

Hello World!入门范例

By www.mcuzone.com

下面以一个通过 44B0 的 UART0 输出字符串 “Hello World!” 的小程序为例简单讲解一下 ADS 的初步使用。本范例的阅读对象是初学者，并假设初学者已经安装 ADS1.2（ARM Developer Suite）软件。

首先通过“开始”->“程序”->“ARM Developer Suite V1.2”->“Codewarrior for ARM Developer Suite”打开 Codewarrior，Codewarrior 是 Metrowerks 公司为 ARM 公司所开发的 IDE。

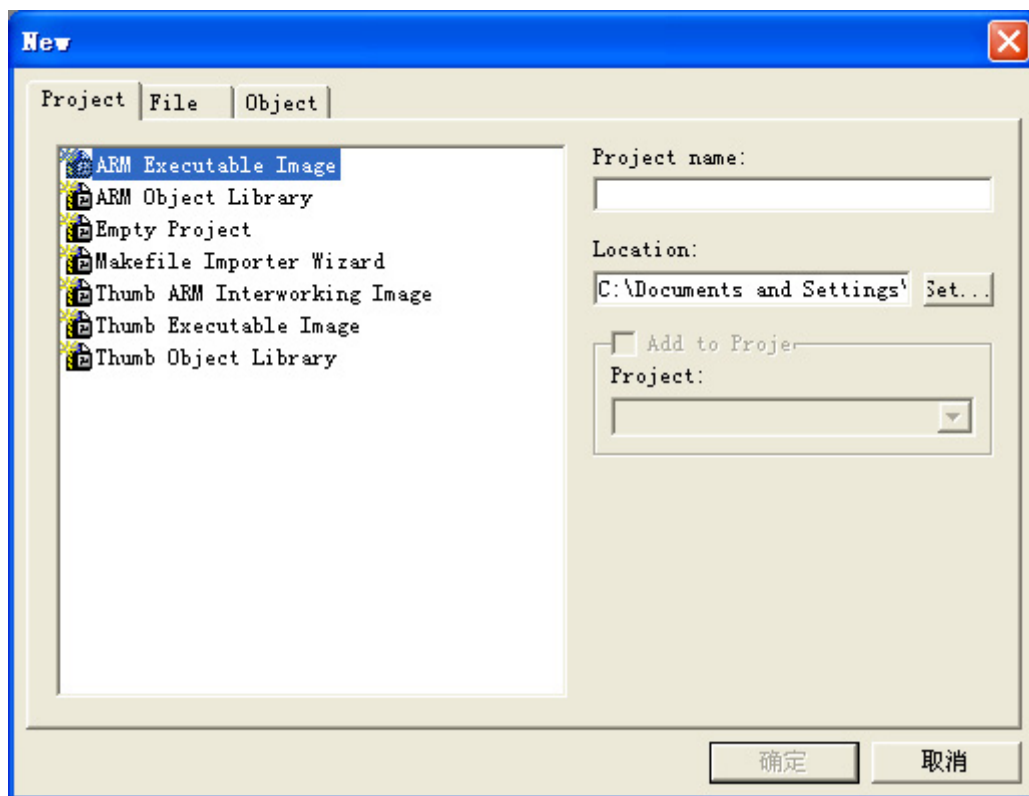


图 1，新建工程

ADS 为用户提供了 7 个模板，分别是：

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

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者源文件的工程；

Makefile Importer Wizard: 用于将 VC 的 nmake 或者 GNU make 文件转入到 Code Warrior IDE 工程文件；

Thumb ARM Executable Image: 用于由 ARM 指令和 Thumb 指令的混合代码生成一个可执行的 ELF 格式的映象文件；

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

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。

一般情况下均选择“ARM Executabel Image”，然后在“Project name:”栏输入工程名称，在“Location:”栏指定路径，本例子的工程名称为“Hello”，点击确定后“Hello”工程建立。

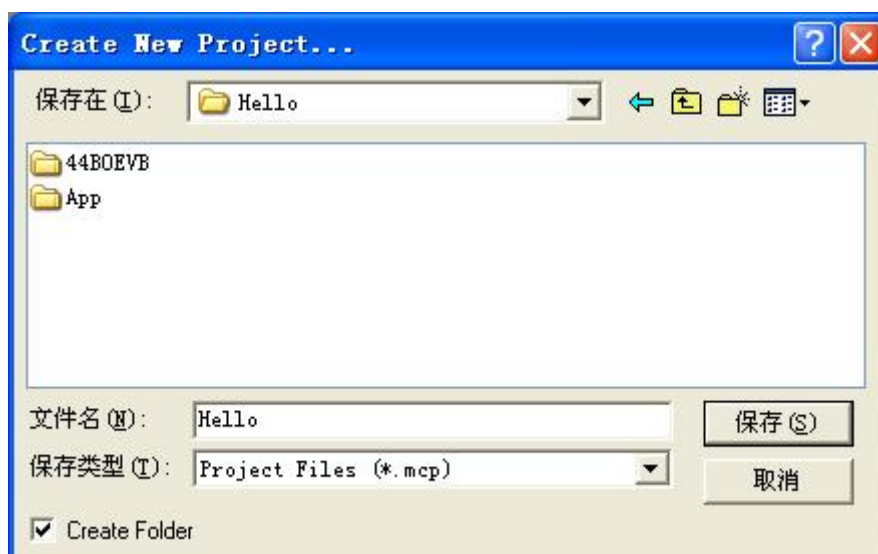


图 2，设置工程路径和工程名

工程建立以后将生成一个空的工程管理窗口，如下图所示：

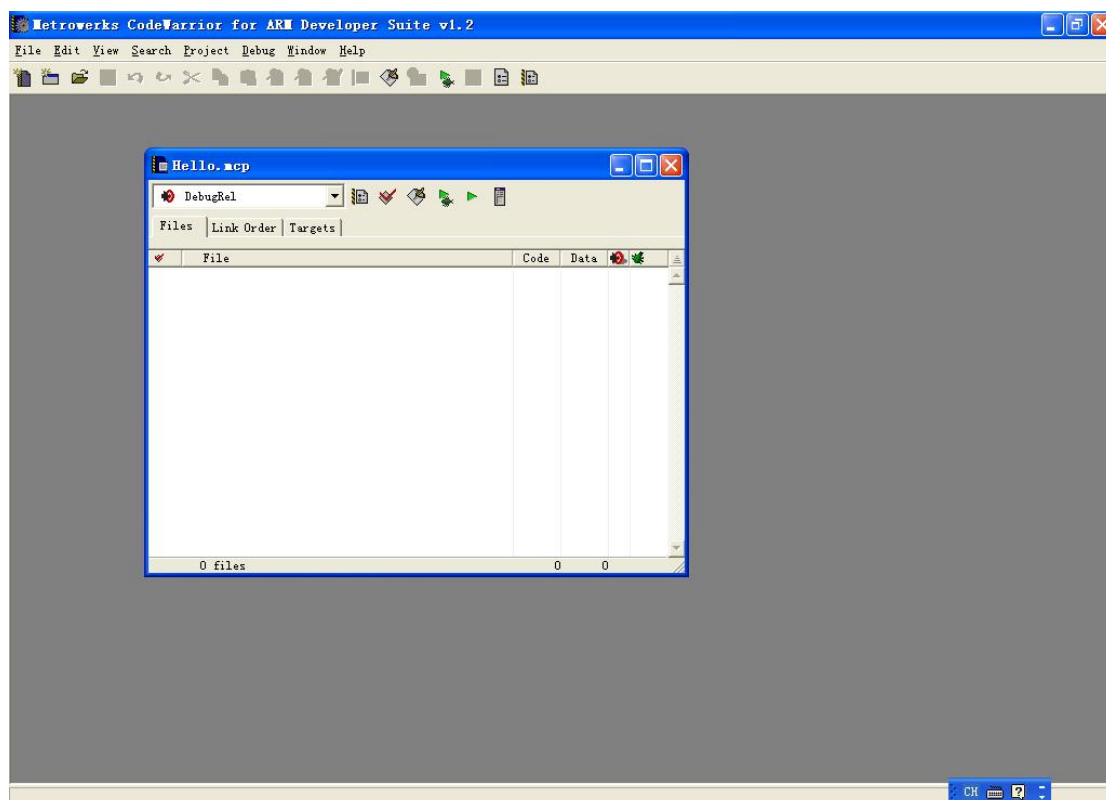


图 3，工程管理窗口

接下来就是向工程内添加和建立目标文件了，由于 44B0 学习评估板具有较多的资源，故其启动文件也较为复杂，所以一般都是直接打包使用，因此在添加启动文件之前请先将“44B0EVB”文件夹复制到“Hello”工程目录下，然后通过

工程管理窗口进行启动文件的添加，如下图所示：

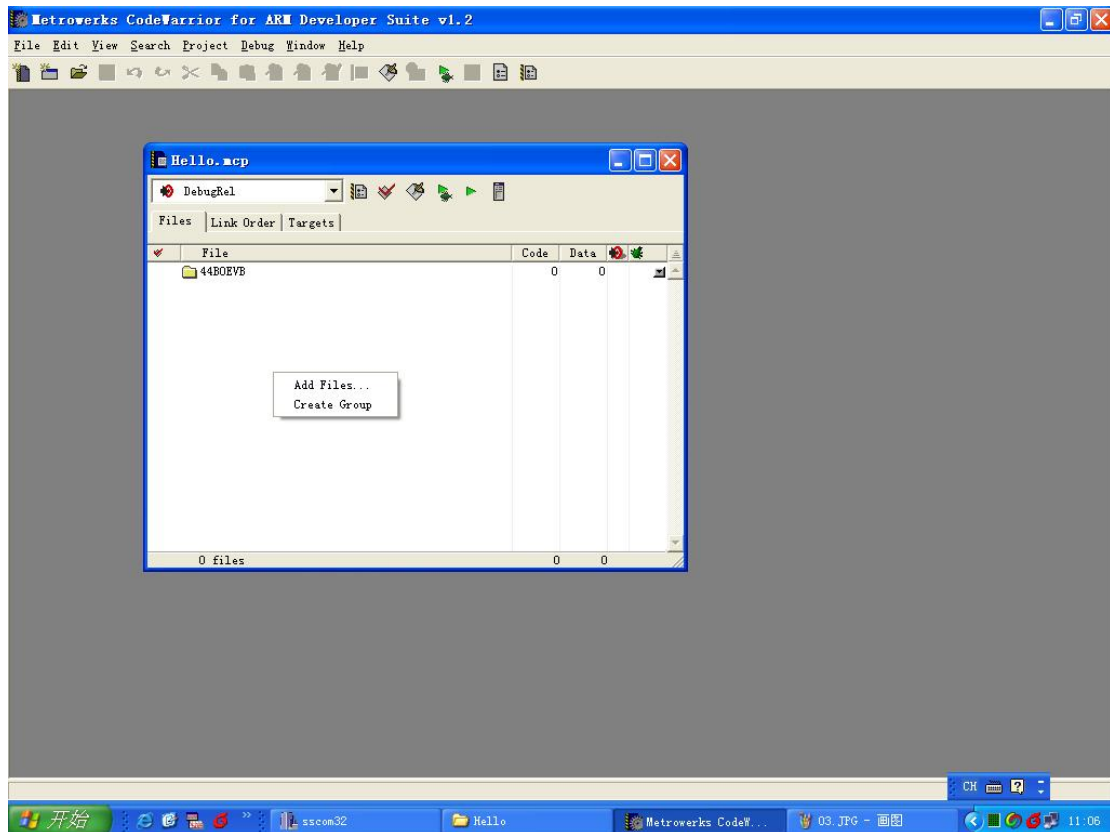


图 4，建立 Group

先通过点击右键选择“Create Group”，建立一个“44B0X”文件夹，然后再右键选择“Add Files...”来添加启动文件，如下图所示：

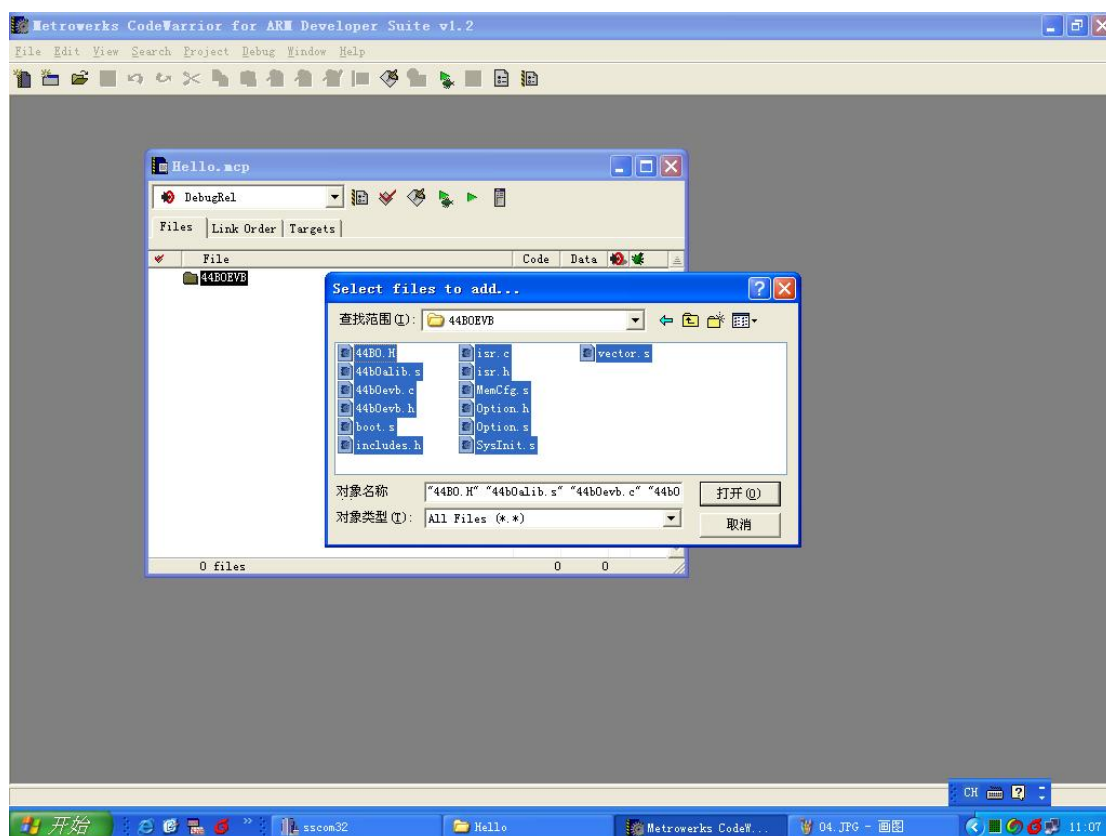


图 5，添加启动文件

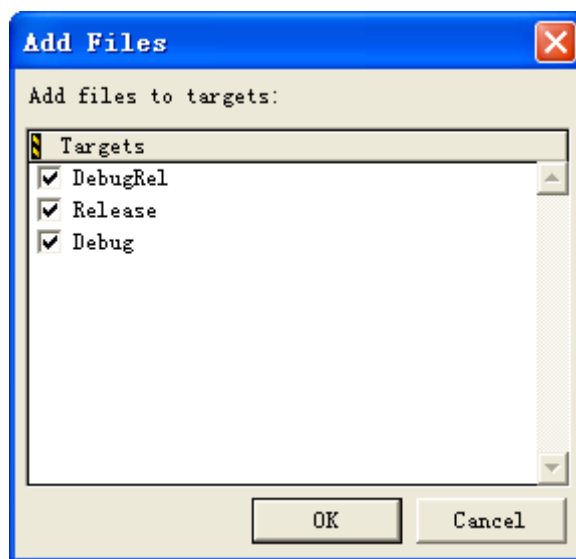


图 6，添加文件到 Target

在添加文件的过程中你可能已经发现了 CodeWarrior IDE 为用户建立了三个 Target，分别是“DebugRel”、“Release”、“Debug”，这三个 Target 分别表示三种调试方式。DebugRel 表示在生成目标的时候会为每一个源文件生成调试信息；Debug 表示为每一个源文件生成最完全的调试信息；Release 表示不生成任何调试信息。一般默认选择“DebugRel”。同时从图 6 可以发现每次添加文件的时候都会询问添加到哪个“Target”，一般默认三个都添加，点击确定。

