

第8章 ARM ADS 集成开发环境的使用

在这一章里，将介绍 ARM 开发软件 ADS(ARM Developer Suite)。通过学习如何在 CodeWarrior IDE 集成开发环境下编写，编译一个工程的例子，使读者能够掌握在 ADS 软件平台下开发用户应用程序。本章还描述了如何使用 AXD 调试工程，使读者对于调试工程有个初步的理解，为进一步的使用和掌握调试工具起到抛砖引玉的作用。

本章主要内容有：

- ADS 软件组成介绍
- 使用 ADS 创建工程
- 用 AXD 进行代码调试

8.1 ADS 集成开发环境组成介绍

ARM ADS 全称为 ARM Developer Suite。是 ARM 公司推出的新一代 ARM 集成开发工具。现在 ADS 的最新版本是 1.2，它取代了早期的 ADS1.1 和 ADS1.0。它除了可以安装在 Windows NT4, Windows 2000, Windows 98 和 Windows 95 操作系统下，还支持 Windows XP 和 Windows Me 操作系统。

ADS 由命令行开发工具，ARM 时实库，GUI 开发环境(Code Warrior 和 AXD)，实用程序和支持软件组成。有了这些部件，用户就可以为 ARM 系列的 RISC 处理器编写和调试自己的开发应用程序了。

下面就详细介绍一下 ADS 的各个组成部分。

8.1.1 命令行开发工具

这些工具完成将源代码编译，链接成可执行代码的功能。

ADS 提供下面的命令行开发工具：

armcc

armcc 是 ARM C 编译器。这个编译器通过了 Plum Hall C Validation Suite 为 ANSI C 的一致性测试。armcc 用于将用 ANSI C 编写的程序编译成 32 位 ARM 指令代码。

因为 armcc 是我们最常用的编译器，所以对此作一个详细的介绍。

在命令控制台环境下，输入命令：

```
armcc -help
```

可以查看 armcc 的语法格式以及最常用的一些操作选项

armcc 最基本的用法为：`armcc [options] file1 file2 ... fileN`

这里的 option 是编译器所需要的选项，file1, file2...fileN 是相关的文件名。

这里简单介绍一些最常用的操作选项。

-c：表示只进行编译不链接文件；

-C：(注意：这是大写的 C)禁止预编译器将注释行移走；

-D<symbol>：定义预处理宏，相当于在源程序开头使用了宏定义语句 `#define symbol`，这里 symbol 默认为 1；

-E：仅仅是对 C 源代码进行预处理就停止；

- g<options>：指定是否在生成的目标文件中包含调试信息表；
 - I<directory>：将 directory 所指的路径添加到 #include 的搜索路径列表中去；
 - J<directory>：用 directory 所指的路径代替默认的对 #include 的搜索路径；
 - o<file>：指定编译器最终生成的输出文件名。
 - O0：不优化；
 - O1：这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 关闭了影响调试结果的优化功能；
 - O2：该优化级别提供了最大的优化功能；
 - S：对源程序进行预处理和编译，自动生成汇编文件而不是目标文件；
 - U<symbol>：取消预处理宏名，相当于在源文件开头，使用语句 #undef symbol；
 - W<options>：关闭所有的或被选择的警告信息；
- 有关更详细的选项说明，读者可查看 ADS 软件的在线帮助文件。

armcpp

armcpp 是 ARM C++ 编译器。它将 ISO C++ 或 EC++ 编译成 32 位 ARM 指令代码。

tcc

tcc 是 Thumb C 编译器。该编译器通过了 Plum Hall C Validation Suite 为 ANSI 一致性的测试。tcc 将 ANSI C 源代码编译成 16 位的 Thumb 指令代码。

tcpp

tcpp 是 Thumb C++ 编译器。它将 ISO C++ 和 EC++ 源码编译成 16 位 Thumb 指令代码。

armasm

armasm 是 ARM 和 Thumb 的汇编器。它对用 ARM 汇编语言和 Thumb 汇编语言写的源代码进行汇编。

armlink

armlink 是 ARM 连接器。该命令既可以将编译得到的一个或多个目标文件和相关的一个或多个库文件进行链接，生成一个可执行文件，也可以将多个目标文件部分链接成一个目标文件，以供进一步的链接。ARM 连接器生成的是 ELF 格式的可执行映像文件。

armsd

armsd 是 ARM 和 Thumb 的符号调试器。它能够进行源码级的程序调试。用户可以在用 C 或汇编语言写的代码中进行单步调试，设置断点，查看变量值和内存单元的内容。

8.1.1.1 armcc 用法详解

下面为读者介绍上述的 4 种 ARM C 和 C++ 编译器的命令通用语法。

```
compiler [PCS-options] [source-language] [search-paths] [preprocessor-options]
[output-format] [target-options] [debug-options] [code-generation-options] [warning-options]
[additional-checks] [error-options] [source]
```

用户可以通过命令行操作选项控制编译器的执行。所有的选项都是以符号“-”开始，有些选项后面还跟有参数。在大多数情况下，ARM C 和 C++ 编译器允许在选项和参数之间存在空格。

命令行中各个选项出现顺序可以任意。

这里的 compiler 是指 armcc，tcc，armcpp 和 tcpp 中的一个；

PCS-options：指定了要使用的过程调用标准；

source-language：指定了编译器可以接受的编写源程序的语言种类。对于 C 编译器默认的语言是 ANSI C，对于 C++ 编译器默认是 ISO 标准 C++；

search-paths：该选项指定了对包含的文件(包括源文件和头文件)的搜索路径；

preprocessor-options：该选项指定了预处理器的行为，其中包括预处理器的输出和宏定义等特性；

output-format：该选项指定了编译器的输出格式，可以使用该项生成汇编语言输出列表文件和目标文件；

target-options：该选项指定目标处理器或 ARM 体系结构；

debug-options：该选项指定调试信息表是否生成，和该调试信息表生成时的格式；

code-generation-options：该选项指定了例如优化，字节顺序和由编译器产生的数据对齐格式等选项；

warning-options：该选项决定警告信息是否产生；

additional-checks：该选项指定了几个能用于源码的附加检查，例如检查数据流异常，检查没有使用的声明等；

error-options：该选项可以关闭指定的可恢复的错误，或者将一些指定的错误降级为警告；

source：该选项提供了包含有 C 或 C++ 源代码的一个或多个文件名，默认的，编译器在当前路径寻找源文件和创建输出文件。如果源文件是用汇编语言编写的(也就是说该文件的文件名是以.s 作为扩展名)，汇编器将被调用来处理这些源文件。

如果操作系统对命令行的长度有限制，可以使用下面的操作，从文件中读取另外的命令行选项：

-via filename

该命令打开文件名为 filename 的文件，并从中读取命令行选项。用户可以对 -via 进行嵌套调用，亦即，在文件 filename 中又通过 -via filename2 包含了另外一个文件。

在下面的例子中，从 input.txt 文件中读取指定的选项，作为 armcpp 的操作选项：

```
armcpp -via input.txt source.c
```

以上是对编译器选项的一个简单概述。它们(包括后面还要介绍的其他一些命令工具)既可以在命令控制台环境下使用，同时由于它们被嵌入到了 ADS 的图形界面中，所以也可以在图形界面下使用。

8.1.1.2 armlink 用法详解

在介绍 armlink 的使用方法之前，先介绍要涉及到的一些术语。

映像文件(image)：是指一个可执行文件，在执行的时候被加载到处理器中。一个映像文件有多个线程。它是 ELF(Executable and linking format)格式的。

段(Section)：描述映像文件的代码或数据块。

RO：是 Read-only 的简写形式。

RW：是 Read-write.的简写形式。

ZI：是 Zero-initialized 的简写形式。

输入段(input section)：它包含着代码，初始化数据或描述了在应用程序运行之前必须要初始化为 0 的一段内存。

输出段(output section)：它包含了一系列具有相同的 RO，RW 或 ZI 属性的输入段。

域(Regions)：在一个映像文件中，一个域包含了 1 至 3 个输出段。多个域组织在一起，就构成了最终的映像文件。

Read Only Position Independent(ROPI)：它是指一个段，在这个段中代码和只读数据的地址在运行时候可以改变。

Read Write Position Independent(RWPI)：它是指一个段，在该段中的可读/写的数据地址在运行期间可以改变。

加载时地址：是指映像文件位于存储器(在该映像文件没有运行时)中的地址。

运行时地址：是指映像文件在运行时的地址。

下面介绍一下 armlink 命令的语法

完整的连接器命令语法如下：

```
armlink [-help] [-vsn] [-partial] [-output file] [-elf] [-reloc][[-ro-base address] [-ropi]
[-rw-base address] [-rwpi] [-split]
[-scatter file][[-debug|-nodebug][[-remove?RO/RW/ZI/DBG]]-noremove] [-entry location ]
[-keep section-id] [-first section-id] [-last section-id] [-libpath pathlist] [-scanlib|-noscanlib]
[-locals|-nolocals] [-callgraph] [-info topics] [-map] [-symbols] [-symdefs file] [-edit file]
[-xref] [-xreffrom object(section)] [-xrefto object(section)] [-errors file] [-list file] [-verbose]
[-unmangled [-mangled] [-match crossmangled]][-via file] [-strict]
[-unresolved symbol][[-MI|-LI|-BI] [input-file-list]
```

上面各选项的含义分别为：

-help

这个选项会列出在命令行中常用的一些选项操作。

-vsn

这个选项显示出所用的 armlink 的版本信息。

-partial

用这个选项创建的是部分链接的目标文件而不是可执行映像文件。

-output file

这个选项指定了输出文件名，该文件可能是部分链接的目标文件，也可能是可执行映像文件。如果输出文件名没有特别指定的话，armlink 将使用下面的默认：

如果输出是一个可执行映像文件，则生成的输出文件名为__image.axf；

如果输出是一个部分链接的目标文件，在生成的文件名为__object.o；

如果没有指定输出文件的路径信息，则输出文件就在当前目录下生成。如果指定了路径信息，则所指定的路径成为输出文件的当前路径。

-elf

这个选项生成 ELF 格式的映像文件，这也是 armlink 所支持的唯一的一种输出格式，这是默认选项。

-reloc

这个选项生成可重定址的映像。

一个可重定址的映像具有动态的段，这个段中包含可重定址信息，利用这些信息可以在链接后，进行映像文件的重新定址；

-reloc，-rw-base 一起使用，但是如果没有-split 选项，链接时会产生错误。

-ro-base address

这个选项将包含有 RO(Read-Only 属性)输出段的加载地址和运行地址设置为 address，该地址必须是字对齐的，如果没有指定这个选项，则默认的 RO 基地址值为 0x8000。

