ADS 中还提供了 ARM C 语言库和 Rogue Wave C++库，合理地使用库函数能大大增加用户程序开发效率。

由于 armcc 是最常用的编译器，所以对此作一个详细的介绍。在安装好 ADS1.2 并正确注册系统环境变量后，可以在 Windows 命令行上键入 armcc -help，这样就看到了这个命令行工具的使用方法，如图 3-1 所示。

```
C:\Documents and Settings\Administrator>armcc -help
ARM C Compiler, ADS1.2 [Build 848]
Software supplied by: Team-EFA

Usage:          armcc [options] file1 file2 ... filen
Main options:

-c              Do not link the files being compiled
-C              Prevent the preprocessor from removing comments (Use with -E)
-D<symbol>      Define <symbol> on entry to the compiler
-E              Preprocess the C source code only
-f<options>     Enable a selection of compiler defined features
-g<options>     Generate tables for high-level debugging
-I<directory>   Include <directory> on the #include search path
-J<directory>   Replace the default #include path with <directory>
-o<file>        Name the file that holds the final output of the compilation
-O0             Minimum optimization
-O1             Restricted optimization for debugging
-O2             Maximum optimization
-S              Output assembly code instead of object code
-U<symbol>      Undefine <symbol> on entry to the compiler
-W<options>     Disable all or selected warning messages
```

图 3-1 armcc 命令行工具

从上图中可以看到，用命令行使用 armcc 时的语法格式为

```
armcc [options] file1 file2 ... filen
```

这里的 options 是编译器所需要的选项，fiel1,file2…filen 是相关的文件名。例如可以用下面的命令语句来将一个名为 main.c 的源文件编译为 32 位的目标文件。

```
armcc -g -O1 -c main.c
```

这里，选项-g 表示在目标中加入调试信息表；-O 是控制代码优化的编译选项，大写字母 O 后面跟的数字不同，表示的优化级别就不同，-O1 表示关闭影响调试结果的优化功能；-c 告诉编译器表示只进行编译，不进行链接。从上图中的各个英文描述中读者可以清楚地看到各个编译选项的意义。这里不再对各个选项做全面的举例讲解了，使用集成开发环境时，可以配置 C 编译器的某些编译特性，实质上就是为 armcc 指定各种编译属性选项，在后面还会有所涉及。

- armasm

armasm 用于将 ARM 汇编语言程序编译为对应的机器码，可以是 32 位 ARM 指令，也可以是 16 位的 Thumb 指令。实际上 armcc 也必须提供与 armasm 相似的功能，它先将 C 程序处理为相应

的汇编程序，然后再调用其内嵌的汇编器（与 armasm 功能相似）来完成汇编程序向最终目标文件的转换。可以在 Windows 命令行上键入 armasm -help，这样就看到了这个命令行工具的使用方法，如图 3-2 所示。

```
C:\Documents and Settings\Administrator>armasm -help
ARM/Thumb Macro Assembler, ADS1.2 [Build 848]

Usage:      armasm [keyword arguments] sourcefile objectfile
            armasm [keyword arguments] -o objectfile sourcefile

Keywords    (Upper case shows allowable abbreviation)
-LIST       listingfile   Write a listing file (see manual for options)
-Depend     dependfile    Save 'make' source file dependencies
-Errors     errorsfile    Put stderr diagnostics to errorsfile
-I          dir[,dir]     Add dirs to source file search path
-PreDefine  directive     Pre-execute a SET{L,A,S} directive
-NOCache                  Source caching off    (default on)
-MaxCache   <n>           Maximum cache size    (default 8MB)
-NOEsc                    Ignore C-style (\c) escape sequences
-NOWarn                   Turn off Warning messages
-G                        Output debugging tables
-APCS       /<quals>      Make pre-definitions to match the
                          chosen proc-call standard
-CheckReglist             Warn about out of order LDM/STM register lists
-Help                     Print this information
-LIttleend                Little-endian ARM
-BIgend                   Big-endian ARM
-MEMACCESS <attributes>   Specify attributes of target memory system
-M                        Write source file dependency lists to stdout
-MD                       Write source file dependency lists to inputfile.d
-KEEP                     Keep local labels in symbol table of object file
-NORegs                   Do not predefine register names
-SPLIT_LDM                Fault long LDM/STM
-UNSAFE                   Downgrade certain errors to warnings
-VIA        <file>        Read further arguments from <file>
-CPU        <target-cpu>  Set the target ARM core type
-FPU        <target-arch> Set target FP architecture version
            one of:       SOFTVFP, SOFTFPA, VFP, FPA, NONE
-16                       Assemble 16 bit Thumb instructions
-32                       Assemble 32 bit ARM instructions
```

图 3-2　armasm 命令行工具

从图中可以看到 armasm 的使用方法，例如可以用下面的命令语句来编译一个名为 file.s 的汇编源文件。

```
armasm -list file.lst file.s
```

这条语句将 file.s 编译为 file.o 的目标文件，-list 选项告诉汇编器同时产生一个名为 file.lst 的列表文件，并将汇编程序列表信息保存到 file.lst 中。从上图中的各个英文描述中读者可以清楚地看到各个汇编选项的意思，在使用集成开发环境时，可以配置汇编器的某些编译特性，实质上就是为 armasm 指定各种汇编属性选项。

- armlink

armlink 是 ARM 连接器，该命令行工具可以将编译得到的一个或多个目标文件与相关的一个或多个库文件进行链接，生成一个 ELF 格式的可执行映像文件，也可以将多个目标文件链接成另一个目标文件，以供后续进一步链接的使用。可以在 Windows 命令行上键入 armlink -help，这样就看到了这个命令行工具的使用方法，如图 3-3 所示。

```
C:\Documents and Settings\Administrator>armlink -help
ARM Linker, ADS1.2 [Build 848]

Usage: armlink option-list input-file-list

where

    option-list      is a list of case-insensitive options.
    input-file-list  is a list of input object and library files.

Main options (abbreviations shown capitalised):

General options:

    -Help            Print this summary.
    -Output file     Specify the name of the output file.

Options for specifying memory map information:

    -partial        Generate a partially linked object.
    -scatter file   Create the memory map as described in file.
    -ro-base n      Set exec addr of region containing RO sections.
    -rw-base n      Set exec addr of region containing RW/ZI sections.
```

图 3-3　armlink 命令行工具

根据上图中的命令行格式，可以用下面的语句将前面两个例子中得到的目标文件进行链接。其中-output 选项指明了输出文件的名称及格式，axf 格式的文件主要用于 AXD 的调试中。

```
armlink main.o file.o -output main.axf
```

这里，笔者不再对上图中的各个链接选项做全面的举例讲解，使用集成开发环境时，可以配置连接器的某些编译特性，实质上就是为 armlink 指定各种链接属性选项。

值得指出的是，GNU 也有类似于 armcc、armasm、armlink 这类针对 ARM 处理器的命令行开发工具，如 arm-elf-gcc、arm-elf-as、arm-elf-ld 等。在基于嵌入式 Linux 的 ARM 程序的开发过程中，GNU 的这些命令行工具会被经常使用到，但是使用这些工具需要有个复杂的开发环境搭建过程，所以 ADS 提供的完整 IDE 环境是非常有利于初学者的。