添加成功后会有一个如图 7 的提示：

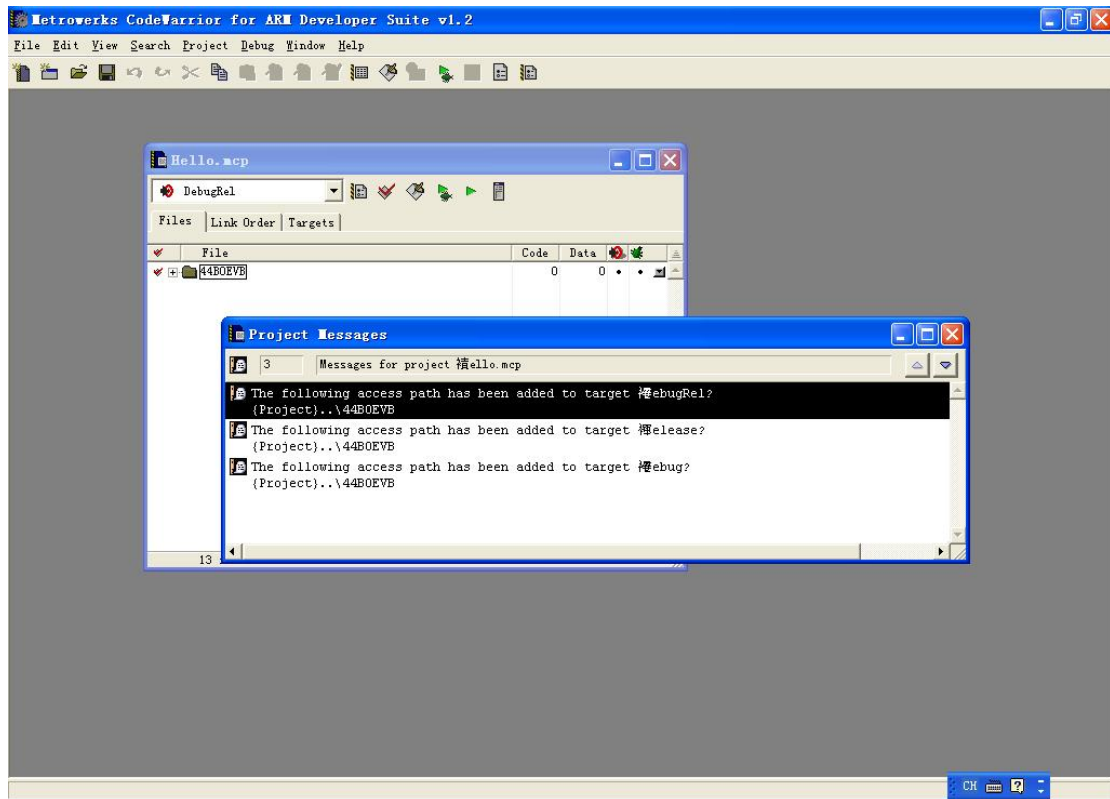


图 7, Project Messages

接下来建立一个 main.c 文件，可以通过“NEW”->“FILE”来建立，如下图：

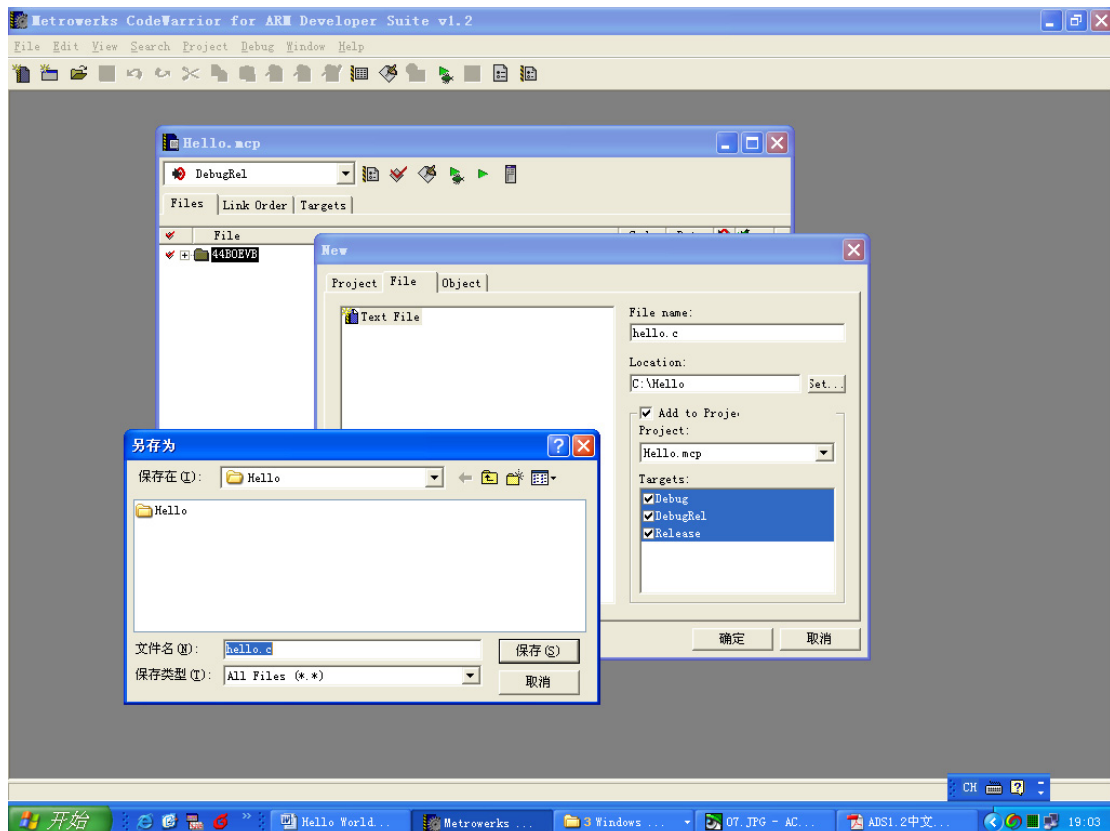


图 8, 建立文件

图 8 中已经把“Add to Project”前面的勾勾上了，所以 main.c 直接加入到了工程中，不然请在 main.c 建立后重新右键选择添加文件到工程。

最终的结果如下图所示：

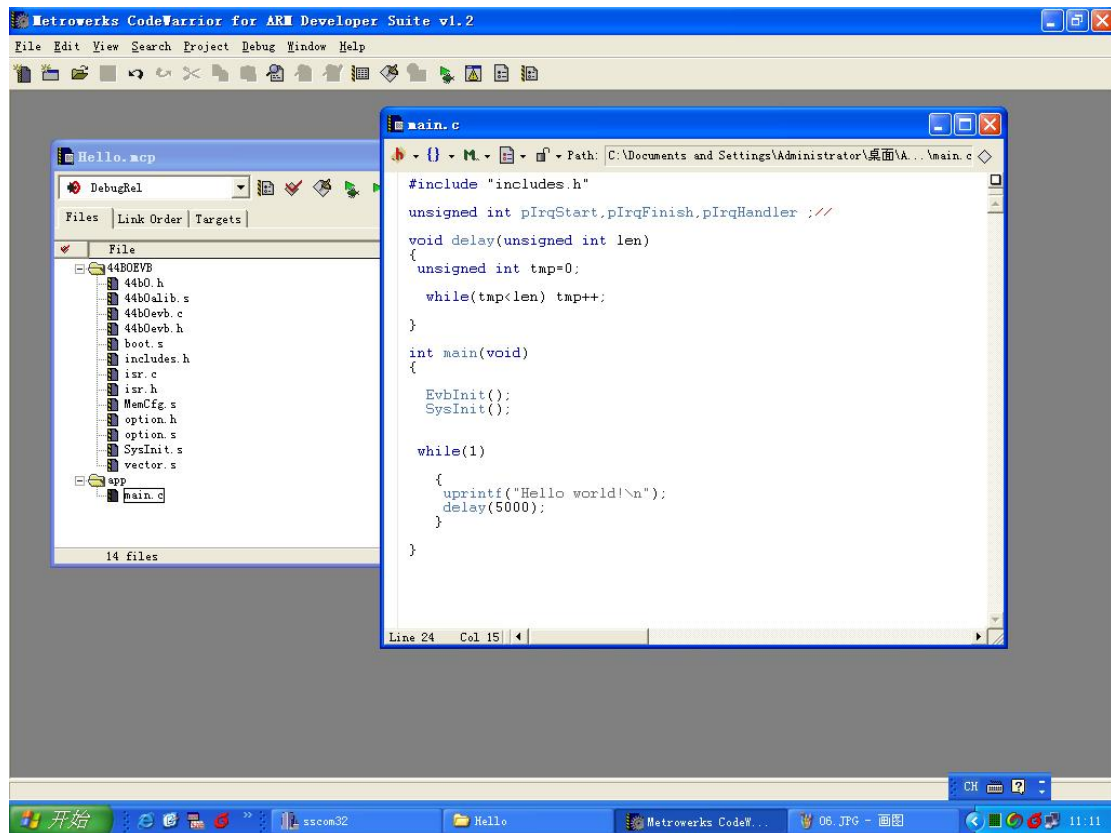


图 9，工程建立完毕

工程建立完毕之后暂时还不能进行编译和链接，还需要进行一些配置。

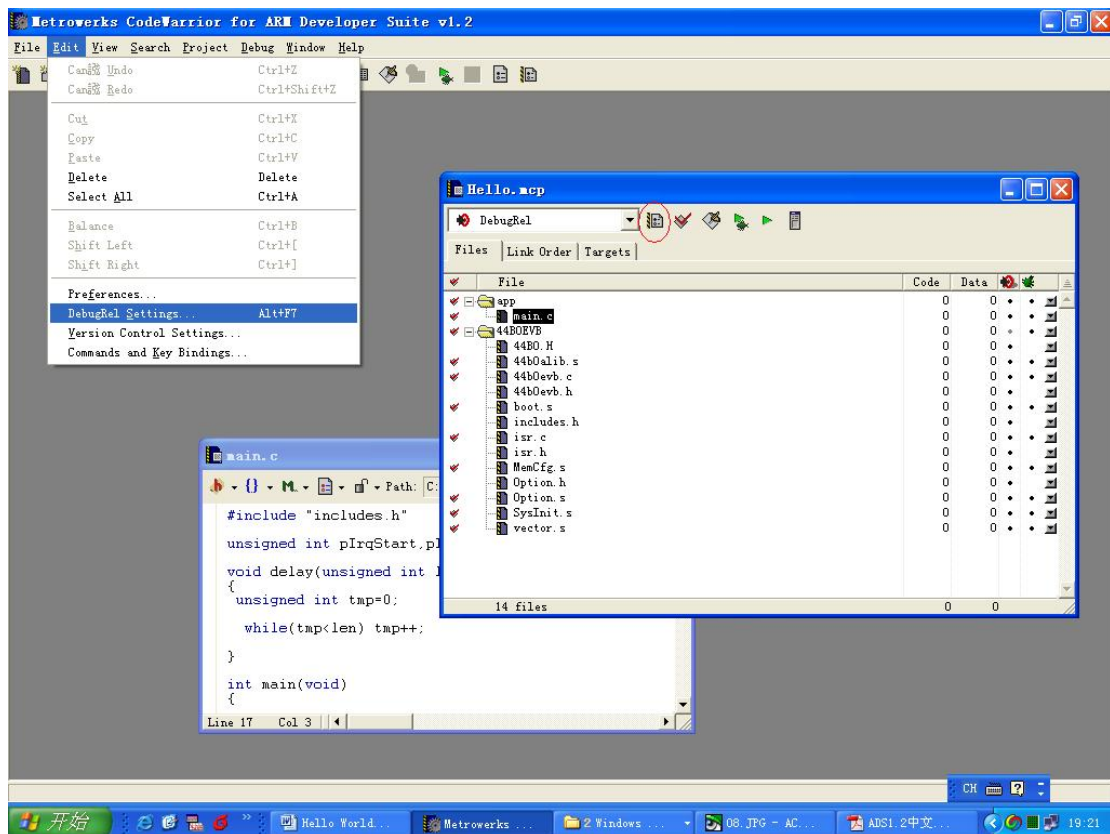


图 10，工程配置

可以通过“Edit”->“DebugRel Settings”或者“ALT+F7”或者点击图 10 中红色小圈内的快捷图标来进入“DebugRel Settings”。如下图所示：

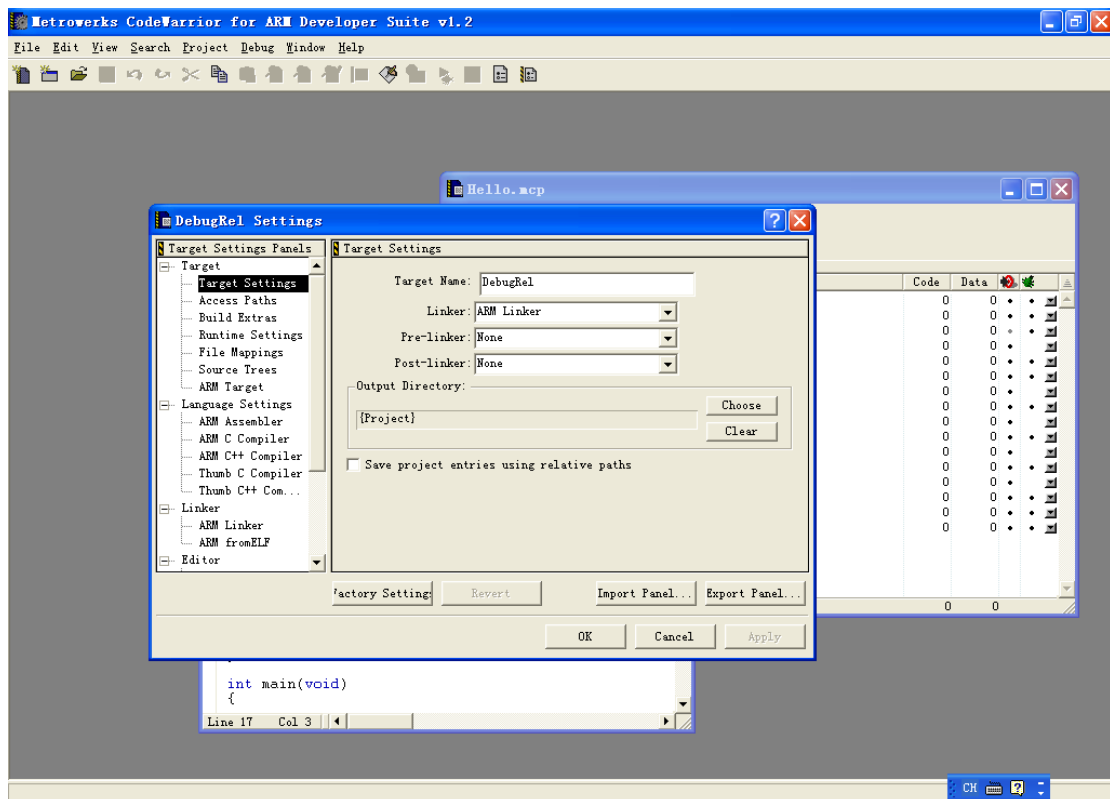


图 11，“DebugRel Settings”

“DebugRel Settings” 里面包含了很多设置信息，在此只说明几个和本范例相关的配置信息，如果了解其他相关信息请参见“ADS 文档”内的相关资料。

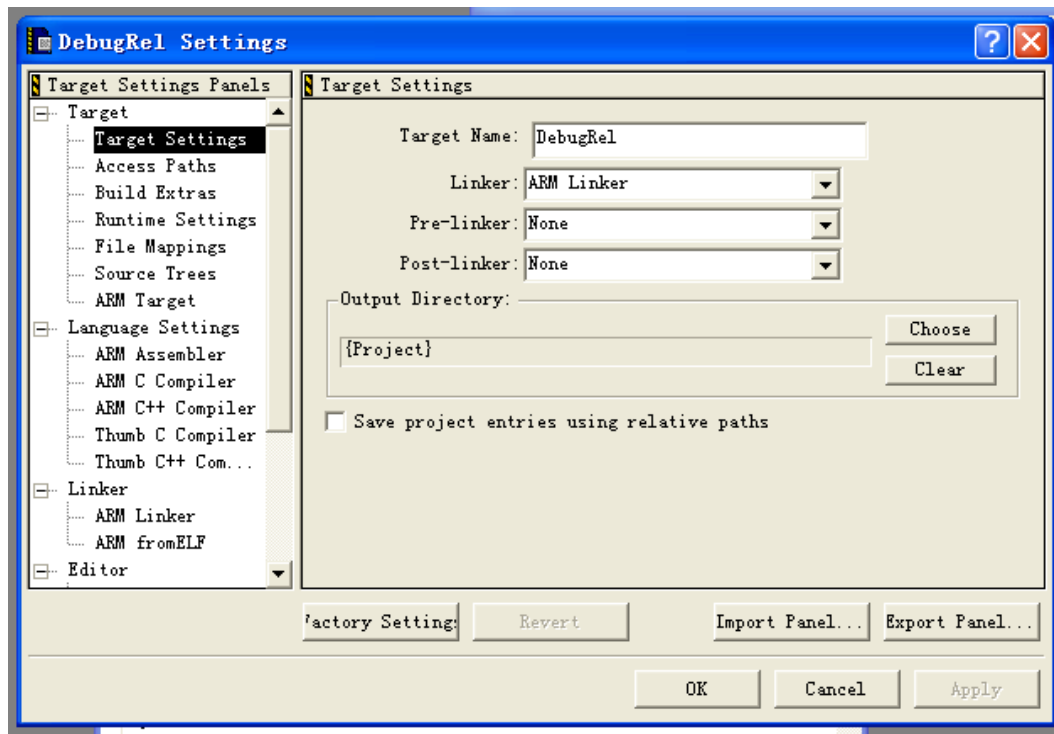


图 12, “Target Settings”

“Target Settings” 里面有一处必须设置，即 “Post-Linker”，“Post-Linker” 用于对输出文件进行操作，由于本范例是需要最终写入到硬件中并运行的，所以必须进行设置（如果纯粹只需要进行软件仿真则此处可以不进行设置）。这里选择 “ARM fromELF”，表示编译后生成映像文件（Image）后再调用 FromELF 命令进行格式转换，以转换成 .bin 或者 .hex 等可以直接烧写到目标芯片执行的文件。

然后是 “Language Settings” 选项，本范例使用了 “ARM Assembler” 和 “ARM C Compiler” 所以请确保在这两个选项内的 “Target” 子选项内为 “ARM7TDMI”（44B0 为 ARM7TDMI），如下图所示：