-ropi

这个选项使得包含有 RO 输出段的加载域和运行域是位置无关的。如果该选项没有使用，则相应的域被标记为绝对的。通常每一个只读属性的输入段必须是只读位置无关的。如果使用了这个选项，armlink 将会进行以下操作：

检查各段之间的重定址是否有效；

确保任何由 armlink 自身生成的代码是只读位置无关的。

这里希望读者注意的是，ARM 工具直到 armlink 完成了对输入段的处理后，才能够决定最终的生成映像是否为只读位置无关的。这就意味着，即使为编译器和汇编器指定了

ROPI 选项, armlink 也可能产生 ROPI 错误信息。

-rw-base address

这个选项设置包含 RW(Read/Write 属性)输出段的域的运行时地址, 该地址必须是字对齐的。

如果这个选项和-split 选项一起使用, 将设置包含 RW 输出段的域的加载和运行时地址都设置在 address 处。

-rwpi

这个选项使得包含有 RW 和 ZI(Zero Initialization, 初始化为 0)属性的输出段的加载和运行时域为位置无关的。如果该选项没有使用, 相应域标记为绝对的。这个选项要求-rw-base 选项后有值, 如果-rw-base 没有指定的话, 默认其值为 0, 即相当于-rw-base 0。通常每一个可写的输入段必须是可读/ 可写的位置无关的。

如果使用了该选项, armlink 会进行以下的操作:

检查可读/可写属性的运行域的输入段是否设置了位置无关属性;

检查在各段之间的重定址是否有效;

生成基于静态寄存器 sb 的条目, 这些在 RO 和 RW 域被拷贝和初始化的时候会用到。

编译器并不会强制可写的的数据一定要为位置无关的, 这就是说, 即使在为编译器和汇编器指定了 RWPI 选项, armlink 也可能生成数据不是 RWPI 的信息。

-split

这个选项将包含 RO 和 RW 属性的输出段的加载域, 分割成 2 个加载域。一个是包含 RO 输出段的加载域, 默认的加载地址为 0x8000, 但是可以用-ro-base 选项设置其他的地址值, 另一个加载域包含 RW 属性的输出段, 由-rw-base 选项指定加载地址, 如果没有使用-rw-base 选项的话, 默认使用的是-rw-base 0。

-scatter file

这个选项使用在 file 中包含的分组和定位信息来创建映像内存映射。

注意, 如果使用了该选项的话, 必须要重新实现堆栈初始化函数 __user_initial_stackheap()。

-debug

这个选项使输出文件包含调试信息, 调试信息包括, 调试输入段, 符号和字符串表。这是默认的选项。

-nodebug

这个选项使得在输出文件中不包含调试信息。生成的映像文件短小, 但是不能进行源码级的调试。armlink 对在输入的目标文件和库函数中发现的任何调试输入段都不予处理, 当加载映像文件到调试器中的时候, 也不包含符号和字符串信息表。这个选项仅仅是对装载到调试器的映像文件的大小有影响, 但是对要下载到目标板上的二进制代码的大小没有任何影响。

如果用 armlink 进行部分链接生成目标文件而不是映像文件, 则虽然在生成的目标文件中不含有调试输入段, 但是会包含符号和字符串信息表。

这里特别请读者注意的是:

如果要在链接完成后使用 fromELF 工具的话, 不可使用-nodebug 选项, 这是因为如果生成的映像文件中不包含调试信息的话, 则有下面的影响:

fromELF 不能将映像文件转换成其他格式的文件;

fromELF 不能生成有意义的反汇编列表。

-remove (RO/RW/ZI/DBG)

使用这个选项会将输入段未使用的段从映像文件中删除。如果输入段中含有映像文件

入口点或者该输入段被一个使用的段所引用，则这样的输入段会当作已使用的段。

在使用这个选项时候要注意，不要删除异常处理函数。使用 `-keep` 选项来标识异常处理函数，或用 `ENTRY` 伪指令标明是入口点。

为了更精确的控制删除未使用的段，可以使用段属性限制符。可以使用以下的段属性限制符：

`RO`

删除所有未使用的 `RO` 属性的段；

`RW`

删除所有未使用的 `RW` 属性的段；

`ZI`

删除所有未使用的 `ZI` 属性的段；

`DBG`

删除所有未使用的 `DEBUG` 属性的段。

这些限制符出现的顺序是任意的，但是它们必须要有“`()`”括住，多个限制符之间要用符号“`/`”进行间隔。ADS 软件中默认选项是 `-remove (RO/RW/ZI/DBG)`。

如果没有指定段属性限制符，则所有未使用的段都会被删除。因为 `-remove` 就等价于 `-remove(RO/RW/ZI/DBG)` 选项。

`-noremove`

这个选项保留映像文件中所有未被使用的段。

`-entry location`

这个选项指定映像文件中唯一的初始化入口点。一个映像文件可以包含多个入口点，使用这个命令定义的初始化入口点是存放在可执行文件的头部，以供加载程序加载时使用。当一个映像文件被装载时，ARM 调试器使用这个入口点地址来初始化 PC 指针。初始化入口点必须满足下面的条件：

映像文件的入口点必须位于运行域内；

运行域必须是非覆盖的，并且必须是固定域(就是说，加载域和运行域的地址相同)。

在这里可以用以下的参数代替 `location` 参数：

1. 入口点地址：这是一个数值，例如 `-entry 0x0`；
2. 符号：该选项指定映像文件的入口点为该符号所代表的地址处，比如：

`-entry int_handler`

表示程序入口点在符号 `int_handler` 所在处。

如果该符号有多处定义存在，`armlink` 将产生出错信息。

`offset+object(section)`：该选项指定在某个目标文件的段的内部的某个偏移量处为映像文件的入口地址，例如：

`-entry 8+startup(startupseg)`

如果偏移量值为 0，可以简写成 `object(section)`，如果输入段只有一个，则可以简化为 `object`。

`-keep section-id`

使用该选项，可以指定保留一个输入段，这样的话，即使该输入段没有在映像文件中使用，也不会被删除。参数 `section-id` 取下面一些格式：

1. `symbol`

该选项指定定义 `symbol` 的输入段不会在删除未使用的段时被删除。如果映像文件中有多处 `symbol` 定义存在，则所有包含 `symbol` 定义的输入段都不会被删除。例如：

`-keep int_handler`

则所有定义 `int_handler` 的符号的段都会保留，而不被删除。

为了保留所有含有以 `_handler` 结尾的符号的段，可以使用如下的选项：

`-keep *_handler`

2. `object(section)`

这个选项指定了在删除未使用段时，保留目标文件中的 `section` 段。输入段和目标名是不区分大小写的，例如，为了在目标文件 `vectors.o` 中保留 `vect` 段，使用：

`-keep vectors.o(vect)`

为了保留 `vectors.o` 中的所有以 `vec` 开头的段名，可以使用选项：

`-keep vectors.o(vec*)`

3. `object`

这个选项指定在删除未使用段时，保留该目标文件唯一的输入段。目标名是不区分大小写的，如果使用这个选项的时候，目标文件中所含的输入段不止一个的话，`armlink` 会给出出错信息。比如，为了保留每一个以 `dsp` 开头的只含有唯一输入段的目标文件，可以使用如下的选项：

`-keep dsp*.o`

`-first section-id`

这个选项将被选择的输入段放在运行域的开始。通过该选项，将包含复位和中断向量地址的段放置在映像文件的开始，可以用下面的参数代替 `section-id`：

1. `symbol`

选择定义 `symbol` 的段。禁止指定在多处定义的 `symbol`，因为多个段不能同时放在映像文件的开始。

2. `object(section)`

从目标文件中选择段放在映像文件的开始位置。在目标文件和括号之间不允许存在空格，例如

`-first init.o(init)`

3. `object`

选择只有一个输入段的目标文件。如果这个目标文件包含多个输入段，`armlink` 会产生错误信息。用这个选项的例子如下：

`-first init.o`

这里希望读者注意的是：

使用 `-first` 不能改变在域中按照 `RO` 段放在开始，接着放置 `RW` 段，最后放置 `ZI` 段的基本属性排放顺序。如果一个域含有 `RO` 段，则 `RW` 或 `ZI` 段就不能放在映像文件的开头。类似地，如果一个域有 `RO` 或 `RW` 段，则 `ZI` 段就不能放在文件开头。

两个不同的段不能放在同一个运行时域的开头，所以使用该选项的时候只允许将一个段放在映像文件的开头。

`-last section-id`

这个选项将所选择的输入段放在运行域的最后。例如，用这个选项能够强制性的将包含校验和的输入段放置在 `RW` 段的最后。使用下面的参数可以替换 `section-id`。

1. `symbol`

选择定义 `symbol` 的段放置在运行域的最后。不能指定一个有多处定义的 `symbol`。使用该参数的例子如下：

`-last checksum`

2. `object(section)`

从目标文件中选择 `section` 段。在目标文件和后面的括号间不能有空格，用该参数的例

子为：

-last checksum.o(check)

3. object

选择只有一个输入段的目标，如果该目标文件中有多个输入段，armlink 会给出出错信息。

和-first 选项一样，需要读者注意的是：

使用-last 选项不能改变在域中将 RO 段放在开始，接着放置 RW 段，最后放置 ZI 段的输出段基本的排放顺序。如果一个域含有 ZI 段，则 RW 段不能放在最后，如果一个域含有 RW 或 ZI 段，则 RO 段不能放在最后。

在同一个运行域中，两个不同的段不能同时放在域的最后位置。

-libpath pathlist

这个选项为 ARM 标准的 C 和 C++库指定了搜索路径列表。

注意，这个选项不会影响对用户库的搜索路径。

这个选项覆盖了环境变量 ARMLIB 所指定的路径。参数 pathlist 是一个以逗号分开的多个路径列表，即为 path1, path2,... pathn，这个路径列表只是用来搜索要用到的 ARM 库函数。默认的，对于包含 ARM 库函数的默认路径是由环境变量 ARMLIB 所指定的。