➢　实用辅助开发软件

ADS 集成的辅助开发软件，旨在为用户的程序开发提供全方面的支持，这里笔者主要说说 fromELF、armar、Flash Downloader 这几个工具。

- fromELF：将 ELF 格式的映像文件转换为其他格式，如普通的二进制文件、Hex 文件等，以满足不同应用场合的需求，这些类型的文件都可以直接下载到 Flash 中运行，因此 fromELF 是用户进行代码测试、产品发布的必要工具。
- armar：可以将 ELF 格式的目标文件归档为函数库文件，实现库文件的有效管理。采用函数库文件，应用程序目标文件就能直接和库文件中的函数映像进行链接，对于应用程序的开发管理是很方便的。
- Flash Downloader：这个工具可用于将最后的可执行二进制文件烧写到 Flash 存储器中。

➢　其他支持软件

主要是 ARMulator，在 AXD 环境中可以用该工具来实现对程序的纯软件调试。ARMulator 是一个 ARM 指令集仿真器，为 ARM 指令和 Thumb 指令提供了精确的模拟。它集成在调试器 AXD

中，用户可以在硬件尚未做好的情况下，开发程序代码。

3.1.2 ARM 汇编器

汇编器的基本功能就是将 ARM 汇编指令编译成为机器码，汇编过程类似于一个查表的操作，正如上一章讲解的指令编码那样，汇编器识别一条指令，然后将该指令转换为相应的机器编码。ARM 汇编语言中提供了多种伪操作、伪指令来控制汇编器的行为和编译结果，汇编器最终将编译好的机器码组织为各个段，例如数据段、代码段，并放在一个目标文件中。每个源程序都会产生一个目标文件，当然这些相互独立的目标文件是不能直接运行的，因为它们之间存在着各种调用关系，而每个目标文件都不知道其他各目标文件的情况，无法实现函数或数据的调用，这时就需要使用连接器来把各个目标文件有效地组织起来。举个例子，在 ARM 汇编语言中，常常使用 LDR 伪指令来装载一个不在本文件中的标号，汇编过程中，若汇编器发现这个标号不在当前段中，它就会在此处放置一条重新定位伪操作（Relocation Directive），这条重定位伪操作将告诉连接器在链接的时候解决这个地址问题，这个地址由包含 LDR 指令的段或内存池被连接器放置的最终位置决定。上述的整个过程，要实现 LDR 的正确装载，都依赖于连接器的功能。

第二章所讲的汇编指令、汇编程序设计等都属于汇编器相关的知识，这里对汇编器也没有什么可多说的了。主要来看看汇编器内部的一些变量，在汇编程序中可以使用这些变量的值来编写程序（可以直接使用这些变量而不用声明），但是注意，这些变量不能被程序赋值。ARM 汇编器中定义的常见变量如表 3-1 所示，其中{CONFIG}表示了当前汇编器是在编译 ARM 代码还是 Thumb 代码，其值为 32 时，汇编器将指令编译为 32 位的机器码，其值为 16 时，汇编器将指令编译为 16 位的机器码；{CPU}记录了目标系统的 CPU 的类型，默认类型为 ARM7TDMI，用户可以在命令行工具中用-cpu 选项指明 CPU 类型，也可以在 IDE 环境中设置；{FPU}指明了目标系统中浮点运算单元的体系，可以为 none、vfpv1 等；{ARCHITECTURE}指明了处理器的体系号，如 4T、5TE 等；{ARMASM_VERSION}和|ads&version|指明了 ADS 的版本号，在 SDT 中没有这两个变量的定义，所以可以通过这两个变量来判断当前编译器类型，如下代码所示：

```
1  IF :DEF: |ads&version|
2  ….;code for ADS
3  ELSE
4  ….;code for SDT
5  ENDIF
```

前面说过，ADS 可以看成是 SDT 的升级版本，在程序中加入上面这种判断的代码，就可使得程序在两种环境下都能正常编译，大大增强了程序的可移植性。

表 3-1　ARM 汇编器内置变量

变量名称	描述	变量名称	描述
{PC}或 .	表示当前指令地址	{CPU}	CPU 名称
{TRUE}	逻辑真	{FPU}	协处理器名称

续表

变量名称	描述	变量名称	描述
{FALSE}	逻辑假	{ARCHITECTURE}	处理器的版本号
{CONFIG}	汇编器在编译状态	{ARMASM_VERSION}	ADS 的版本号
{ENDIAN}	处理器大小端	|ads&version|	ADS 的版本号

在安装好 ADS 后，用户就可以使用 CodeWarrior IDE 进行程序的开发了，在编译选项设置时，可以通过如图 3-4 所示的用户界面设置 ARM 汇编器的特性。

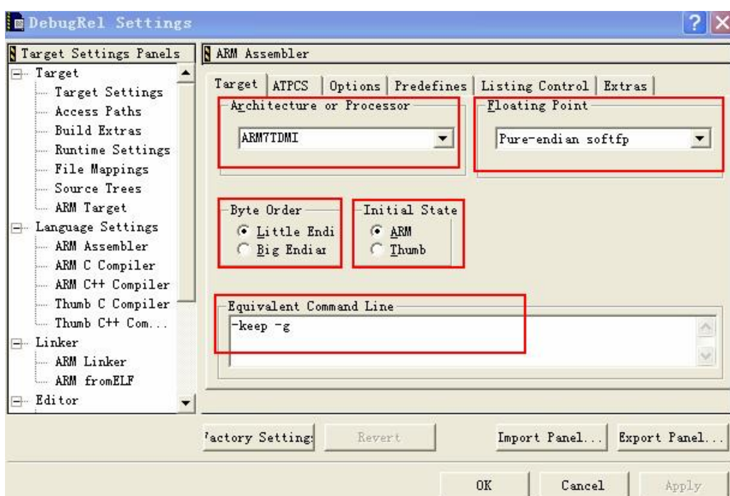

图 3-4　ARM 汇编器选项设置

如上图中可以对汇编器的 Target、ATPCS、Options 等选项卡进行设置。在 Target 选项卡中，如图中标注所示，可以对处理器类型、浮点处理器类型、大小端模式、汇编器初始状态等进行设置。对应这些设置的值会直接影响到 ARM 汇编器的内置变量，如{CPU}、{ARCHITECTURE}、{ENDIAN}、{CONFIG}、{FPU}等。与所有汇编器设置等效的命令行语句在 Equivalent Command Line 栏中给出，如图 3-4 中最下方的标记所示，-keep 命令告诉汇编器将局部符号保留在目标文件的符号表中，以供调试器调试使用；-g 命令告诉汇编器产生 DRAWF2 格式的调试信息表。

3.1.3　ARM 编译器

ARM 编译器包括了 armcc、armcpp、tcc、tcpp 四类，它们也遵循相同的命令行格式，本节对 ARM 编译器做简要的讲解。

编译器中集成了以下几种标准库文件，使用库文件可以提高用户应用程序开发效率。

（1）ARM C 语言库，ARM C 语言包括了 ISO C 语言标准中的函数集（注意，标准中的有些函数是与平台相关的，例如 printf 函数，ARM C 语言库通过引进 semihost 机制实现了对这类函数