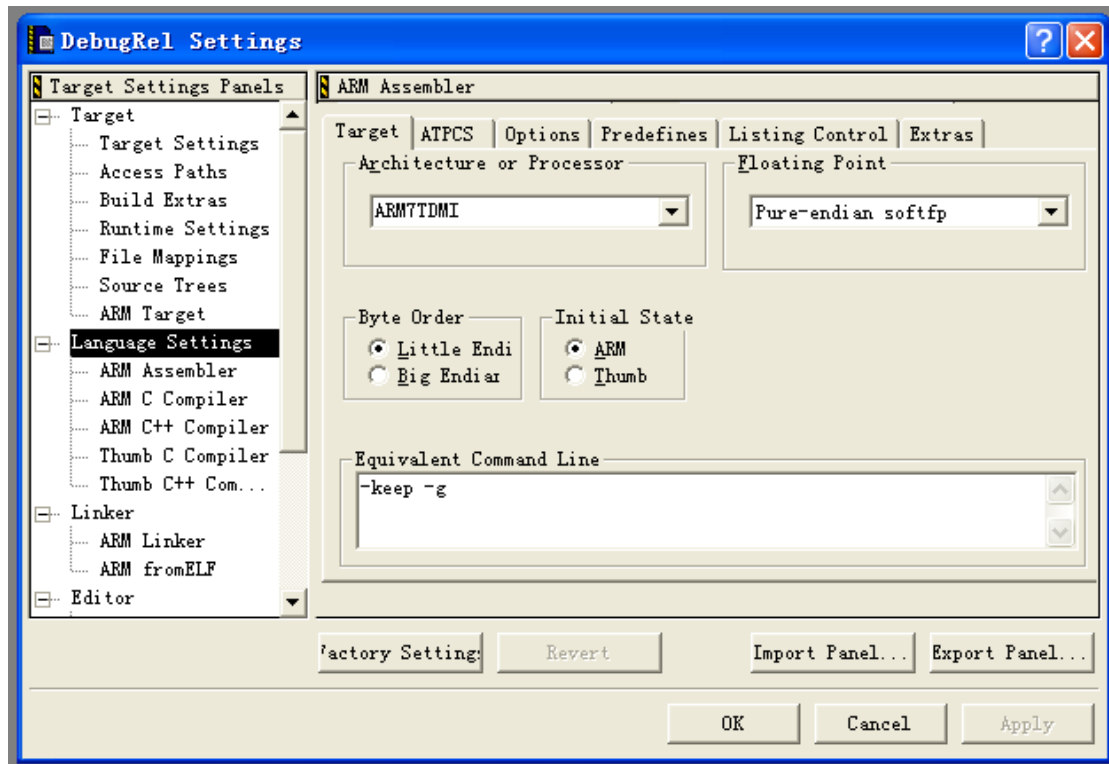


图 13, “Language Settings”

设置完了“Language Settings”后接下来就是“Linker”选项了,在该选项里面有和硬件紧密相关的设置,首先进入“ARM Linker”子选项,在该子选项内需要设置入口地址(entry)、RO地址、RW地址。

首先设置RO/RW地址,RO表示Read Only,RW表示Read Write,RO栏默认是0x8000,需要根据实际硬件进行更改,一般为FLASH地址,RO也可以设置在SDRAM中,启动代码中包含COPY的部分,会将代码移动到SDRAM中运行,这对44B0特别适合,因为SDRAM很大,而且对速度的提升也很明显。entry point必须同RO一致,以提供连接器定位信息

本44B0板的FLASH地址即为0x00000000;RW栏默认为空,一般为SDRAM地址,本44B0板的地址即为0x0C000000(RO也可以设置在SDRAM中但entry point必须同RO一致)。

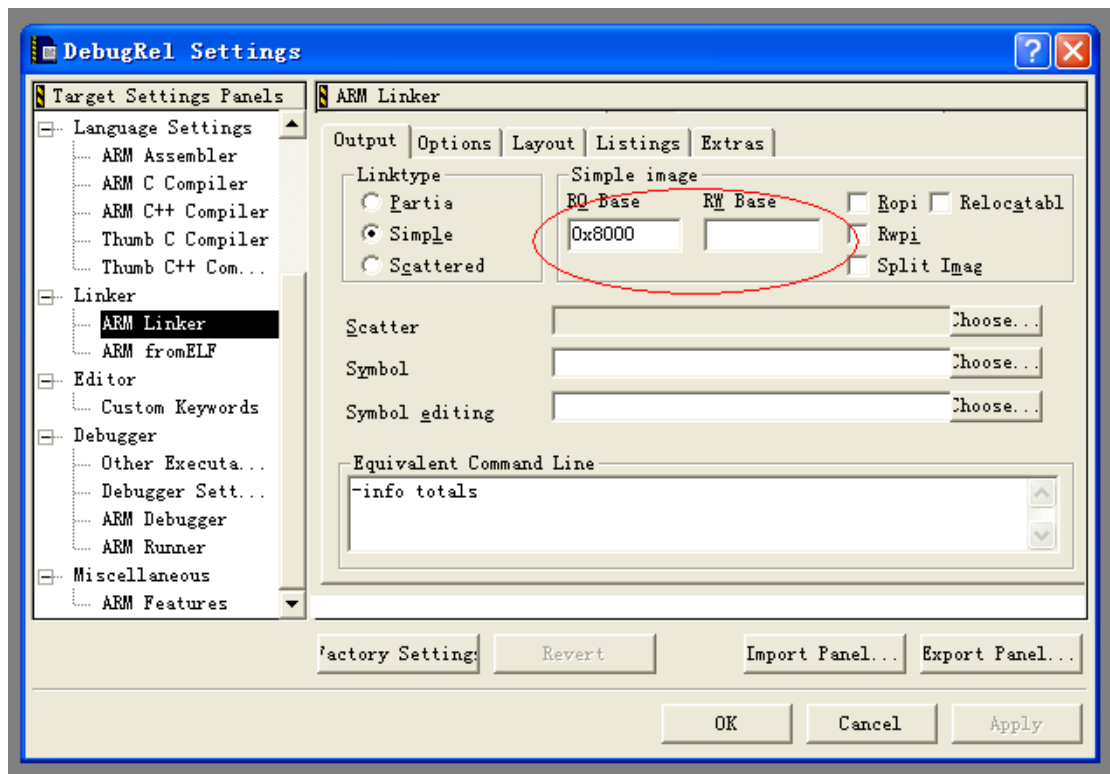


图 14, “ARM Linker” -> “Output”

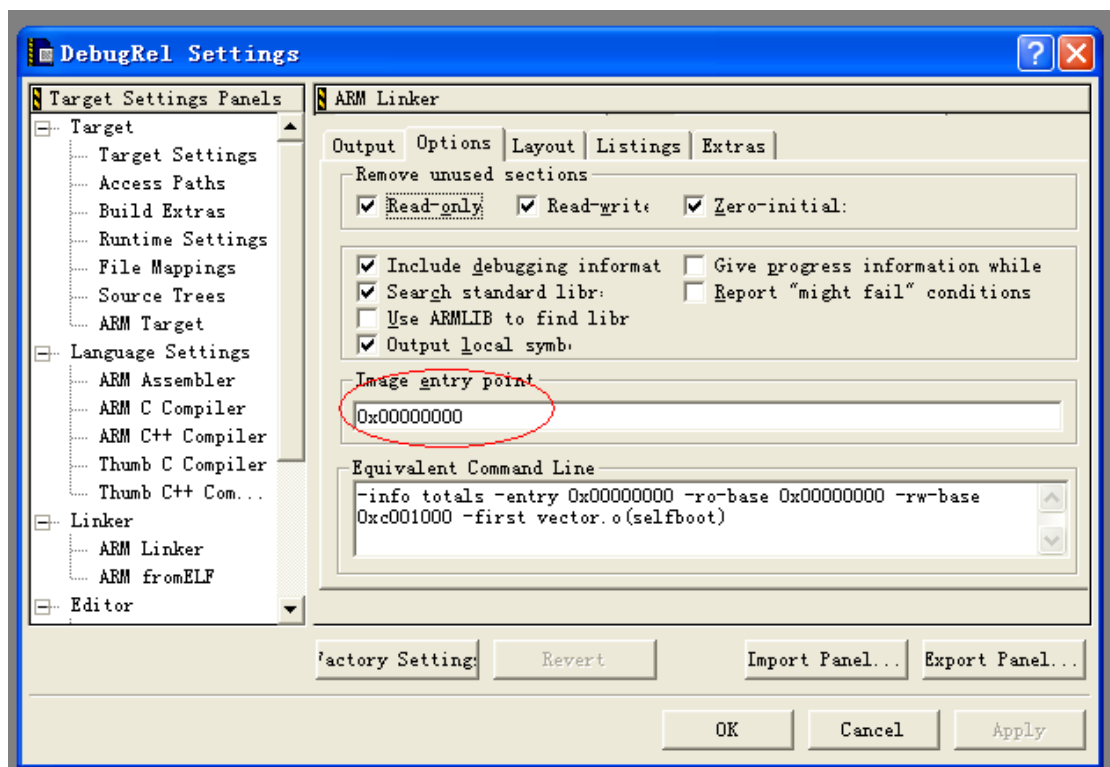


图 15, “ARM Linker” -> “Options”

在“ARM Linker”->“Options”内还需要设置一个“Image entry point”，表示映象文件的入口地址。

设置完“Image entry point”后继续在“Layout”里面设置代码中的哪一段置于 IMAGE 的起始位置：

图中 vector.o SelfBoot 的意思就是将 vector.o 置于 IMAGE 的起始位置,必须设置!

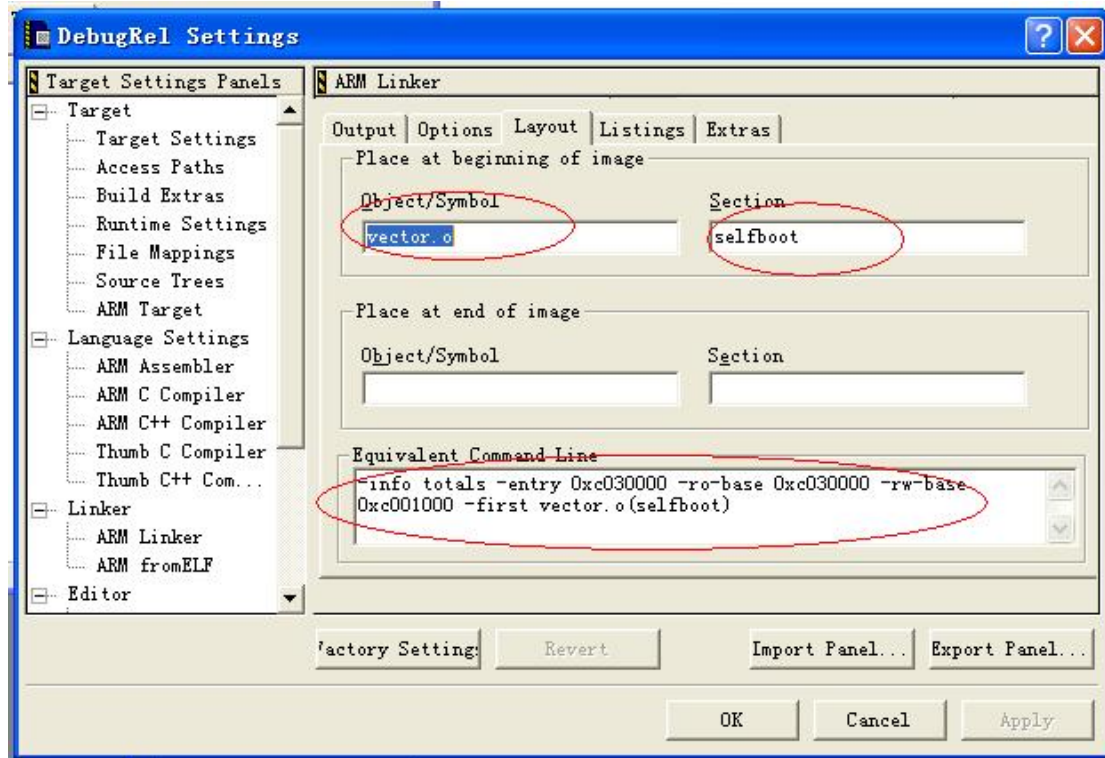


图 16, “ARM Linker” -> “Layout”

在“Linker”选项下还有一个子选项需要进行设置,即“ARM fromELF”,在“Target”的“Post-Linker”设置成“FromELF”后在这里需要进行设置。

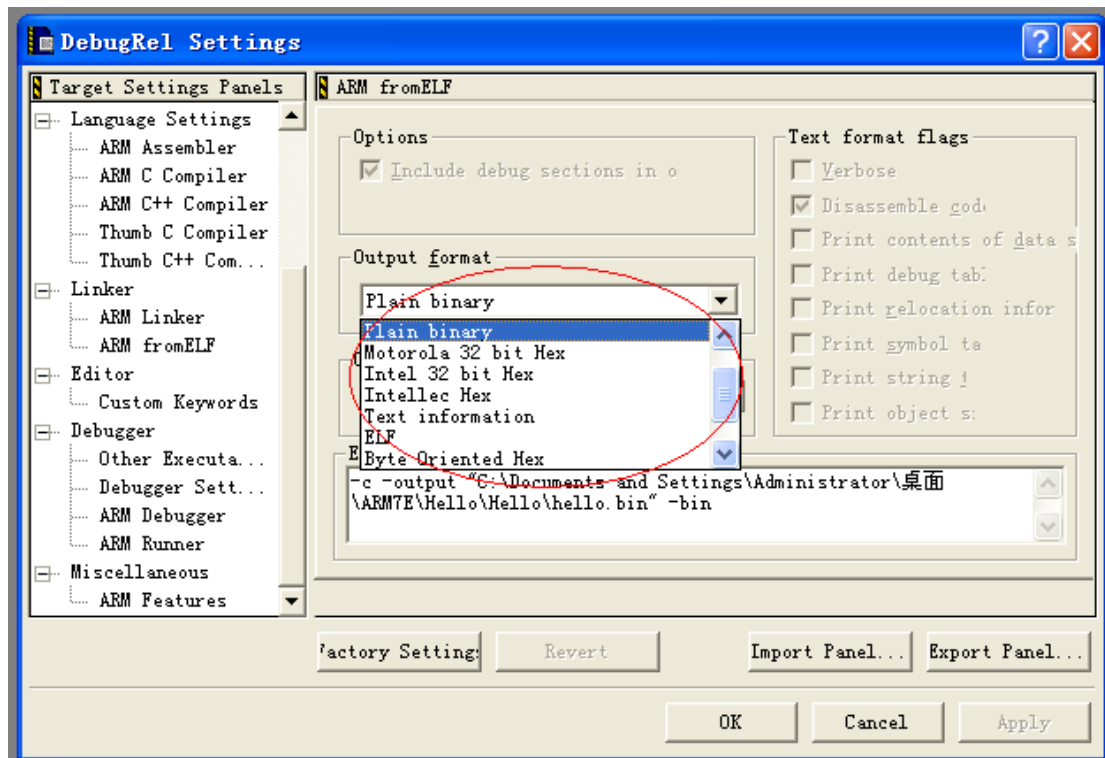


图 17, “Linker” -> “ARM fromELF” -> “Output format”

参照图 17 进行输出文件格式的设置, 一般设置成 “Plain binary” 或者 “Intel 32 bit hex”。

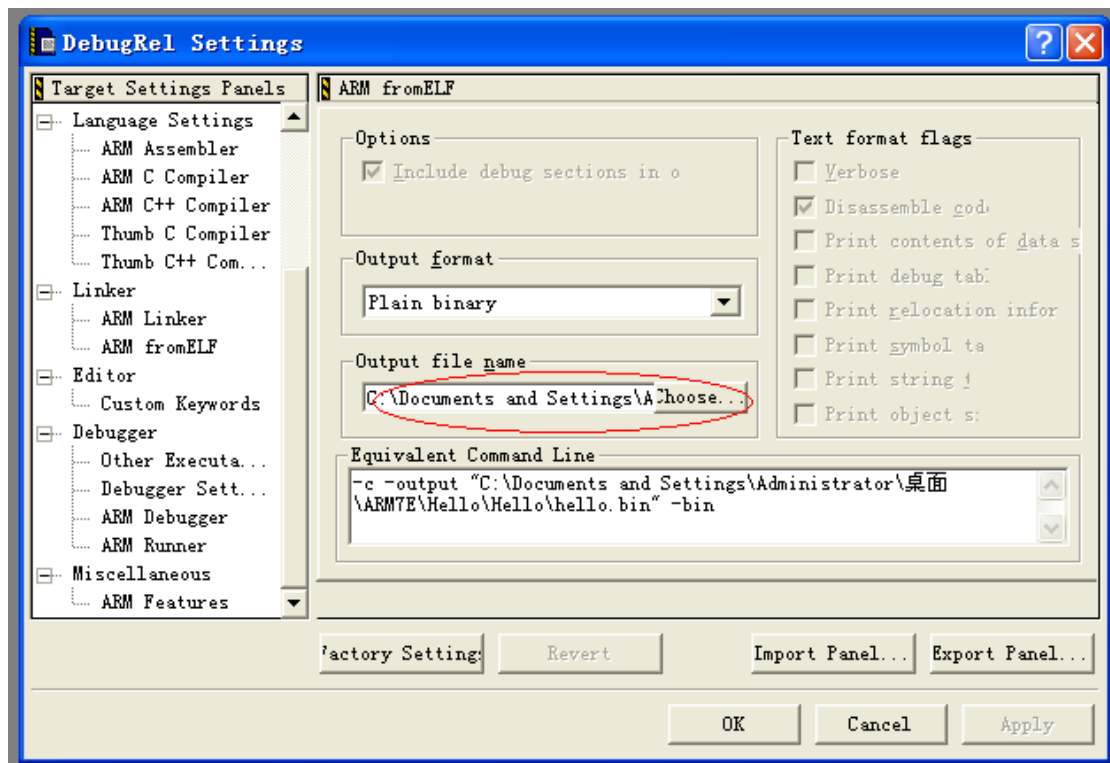


图 18, “Linker” -> “ARM fromELF” -> “Output file name”

参照图 18 进行输出文件名称和路径的设置, 请带上扩展名.bin 或者.hex。当工程文件是从别处 copy 过来的时候请记得对该路径进行重新设置, 不然将出现警告提示。

到此为止已经完成基本设置, 接下来可以进行编译和链接。

如果你的工程文件是从别处 copy 过来的则在编译之前还有一件事情需要做, 即 “Remove Object Code”, 在选项在 “Project” 菜单下面, 如图 19 所示, 如果红圈内的.c 文件前面没有勾的图标的话表示该工程已经编译链接通过, 如果需要重新编译则需要先进行 “Remove Object Code” 操作以去除一些和路径、目标文件等有关联的信息。