-scanlib

这个选项启动对默认库(标准 ARM C 和 C++库)的扫描以解析引用的符号。这个选项是默认的设置。

-noscanlib

该选项禁止在链接时候扫描默认的库。

-locals

这个选项指导链接器在生成一个可执行映像文件的时候，将本地符号添加到输出符号信息表中。该选项是默认设置。

-nolocals

这个选项指导链接器在生成一个可执行映像文件的时候，不要将本地符号添加到输出符号信息表中。如果想减小输出符号表的大小，可以使用该选项。

-callgraph

该选项创建一个 HTML 格式的静态函数调用图。这个调用图给出了映像文件中所有函数的定义和引用信息。对于每一个函数它列出了：

1. 函数编译时候的处理器状态(ARM 状态还是 Thumb 状态)；
2. 调用 func 函数的集合；
3. 被 func 调用的函数的集合；
4. 在映像文件中使用的 func 寻址的次数。

此外，调用图还标识了下面的函数：

1. 被 interworking veneers 所调用的函数；
2. 在映像文件外部定义的函数；
3. 允许未被定义的函数(以 weak 方式的引用)；

静态调用图还提供了堆栈使用信息，它显示出了：

1. 每个函数所使用的堆栈大小；
2. 在全部的函数调用中，所用到的最大堆栈大小。

-info topics

这个选项打印出关于指定种类的信息，这里的参数 topics 是指用逗号间隔的类型标识符列表。类型标识符列表可以是下面所列出的任意一个：

1. sizes

为在映像文件中的每一个输入对象和库成员列出了代码和数据(这里的数据包括 RO 数据, RW 数据, ZI 数据和 Debug 数据)的大小;

2. totals

为输入对象文件和库, 列出代码和数据(这里的数据包括, RO 数据, RW 数据, ZI 数据和 Debug 数据) 总的大小;

3. veneers

给出由 armlink 生成的 veneers 的详细信息;

4. unused

列出由于使用 -remove 选项而从映像文件中被删除的所有未使用段。

注意: 在信息类型标识符列表之间不能存在空格, 比如可以输入

-info sizes,totals

但是不能是

-info sizes, totals(即在逗号和 totals 之间有空格是不允许的)

-map

这个选项创建映像文件的信息图。映像文件信息图包括映像文件中的每个加载域, 运行域和输入段的大小和地址, 这里的输入段还包括调试信息和链接器产生的输入段。

-symbols

这个选项列出了链接的时候使用的每一个局部和全局符号。该符号还包括链接生成的符号。

-symdefs file

这个选项创建一个包含来自输出映像文件的全局符号定义的符号定义文件。

默认的, 所有的全局符号都写入到符号定义文件中。如果文件 file 已经存在, 链接器将限制生成在已存在的 symdefs 文件中已列出的符号。

如果文件 file 没有指明路径信息, 链接器将在输出映像文件的路径搜索文件。如果文件没有找到, 就会在该目录下面创建文件。

在链接另一个映像文件的时候, 可以将符号定义文件作为链接的输入文件。

-edit file

这个选项指定一个 steering 类型的文件, 该文件包含用于修改输出文件中的符号信息表的命令。可以在 steering 文件中指定具有以下功能的命令:

隐藏全局符号。使用该选项可以在目标文件中隐藏指定的全局符号。

重命名全局符号。使用这个选项可以解决符号命名冲突的现象。

-xref

该选项列出了在输入段间的所有交叉引用。

-xreffrom object(section)

这个选项列出了从目标文件中的输入段对其他输入段的交叉引用。如果想知道某个指定的输入段中的引用情况, 就可以使用该选项。

-xrefto object(section)

该选项列出了从其他输入段到目标文件输入段的引用。

-errors file

使用该选项会将诊断信息从标准输出流重定向到文件 file 中。

-list file

该选项将 -info, -map, -symbols, -xref, -xreffrom 和 -xrefto 这几个选项的输出重新定向到文件 file 中。

如果文件 file 没有指定路径信息，就会在输出路径创建该文件，该路径是输出映像文件所在的路径。

-verbose

这个选项将有关链接操作的细节打印出来，包括所包括的目标文件和要用到的库。

-unmangled

该选项指定链接器在由 xref, -xreffrom, -xrefsto, 和-symbols 所生成的诊断信息中显示出 unmangled C++符号名。

如果使用了这个选项，链接器将 unmangle C++符号名以源码的形式显示出来。这个选项是默认的。

-mangled

这个选项指定链接器显示由-xref, -xreffrom, -xrefsto, 和-symbols 所产生的诊断信息中的 mangled C++符号名。如果使用了该选项，链接器就不会 unmangle C++符号名了。符号名是按照它们在目标符号表中显示的格式显示的。

-via file

该选项表示从文件 file 中读取输入文件名列表和链接器选项。

在 armlink 命令行可以输入多个-via 选项,当然 ,-via 选项也能够不含在一个 via 文件中。

-strict

这个选项告诉链接器报告可能导致错误而不是警告的条件。

-unresolved symbol

这个选项将未被解析的符号指向全局符号 symbol。Symbol 必须是已定义的全局符号，否则，symbol 会当作一个未解析的符号，链接将以失败告终。这个选项在自上而下的开发中尤为有用，在这种情况下，通过将无法指向相应函数的引用指向一个伪函数的方法，可以测试一个部分实现的系统。

该选项不会显示任何警告信息。

input-file-list

这是一个以空格作为间隔符的目标或库的列表。

有一类特殊的目标文件，即 symdef 文件，也可以包含在文件列表中，为生成的映像文件提供全局的 symbol 值。

在输入文件列表中有两种使用库的方法。

1. 指定要从库中提取并作为目标文件添加到映像文件中的特定的成员。
2. 指定某库文件，链接器根据需要从其中提取成员。

armlink 按照以下的顺序处理输入文件列表：

1. 无条件的添加目标文件
2. 使用匹配模式从库中选择成员加载到映像文件中去。例如使用下面的命令：

```
armlink main.o mylib(stdio.o) mylib(a*.o).
```

将会无条件的把 mylib 库中所有的以字母 a 开头的目标文件和 stdio.o 在链接的时候链接到生成的映像文件中去。

3. 添加为解析尚未解析的引用的库到库文件列表。

8.1.2 ARM 运行时库

本小节为读者介绍一下 ARM C/C++库方面的相关内容。

8.1.2.1 运行时库类型和建立选项

ADS 提供以下的运行时库来支持被编译的 C 和 C++代码：

ANSI C 库函数：

这个 C 函数库是由以下几部分组成：

1. 在 ISO C 标准中定义的函数；
2. 在 semihosted 环境下(semihosting 是针对 ARM 目标机的一种机制,它能够根据应用程序代码的输入/输出请求,与运行有调试功能的主机通讯。这种技术允许主机为通常没有输入和输出功能的目标硬件提供主机资源)用来实现 C 库函数的与目标相关的函数；
3. 被 C 和 C++编译器所调用的支持函数。

ARM C 库提供了额外的一些部件支持 C++，并为不同的结构体系和处理器编译代码。

C++库函数：

C++库函数包含由 ISO C++库标准定义的函数。C++库依赖于相应的 C 库实现与特定目标相关的部分,在 C++库的内部本身是不包含与目标相关的部分。这个库是由以下几部分组成的：

1. 版本为 2.01.01 的 Rogue Wave Standard C++库；
2. C++编译器使用的支持函数；
3. Rogue Wave 库所不支持的其他的 C++函数。

正如上面所说,ANSI C 库使用标准的 ARM semihosted 环境提供例如,文件输入/输出的功能。Semihosting 是由已定义的软件中断(Software Interrupt)操作来实现的。在大多数的情况下,semihosting SWI 是被库函数内部的代码所触发,用于调试的代理程序处理 SWI 异常。调试代理程序为主机提供所需要的通信。Semihosted 被 ARMulator ,Angel 和 Multi-ICE 所支持。用户可以使用在 ADS 软件中的 ARM 开发工具去开发用户应用程序,然后在 ARMulator 或在一个开发板上运行和调试该程序。

用户可以把 C 库中的与目标相关的函数作为自己应用程序中的一部分,重新进行代码的实现。这就为用户带来了极大的方便,用户可以根据自己的执行环境,适当的裁剪 C 库函数。

除此之外,用户还可以针对自己的应用程序的要求,对与目标无关的库函数进行适当的裁剪。

在 C 库中有很多函数是独立于其他函数的,并且与目标硬件没有任何依赖关系。对于这类函数,用户可以很容易地从汇编代码中使用它们。

在建立自己的用户应用程序的时候,用户必须指定一些最基本的操作选项。例如：

字节顺序,是大端模式(big endian:字数据的高字节存放在低地址,低字节存放在高地址),还是小端模式(little endian:字数据的高字节存放在高地址,低字节存放在低地址)；

浮点支持:可能是 FPA,VFP,软件浮点处理或不支持浮点运算；

堆栈限制:是否检查堆栈溢出；

位置无关(PID):数据是从与位置无关的代码还是从与位置相关的代码中读/写,代码是位置无关的只读代码还是位置相关的只读代码。

当用户对汇编程序,C 程序或 C++程序进行链接的时候,链接器会根据在建立时所指定的选项,选择适当的 C 或 C++运行时库的类型。选项各种不同组合都有一个相应的 ANSI C 库类型。

8.1.2.2 库路径结构

库路径是在 ADS 软件安装路径的 lib 目录下的两个子目录。假设,ADS 软件安装在 e:\arm\adsv1_2 目录,则在 e:\arm\adsv1_2\lib 目录下的两个子目录 armlib 和 cpplib 是 ARM 的库所在的路径。

armlib

这个子目录包含了 ARM C 库,浮点代数运算库,数学库等各类库函数。与这些库相应的头文件在 e:\arm\adsv1_2\include 目录中。

cpplib

这个子目录包含了 Rogue Wave C++库和 C++支持函数库。Rogue Wave C++库和 C++支持函数库合在一起被称为 ARM C++库。与这些库相应的头文件安装在 e:\arm\adsv1_2\include 目录下。

环境变量 ARMLIB 必须被设置成指向库路径。另外一种指定 ARM C 和 ARM C++库路径的方法是,在链接的时候使用操作选项-libpath directory(directory 代表库所在的路径),来指明要装载的库的路径。

无需对 armlib 和 cpplib 这两个库路径分开指明,链接器会自动从用户所指明的库路径中找出这两个子目录。

这里需要让读者特别注意的以下几点:

1. ARM C 库函数是以二进制格式提供的;
2. ARM 库函数禁止修改。如果读者想对库函数创建新的实现的话,可以把这个新的函数编译成目标文件,然后在链接的时候把它包含进来。这样在链接的时候,使用的是新的函数实现而不是原来的库函数。
3. 通常情况下,为了创建依赖于目标的应用程序,在 ANSI C 库中只有很少的几个函数需要实现重建。
4. Rogue Wave Standard C++函数库的源代码不是免费发布的,可以从 Rogue Wave Software Inc., 或 ARM 公司通过支付许可证费用来获得源文件。