的支持，在后面会说到）。

（2）Rogue Wave C++库，包含了基本的C++库和基本的C++对象，该C++库中不再包含与目标环境相关的内容，因为它通过ARM C语言库来提供与目标环境相关的功能。

值得指出的是，这些库文件都是以二进制的形式提供的，对于不同的ATPCS格式，都有不同格式的C/C++库。这里有两点亟待说明：①什么是Semiohost机制；②什么是ATPCS，这两个术语已在前面多次出现。

- Semiohost 机制

对于标准C库中许多函数，它们是与平台密切相关的，在嵌入式平台下，标准C库中的许多函数，如输入/出函数，无法直接实现。对于这些函数，ARM C库通过引入Semiohost机制来实现对它们的支持。所谓 Semiohost，就是利用主机资源来完成在目标机上运行的程序所需要的输入/出功能。ARM提供的调试开发工具ARMulator、Angel、Multi-ICE和EmbededICE都支持Semiohost技术。举个简单的例子，例如C标准库中的输入/出函数，嵌入式系统没有键盘，没有显示器，根本无法实现这些函数。Semiohost提供了一种间接的机制，使得这些函数可以在嵌入式系统中运行。例如，当嵌入式系统调用printf语句输出一条字符串，此时会产生一条软件中断指令，该中断使得CPU进入中断状态，同时将控制权交给调试主机，主机根据printf函数中的内容在PC显示器上输出相关的内容，以此实现了嵌入式系统的标准输出功能。注意：Semiohost功能只能用在系统的调试过程中，但是许多嵌入式产品在实际使用时，是没有主机的支持的。因此，若要编写可以烧写到Flash存储器上并作为最终产品发布的程序，此时不能再调用printf函数实现字符串的输出。当然也有其他办法解决这个问题，开发人员要么不在程序中使用任何printf语句，要么实现自己的输入/出函数，如定义一个串口输出函数uart_printf，将字符串输出到串口上，以此完成显示的功能，这样就可以避免使用Semiohost机制了。

在我们的程序中，若不使用Semiohost功能，就不能使用系统定义好的许多函数，例如标准输入/出、文件打开/关闭、定时函数等。如果在程序中必须要使用C库中的这些函数，而此时又没有Semiohost的支持，那该怎么办呢？实际上，用户完全可以自己提供一套机制，以实现对那些需要基于Semiohost才能完成其功能的函数的支持。例如，用户怎样使用自己的机制来完成对prinf函数的支持呢，很简单，用户根据自己的硬件情况实现一个fputc()函数就行了，对于fputc()函数怎样去实现，完全由用户自己决定，例如可以将该函数定义为在串口上输出一个字符。

- 对于C库的使用

上面说过，ARM C标准库里面的有些函数需基于Semiohost的功能才能实现，若用户程序没使用Semiohost功能，那自然也不能在工程中调用这些函数，但是用户可以按照自己的系统需求自己实现这些函数，例如要输出，可以定义一个串口输出函数，在需要输出的时候，调用该函数就可以了。另外，C库中的某些函数必须初始化，否则不能成功调用，用户程序编写时通常会有个main函数，这个函数就是间接告诉编译器：我的程序要使用C库中的那些需初始化的函数，请将这些函数初始化好。

函数main在ADS中有特殊的意义。当一个工程项目中存在main函数时，连接器会把另外两

个库函数（_main 和_rt_entry）的代码链接到目标文件中；当然，若没有 main 函数，这两个初始化过程函数就不会被链接，结果就会导致一些标准的 C 库函数无效。_main 函数主要完成代码和数据的复制，并把 ZI 数据区清零，关于这里的一些术语我们在连接器部分讲解；_rt_entry 主要进行程序堆栈的初始化和库函数的初始化；最后_rt_entry 跳到应用程序的入口 main，这才开始用户程序的执行。

若 C 标准库未被正常初始化，则有些库函数是不能使用的。当然若应用程序不需要 C 库自己初始化，也不需要调用 C 库中那些必须初始化后才能使用的函数（例如 signal.h 中定义的 signal 和 raise 函数、数据栈溢出检查等），那么用户不要在工程文件中出现 main 函数就可以了。但是这带来了麻烦，关于程序运行的环境必须用户自己去搭建，包括代码搬运、堆栈设置等。我们的程序中不会使用这些需要初始化的 C 库函数，所以整个程序代码中不会出现 main 函数，也因此，整个代码的搬移、数据堆栈的初始化都需要用户自行编程实现，这一点后面慢慢道来。

- ATPCS

为了使单独编译的 C 语言程序和汇编程序之间能够相互调用，必须规定好子程序之间的调用规则。下面来说说 ARM 程序和 Thumb 程序中子程序调用的基本规则，即 ATPCS（ARM-Thumb Procedure Call Standard）。ATPCS 规定了一些子程序之间调用的基本规则，这些基本规则包含了子程序调用过程中寄存器的使用规则、数据栈的使用规则、参数的传递规则、函数返回值规则等。基本的 ATPCS 规则如下。

（1）寄存器使用规则：子程序间通过寄存器 R0～R3 来传递参数，这时它们又可以记作 A0～A3，被调用的子程序无需恢复寄存器 R0～R3 的内容。当参数个数大于 3 个时，需要使用堆栈来传递数据。

子程序中，使用寄存器 R4～R11 来保存局部变量，这时它们可以记作 V1～V8。如果子程序会改变 R4～R11 中的某个值，则子程序必须先保存了这些寄存器的值后才能使用，且在返回前必须恢复它们的值；在 Thumb 子程序中往往只能用 R4～R7 来保存局部变量；寄存器 R12 是内部调用暂存寄存器，记作 IP，该寄存器无其他特殊用途，在过程调用期间，用户可以将它用于任何目的，也不用担心它的返回值；对于其他特殊功能寄存器 R13～R15，保留它们原来的特殊寄存器功能。

（2）数据栈的使用规则：数据栈都使用 FD 类型（什么是 FD 类型，这点相信读者已经在前一章的学习中有所了解），且要求对数据栈的操作是 8 字节对齐的。异常中断可以使用被中断程序的数据栈，这时用户要保证被中断程序的数据栈空间足够大。

（3）参数的传递规则：根据参数个数将子程序分为参数个数固定的子程序和参数个数可变的子程序，这两种类型程序的参数传递规则是不同的。

对于参数个数可变的子程序，当参数个数不超过 4 个时，可以使用 R0～R3 来传递参数，当参数个数大于 4 个时，需要使用堆栈来传递数据。在参数传递时，将所有参数看作是存放在连续内存地址单元的字数据，然后，依次将各个数据送到寄存器 R0～R3 中；如果参数多余 4 个，则将剩余的数据依次传送到堆栈中，即通过堆栈来传递多余的参数，入栈的顺序与参数顺序相反，即最后一个字数据先入栈。按照上面的规则，一个 long long 类型的参数（8 字节）可能通过寄存器传递，也可能通过数据栈传递，也可能一半通过寄存器传递，一半通过数据栈传递；若参数中有浮点数，

且处理器恰好有浮点处理单元，则浮点数参数的传递不加入上述行列，它是直接通过浮点单元寄存器来传递，如果浮点空间不够，才将浮点数通过堆栈传递。注意，用户应尽量使函数参数个数少于5 个，因为堆栈访问需要消耗更多的时间。