“Remove Object Code” 操作后如图 20 所示, 请对比图 19 和 20。

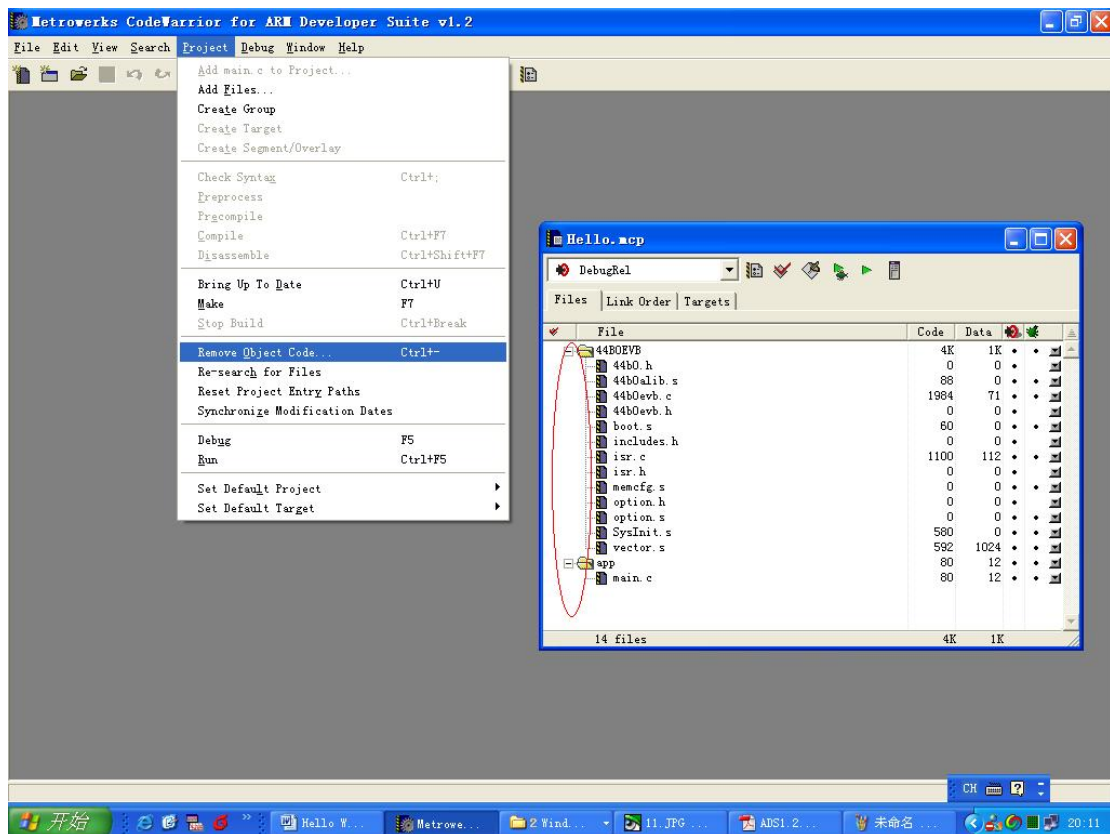


图 19, “Remove Object Code”

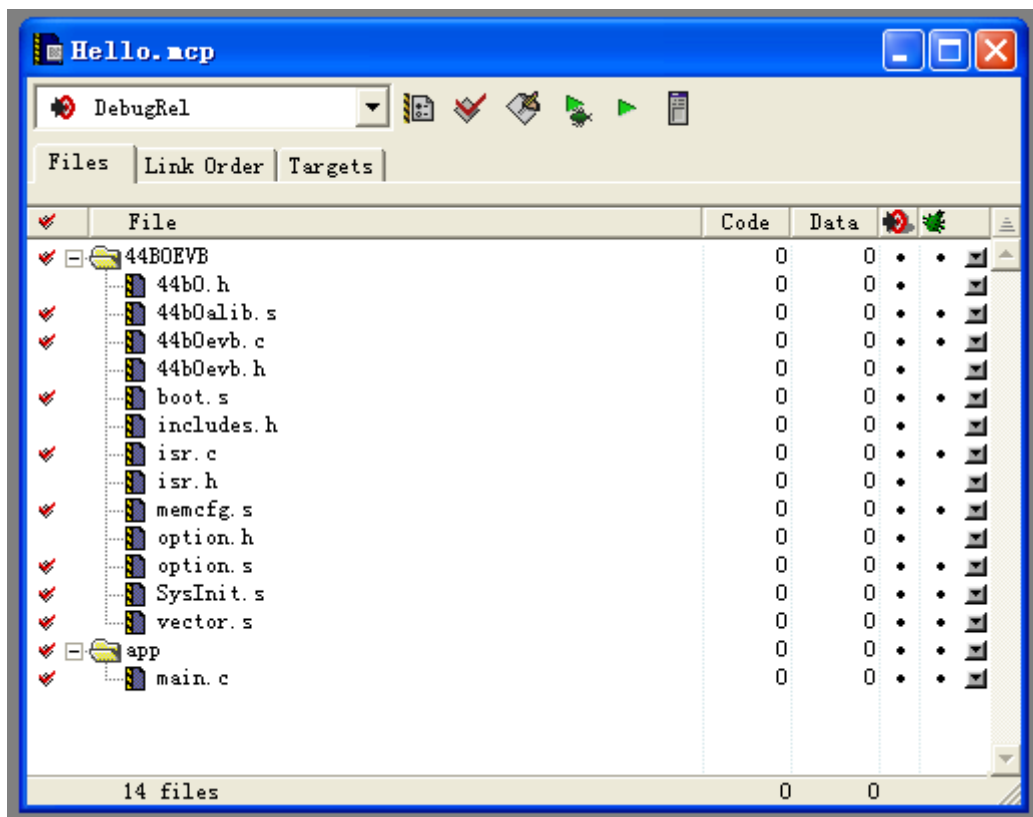


图 20, 需要编译的文件前有勾状图标

然后 “Project” -> “Make” 或者 F7, 如果一切设置正确则出现如下提示:

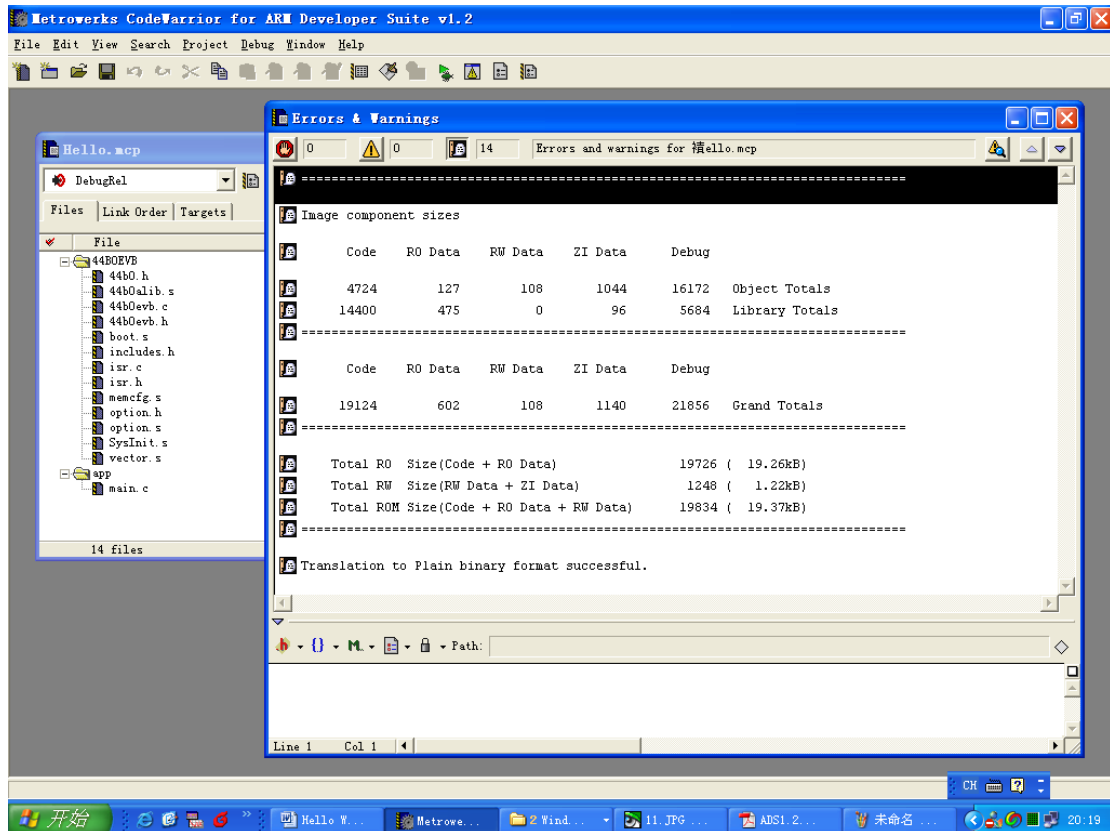


图 21, MAKE 结果

如果有错误和警告请一一改正，直至编译通过。

最终生成可执行文件 `hello.bin/hello.hex`（因具体设置而异）和调试文件 `hello.axf`（映像文件）。其中可执行文件的路径由用户在“LINKER”->“ARM from ELF”中的设置所决定；而调试文件（映像文件）“`hello.axf`”则默认生成在“`..\Hello\Hello_Data\DebugRel`”下。`Hello.bin/hello.hex` 可以通过工具软件直接烧入目标芯片运行，`hello.axf` 可以通过 ADS 里面集成的 AXD 调试工具进行调试，可以软件仿真或者外部通过第三方 JTAG 调试工具进行调试。

下面简单讲解一下用 AXD 进行代码调试的流程：

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，即软件仿真，下面内容主要是结合 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，在介绍完 ARMulator 之后将会在后文继续简单讲解如何通过 JTAG 开发工具进行硬件仿真。

要使用 AXD 必须首先要生成包含有调试信息的程序，在前面的 `hello` 工程中，已经生成的 `hello.axf` 就是包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

可以有两种方法打开 AXF 文件，一是在 `make` 完成后通过“Project”->“Debug”或者 F7 进入 AXD，另外也可以通过“开始”->“程序”->“ARM Developer Suite”

-> “AXD” 打开 AXD，然后再在菜单 “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 22 所示：

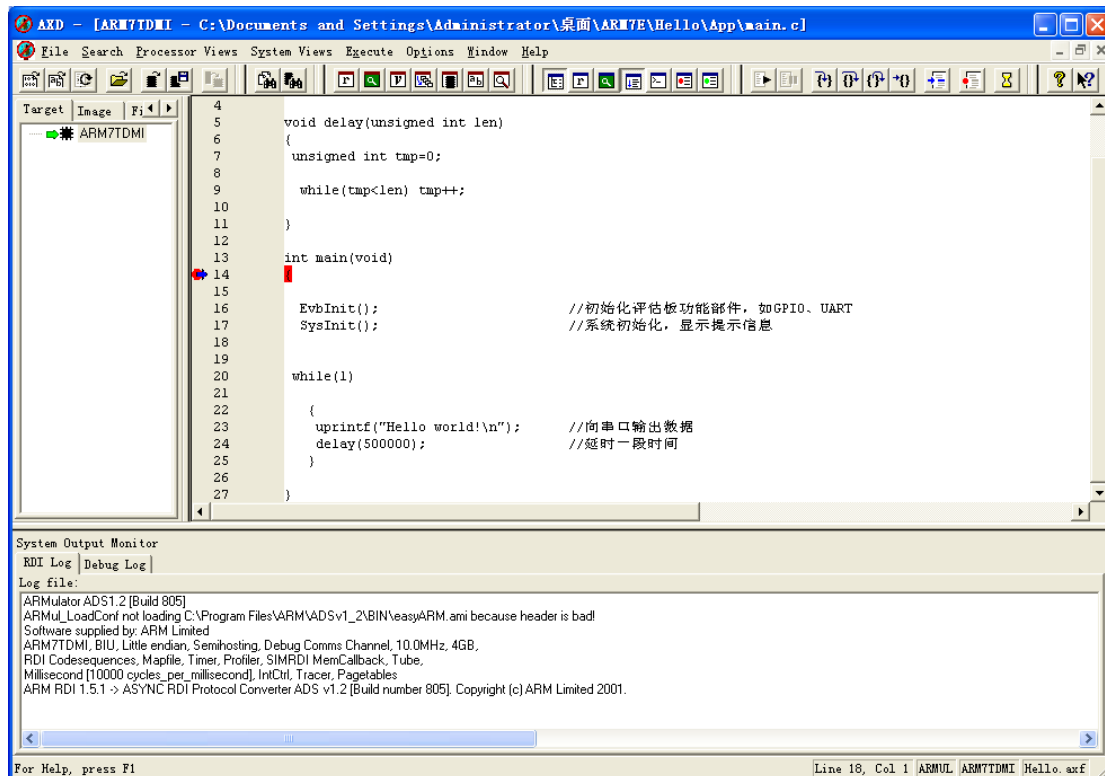


图 22 在 axd 下打开映像文件

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。

注意：

在打开 AXD 的时候可能会出现如下的错误，请按照下文指示进行处理。

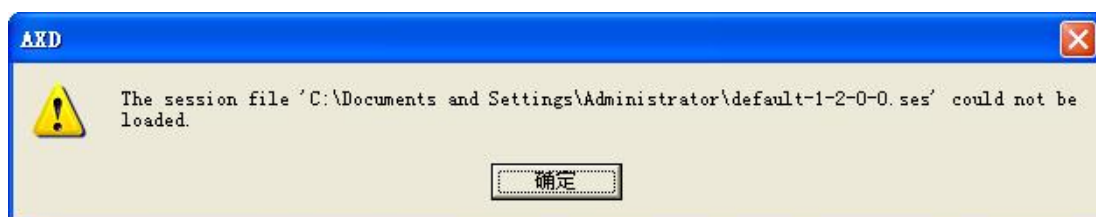


图 23, Warning

点击确定，出现另一对话框：

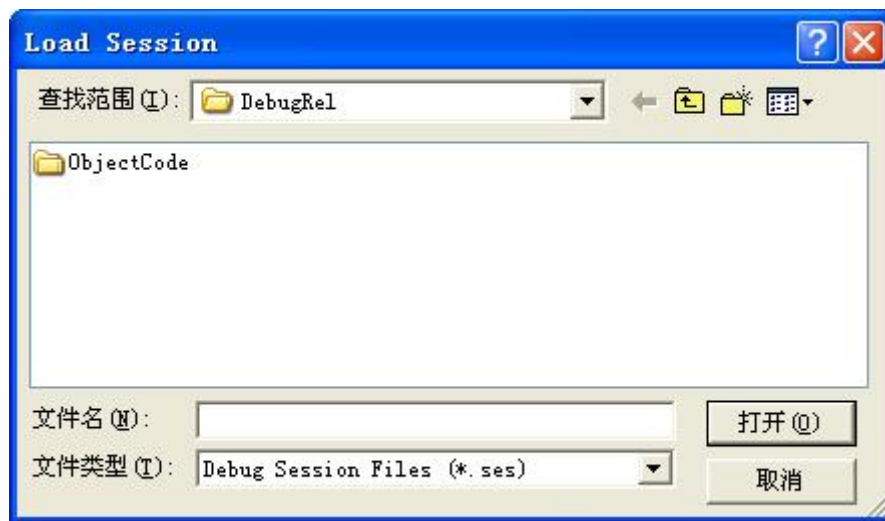


图 24, Load session file

点击“取消”进入 AXD。

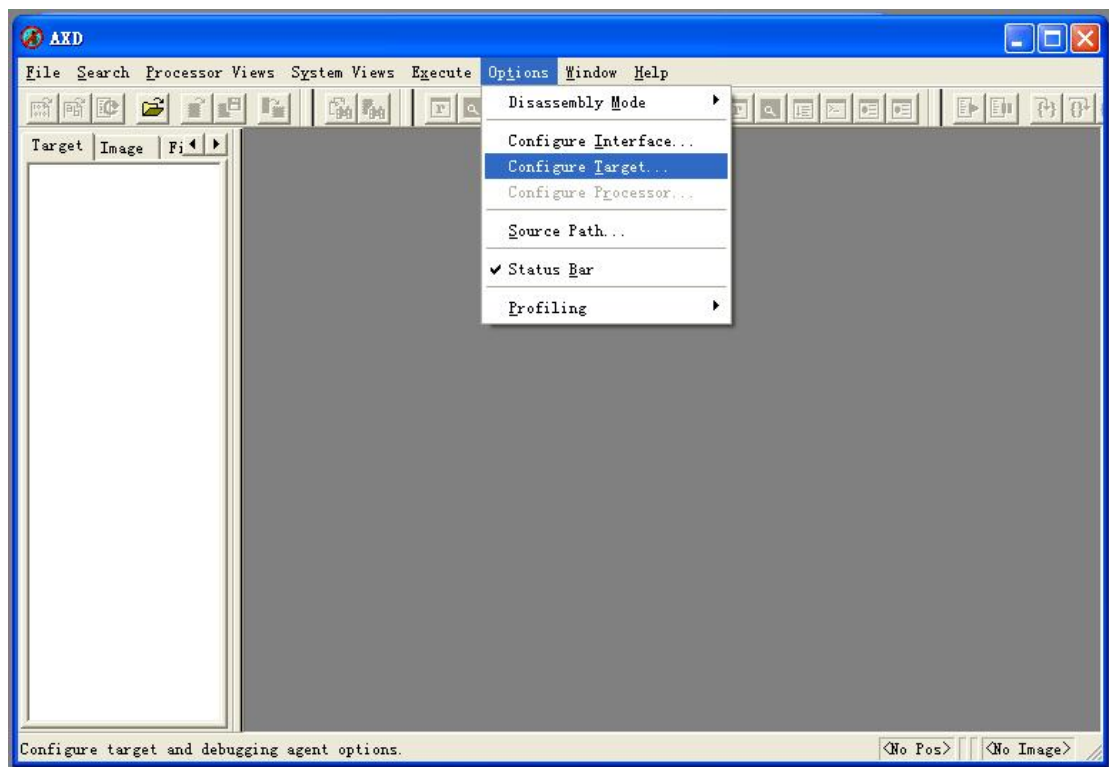


图 25, Configure Target

进入 AXD 后通过“Option”->“Configure Target...”选择调试目标。
由于当前采用“ARMulator”进行软件仿真，故选择 ARMUL，如下图所示：