8.1.3 GUI 开发环境(Code Warrior 和 AXD)

8.1.3.1 CodeWarrior 集成开发环境

CodeWarrior for ARM 是一套完整的集成开发工具,充分发挥了 ARM RISC 的优势,使产品开发人员能够很好的应用尖端的片上系统技术。该工具是专为基于 ARM RISC 的处理器而设计的,它可加速并简化嵌入式开发过程中的每一个环节,使得开发人员只需通过一个集成软件开发环境就能研制出 ARM 产品,在整个开发周期中,开发人员无需离开 CodeWarrior 开发环境,因此节省了在操做工具上花的时间,使得开发人员有更多的精力投入到代码编写上来,

CodeWarrior 集成开发环境(IDE)为管理和开发项目提供了简单多样化的图形用户界面。用户可以使用 ADS 的 CodeWarrior IDE 为 ARM 和 Thumb 处理器开发用 C, C++, 或 ARM 汇编语言的程序代码。通过提供下面的功能, CodeWarrior IDE 缩短了用户开发项目代码的周期。

1. 全面的项目管理功能;
2. 子函数的代码导航功能,使得用户迅速找到程序中的子函数。

可以在 CodeWarrior IDE 为 ARM 配置在 8.1.1 中介绍的各种命令工具,实现对工程代码的编译,汇编和链接。

在 CodeWarrior IDE 中所涉及到的 target 有两种不同的语义。

目标系统(Target system)

是特指代码要运行的环境,是基于 ARM 的硬件。比如,要为 ARM 开发板上编写要运行在它上面的程序,这个开发板就是目标系统。

生成目标(Build target)

是指用于生成特定的目标文件的选项设置(包括汇编选项,编译选项,链接选项以及链

接后的处理选项)和所用的文件的集合。

CodeWarrior IDE 能够让用户将源代码文件,库文件还有其他相关的文件以及配置设置等放在一个工程中。每个工程可以创建和管理生成目标设置的多个配置。例如,要编译一个包含调试信息的生成目标和一个基于 ARM7TDMI 的硬件优化生成目标,生成目标可以在同一个工程中共享文件,同时使用各自的设置。

CodeWarrior IDE 为用户提供下面的功能:

源代码编辑器,它集成在 CodeWarrior IDE 的浏览器中,能够根据语法格式,使用不同的颜色显示代码;

源代码浏览器,它保存了在源码中定义的所有符号,能够使用户在源码中快速方便的跳转;

查找和替换功能,用户可以在多个文件中,利用字符串通配符,进行字符串的搜索和替换;

文件比较功能,可以使用户比较路径中的不同文本文件的内容。

ADS 的 CodeWarrior IDE 是基于 Metrowerks CodeWarrior IDE 4.2 版本的。它经过适当的裁剪以支持 ADS 工具链。

针对 ARM 的配置面板为用户提供了在 CodeWarrior IDE 集成环境下配置各种 ARM 开发工具的能力,这样用户可以不用在命令控制台下就能够使用在 8.1.1 和将在 8.1.4 中介绍的各种命令。

以 ARM 为目标平台的工程创建向导,可以使用户以此为基础,快速创建 ARM 和 Thumb 工程。

尽管大多数的 ARM 工具链已经集成在 CodeWarrior IDE,但是仍有许多功能在该集成环境中没有实现,这些功能大多数是和调试相关的,因为 ARM 的调试器没有集成到 CodeWarrior IDE 中。

由于 ARM 调试器(AXD)没有集成在 CodeWarrior IDE 中,这就意味着,用户不能在 CodeWarrior IDE 中进行断点调试和查看变量。

对于熟悉 CodeWarrior IDE 的用户会发现,有许多的功能已经从 CodeWarrior IDE For ARM 中移走,比如快速应用程序开发模板等。

在 CodeWarrior IDE For ARM 中有很多的菜单或子菜单是不能使用的。下面介绍一下这些不能使用的选项。

1. View 菜单下不能使用的菜单选项有:

Processes, Expressions, Global Variable, Breakpoints, Registers。

2. Project 菜单不能使用的菜单选项:

Precompile 子菜单。因为 ARM 编译器不支持预编译的头文件。

3. Debug 菜单

该菜单中没有一个子菜单是可以使用的。

4. Browser 菜单中不能使用的菜单选项:

New Property, New Method 和 New Event Set。

5. Help menu 中不能用于 ADS 的菜单选项有:

CodeWarrior Help, Index, Search 和 Online Manuals。

有关 CodeWarrior IDE 中一些常用菜单的使用,将在后面的举例中具体说明的,在此,不在赘述。

8.1.3.2 ADS 调试器

调试器本身是一个软件,用户通过这个软件使用 debug agent 可以对包含有调试信息的,正在运行的可执行代码进行比如变量的查看,断点的控制等调试操作。

ADS 中包含有 3 个调试器：

AXD(ARM eXtended Debugger)：ARM 扩展调试器；

armsd(ARM Symbolic Debugger)：ARM 符号调试器；

与老版本兼容的 Windows 或 Unix 下的 ARM 调试工具 ADW/ADU(Application Debugger Windows/Unix)。

下面对在调试映像文件中所涉及到的一些术语做一个简单的介绍。

Debug target

在软件开发的最初阶段，可能还没有具体的硬件设备。如果要测试所开发的软件是否达到了预期的效果，这可以由软件仿真来完成。即使调试器和要测试的软件运行在同一台 PC 上，也可以把目标当作一个独立的硬件来看待。

当然，也可以搭建一个 PCB 板，这个板上可以包含一个或多个处理器，在这个板上可以运行和调试应用软件。

只有当通过硬件或者是软件仿真所得到的结果达到了预期的效果，才算是完成了应用程序的编写工作。

调试器能够发送以下指令：

1. 装载映像文件到目标内存；
2. 启动或停止程序的执行；
3. 显示内存，寄存器或变量的值；
4. 允许用户改变存储的变量值。

Debug agent

Debug agent 执行调试器发出的命令动作，比如：设置断点，从存储器中读数据，把数据写到存储器等。

Debug agent 既不是被调试的程序，也不是调试器。在 ARM 体系中，它有这几种方式：Multi-ICE(Multi-processor in-circuit emulator)，ARMulator 和 Angel。其中 Multi-ICE 是一个独立的产品，是 ARM 公司自己的 JTAG 在线仿真器，不是由 ADS 提供的。

AXD 可以在 Windows 和 UNIX 下，进行程序的调试。它为用 C，C++，和汇编语言编写的源代码提供了一个全面的 Windows 和 UNIX 环境。

在后面的章节中，会结合具体实例为读者介绍如何使用 AXD 调试器。

8.1.4 实用程序

ADS 提供以下的实用工具来配合前面介绍的命令行开发工具的使用

fromELF

这是 ARM 映像文件转换工具。该命令将 ELF 格式的文件作为输入文件，将该格式转换为各种输出格式的文件，包括 plain binary(BIN 格式映像文件)，Motorola 32-bit S-record format(Motorola 32 位 S 格式映像文件)，Intel Hex 32 format(Intel 32 位格式映像文件)，和 Verilog-like hex format(Verilog 16 进制文件)。FromELF 命令也能够为输入映像文件产生文本信息，例如代码和数据长度。

armar

ARM 库函数生成器将一系列 ELF 格式的目标文件以库函数的形式集合在一起，用户可以把一个库传递给一个链接器以代替几个 ELF 文件。

Flash downloader

用于把二进制映像文件下载到 ARM 开发板上的 Flash 存储器的工具

8.1.5 支持的软件

ADS 为用户提供下面的软件，使用户可以在软件仿真的环境下或者在基于 ARM 的硬件环境调试用户应用程序。

ARMLulator

这是一个 ARM 指令集仿真器，集成在 ARM 的调试器 AXD 中，它提供对 ARM 处理器的指令集的仿真，为 ARM 和 Thumb 提供精确的模拟。用户可以在硬件尚未做好的情况下，开发程序代码。

8.2 使用 ADS 创建工程

本节通过一个具体实例，为读者介绍如何使用该集成开发环境，利用 CodeWarrior 提供的建立工程的模板建立自己的工程，并学会如何进行编译链接，生成包含调试信息的映像文件和可以直接烧写的 Flash 中的 .bin 格式的二进制可执行文件。

8.2.1 建立一个工程

工程将所有的源码文件组织在一起，并能够决定最终生成文件存放的路径，输出的格式等。

在 CodeWarrior 中新建一个工程的方法有两种，可以在工具栏中单击“New”按钮，也可以在“File”菜单中选择“New...”菜单。这样就会打开一个如图 8.1 所示的对话框。

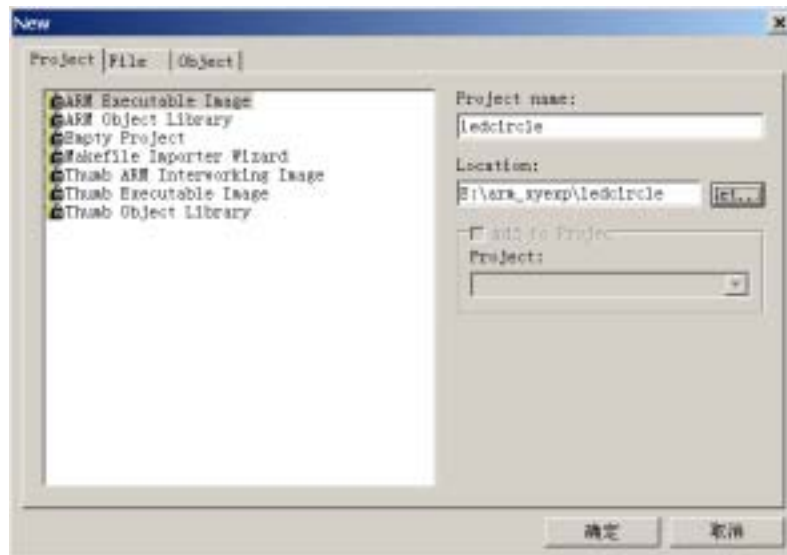


图 8.1 新建工程对话框

在这个对话框中为用户提供了 7 种可选择的工程类型。

ARM Executable Image :用于由 ARM 指令的代码生成一个 ELF 格式的可执行映像文件；

ARM Object Library :用于由 ARM 指令的代码生成一个 armar 格式的目标文件库；

Empty Project :用于创建一个不包含任何库或源文件的工程；

Makefile Importer Wizard :用于将 Visual C 的 nmake 或 GNU make 文件转入到 CodeWarrior IDE 工程文件；