对于参数个数固定的子程序，当参数个数不超过 4 个时，可以使用 R0～R3 来传递参数，当参数个数大于 4 个时，需要使用堆栈来传递数据。在参数传递时，将所有参数看作是存放在连续内存地址单元的字数据，然后，依次将各自数据送到寄存器 R0～R3 中，如果参数多于 4 个，则将剩余的数据传送到堆栈中，入栈的顺序与参数顺序相同，即最后一个字数据最后入栈，注意这里与参数个数可变类型传递的区别。

（4）子程序的结果返回规则：结果为 1 字长的整数时，可以通过寄存器 R0 返回；结果为 2～4 字长的整数时，对应的可通过 R0～R1、R0～R2、R0～R3 返回；如果返回参数为浮点数时，可以通过浮点部件来返回；如果参数字长很长，必须通过间接的方式返回，如内存传递。

为了适应一些特定处理函数的需要，往往需要在上述基本 ATPCS 基础上加以一定的改进，增加一些功能。这样，就派生出了以下几种子程序间的调用规则，它们都是在基本 ATPCS 规则上进行增加或改进后定义出来的。

- 支持数据栈限制检查的 ATPCS；
- 支持代码段位置无关的 ATPCS；
- 支持数据段位置无关的 ATPCS；
- 支持 Thumb 程序和 ARM 程序混合调用的 ATPCS；
- 浮点处理运算的 ATPCS。

这里就不对各类特殊应用场合的 ATPCS 进行介绍了，它们在实际中也很少被用到。有调用关系的所有子程序必须使用同一种 ATPCS，编译器和汇编器都会在 ELF 格式的目标文件中设置相应属性，以标识出用户的 ATPCS 类型。对于不同类型的 ATPCS 规则，对应的 C 函数库有着不同的实现。连接器会根据选择的 ATPCS 来连接相应的 C 库函数。

————分割线————

接下来，回归正题，讲解一些与 ARM 编译器相关的特性。

➢ 编译器中的 pragma 命令

pragma 命令是 ARM 编译器自定义的特性，它用来控制编译器的编译特性，其命令格式如下：

```
#pragma [no_]feature-name
```

[no_]表示关闭相应的特性，否则为开启。feature-name 可以为表 3-2 中所示的值，它们的意义及默认值也已在表中列出。

表 3-2　各种编译器属性

特性名称	默认状态	含义
check_printf_format	off	检查 printf 类函数中的字符串格式
check_scanf_format	off	检查 scanf 类函数中的字符串格式

续表

特性名称	默认状态	含义
check_stack	on	检查数据栈是否溢出
debug	on	是否产生调试信息表
import	--	引入外部符号
Ospace	--	编译器对代码大小进行优化
Otime	--	编译器对生成代码运行速度优化
Onum	--	指定编译器的优化级别
Softfp linkage	off	是否使用软件浮点连接

我们对 check_printf_format 举个例子，它能告诉编译器对 printf 类函数中的字符串变量进行格式检查，如下所示：

```
1   #pragma check_printf_format
2   extern void Uart_printf(const char *format,…);
3   #pragma no_check_printf_format
```

可以使用#pragma import(symbol_name)引入外部符号 symbol_name，其功能和汇编中的 IMPORT 伪操作相同，对于其他特性就不一一讲解了，读者可参考 ADS 编译器手册。

➢ 编译器中的特殊关键字

（1）函数声明相关的关键字。

ARM 编译器中定义了几个关键字，这些关键字用来修饰函数，告诉编译器对这些函数进行特殊对待，重点说说经常使用到的_asm、_inline、_irq 这三个关键字。

_asm 用于告诉编译器下面的代码是用汇编语言写成的，使用它就可以在 C 程序语言中直接嵌入汇编语言，实现 C 与汇编的混合编程。当然，这时候的参数使用规则要满足相应的 ATPCS 规则。

_inline 告诉编译器在合适的场合下，可以将该关键字声明的函数在其调用地方展开，以达到代码执行效率的最优化。所谓合适的地方，即编译器认为这种处理是合适的，例如处于循环中的简单函数，函数调用开销会很大，采用_inline 就可以很好解决这个问题。若_inline 修饰的函数展开后会影响代码的紧凑性和性能，这时编译器就可能无视_inline，将函数当作普通函数来处理。

_irq 关键字用于声明一个函数，这个函数可以被用作 irq 或 fiq 的异常中断处理程序。这不是说没有_irq 关键字修饰的函数就不能用作中断处理程序，只是有_irq 时编译器会自动保存函数中会使用到的寄存器，在函数返回时直接恢复寄存器的值，用户处理程序不用再考虑寄存器保护的问题。

还有其他的几个关键字就不再讨论了，如_pure、_softfp、_swi、_weak 等。

（2）数据定义相关的特殊关键字。

ARM 编译器中还定义了一些用于数据声明的特殊关键字，主要包括两大类：用于特殊数据类型定义的关键字和对数据作限定的关键字。

register，当一个数据声明为该类型时，它告诉编译器最好将该数据保存在寄存器中，某些数据会被经常访问到，例如循环语句中的控制变量，若把这类变量放在寄存器中，会加快程序执行效率。注意，这个关键字只能建议编译器这样做，但不能强制编译器这样做，编译器只能根据具体情况来处理这个数据，使用了该关键字后，可能影响编译器的优化功能，使得编译出的代码更长。

_int64，这个数据类型实际上是 long long 的同义词，它用于定义一个 64 位的整数。

以下几个关键字用于限制数据，告诉编译器对定义的这个数据变量做特殊处理。

_align 告诉编译器，这个变量在存放时需要按照指定的字节数对齐，由于数据栈操作要求是 8 字节对齐的，因此 LDM/STM 处理的数据应该是 8 字节对齐的，所以这些被访问数据定义要加_align(8)进行限制，以保证指令执行速度。

_packed 关键字用于告诉编译器，其限定的数据是 1 字节对齐的，即防止编译器的自动对齐。举个例子：


```
1  struct mydata{                1  _packed struct mydata{
2      char b;                   2      char b;
3      int a;                    3      int a;
4  }                             4  }
5  sizeof(struct mydata);        5  sizeof(struct mydata);
```

上面的两个 sizeof 运算符输出的结果是不一样的，前者输出为 8，后者输出为 5，没有使用_packed 时，编译器会自动将变量 a 的起始地址变为字对齐，这样由于结构体中字段 b 只占用了 1 个字节的空间，所以编译器会填充 3 个字节，在字地址处才为 a 分配空间，因此，第一个 sizeof 输出为 8，后一个输出为 5。

volatile 限制的变量，表示其值可能在程序之外被修改，如果要使用变量，需要重新对变量进行读取。这样的话编译器将不会对该变量进行优化操作，对于系统中的 I/O 寄存器，通常需要使用 volatile 类型的变量来访问。