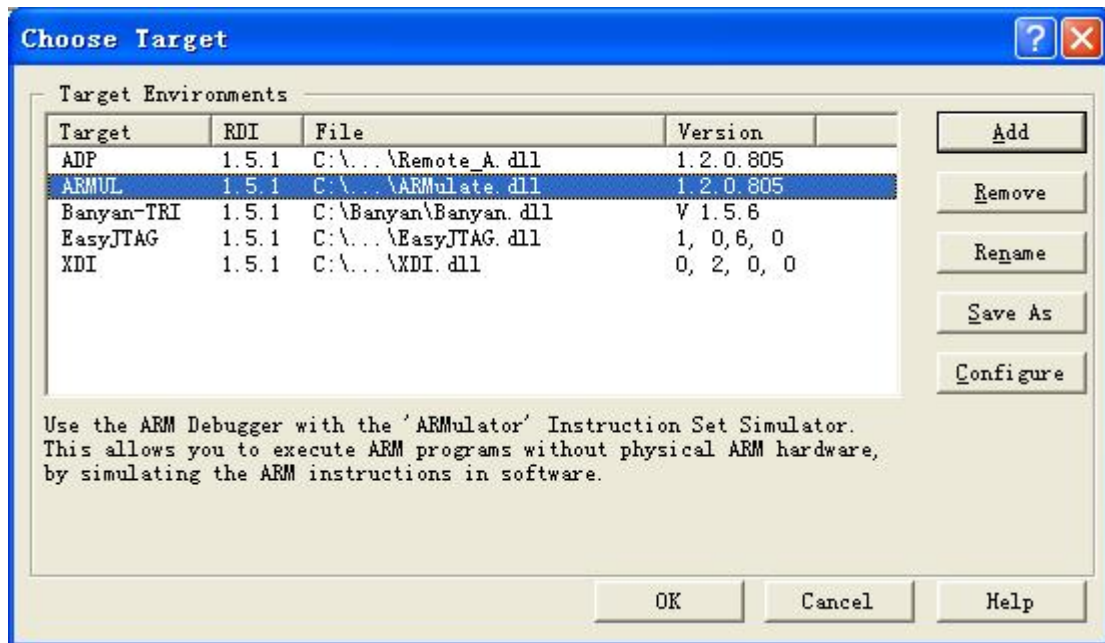


图 26, 选择 ARMUL

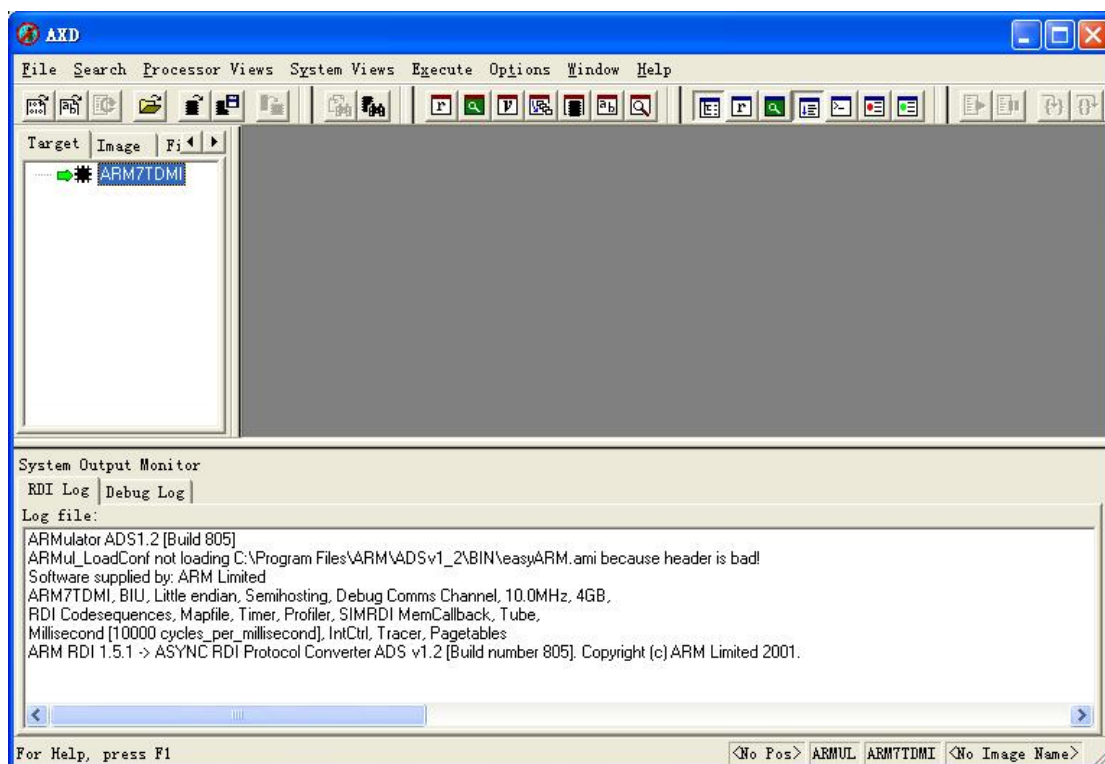


图 27,
点击 OK, 进入图 27 所示状态。

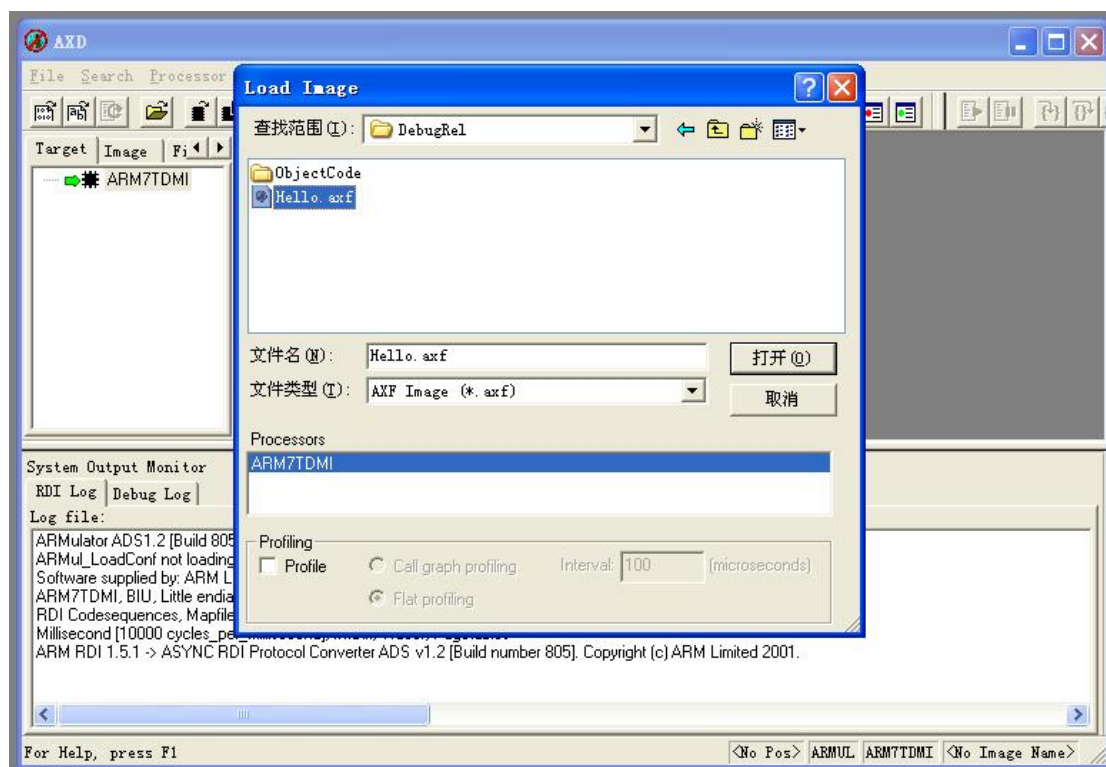


图 28, Load Image

然后 Load Image, 在 hello 工程下面的“DebugRel”文件夹内找到 hello.axf 文件, 然后调入。

另外在使用 JTAG 调试的时候还可能出现弹出 Fatal AXD Error 窗口, 如下图所示, 此时可以点击“Connect mode...”, 然后选择“ATTACH...”项确定, 再点击“Restart”。接下来就可以使用【File】->【Load Image...】加载调试文件, 进行 JTAG 调试。

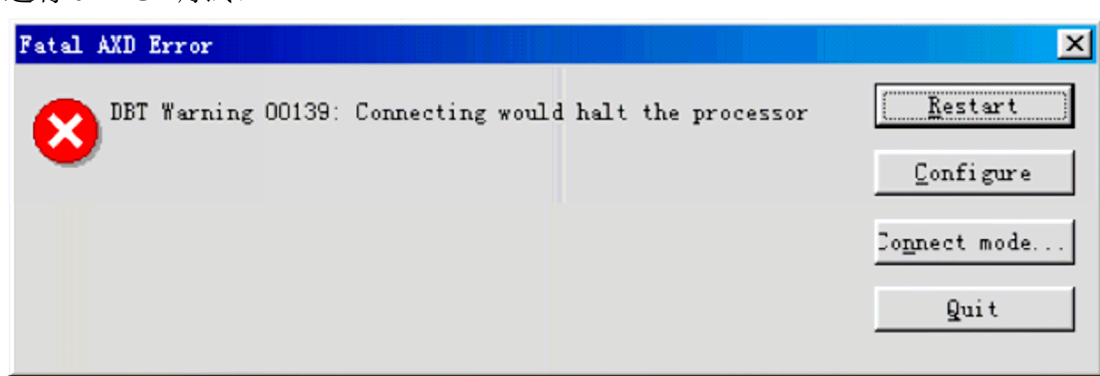
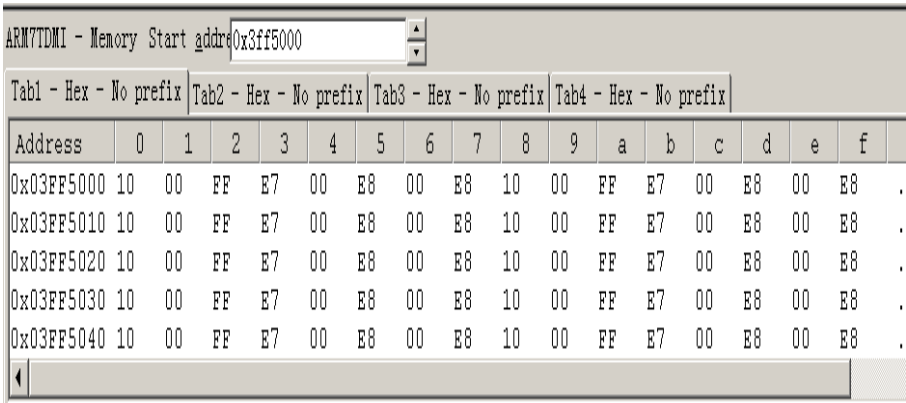


图 29, Fatal AXD Error

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 30 所示。



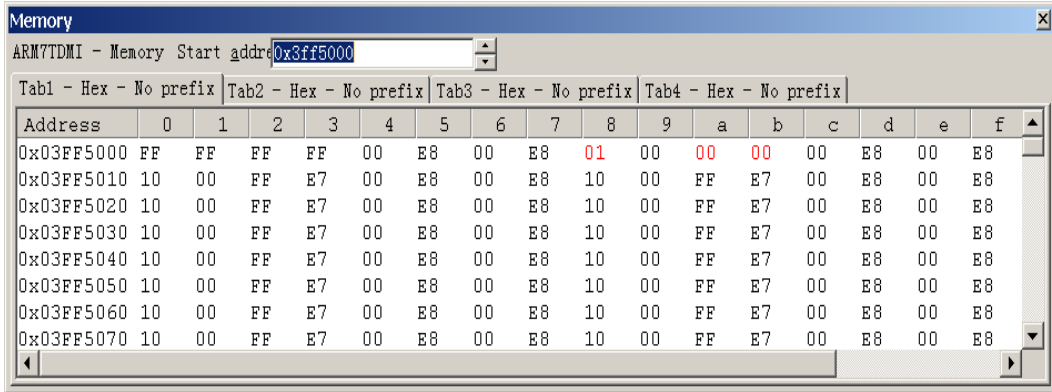
ARM7TDMI - Memory Start address: 0x3ff5000

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 30 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 23 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 31 所示：



ARM7TDMI - Memory Start address: 0x3ff5000

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 31 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(500000);” 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 32 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 32 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 32 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

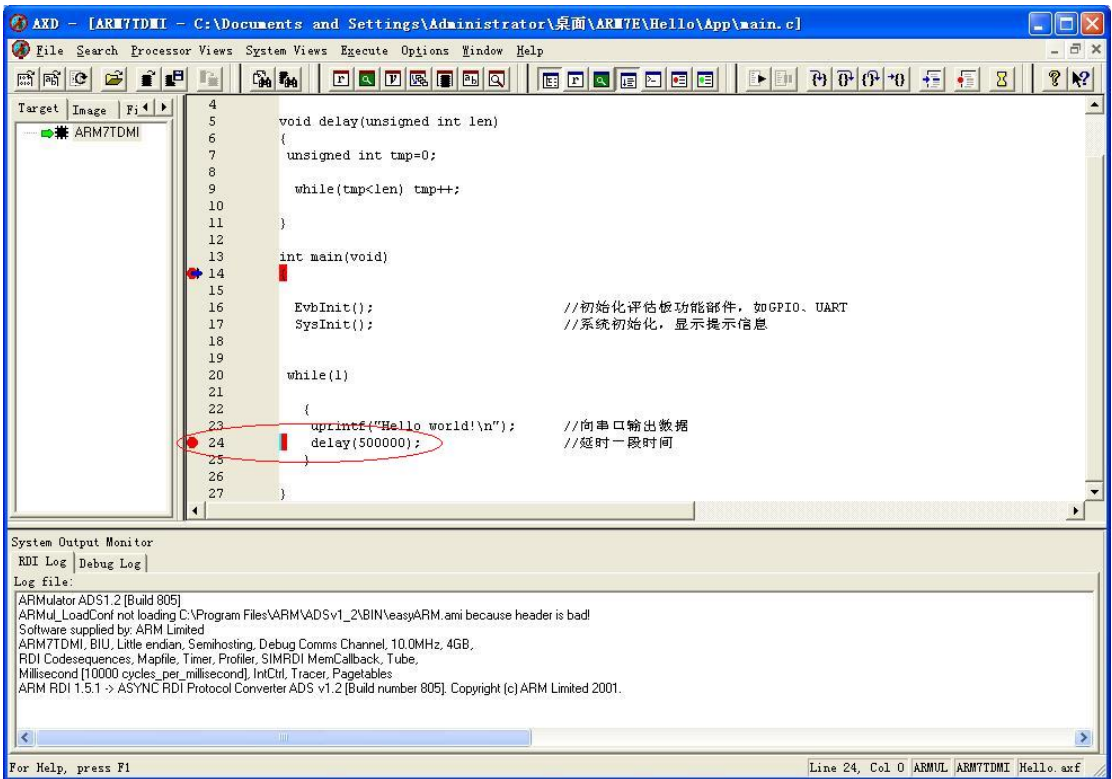


图 32 设置断点

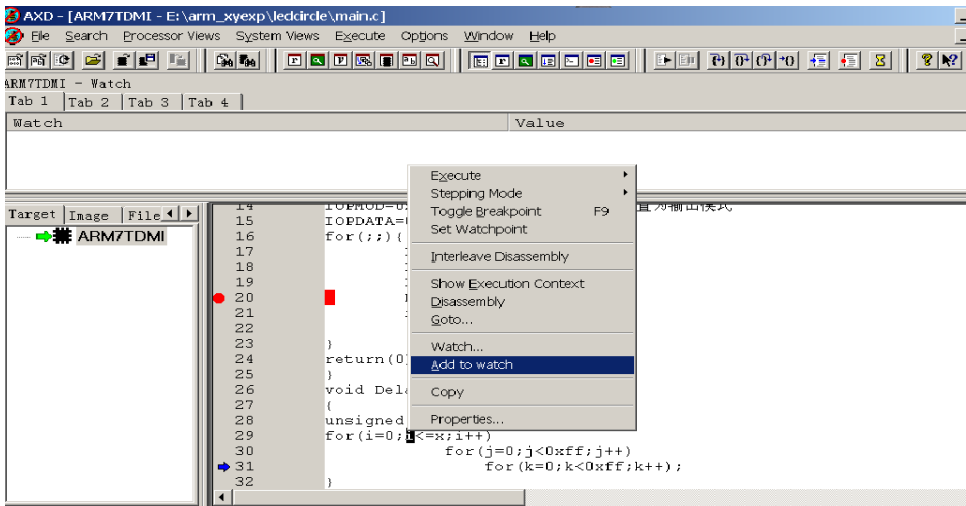


图 33 查看变量

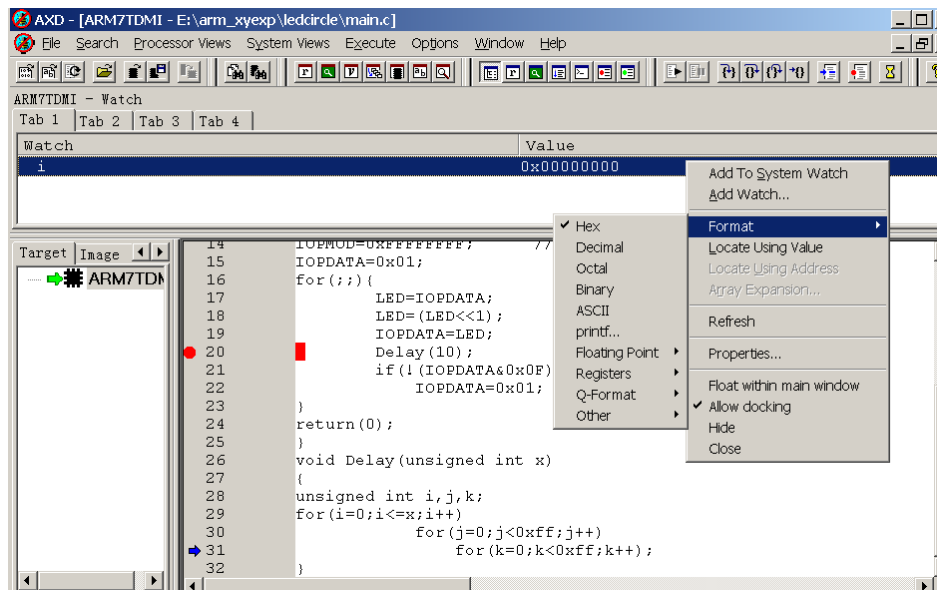


图 34 改变变量的格式

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 34 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。