Thumb ARM Executable Image :用于由 ARM 指令和 Thumb 指令的混和代码生成一个可执行的 ELF 格式的映像文件；

Thumb Executable image :用于由 Thumb 指令创建一个可执行的 ELF 格式的映像文件；

Thumb Object Library :用于由 Thumb 指令的代码生成一个 armar 格式的目标文件库。

在这里选择 ARM Executable Image，在“Project name:”中输入工程文件名，本例为“ledcircle”，点击“Location:”文本框的“Set...”按钮，浏览选择想要将该工程保存的路径，将这些设置好后，点击“确定”，即可建立一个新的名为 ledcircle 的工程。

这个时候会出现 ledcircle.mcp 的窗口，如图 8.2 所示，有三个标签页，分别为 files,link order,target 默认的是显示第一个标签页 files。通过在该标签页点击鼠标右键，选中“Add Files...”可以把要用到的源程序添加到工程中。

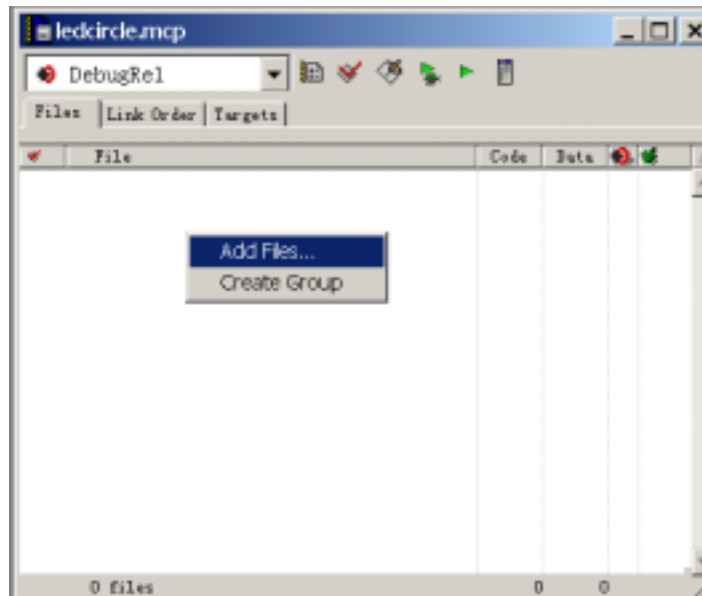


图 8.2 新建工程打开窗口

对于本例，由于所有的源文件都还没有建立，所以首先需要新建源文件。

在“File”菜单中选择“New”，在打开的如图 8.1 所示的对话框中，选择标签页 File，在 File name 中输入要创建的文件名，输入“Init.s”，点击“确定”关闭窗口。

在打开的文件编辑框中输入下面的汇编代码：

```

;*****
;Chinese Academy of Sciences, Institute of Automation
;File Name:   Init.s
;Description:
;Author:     JuGuang.Li
;Date:
;*****

IMPORT      Main
AREA       Init, CODE, READONLY
ENTRY
LDR R0,    =0x3FF0000
LDR R1,    =0xE7FFFF80 ;配置 SYSCFG,片内 4K Cache,4K SRAM
STR       R1, [R0]
LDR SP,    =0x3FE1000 ;SP 指向 4K SRAM 的尾地址,堆栈向下生成
BL        Main
  
```



```
B
END
```

在这段代码中，伪操作 IMPORT 告诉编译器符号 Main 不是在该文件中定义的，而是在其他源文件中定义的符号，但是本源文件中可能要用到该符号。接下来用伪指令 AREA 定义段名为 Init 的段为只读的代码段，伪指令 ENTRY 指出了程序的入口点。下面就是用汇编指令实现了配置 SYSCFG 特殊功能寄存器，将 S3C4510B 片内的 8K 一体化的 SRAM 配置为 4K Cache，4K SRAM，并将用户堆栈设置在片内的 SRAM 中。

4K SRAM 的地址为 0x3FE,0000 ~ (0x3FE,1000-1)，由于 S3C4510B 的堆栈由高地址向低地址生成，将 SP 初始化为 0x3FE,1000。

完成上述操作后，程序跳转到 Main 函数执行。

保存 Init.s 汇编程序。

用同样的方法，再建立一个名为 main.c 的 C 源代码文件。具体代码如下：

```

/*****
//Chinese Academy of Sciences, Institute of Automation
//File Name:   main.c
//Description:
//Author:     JuGuang.Li
//Date:
/*****
#define IOPMOD (*(volatile unsigned *)0x03FF5000) //IO port mode register
#define IOPDATA (*(volatile unsigned *)0x03FF5008) //IO port data register
void Delay(unsigned int);
int Main()
{
    unsigned long LED;
    IOPMOD=0xFFFFFFFF; //将 IO 口置为输出模式
    IOPDATA=0x01;
    for(;;){
        LED=IOPDATA;
        LED=(LED<<1);
        IOPDATA=LED;
        Delay(10);
        if(!(IOPDATA&0x0F))
            IOPDATA=0x01;
    }
    return(0);
}
void Delay(unsigned int x)
{
    unsigned int i,j,k;
    for(i=0;i<=x;i++)
        for(j=0;j<0xff;j++)
            for(k=0;k<0xff;k++);
}

```

该段代码首先将 I/O 模式寄存器设置为输出模式，为 I/O 数据寄存器赋初值为 0x1，通过将 I/O 数据寄存器的数值进行周期性的左移，实现使接在 P0~P3 口的 LED 显示器轮流被点亮的功能。（注意这里的 if 语句，是为了保证当 I/O 数据寄存器中的数在移位过程中，第 4 位为数字“1”时，使数字 1 通过和 0xFF 相与，又重新回到 I/O 数据寄存器的第 0 位，从而

保证了数字 1 一直在 I/O 数据寄存器的低四位之间移位。)

在这里还有一个细节,希望读者注意。在建立好一个工程时,默认的 target 是 DebugRel, 还有另外两个可用的 target, 分别为 Realse 和 Debug, 这三个 target 的含义分别为:

DebugRel: 使用该目标,在生成目标的时候,会为每一个源文件生成调试信息;

Debug: 使用该目标为每一个源文件生成最完全的调试信息;

Release: 使用该目标不会生成任何调试信息。

在本例中,使用默认的 DebugRel 目标。

现在已经新建了两个源文件,要把这两个源文件添加到工程中去。

为工程添加源码常用的方法有两种,既可以使用入图 8.2 所示方法,也可以在“Project”菜单项中,选择“Add Files...”,这两种方法都会打开文件浏览框,用户可以把已经存在的文件添加到工程中来。当选中要添加的文件时,会出现一个对话框,如图 8.3 所示,询问用户把文件添加到何类目标中,在这里,我们选择 DebugRel 目标。把刚才创建的两个文件添加到工程中来。

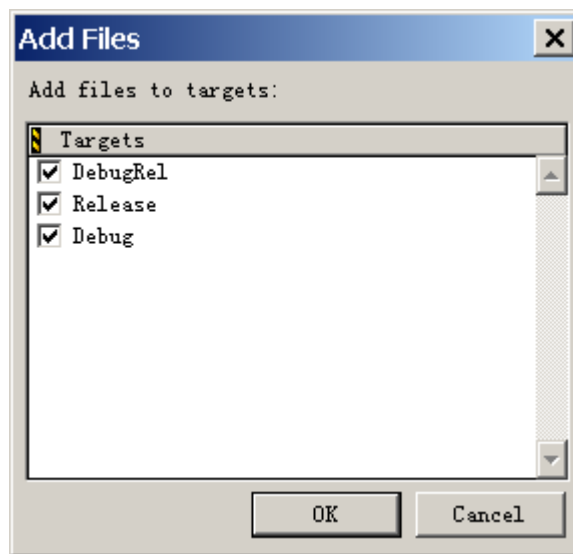


图 8.3 选择添加文件到指定目标

到目前为止,一个完整的工程已经建立。

下面该对工程进行编译和链接工作。

8.2.2 编译和链接工程

在进行编译和链接前,首先讲述一下如何进行生成目标的配置。

点击 Edit 菜单,选择“DebugRel Settings...”(注意,这个选项会因用户选择的不同目标而有所不同),出现如图 8.2 所示的对话框。

这个对话框中的设置很多,在这里指介绍一些最为常用的设置选项,读者若对其他未涉及到的选项感兴趣,可以查看相应的帮助文件。

1. target 设置选项

Target Name 文本框显示了当前的目标设置。

Linker 选项供用户选择要使用的链接器。在这里默认选择的是 ARM Linker,使用该链接器,将使用 armlink 链接编译器和汇编器生成的工程中的文件相应的目标文件。

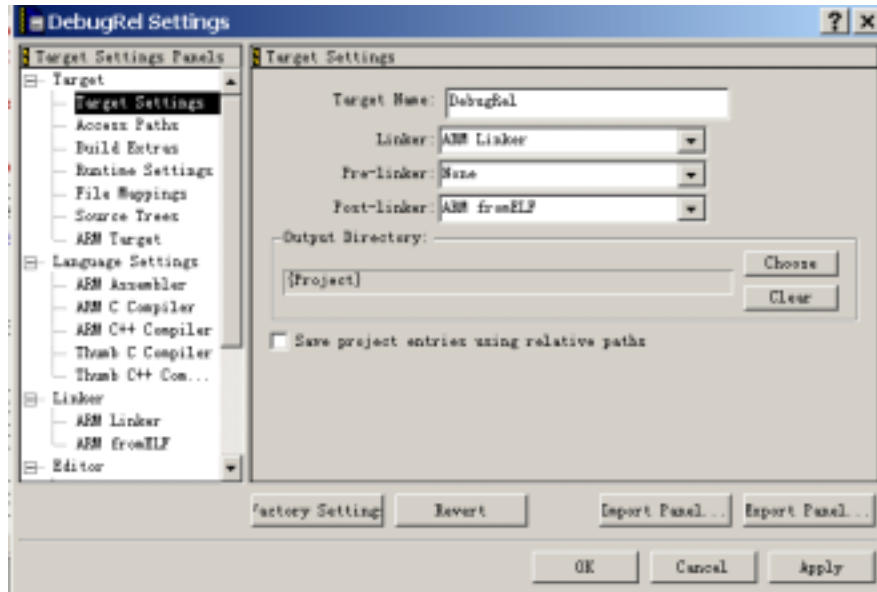


图 8.4 DebugRel 设置对话框

这个设置中还有两个可选项，None 不是不用任何链接器，如果使用它，则工程中的所有文件都不会被编译器或汇编器处理。ARM Librarian 表示将编译或汇编得到的目标文件转换为 ARM 库文件。对于本例，使用默认的链接器 ARM Linker。