_weak 关键词用于限定一个对象，如果该对象在连接时不存在，连接器也不会产生相应的错误信息。

➢ 编译器预定义宏

ARM 编译器内部还定义了一些宏，这些宏可以在程序中被使用。来看看几个最常用的宏，如表 3-3 所示，编译器中这样的宏还很多，读者可以参阅 ARM 编译器手册。

表 3-3　常见的编译器预定义宏

预定义宏	默认值	含义
_arm	--	编译器使用 armcc、tcc、armcpp、tcpp 时
_ARMCC_VERSION	Ver	代表编译器的版本号
_BIG_ENDIAN	--	大小端模式
DATE	date	编译源文件时的日期
FILE	name	当前被编译源文件的全路径名称

续表

预定义宏	默认值	含义
func	name	当前被编译的函数名称
LINE	name	当前被编译的代码的行号
_sizeof_int	4	Int 类型的长度
_TARGET_ARCH_xx	--	处理器体系编号，xx 为 4T、5TE 等
_TARGET_CPU_xx	--	目标 CPU 编号，xx 在编译时确定
_TARGET_FPU_xx	--	协处理器类型，xx 为 VFP、FPA 等
TIME	--	源文件被编译的时间

在 ADS 集成环境中，可以对编译器进行配置，其配置的本质也就是配置上面各个宏的取值，以及命令行属性等。在使用 CodeWarrior IDE 进行程序的开发时，可以在编译选项设置时，按照图 3-5 所示对编译器属性进行设置。

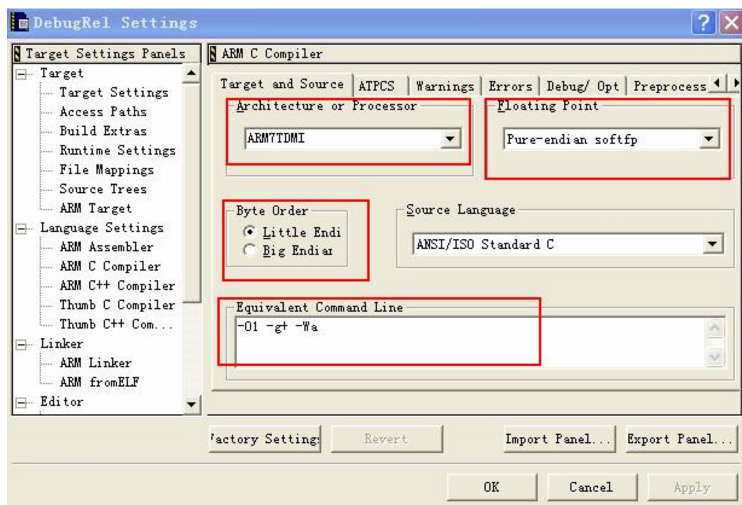

图 3-5　ARM 编译器选项设置

上图中可以对 C 编译器的 Target and Source、ATPCS、Warnings 等选项卡进行设置。在 Target and Source 选项卡中，如图中标注所示，可以对处理器类型、浮点处理器类型、大小端模式等进行设置，这点同上小节所讲的汇编器设置很类似。与所有编译器设置等效的命令行语句在 Equivalent Command Line 栏中给出，如图 3-5 中最下方的标记所示，-O1 表示了代码优化级别，它告诉编译器关闭那些严重影响调试效果的优化功能；-g+告诉编译器在目标文件中包含调试信息；-Wa 表示关闭某类警告信息，在这里不深究。

当然，也可以对编译器的 ATPCS 属性进行设置，如图 3-6 所示，用户可以根据具体情况选择需要的特殊 ATPCS 类型，如图中标记所示，它一共包含了四种特殊的 ATPCS 类型：支持 Thumb

程序和 ARM 程序混合调用的 ATPCS；支持数据栈限制检查的 ATPCS；支持代码段位置无关的 ATPCS；支持数据段位置无关的 ATPCS。这里没有选择任何的特殊 ATPCS，采用基本的 ATPCS 就可以了。

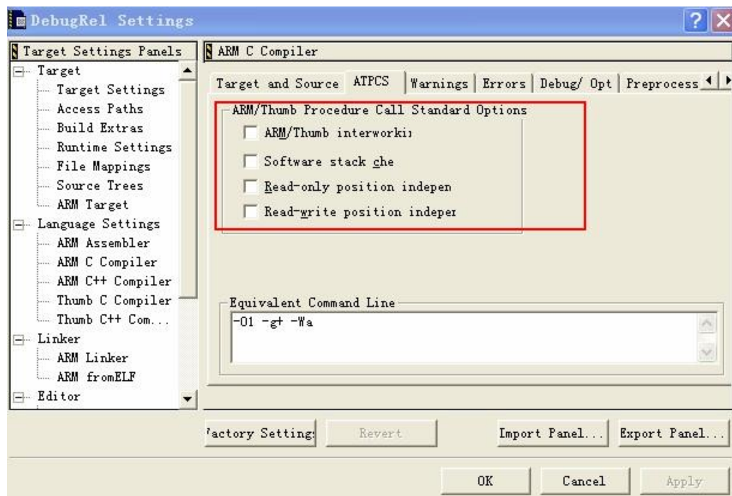

图 3-6　ARM 编译器的 ATPCS 属性

3.1.4　ARM 连接器

编译器或汇编器将各源文件编译为独立的 ELF 目标文件，但是这些目标文件最终要被放入到程序存储器中才能被执行。那么这些目标文件是怎样放入存储器的呢？或者说，各个 ELF 在存储器中的放置位置以及它们之间的关系是怎么样的呢？这些都需要有个工具来进行统一的组织和管理，这样我们的连接器 armlink 就粉墨登场了。它的功能就是将各目标文件连接为 ELF 格式的可执行映像文件（image）。映像文件的结构由以下几个因素来决定：映像文件中的域个数，映像文件下载到 Flash 中保存时的保存地址、映像文件在 RAM 中运行时的运行地址，这些本节都会一一道来。

➢　映像文件的结构

映像文件是一个层次性结构，最根本的，它由域（region）组成。同时，域由输出段（output section）组成，而输出段又是由输入段（input section）组成的。

各个输入段存在于编译器编译后的目标文件或者库文件中，通常一个目标文件由 RO 段、RW 段和 ZI 段组成（RO 段就是我们所说的只读段，可以是只读的代码，也可以是只读的数据；RW 段主要代表可读可写的数据，且在程序运行前已经被初始化为某个非 0 值；ZI 段也主要代表可读可写的数据，但在程序运行前它们未被初始化或者已经被初始化为 0。关于这三种段上数据的特性在本章结束处会有个具体的例子来说明）。输出段是通过将所有目标文件中具有相同属性的段组织到一起而形成的，一个域包含 1～3 个输出段（RO 段、RW 段和 ZI 段），在域中各个输出段的排列顺序一般为：首先是 RO 段，然后是 RW 段，最后才是 ZI 段。举一个例子来看看一个映像文件的组