“ARMulator”仿真一般是在不具备硬件条件的情况下进行软件仿真，多用于学习状态，并不能完全反映实际硬件运行状态。接下来简单讲解用 JTAG 工具进行硬件调试的流程。

很多公司都有支持 ADS 的 JTAG 调试工具，象 ARM 公司，IAR 公司，周立功公司等等，但是一般大公司的这些 JTAG 工具都是性能强大但是价格昂贵，其实也可以通过一些第三方的免费或者评估软件配合简易的 JTAG 小板进行调试开发，这其中流传的最广的可以算 BANYAN II 软件和 Wiggler 板了。

首先需要安装 BANYAN II 软件，并在调试前先运行它，具体安装和使用请参见 BANYAN II 的用户手册。

安装完Banyan 后，在：开始—>程序—>Banyan 中打开Daemon，（即任务栏右下脚那只爬虫），出现如下所示图35：

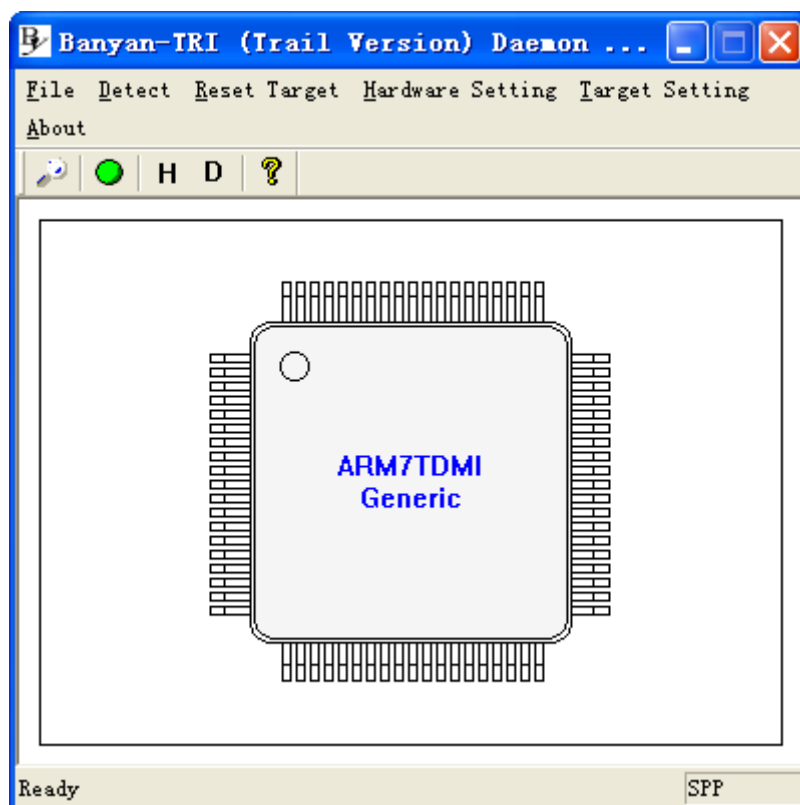


图 35，启动 BANYAN

启动 BANYAN 并将目标板通过 Wiggler JTAG 板连接到计算机的并口后首先需要通过 BANYAN 来检测目标是否连接正确。

虽然图示出现 ARM7TDMI 字样，但是该提示未必表示真正找到了 44B0X 板，并检测到 44B0X，首先需要进行硬件设置，请点击 Hardware Setting 菜单，出现如下页面：

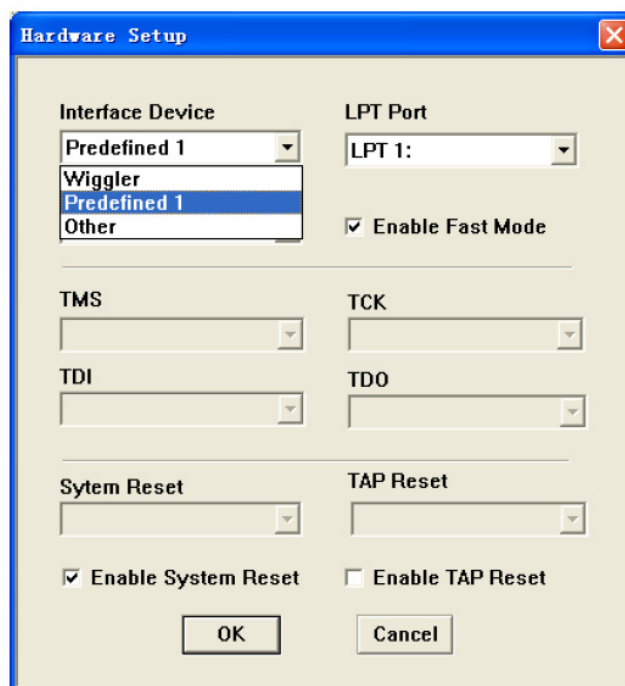


图36，Hardware Setup

BANYAN支持Wiggler和Predefined两种简易JTAG调试工具，由于我们使用的是兼容的简易Wiggler JTAG调试工具，故需在Interface Device里面设置成“Wiggler”，一般默认情况即为“Wiggler”，如果设置不正确将不能找到目标芯片44BOX。**请切记完成该步骤!!!**

在完成硬件设置后即可检测JTAG 工具能否连上目标板上的44BOX 了，请点击Detect 菜单选项，如果连接正确将真正检测ARM7TDMI 核，如果连接错误将出现错误提示，如下图37：

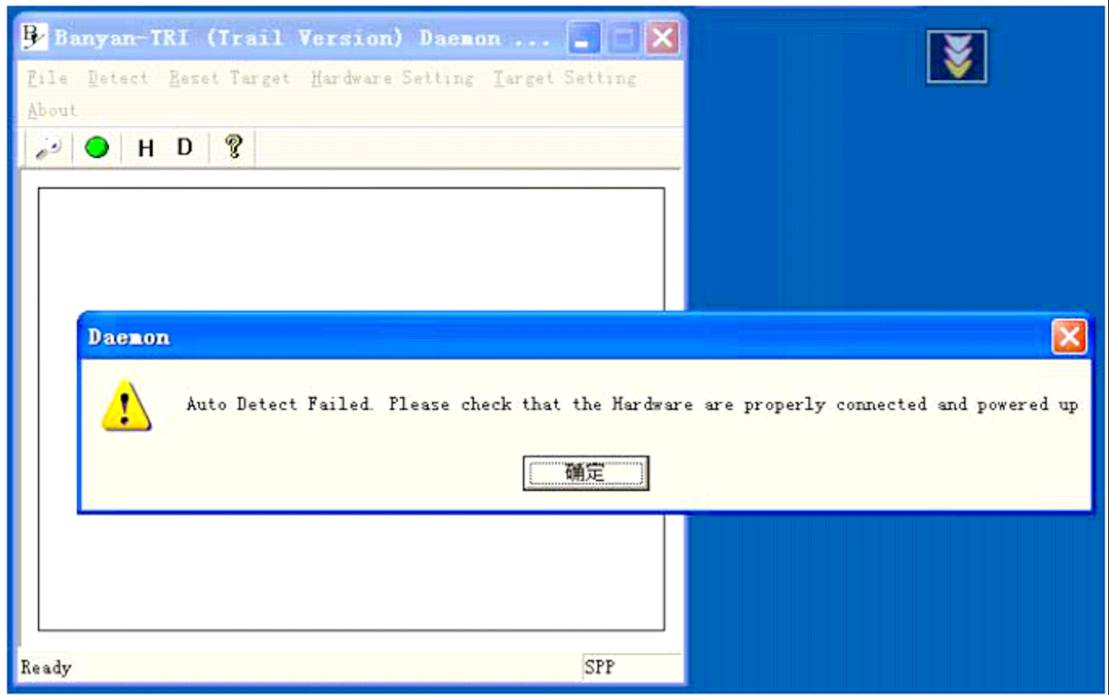


图37，错误提示

如果出现此错误提示请仔细检查电路连接和电源情况，直至按下“Detect”后出现图35的信息为止。

接下来就可以在AXD软件里面通过“Option”->“Configure Target”设置“Target Enviroments”，如下图所示：

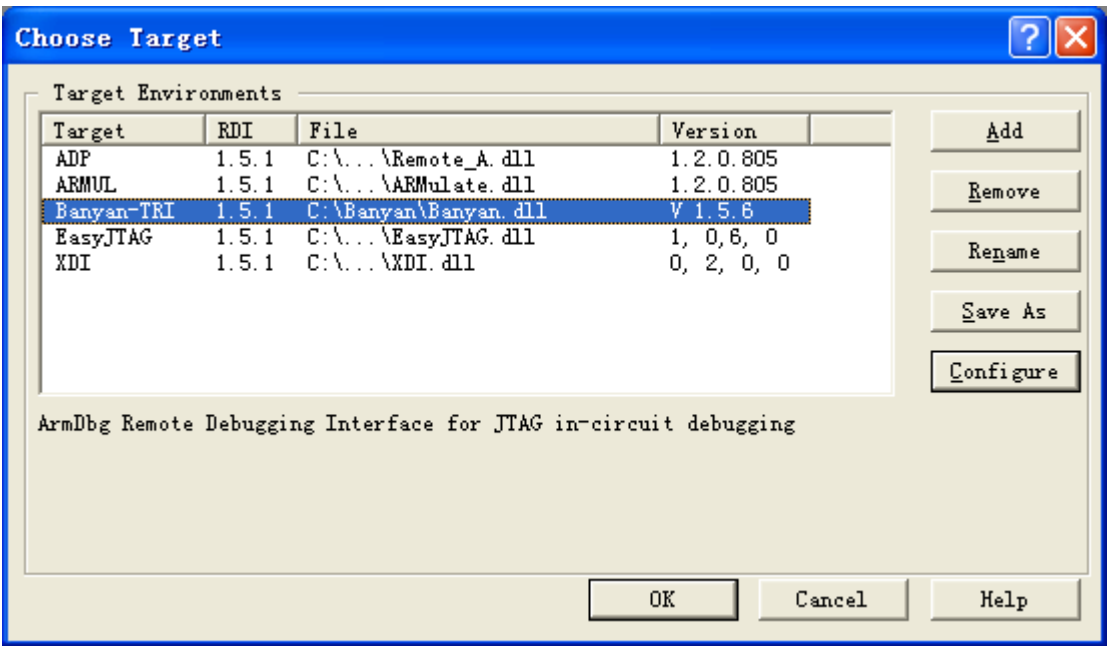


图38，选择Target为BANYAN

接下来的流程就和使用“ARMulator”相差无几了，不过使用试用版本的BANYAN有不少功能限制，具体内容请参照BANYAN软件的用户手册。

最近网上还有一个叫XJTAG的调试代理软件也具备BANYAN的功能，而且是免费的，如果有兴趣可以下载安装并评估。安装和配置过程简要如下：



点击 **Setup.exe** 进行安装，点击两次NEX后即可完成安装。安装完成后通过点击



快捷图标 **XJTAG BETA.1nk** 或者“开始”->“程序”->“XJTAG BETA 0.2”->“XJTAG BETA”运行，该软件类似BANYAN，需要在ADS调用XJTAG的“XDI.DLL”之前先行开启代理软件“XJTAG.exe”。界面如下图39：

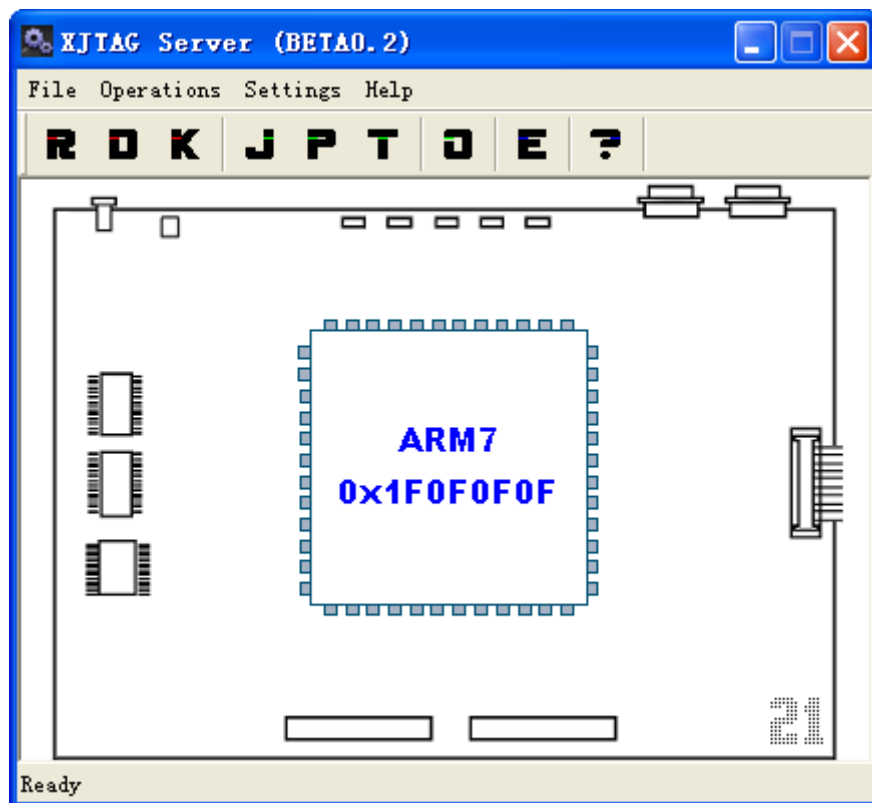


图39，XJTAG代理软件

对于本44B0板无须另行设置，默认设置即可正确运行，如有需要可以通过“Settings”菜单进行“JTAG Settings”、“Port Settings”、“Target Settings”进行设置。其中“JTAG Settings”用于设置JTAG类型，如下图，对于本44B0板应选择Wiggler JTAG、另外“Port Settings”用于设置采用并口1还是2，一般都是1，“Target Settings”用于设置目标芯片的内核，当前XJTAG支持ARM7和ARM9，可以直接选择或者设置成AUTO。

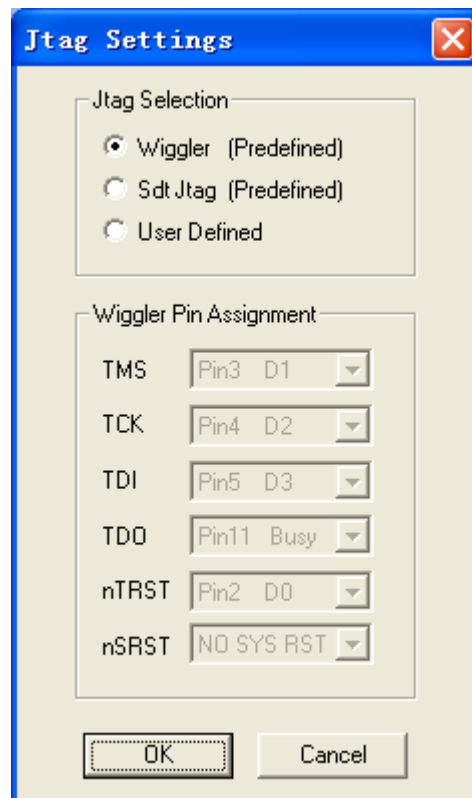


图40, JTAG Settings

另外“Options”可以对复位进行选择，对于本44B0板默认即可，该选项应和实际硬件（JTAG板）相结合进行设置。

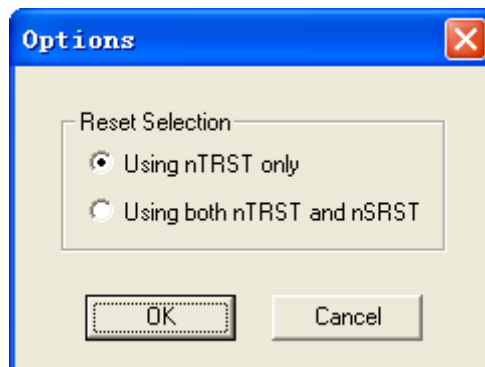


图41, Options

XJTAG在启动的时候即会通过默认配置对目标芯片进行搜索，如果连接正确应该出现如图39所示。如果连接不上则会出现如图42的错误提示，请检查电源和连接然后重新“Detect”直至出现图39的提示。