Pre-linker：目前 CodeWarrior IDE 不支持该选项。

Post-Linker：选择在链接完成后，还要对输出文件进行的操作。因为在本例中，希望生成一个可以烧写到 Flash 中去的二进制代码，所以在这里选择 ARM fromELF，表示在链接生成映像文件后，再调用 FromELF 命令将含有调试信息的 ELF 格式的映像文件转换成其他格式的文件。

2. Language Settings

因为本例中包含有汇编源代码，所以要用到汇编器。首先看 ARM 汇编器。这个汇编器实际就在 8.1 节中谈到的 armasm，默认的 ARM 体系结构是 ARM7TDMI，正好符合目标板 S3C4510B，无需改动。字节顺序默认就是小端模式。其他设置，就用默认值即可。

还有一个需要注意的就是 ARM C 编译器，它实际就是调用的命令行工具 armcc。使用默认的设置就可以了。

细心的读者可能会注意到，在设置框的右下脚，当对某项设置进行了修改，该行中的某个选项就会发生相应的改动，如图 8.5 所示。实际上，这行文字就显示的是在 8.1 中介绍的相应的编译或链接选项，由于有了 CodeWarrior，开发人员可以不用再去查看繁多的命令行选项，只要在界面中选中或撤消某个选项，软件就会自动生成相应的代码，为不习惯在 DOS 下键入命令行的用户提供了极大的方便。

3. Linker 设置

鼠标选中 ARM Linker，出现如图 8.6 所示对话框。这里详细介绍该对话框的主要的标签页选项，因为这些选项对最终生成的文件有着直接的影响。

在标签页 Output 中，Linktype 中提供了三种链接方式。Partial 方式表示链接器只进行部分链接，经过部分链接生成的目标文件，可以作为以后进一步链接时的输入文件。Simple 方式是默认的链接方式，也是最为频繁使用的链接方式，它链接生成简单的 ELF 格式的目标文件，使用的是链接器选项中指定的地址映射方式。Scattered 方式使得链接器要根据 scatter 格式文件中指定的地址映射，生成复杂的 ELF 格式的映像文件。这个选项一般情况

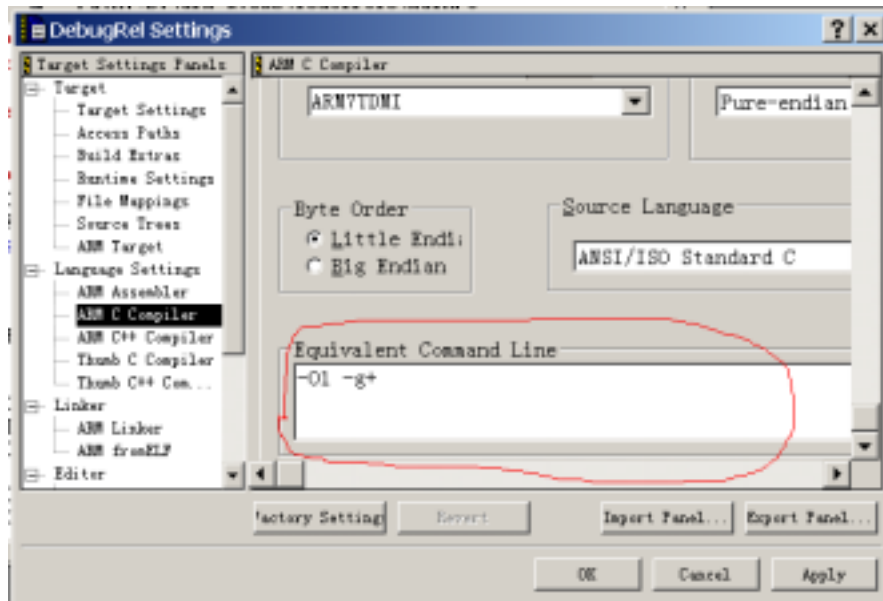


图 8.5 命令行工具选项设置

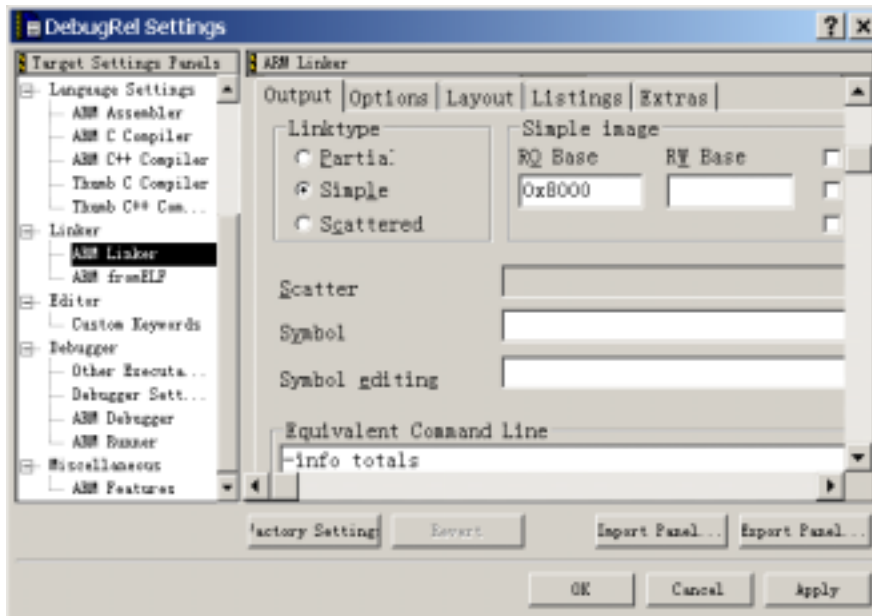


图 8.6 链接器设置

下，使用不太多。

因为我们所举的例子比较简单，选择 Simple 方式就可以了。

在选中 Simple 方式后，就会出现 Simple image。

RO Base：这个文本框设置包含有 RO 段的加载域和运行域为同一个地址。默认是 0x8000。这里用户要根据自己硬件的实际 SDRAM 的地址空间来修改这个地址，保证在这里填写的地址，是程序运行时，SDRAM 地址空间所能覆盖的地址。针对本书所介绍的目标板，就可以使用这个默认地址值。

RW Base：这个文本框设置了包含 RW 和 ZI 输出段的运行域地址。如果选中 split 选项，链接器生成的映像文件将包含两个加载域和两个运行域，此时，在 RW Base 中所输入的地址为包含 RW 和 ZI 输出段的域设置了加载域和运行域地址

Ropi：选中这个设置将告诉链接器使包含有 RO 输出段的运行域位置无关。使用这个选

项，链接器将保证下面的操作：

检查各段之间的重定址是否有效；

确保任何由 armlink 自身生成的代码是只读位置无关的。

Rwpi：选中该选项将会告诉链接器使包含 RW 和 ZI 输出段的运行域位置无关。如果这个选项没有被选中，域就标识为绝对。每一个可写的输入段必须是读写位置无关的。如果这个选项被选中，链接器将进行下面的操作，

检查可读/可写属性的运行域的输入段是否设置了位置无关属性；

检查在各段之间的重地址是否有效；

在 Region\$\$Table 和 ZISection\$\$Table 中添加基于静态存储器 sb 的选项。

该选项要求 RW Base 有值，如果没有给它指定数值的话，默认为 0 值。

Split Image：选择这个选项把包含 RO 和 RW 的输出段的加载域分成 2 个加载域：一个是包含 RO 输出段的域，一个是包含 RW 输出段的域。

这个选项要求 RW Base 有值，如果没有给 RW Base 选项设置，则默认是 -RW Base 0。

Relocatable：选择这个选项保留了映像文件的重定址偏移量。这些偏移量为程序加载器提供了有用信息。

在 Options 选项中，需要读者引起注意的是 Image entry point 文本框。它指定映像文件的初始入口点地址值，当映像文件被加载程序加载时，加载程序会跳转到该地址处执行。如果需要，用户可以在这个文本框中输入下面格式的入口点：

入口点地址：这是一个数值，例如 -entry 0x0

符号：该选项指定映像文件的入口点为该符号所代表的地址处，比如：

-entry int_handler

如果该符号有多处定义存在，armlink 将产生出错信息。

offset+object(section)：该选项指定在某个目标文件的段的内部的某个偏移量处为映像文件的入口地址，例如：

-entry 8+startup(startupseg)

在此处指定的入口点用于设置 ELF 映像文件的入口地址。

需要引起注意的是，这里不可以用符号 main 作为入口点地址符号，否则将会出现类似“Image dose not have an entry point(Not specified or not set due to multiple choice)”的错误信息。

关于 ARM Linker 的设置还有很多，对于想进一步深入了解的读者，可以查看帮助文件，都有很详细的介绍。

在 Linker 下还有一个 ARM fromELF，如图 8.7 所示：

fromELF 就是在 8.1 节中介绍的一个实用工具，它实现将链接器，编译器或汇编器的输出代码进行格式转换的功能。例如，将 ELF 格式的可执行映像文件转换成可以烧写到 ROM 的二进制格式文件；对输出文件进行反汇编，从而提取出有关目标文件的大小，符号和字符串表以及重定址等信息。

只有在 Target 设置中选择了 Post-linker，才可以使用该选项。

在 Output format 下拉框中，为用户提供了多种可以转换的目标格式，本例选择 Plain binary，这是一个二进制格式的可执行文件，可以被烧些的目标板的 Flash 中。

在 Output file name 文本域输入期望生成的输出文件存放的路径，或通过点击 Choose... 按钮从文件对话框中选择输出文件。如果在这个文本域不输入路径名，则生成的二进制文件存放在工程所在的目录下。