织过程。

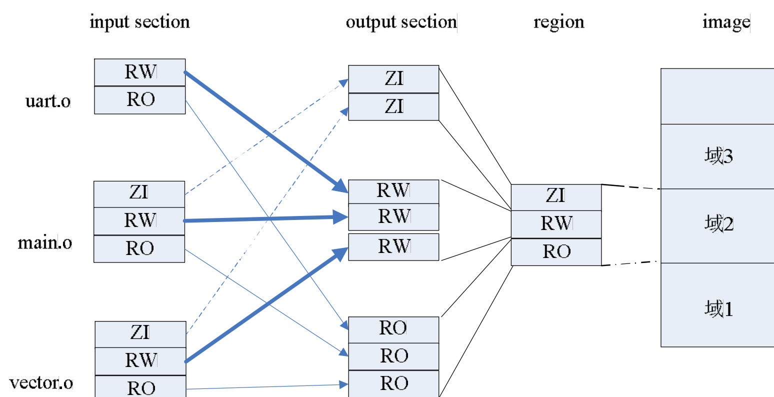

图 3-7　ARM 映像文件的组成

如图 3-7 所示，假设经过编译器编译后产生了三个目标文件：uart.o、main.o、vector.o，其中 vector.o 中包含了只读代码段 RO、初始化数据段 RW、未初始化数据段 ZI；main.o 也包含了 RO、RW、ZI 段；uart.o 只包含了 RO 和 RW 段。每个输出段（output section）都是将三个目标文件中的相同属性的段组织在一起，输出段也有自己的属性，它与组成它的输入段有相同的属性，至于各个输入段是以怎样的位置在输出段中排列的，这不在我们的讨论范围（连接器对于各个输入段的排放有着自己的一套规则，当然读者也可以在命令行中采用-first 或-last 来简单指定段排放的顺序，或者用一个 scatter 文件来告诉连接器各个段的详细排放顺序）；1~3 个不同属性的输出段可以组成一个域，在域中，输出段的排列顺序为，先放 RO 段，再放 RW 段，最后才是 ZI 段。一个映像文件（image）一般由一个或多个域组成，而往往域在 image 中的位置同域在实际存储器中的位置具有对应关系。

➢　映像文件的存储地址映射

ARM 映像文件各个组成部分在存储系统中的地址有两种：一是映像文件处于存储器中时的地址，称为加载时地址（load view），此时映像并没有运行。映像在运行之前，必须将自身的某些域搬移到某个地址才能运行，例如 RW 段的数据是可读可写的，它的加载时地址位于 Flash 中，而 Flash 中的数据并不是可读可写的（Flash 不能随机写，只能块擦出后再写），若不将它搬运到 RAM 中，程序是不能正常运行的。通常 RW 和 ZI 段数据都必须放在 RAM 中才能运行，所以这里又有了映像文件的运行时地址（Excution view）的概念。例如图 3-8 所示，RW 的加载时地址（load view）为 0x6000（该段所占存储区域的起始地址），RW 的运行时地址为 0x8000，这个地址位于 RAM 中，则整个映像文件两种地址间的对应关系如图 3-8 所示。

通常当存储在 ROM 中时，ZI 段是不需要被存储的，因为它们都未被初始化或者被初始化为 0，这样只需用一个参数保存该段的大小就可以了，在代码从加载时地址往运行时地址搬运的过程中，由于 ZI 段紧跟在 RW 段后面，所以只要将 RW 段搬运完毕，然后在紧跟其后的内存空间开辟出 ZI

段的大小，并将对应内存空间全部初始化为 0 就行了。如上图中，在 RW 段后面初始化了大小为 0x1000 的 ZI 段。在很多情况下，RO 段的代码也可以被搬运到 RAM 中，因为在 RAM 中的访问速度往往比 ROM 中的快，将代码段搬运到 RAM 中可以提高程序执行效率。我们使用的 PC 机也是相似的一个过程，处于 ROM 中的代码只用完成系统的初始化，然后它会将操作系统内核拷贝到内存中（称为内核加载），最后再跳到内存中的内核入口执行。

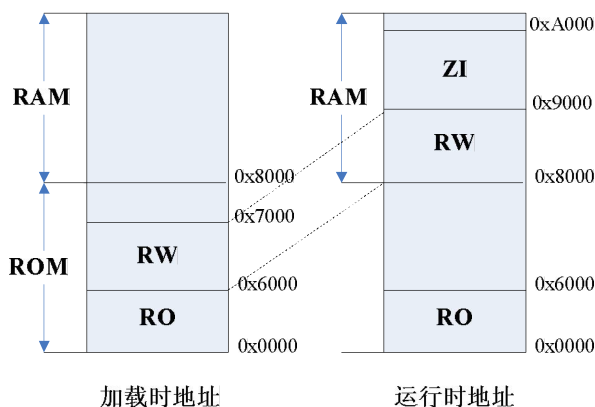

图 3-8　ARM 映像文件的地址映射

➢　映像文件结构描述信息

通常一个映像文件可以包含若干个域，各个域包含了若干个输出段，输出段又由多个输入段复杂地组合在一起，ARM 连接器需要很多信息以生成最终满足要求的映像文件。这些信息主要包括分组信息和定位信息。

（1）分组信息（Grouping）描述了输入段的排放规则，以组成最后的输出段和域；

（2）定位信息（Placment）描述了在 iamge 中，各个域是怎样组织在一起的，也可以说成各个域在存储空间中位置是怎样组织的，它们的保存或运行时的起始地址如何。

用户可以用两种方法来告诉连接器这些信息：一是通过简单的命令行选项，或者更简单说，直接在 Code Warrior IDE 上配置 armlink 就可以了（例如可以用-ro-base 指定 RO 段的运行时地址、-rw-base 指定 RW 段的运行时地址），这种方式一般用在简单的地址映射上；第二种，当映像文件中的各个段映射关系比较复杂时，如果用户想完全控制各个输入段的位置以及它们之间的联系，就需要一个专门的配置文件来告诉连接器其中的地址映射关系，通常将这样的文件称为分散加载文件（scatter 文件），关于分散加载文件的写法和格式不再讨论，但只要在命令行中使用-sactter filename 选项，就可以使用 scatter 文件了。

➢　映像文件的入口点

程序映像的入口点是整个程序可以开始执行的地方，通常可以将该入口点称为初始入口点（initial entry point）。每个映像必须有一个初始入口点，否则在程序连接过程中会产生警告信息。映像入口点一般保存在 ELF 的文件头中，当操作系统加载一个 ELF 的程序运行时，它就是通过跳

转到该程序入口点来实现程序运行。事实上，操作系统本身也是一个映像文件，它也有一个程序入口点，往往我们的系统启动程序将操作系统映像加载到内存后，也是直接跳到该入口点，然后操作系统就运行起来了。可以通过两种方法来为映像指明程序入口点：

（1）在连接命令行中使用-entry address 来为程序指明入口点，注意这里的 address 为映像的运行时地址。对于 ARM 处理器来说，由于上电后它会到 0x0 地址处取值执行，所以对于 imange 映像来说，其程序入口点也通常设置在 0x0 处，即用-entry 0x0 来指定，即 0x0 处是整个映像文件的第一条指令。

（2）在汇编指令中用了伪指令 ENTRY 指明程序入口，且工程所有文件中只有一处 ENTRY 的使用，这样连接器就会直接将该 ENTRY 作为程序的入口点了。注意，若程序中有多处使用 ENTRY 伪指令，那么连接器是不会选择任何 ENTRY 指定的地址作为入口点的，因为它不知道何从选择，这种情况下就只有使用（1）中所示的方法了。