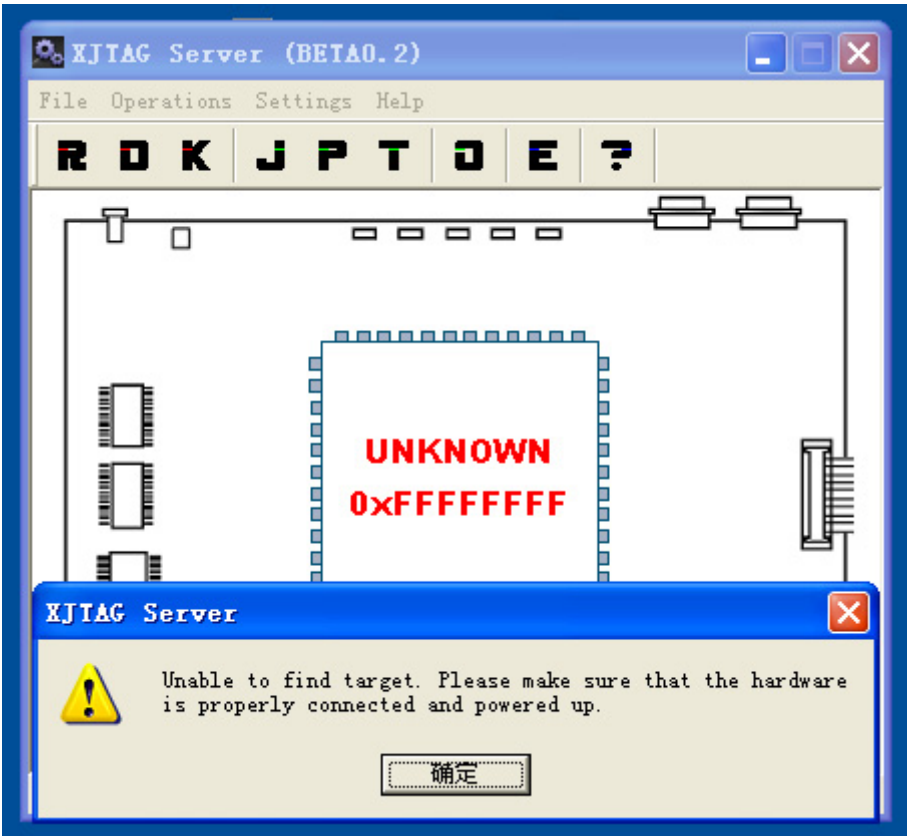


图42, Error

检测到目标芯片后即可在ADS里面进行调试了，只需要在AXD的“Option” -> “Configure Target” 里面设置成“XDI” 即可。如下图所示，XDI即对应XJTAG。

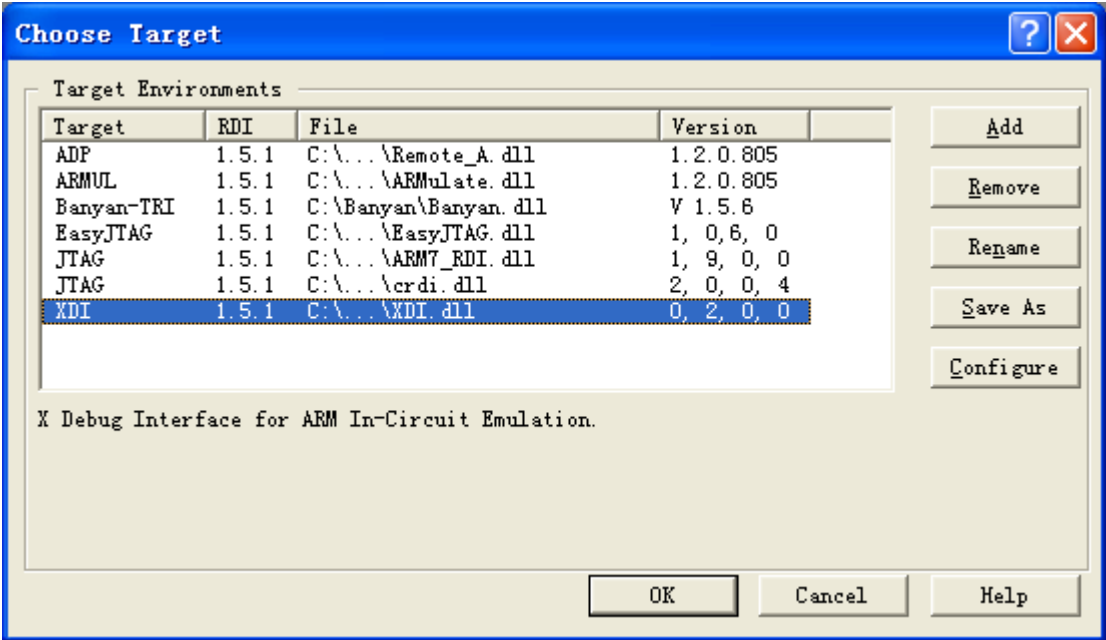


图43, Choose Target

设置完成即可和BANYAN一样进行调试，不过XJTAG没有BANYAN的爬虫直观，在调试过程中不会有其他提示，而BANYAN软件在调试过程中右下角任务栏上的爬虫会闪烁，表示有数据传输。

另外还有一个支持ARM7 ARM9 的Wiggler&SDT Jtag Cable 的RDI接口驱动 crdi.dll(Ver2.0 beta4 2005-5-4),也可以在通过ADS设置然后配合Wiggler进行JTAG 调试。该驱动无须安装，下载后解压，然后在AXD的“Choose Target”菜单下选择“Add”，然后将目录指向该驱动所在位置即可，如图：

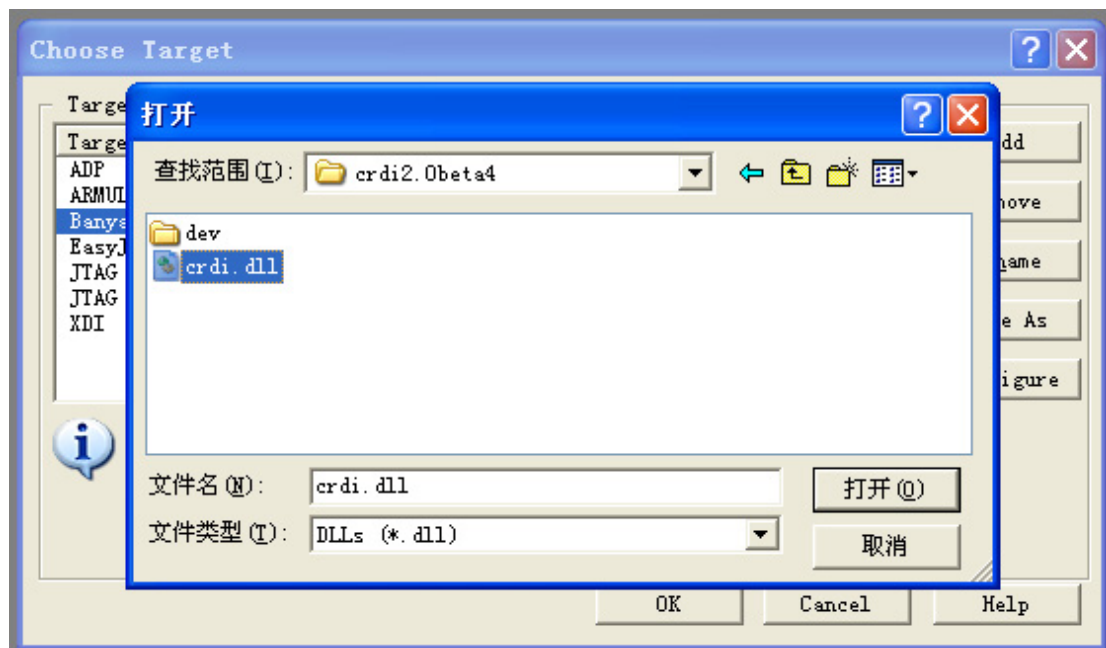


图44，添加“Target”

添加完成后选择“JTAG”，然后点击“Configure”进行配置，出现下图所示内容：

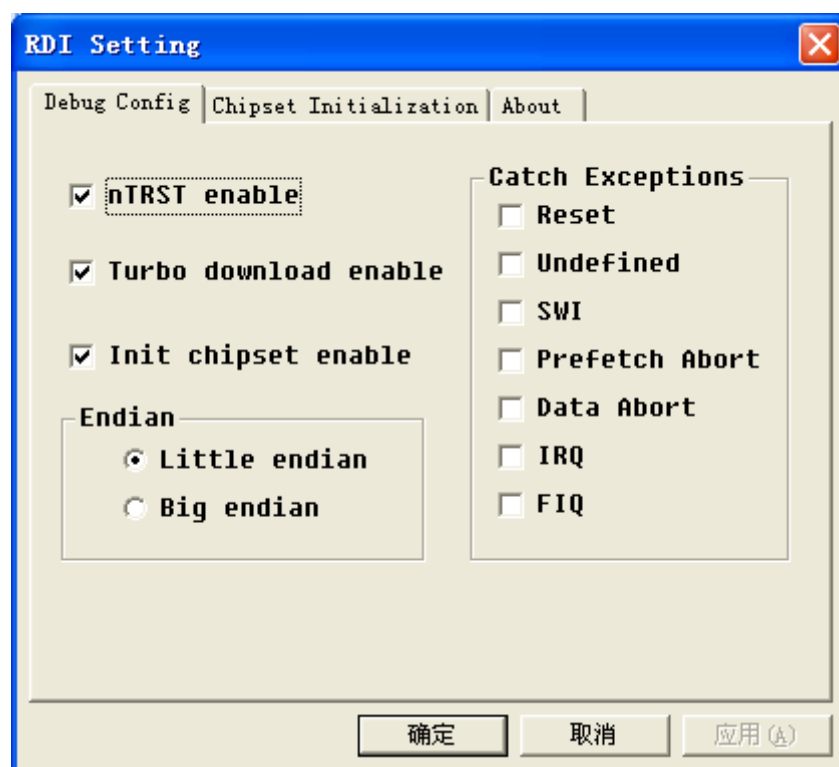


图55，RDI Setting

第一栏为“Debug Config”，可以对复位、下载、初始化、大小端等进行设置。第二栏为“Chipset Initialization”，可以选择或者修改或者自行新建适合自己的目标芯片的SFR列表，当前支持的目标芯片有85个之多，可以选择目标芯片任何参照现有的CFG文件进行修改以适合自己的系统。

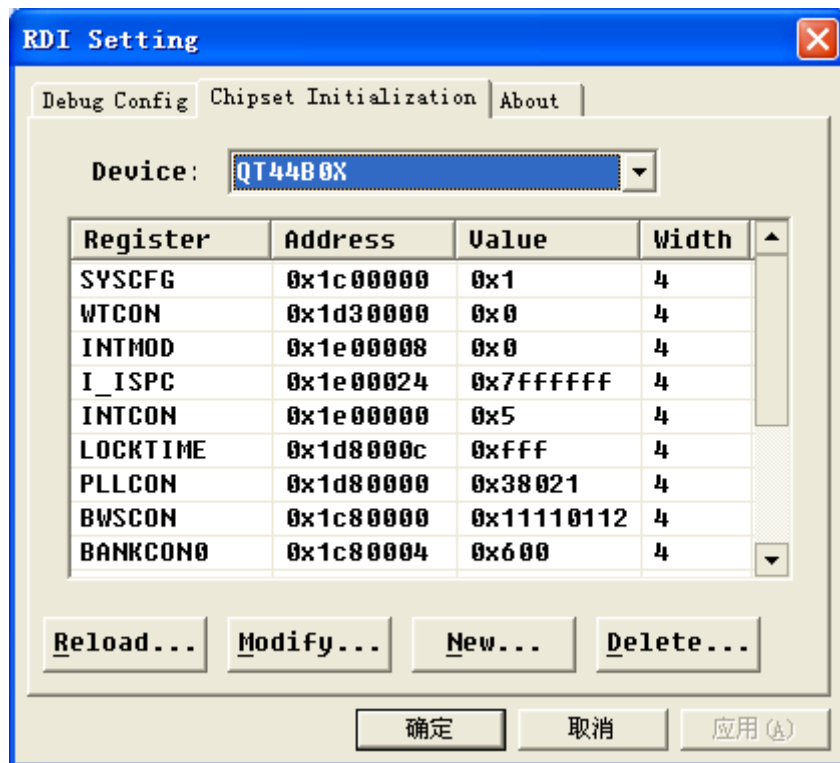


图56, Chipset Initialization

如果配置正确点击确定后AXD应该会提示找到目标芯片，如下图：

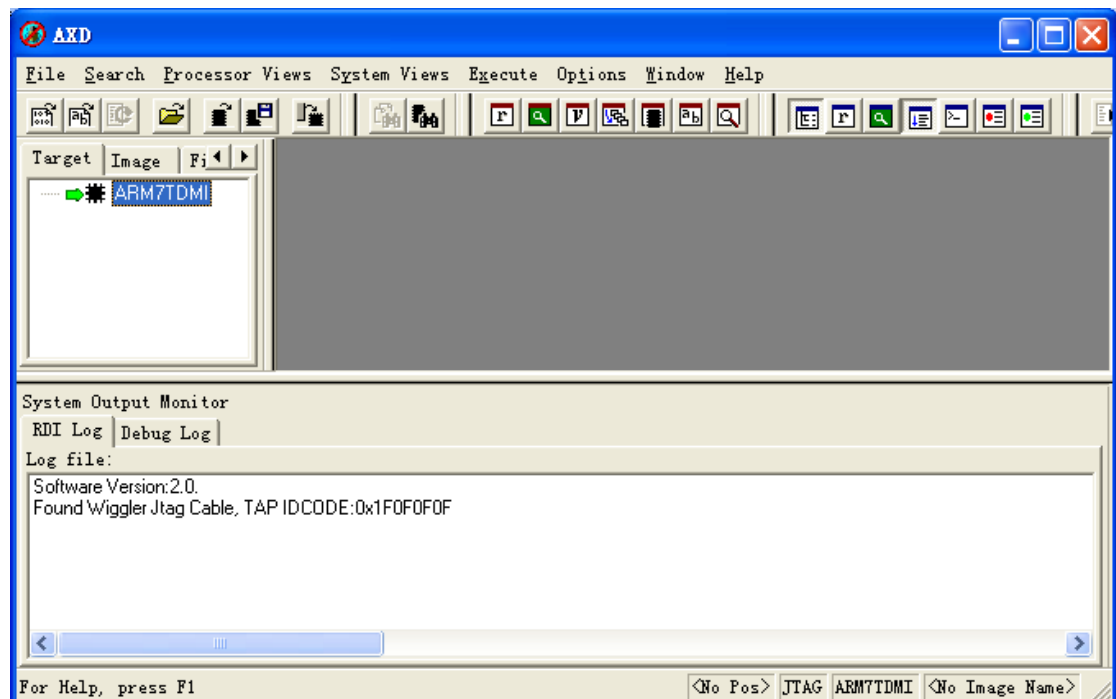


图57，找到44B0

AXD提示找到Wiggler JTAG Cable，并发现代码为“0x1F0F0F0F”，此代码

即为44B0的ID。

由于我们使用的都是简易的Wiggler JTAG工具配合一些免费或者试用版本的第三方软件，而这些工具一般都有Flash下载的限制，不能直接在AXD里面完成代码的下载，而需要另外的第三方软件来实现可执行文件(一般为BIN或HEX文件)的下载，感谢这几年如火如荼的ARM推广运动，使得我们有不少免费的或者有功能限制的FLASH 烧写软件可以使用。下面简单介绍几种比较常见的工具。

首先是最为常见但是也是性能最差的免费软件FLUTED.EXE, FLUTED是一个基于命令行的程序，可以在2KXP的命令提示符下进行操作，其参数如下：

Flash Programming Tool via IEEE1149.1 BSR (Version 0.8a)
Designed By David Geng April, 2000

Usage: <this> command [options]

command: (case insensitive)

R	Read from Flash
W	Write to Flash
E	Erase Flash
V	Verify data in Flash
T	Test scan chain
A	Automatic erase + write

option: (case insensitive)

-V	Verify(off)
-D	Debug mode (off)
-C	FCD file name (default.fcd)
-F	Data file name (flute.bin)
-S	Start address (0, decimal in byte)
-L	Data Length (256 or size of file, decimal in

byte)

也可以利用一些常用的参数，然后编写一个.bat文件，只需要更改一下需要下载的.bin文件名称然后双击运行即可，不过该软件下载速度极慢，只适合下载小型程序，如果有兴趣也可以自己编写一个GUI方便操作。

下面介绍一个下载速度非常快的软件，FLASHP，当前版本是 3.1。

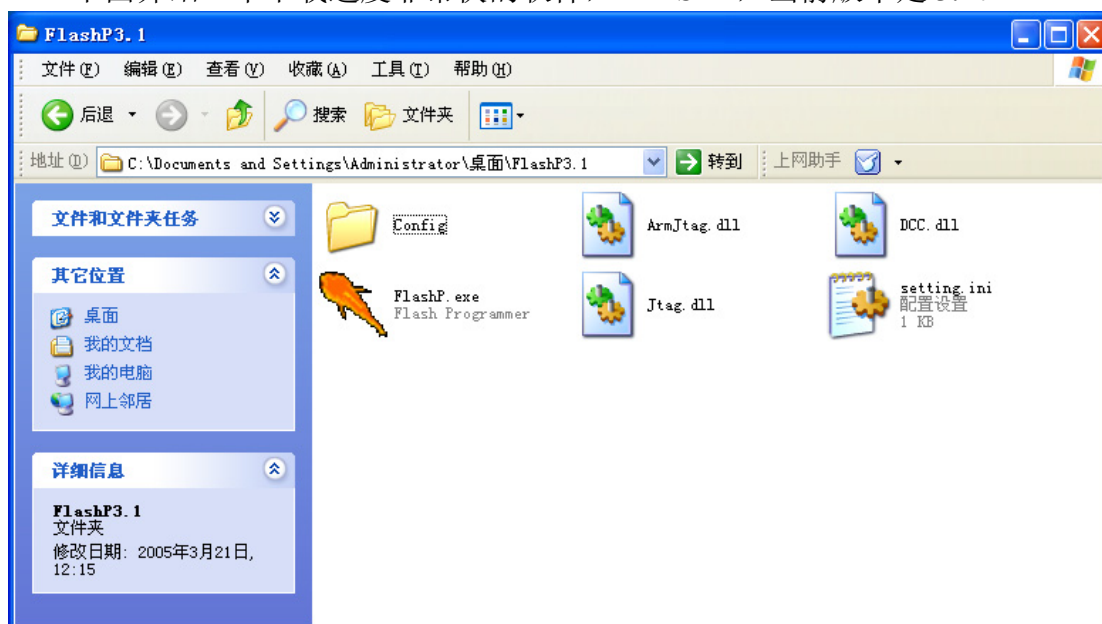


图 58，文件目录

双击 FLASHP.EXE 运行，选择适合本 44B0 开发板的 ARM7E 配置文件：

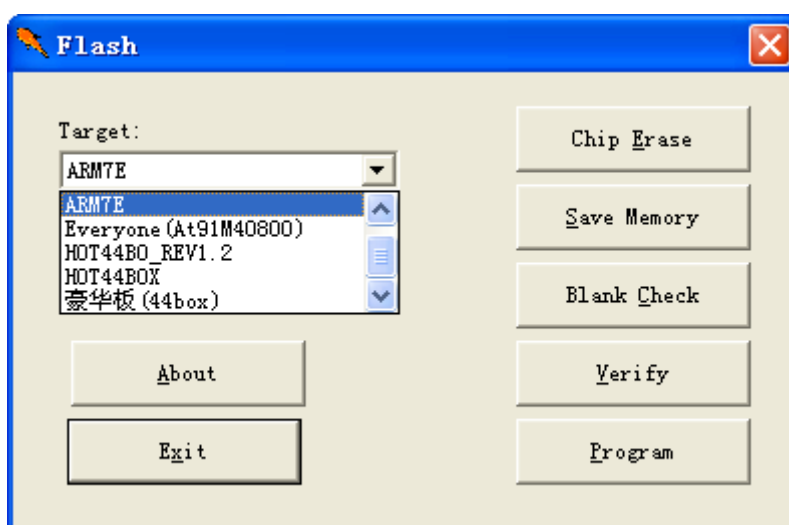


图 59，Target: ARM7E

Cable 可以选择“Wiggler”或者“SDT”，如果不进行选择的话在执行右边的按钮的时候 FLASHP 会自行判别 Cable 类型并进行下一步操作。

一般操作是先“Chip Erase”，然后“Program”，出现图 60 所示内容，“Browse”按钮用于选择需要下载的 bin 文件，还可以设置偏移地址，点击 OK 后开始烧写，如果文件比较大请稍等片刻，烧写完成后会有图 61 所示提示，一般速度可以达到 38K 左右，几个数量级于 FLUTED！