进行好这些相关的设置后，以后在对工程进行 make 的时候，CodeWarrior IDE 就会在链接完成后调用 fromELF 来处理生成的映像文件。

对于本例的工程而言，到此，就完成了 make 之前的设置工作了。

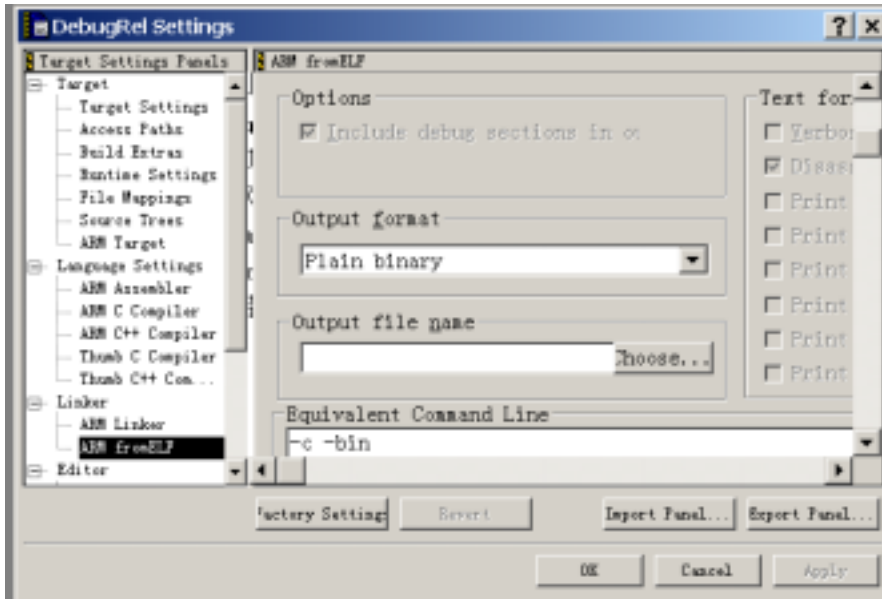


图 8.7 ARM fromELF 可选项

点击 CodeWarrior IDE 的菜单 Project 下的 make 菜单 就可以对工程进行编译和链接了。整个编译链接过程如图 8.8 所示：

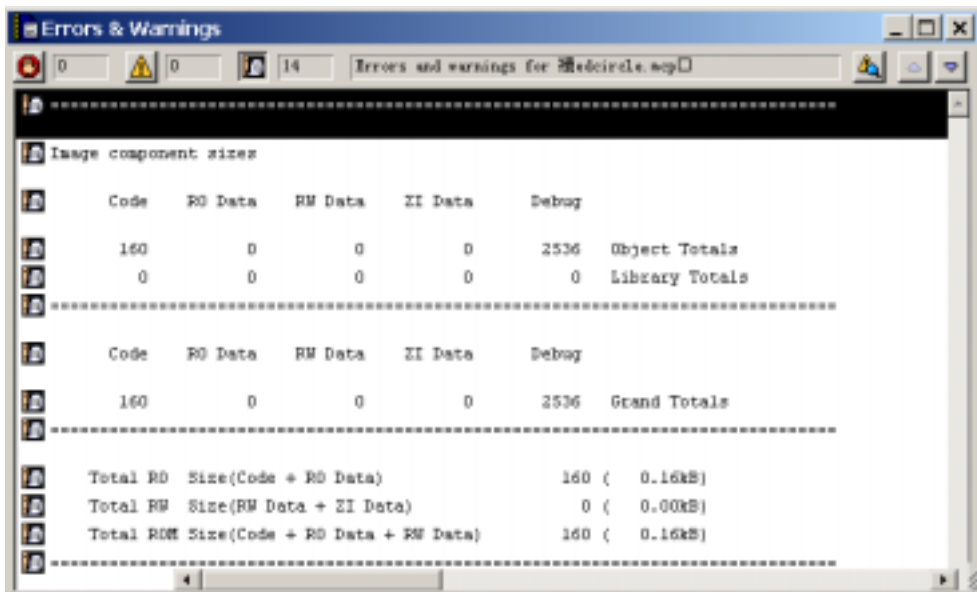


图 8.8 编译和链接过程

在工程 ledcircle 所在的目录下，会生成一个名为：工程名_data 目录，在本例中就是 ledcircle_data 目录，在这个目录下不同类别的目标对应不同的目录。在本例中由于我们使用的是 DebugRe 目标，所以生成的最终文件都应该在该目录下。进入到 DebugRel 目录中去，读者会看到 make 后生成的映像文件和二进制文件，映像文件用于调试，二进制文件可以烧写到 S3C4510B 的 Flash 中运行。

8.2.3 使用命令行工具编译应用程序

如果用户开发的工程比较简单，或者只是想用到 ADS 提供的各种工具，但是并不想在

CodeWarrior IDE 中进行开发。在这种情况下，再为读者介绍一种不在 CodeWarrior IDE 集成开发环境下，开发用户应用程序的方法，当然前提是用户必须安装了 ADS 软件，因为在编译链接的过程中要用到 ADS 提供的各种命令工具。

这种方法对于开发包含较少源代码的工程是比较实用的。

首先用户可以用任何编辑软件(比如 UltraEdit)编写 8.2.1 中所提到的两个源文件 Init.s 和 main.c。接下来，可以利用在第 7 章中介绍的 makefile 的知识，编写自己的 makefile 文件。对于本例，编写的 makefile 文件(假设该 makefile 文件保存为 ads_mk.mk)如下：

```
PAT = e:/arm/adsv1_2/bin
CC = $(PAT)/armcc
LD = $(PAT)/armlink
OBJTOOL = $(PAT)/fromelf

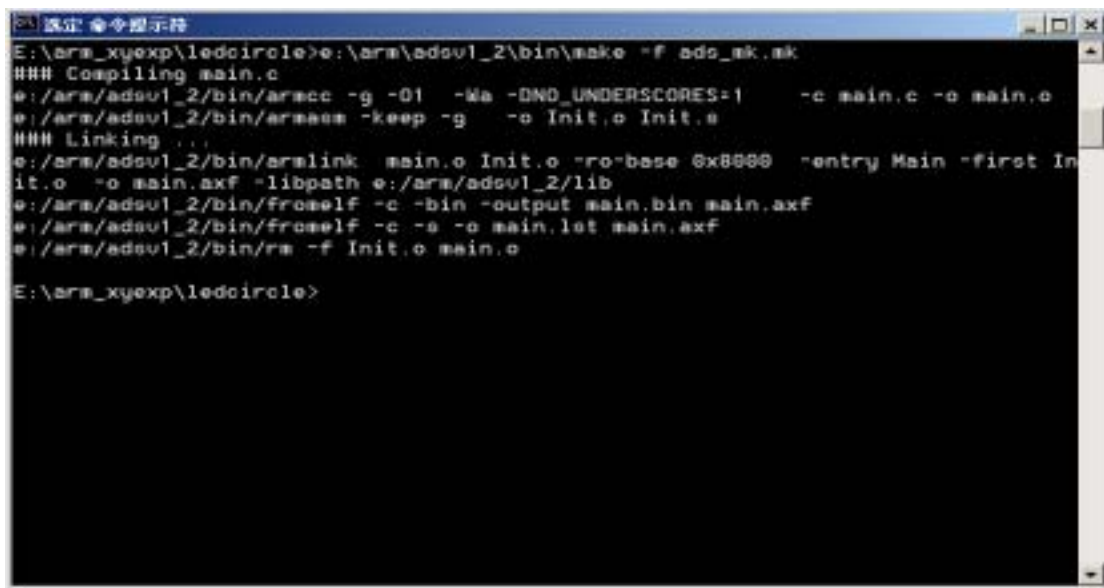
RM = $(PAT)/rm -f
AS = $(PAT)/armasm -keep -g
ASFILE = e:/arm_xyexp/Init.s
CFLAGS = -g -O1 -Wa -DNO_UNDERSCORES=1
MODEL = main
SRC = $(MODEL).c
OBJS = $(MODEL).o
all: $(MODEL).axf $(MODEL).bin clean

%.axf:$(OBJS) Init.o
    @echo "### Linking ..."
    $(LD) $(OBJS) Init.o -ro-base 0x8000 -entry Main -first Init.o -o $@
-libpath e:/arm/adsv1_2/lib
%.bin: %.axf
    $(OBJTOOL) -c -bin -output $@ $<
    $(OBJTOOL) -c -s -o $(<:.axf=.lst) $<

%.o:%.c
    @echo "### Compiling $<"
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    $(RM) Init.o $(OBJS)
```

由于 ADS 在安装的时候没有提供 make 命令，可以将在第 7 章中用到的 make 命令直接拷贝到 ADS 安装路径的 bin 目录下，比如 ADS 安装在目录 e:\arm\adsv1_2 下，可以将 make 命令拷贝到 e:\arm\adsv1_2\bin 目录下，在 command console 下的编译过程如图 8.9 所示：



```
E:\arm_xyexp\ledcircle>e:\arm\adsv1_2\bin\make -f ads_mk.mk
### Compiling main.c
e:/arm/adsv1_2/bin/armcc -g -O1 -Wa -DNO_UNDERSCORES=1 -c main.c -o main.o
e:/arm/adsv1_2/bin/armasm -keep -g -o Init.o Init.a
### Linking ...
e:/arm/adsv1_2/bin/armlink main.o Init.o -ro-base 0x8000 -entry Main -first In
it.o -o main.axf -libpath e:/arm/adsv1_2/lib
e:/arm/adsv1_2/bin/froelf -c -bin -output main.bin main.axf
e:/arm/adsv1_2/bin/froelf -c -a -o main.lst main.axf
e:/arm/adsv1_2/bin/ra -f Init.o main.o

E:\arm_xyexp\ledcircle>
```

图 8.9 在 command console 下编译过程

经过上述编译链接,以及链接后的操作,在 `e:\arm_xyexp\ledcircle` 目录下会生成两个新的文件, `main.axf` 和 `main.bin`。

用这种方式生成的文件与在 CodeWarrior IDE 界面通过各个选项的设置,生成的文件是一样的。

在所举的例子中,都生成了包含有调试信息的可执行映像文件,即以 `.axf` 结尾的文件。下面一节将介绍如何用 AXD 对程序进行源码级的调试。

8.3 用 AXD 进行代码调试

AXD(ARM eXtended Debugger)是 ADS 软件中独立于 CodeWarrior IDE 的图形软件,打开 AXD 软件,默认是打开的目标是 ARMulator。这个也是调试的时候最常用的一种调试工具,本节主要是结合 ARMulator 介绍在 AXD 中进行代码调试的方法和过程,使读者对 AXD 的调试有初步的了解。

要使用 AXD 必须首先要生成包含有调试信息的程序,在 8.2 节中,已经生成的 `ledcircle.axf` 或 `main.axf` 就是含有调试信息的可执行 ELF 格式的映像文件。