在程序中什么情况下会使用多个 ENTRY 呢？事实上，对于异常中断处理程序，由于其不能直接与外部产生调用关系（而是在中断发生时，处理器直接跳转来执行的，它们也可以看成是中断发生时程序的入口点），因此在很多情况下，连接器会认为这些程序没有被用到，它在代码优化的过程中就会删除这些没用的段，这样的话这些代码段链接后就不见了，因此在这些处理程序的段中，经常被加上 ENTRY 伪操作，告诉编译器在优化的时候不要优化我，因为我也可能是程序的一个入口点。

➢ 连接器生成的符号

ARM 连接器内部定义了一些符号，用来表示连接器对映像文件的连接结果，如各个域、输入段、输出段的起始地址和结束地址、大小等信息。这些符号都包含$$字符（包含$$字符的符号一般都是 ARM 开发工具内部使用的符号），它们可以在用户的汇编程序中直接使用，也可以在 C 程序中通过 extern 来直接引用，在连接过程中，连接器会自动将这些值替换。

如表 3-4 所示为连接器所生成的与域相关的符号。

表 3-4 连接器中与域相关的符号

符号	含义
Load$$region_name$$Base	域 region_name 的加载时起始地址
Image$$region_name$$Base	域 region_name 的运行时起始地址
Image$$region_name$$Length	域 region_name 的运行时长度（以字为单位）
Image$$region_name$$Limit	域 region_name 运行时的结束处的后一字节地址
Image$$region_name$$ZI$$Base	域 region_name 的 ZI 段运行时的起始地址
Image$$region_name$$ZI$$Length	域 region_name 的 ZI 段运行时的长度（以字为单位）
Image$$region_name$$ZI$$Limit	域 region_name 的 ZI 段运行时结束处的后一字节地址

注意上面的 region_name 表示了域的名称，当使用了 scatter 文件时，文件中已经指定了各个域的名称。未使用 scatter 文件时，则默认采取下面的名称：对于只读的域，使用 ER_RO；对于能读

能写的域，使用 ER_RW；对于全部 0 初始化的域，用 ER_ZI。

如果未使用 scatter 文件规定各个段的名字，则连接器还会为各个输入段定义另一些与段相关的符号，如表 3-5 所示。如果用户使用了 scatter 文件，则表中所述的这些符号都是无效的，用户程序中不应该试图引用这些符号。

表 3-5　连接器中与段相关的符号

符号	含义
Image$$RO$$Base	RO 输出段运行时起始地址
Image$$RO$$Limit	RO 输出段运行时的结束处的后一字节地址
Image$$RW$$Base	RW 输出段运行时起始地址
Image$$RW$$Limit	ZI 输出段运行时的结束处的后一字节地址
Image$$ZI$$Base	ZI 输出段运行时起始地址
Image$$ZI$$Limit	ZI 输出段运行时的结束处的后一字节地址

注意，上表中 Image$$RW$$Limit 项记录的是 ZI 输出段的值 Image$$ZI$$Limit，这点没错，在数据手册中是这样描述的，为了保证与以往代码的兼容性，Image$$RW$$Limit 用来保存 ZI 段的 limit 值，而不是 RW 段的。输出段的符号是我们在启动代码中会用到的，举个例子对它加以说明，假如在图 3-8 的情况下，假设没有使用 scatter 文件，则 Image$$RO$$Base 为 0x0000；Image$$RO$$Limit 为 0x6000；Image$$RW$$Base 为 0x8000；Image$$ZI$$Base 为 0x9000；Image$$RW$$Limit 和 Image$$ZI$$Limit 都为 0xA000。

在使用 CodeWarrior IDE 进行程序的开发时，可以在连接选项设置时，按照图 3-9 所示对连接器属性和连接信息进行设置。

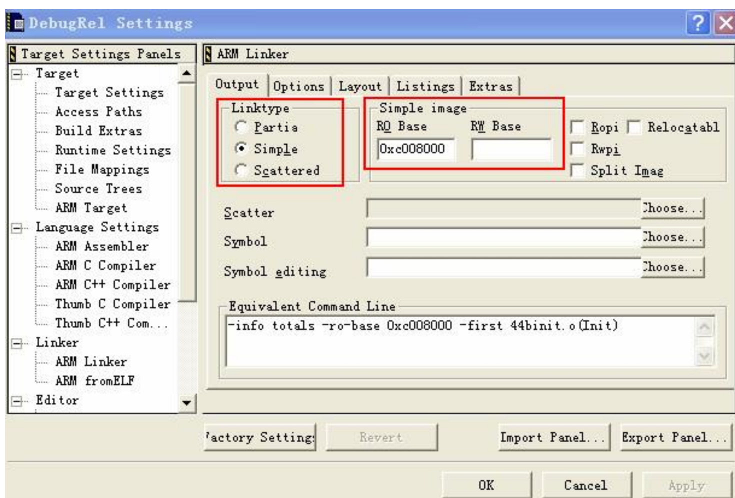

图 3-9　映像文件定位信息

如上图中可以对连接器的 Output、Options、Layout 等选项卡进行设置。在 Output 选项卡中，如图中标注所示，可以进行映像文件定位信息的设置，可以选择 Simple 方式，这样就只用在右侧 Simple image 栏填入映像的 RO 起始地址等就可以了，这里也有很多可以继续深入说的，在后续根据具体的工程情况加以说明，更有利于读者的理解；另外，也可以选择 Scattered 方式，这样用户就可以在下方的 Scatter 栏中选择相应的 Scatter 文件了。

另外，对于映像文件的分组信息，可以在 Layout 选项卡中进行简单设置，如图 3-10 所示。

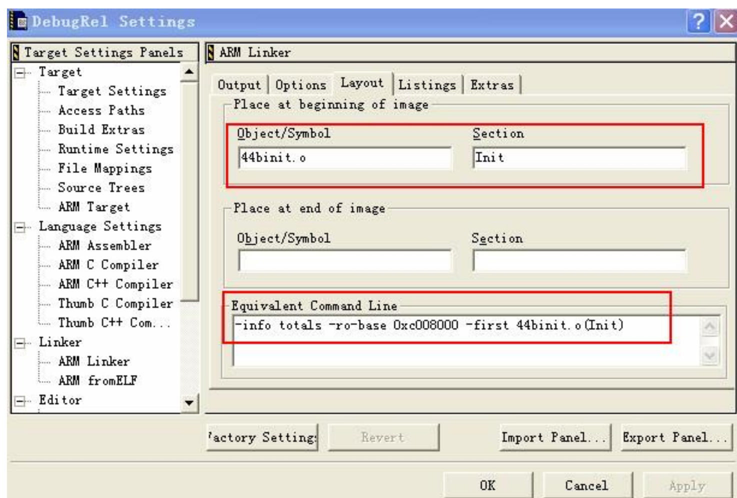

图 3-10　映像文件分组信息

如上图中，第一个标记框内指出了哪一个目标文件的哪一个段放在整个映像文件的起始处，这点是非常必要的。图中第二个标记框显示了与所有连接器设置等效的命令行语句。

3.2　程序的调试

3.2.1　几种调试方式

程序调试是用户程序开发过程中的重要阶段，ARM 处理器和 ARM 开发工具为用户提供了完善的程序调试手段。本节介绍用户程序调试的相关知识，后续会用到一些术语，先在这里做个总的交代。