1. 在 AXD 中打开调试文件

在菜单 File 中选择“Load image...”选项,打开 Load Image 对话框,找到要装载的 `.axf` 映像文件,点击“打开”按钮,就把映像文件装载到目标内存中了。

在所打开的映像文件中会有一个蓝色的箭头指示当前执行的位置。对于本例,打开映像文件后,如图 8.10 所示:

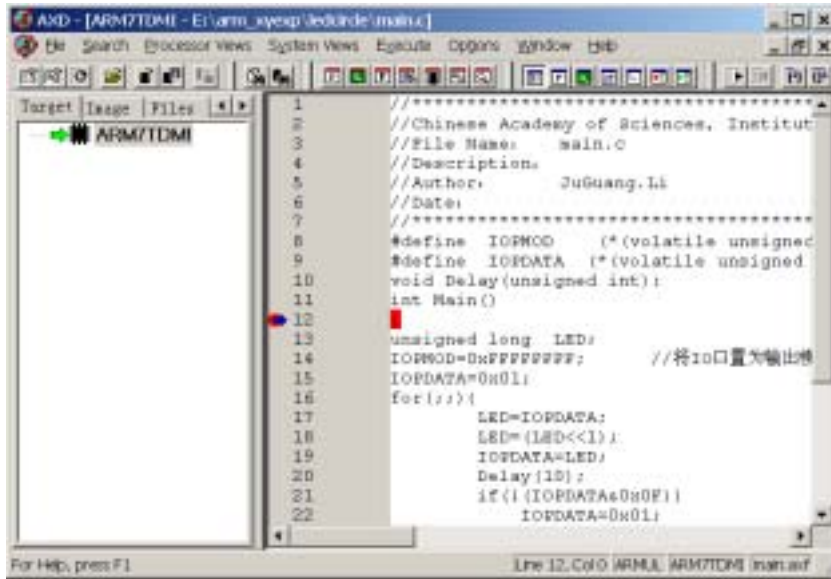


图 8.10 在 axd 下打开映像文件

在菜单 Execute 中选择“Go”，将全速运行代码。要想进行单步的代码调试，在 Execute 菜单中选择“Step”选项，或用 F10 即可以单步执行代码，窗口中蓝色箭头会发生相应的移动。

有时候，用户可能希望程序在执行到某处时，查看一些所关心的变量值，此时可以通过断点设置达到此要求。将光标移动到要进行断点设置的代码处，在 Execute 菜单中，选择“Toggle Breakpoint”或按 F9，就会在光标所在位置出现一个实心圆点，表明该处为断点。

还可以在 AXD 中查看寄存器值，变量值，某个内存单元的数值等等。

下面就结合本章中的例子，介绍在 AXD 中调试过程。

2. 查看存储器内容

在程序运行前，可以先查看两个宏变量 IOPMOD 和 IOPDATA 的当前值。方法是：从 Processor Views 菜单中选择“Memory”选项，如图 8.11 所示。

ARM7TDMI - Memory Start address 0x3ff5000																	
Tab1 - Hex - No prefix				Tab2 - Hex - No prefix				Tab3 - Hex - No prefix				Tab4 - Hex - No prefix					
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
0x03FF5000	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8	.
0x03FF5010	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8	.
0x03FF5020	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8	.
0x03FF5030	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8	.
0x03FF5040	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8	.

图 8.11 查看存储器内容

在 Memory Start address 选择框中，用户可以根据要查看的存储器的地址输入起始地址，在下面的表格中会列出连续的 64 个地址。因为 I/O 模式控制寄存器和 I/O 数据控制寄存器都是 32 位的控制寄存器，所以从 0x3ff5000 开始的连续四个地址空间存放的是 I/O 模式控制寄存器的值，从图 8.11 可以读出该控制寄存器的值开始为 0xE7FF0010，I/O 数据控制寄存器的内容是从地址 0x3FF5008 开始的连续四个地址空间存放的内容。从图 8.11 中可以看出 IODATA 中的初始值为 0x E7FF0010，注意因为用的是小端模式，所以读数据的时候注意高

地址中存放的是高字节，低地址存放的是低字节。

现在对程序进行单步调试，当程序运行到 for 循环处时，可以再一次查看这两个寄存器中的内容，此时存储器的内容如图 8.12 所示：

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x03FF5000	FF	FF	FF	FF	00	E8	00	E8	01	00	00	00	00	E8	00	E8
0x03FF5010	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5020	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5030	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5040	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5050	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5060	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x03FF5070	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8

图 8.12 单步运行后的存储器内容

从图中可以看出运行完两个赋值语句后，两个寄存器的内容的确发生了变化，在地址 0x3FF5000 作为起始地址的连续四个存储单元中，可以读出 I/O 模式控制寄存器的内容为 0xFFFFFFFF，在地址 0x3FF5008 开始的连续的四个存储单元中，可以读出 I/O 数据控制寄存器的内容为 0x00000001。

3. 设置断点

可以在 for 循环体的“Delay(10);”语句处设置断点，将光标定位在该语句处，使用快捷键 F9 在此处设置断点，按 F5 键，程序将运行到断点处，如果读者想查看子函数 Delay 是如何运行的，可以在 Execute 菜单中选择“Step In”选项，或按下 F8 键，进入到子函数内部进行单步程序的调试。如图 8.13 所示。

4. 查看变量值

在 Delay 函数的内部，如果用户希望查看某个变量的值，比如查看变量 i 的值，可以在 Processor Views 菜单中选择“Watch”，会出现如图 8.14 所示的 watch 窗口，然后用鼠标选中变量 i，点击鼠标右键，在快捷菜单中选中“Add to watch”，如图 8.14 所示，这样变量 i 默认是添加到 watch 窗口的 Tab1 中。程序运行过程中，用户可以看到变量 i 的值在不断的



图 8.13 设置断点

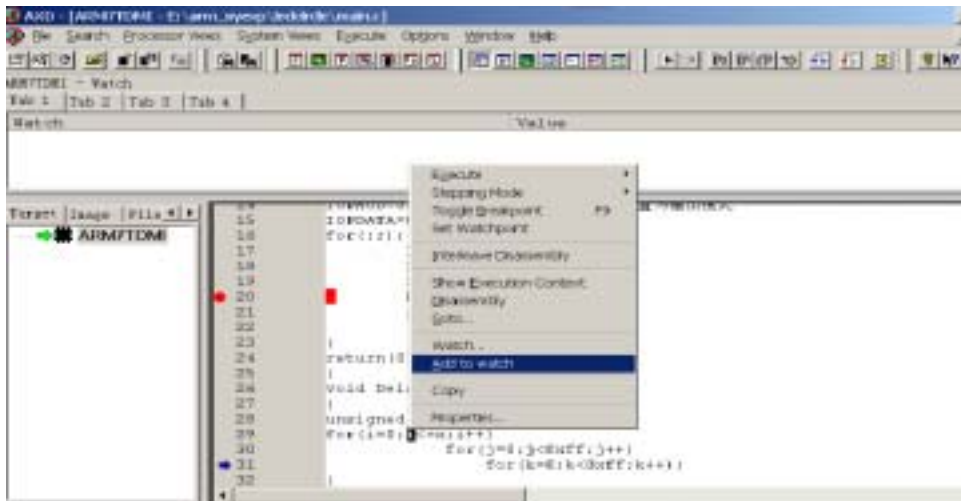


图 8.14 查看变量

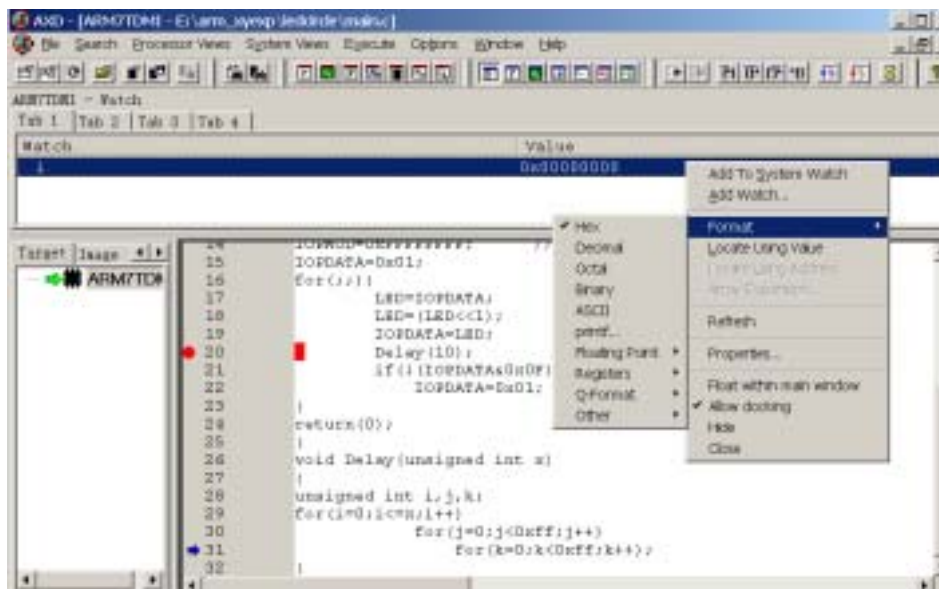


图 8.15 改变变量的格式

变化。默认显示变量数值是以十六进制格式显示的，如果用户对这种显示格式不习惯的话，可以通过在 watch 窗口点击鼠标右键，在弹出的快捷菜单中选择“Format”选项，如图 8.15 所示，用户可以选择所查看的变量显示数据的格式。如果用户想从 Delay 函数中跳出到主函数中去，最简单的方法就是将光标定位到你想要跳转到的主函数处，在 Execute 菜单中选择“Run to Cursor”选项，则程序会从 Delay 函数中跳转到光标所在位置。

8.4 本章小结

本章主要介绍了 ADS 软件。首先介绍了 ADS 软件的基本组成部分，接着重点地介绍了最常用的 2 个命令工具 armcc 和 armlink 的使用语法和各个操作选项。然后结合一个具体的应用实例，介绍如何在 CodeWarrior IDE 环境下建立自己的新工程，编译和链接工程生成可以调试的映像文件和二进制文件的过程，同时补充了一种不在 CodeWarrior IDE 集成开发环

境下，利用第 7 章介绍的有关 make 的知识，利用 ADS 提供的编译和链接等命令工具编写自己的 makefile 文件，来开发和编译程序的方法。最后介绍了如何使用 ADS 的调试软件 AXD 调试应用程序。