宿主机：或叫主机，即调试时要用到的 PC 机，PC 机是程序开发、调试的重要依赖体。

目标机：被调试程序的运行环境，在这里就是我们的目标开发板。

调试器：我们用的 AXD 就属于一种调试器，主要用来向目标机发送调试命令等。

调试代理：调试代理可以运行在主机或者目标机上，可以用来完成目标机和主机之间数据或命令的解析、转发。调试代理种类很复杂，往往需要一系列的软硬件支持才能完成其功能。上述四者

之间的关系可以通过图 3-11 来描述。

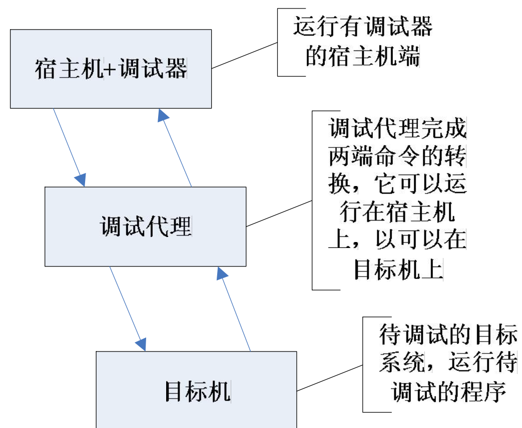

图 3-11　调试系统的基本组成

好吧，总的来说，可以通过下面四种方式来调试我们的 ARM 程序。

- 指令级仿真

指令级仿真是通过在 PC 机上运行仿真软件来实现程序调试的方法。这种方式通过 PC 机上的软件来模拟目标系统的 CPU，例如寄存器、存储单元等，这样嵌入式软件的开发可以在目标硬件系统并不存在的条件下进行。指令级仿真的过程很好理解，虚拟的 CPU 会对可执行映像文件中的每一条指令依次解释、执行，例如 MOV R1, #0，就代表往虚拟的寄存器 R1 中写 0，等等。指令仿真的执行速度较慢，但是可以验证程序逻辑，在某些条件下是一种重要的仿真手段。指令集仿真调试属于纯软件仿真，它比较特殊，不需要目标机，所以调试器与调试代理都运行在同一台主机上。后面会说到的调试代理 ARMulator 就属于这种类型，它是一种最简单最便宜的仿真方法。指令集仿真也存在很多的不足，例如无法仿真外部中断、无法反映外设情况、仿真的时序和实际硬件时序也存在差异等。

- 在线仿真

在线仿真（In Circuit Emulator，ICE）通过使用仿真头完全取代目标板上的 CPU，以实现调试的目的。用这种方式仿真时，通常将目标机上的 CPU 取下，同时将 ICE 的 CPU 引线接到目标机的 CPU 插槽中。ICE 可以完全仿真处理器芯片的行为，且在仿真头上会有更多的引脚或寄存器，以提供更加深入的调试功能。ICE 在一些实时性能分析、时序要求较高的场合有很大的优势，特别适用在调试实时应用程序、设备驱动程序以及对硬件进行功能和性能测试的场合。但通常这类仿真器必须采用极其复杂的设计和工艺，价格比较昂贵，因此目前在线仿真器主要用在 ARM 的硬件开发中，在软件的开发中较少使用。

- 驻留监控程序

驻留监控程序是运行在目标机上的一段程序，它负责监控目标机上被调试程序的运行，它可以

和宿主机端的调试器一起完成对应用程序的调试。驻留监控程序通常被固化到目标机的 ROM 空间中，在目标机复位后，首先执行的就是这段程序，它对目标机进行必要的初始化后，初始化自己的运行空间，然后等待宿主机的命令。宿主机上的调试器可以使用串口、并口、网口与监控程序实现交互，以完成程序的执行、存储器或寄存器的读写、断点设置等工作。这种调试方式的优点在于不需要额外的硬件就能实现调试工作，但它需要占用目标板的资源：处理器、存储器和通信接口等。ANGEL 就是一个在 ARM 的调试过程中被广泛使用的监控程序，基于 ANGEL 的调试系统可以用来调试目标机上运行的 ARM 程序或 Thumb 程序。主机上的调试器向目标机器上的 ANGEL 发送请求，目标机器上的 ANGEL 截取这些请求，并根据请求来执行相应的操作。例如，当主机上的调试器请求设置断点时，ANGEL 在目标程序的相应位置插入一条未定义指令，当程序运行到这条指令时，就会产生未定义指令异常中断，在异常指令处理程序中，完成断点需要的功能。

- JTAG 片上调试

片上调试（On Chip Debugging，OCD）是 CPU 芯片自身提供的一种调试功能，即在 CPU 内部集成了调试电路。JTAG 属于片上调试的一种类型，是目前使用最广泛的一种调试方式。JTAG 调试方式是我们这节将重点讲解的方式，目前市场上的有很多基于 JTAG 的调试产品，如 Multi-ICE、Ulink、Jlink 等。在下面的内容中，先简单说说 JTAG 调试的原理及特点，然后再看看 ARM7TDMI 内部的调试电路组成。

（1）JTAG 标准。

联合测试行动小组 JTAG（Joint Test Action Group）在 20 世纪 80 年代后期起草了关于边界扫描测试 BST（Boundary-Scan Testing）规范，这个规范主要用于芯片内部测试以及对系统进行仿真和测试，这个规范于 1990 年正式成为 IEEE1149.1 工业标准，简称 JTAG 标准。JTAG 标准通过在芯片内部封装专门的测试电路以及测试访问口 TAP（Test Access Port），可以实现对芯片内部各部件的访问。JTAG 调试方式目前是开发、调试嵌入式系统的有效手段，许多嵌入式器件，包括复杂的 ARM、DSP、FPGA，简单的 MSP430、C8051F 等单片机都提供了对 JTAG 的支持。除开调试功能外，JTAG 接口还常用于实现 ISP（In-System Programmable，在系统编程）。

在 JTAG 调试中，边界扫描是一个很重要的概念。边界扫描技术的基本思想是在靠近芯片的输入/出管脚上增加一个移位寄存器单元。因为这些移位寄存器单元分布在芯片的边界上（周围），所以被称为边界扫描寄存器（Boundary-Scan Register Cell）。当芯片处于调试状态时，这些寄存器就可以将芯片和外围的输入/出隔离开来，通过这些边界扫描寄存器单元，可以实现对芯片输入/出信号的观察和控制。在非调试状态下，这些边界寄存器对于芯片来说是完全透明的，所以不会对芯片的正常运行带来影响。另一方面，芯片上的这些寄存器单元相互间被串行的连接了起来，这样就在芯片周围形成了一个边界扫描链（Boundary-Scan Register Chain）。通过相应的时钟信号和控制信号，可以在边界扫描链上实现数据的串行输入或输出，这样便可方便地观察和控制处在调试状态下的芯片。一般来说，芯片都会提供几条独立的边界扫描链，以实现完整的测试功能。

那么如何来控制和访问这些边界扫描链呢？这就是 TAP 控制器的功能了。TAP 控制器通过 TAP 口，可以访问到与扫描链相关的所有数据寄存器和指令寄存器，从而实现了对所有边界扫描链的控