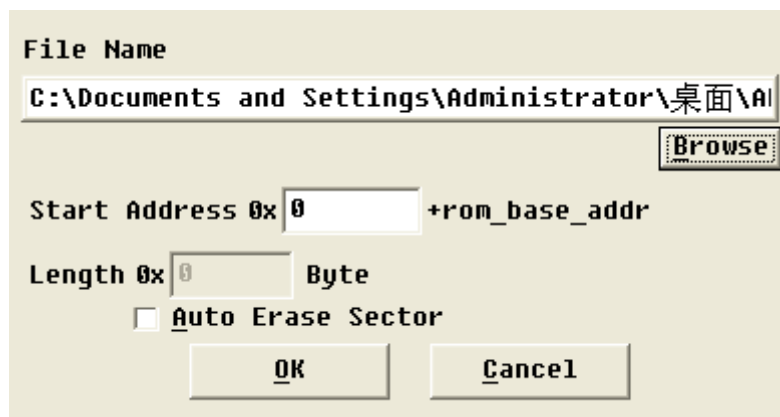


图 60, Program 选项



图 61, 完成报告

FLASHP 还可以通过自行编写（参考示例）或者在线生成一个适合目标系统的 CFG 文件，几乎支持了所有市面上的主流 FLASH。

另外还有一个国外的软件，速度也比较快，不过需要注册，在未注册的情况下只能擦除，不能写入。

Macraigor Systems 公司的 FlashProgrammer 当前版本是 2.5.4，支持非常多的 CPU 和 FLASH，对于本 44B0 板完全可以通过 FlashProgrammer 来进行烧写 FLASH。FlashProgrammer 的具体安装方法请参见该软件所在的根目录下的说明文档，下面简单介绍一下该软件的使用方法。

运行 FlashProgrammer，需要在一些地方进行设置，如下图 61：

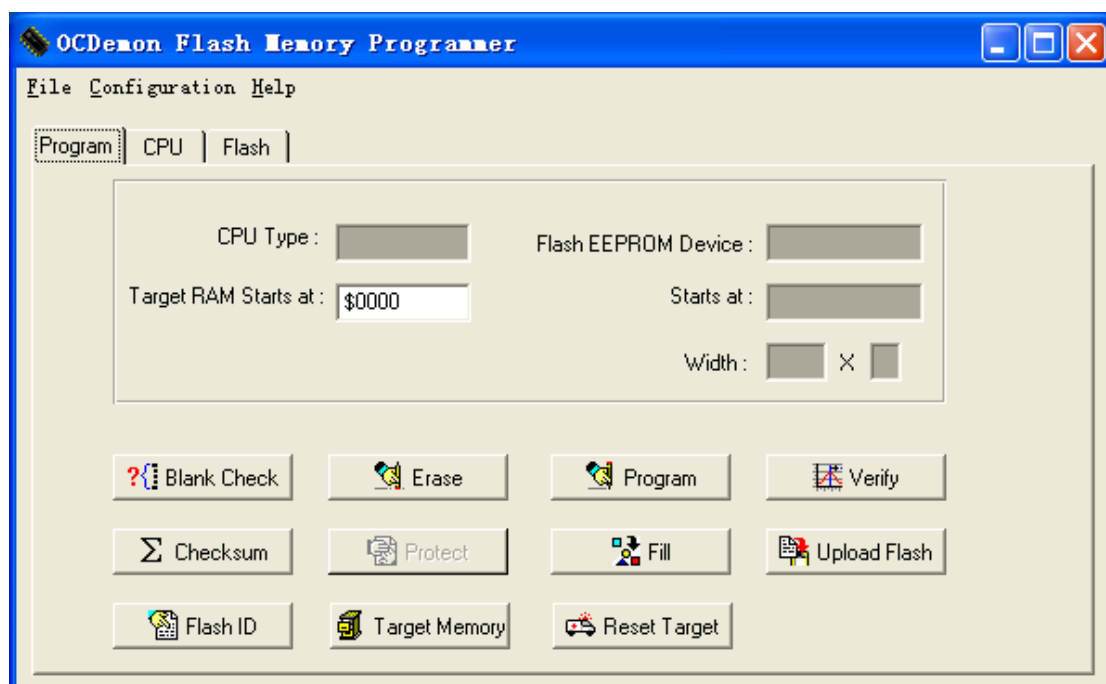


图 61，打开 FlashProgrammer

首先选择 CPU 和 FLASH，对于本 44B0 板 CPU 当然选择 SAMSUNG 的 S3C44B0X，FLASH 则选择 SST 的 SST39VF1601，然后通过“File”->“Open”来选择适合本 44B0 板的配置文件“44B0ForHziee.ocd”，并按照图 62 修改“Target RAM Start At”的值，完成后如下图：

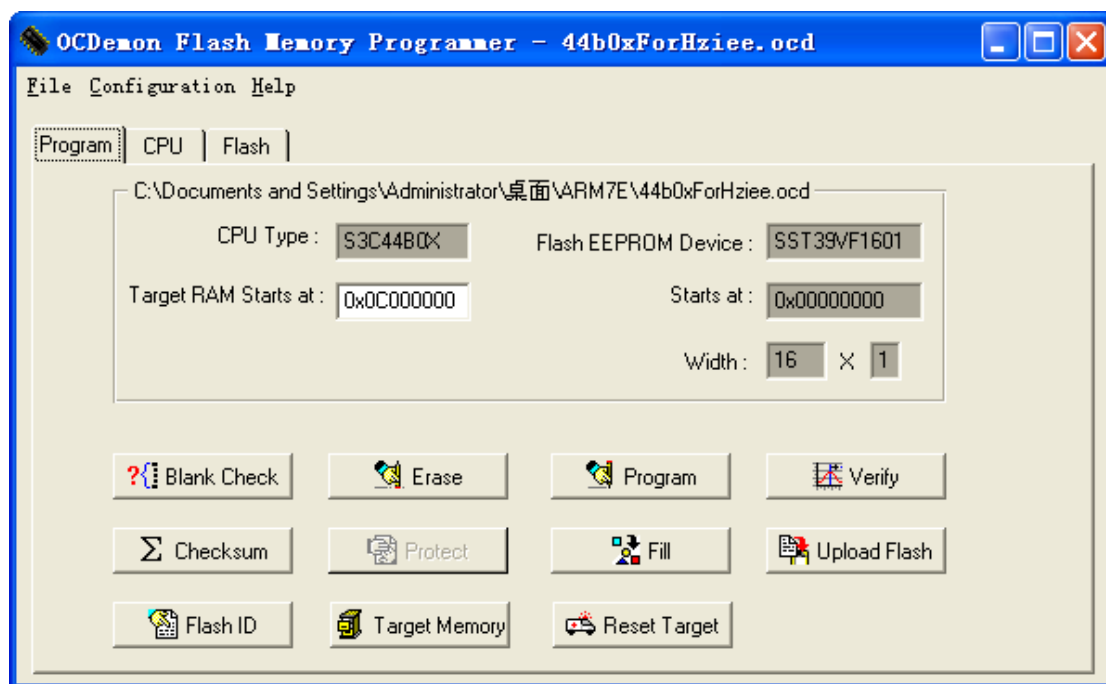


图 62，设置 FlashProgrammer

然后继续进行设置，打开“Configuration”菜单，选择“Communications”，进行设置，如下图所示：

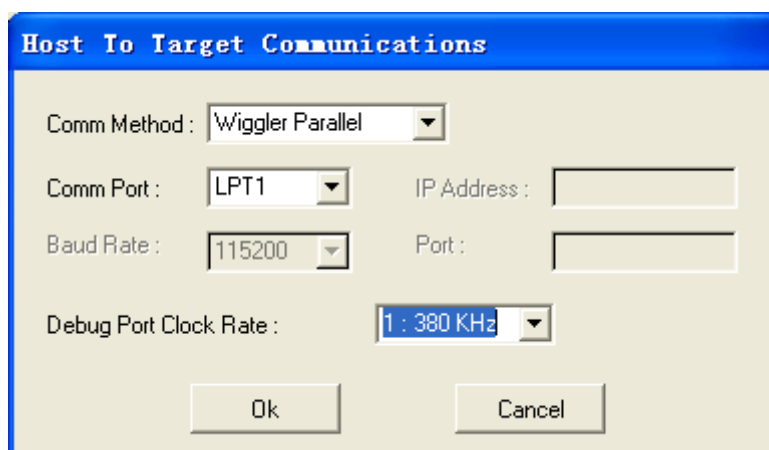


图 63, Communications

选择 Wiggler 并口线，并设置好速度，一般建议设置成较高速度。
接下来设置默认的编程文件的类型，选择 .hex 类型，如图 64:

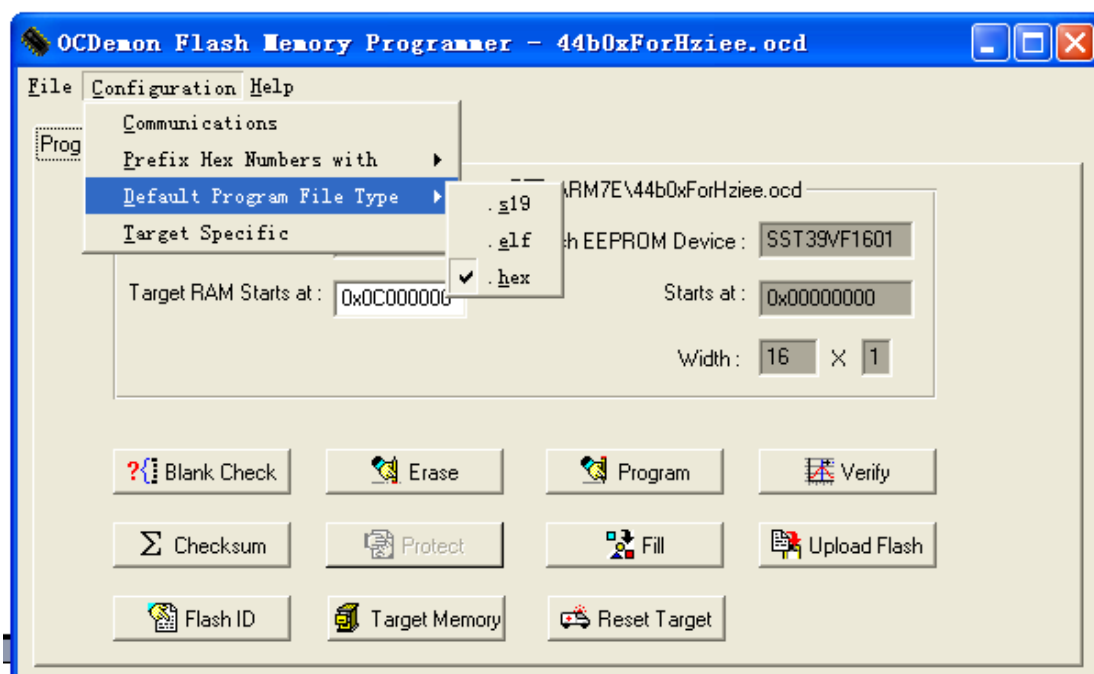


图 64, Default Program File Type

关于其他的设置选项和 OCD 文件的编写可以参照“Help”菜单下的“Contents”。

完成设置后即可开始烧写，一般操作流程是先点击“Flash ID”，以查看是否正确连接，FLASH ID 是否正确。点击按钮后可能需要几秒才会出现如下提示，具体视操作系统版本和软件版本以及 JTAG 速度设置而异。

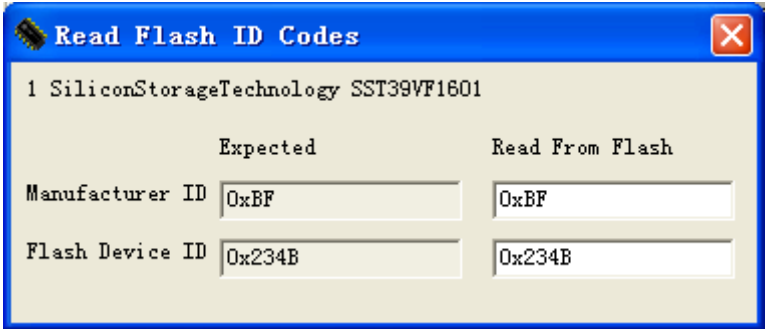


图 65, FLASH ID

如果出现图 64 所示即可进行下一步操作,如出现图 66 所示提示则需要检查硬件连接:



图 66, Disconnected

请仔细检查硬件直到出现图 65 提示为止。

FLASH ID 读取正确后即可以进行下一步操作,一般流程是先查空再擦除再编程再校验。

首先查空, 点击“Blank Check”出现下图所示, 选择查空区域点击“Blank Check”按钮, 软件将提示目标 FLASH 是否为空, 如果非空将告之第一个非空地址, 如下图。

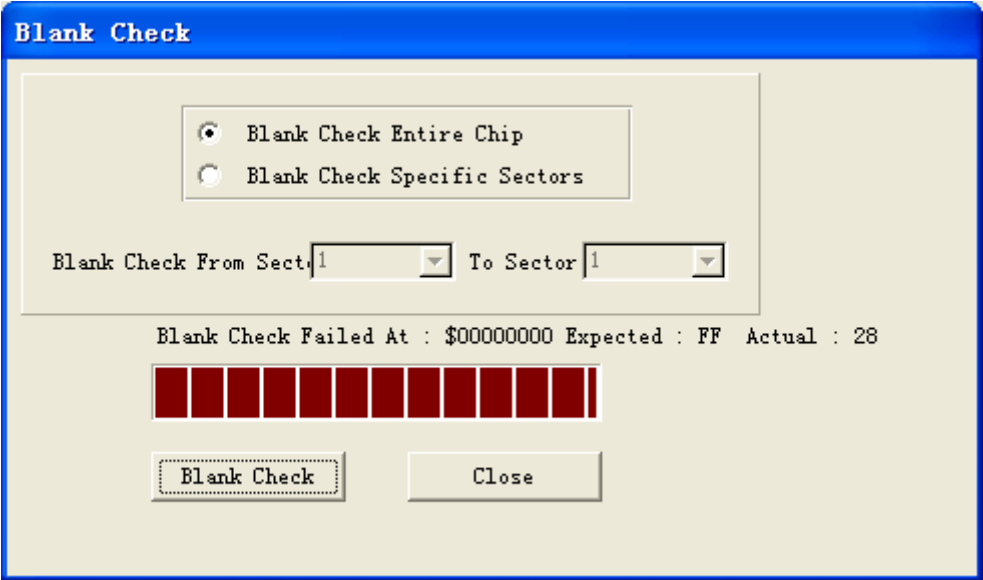


图 67, 查空

如果非空则在编程前请先点击“Erase”进行擦除, 如下图:

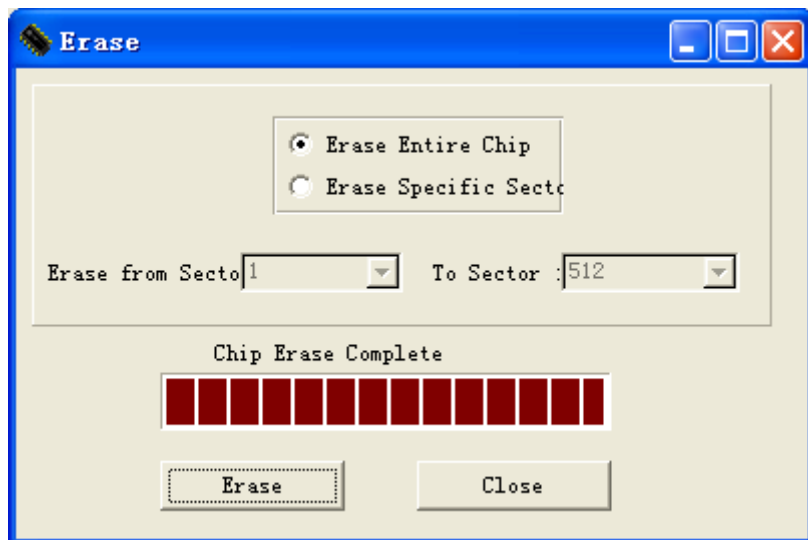


图 68, Erase

选择整片擦除。

擦除完成后即可使用编程功能，

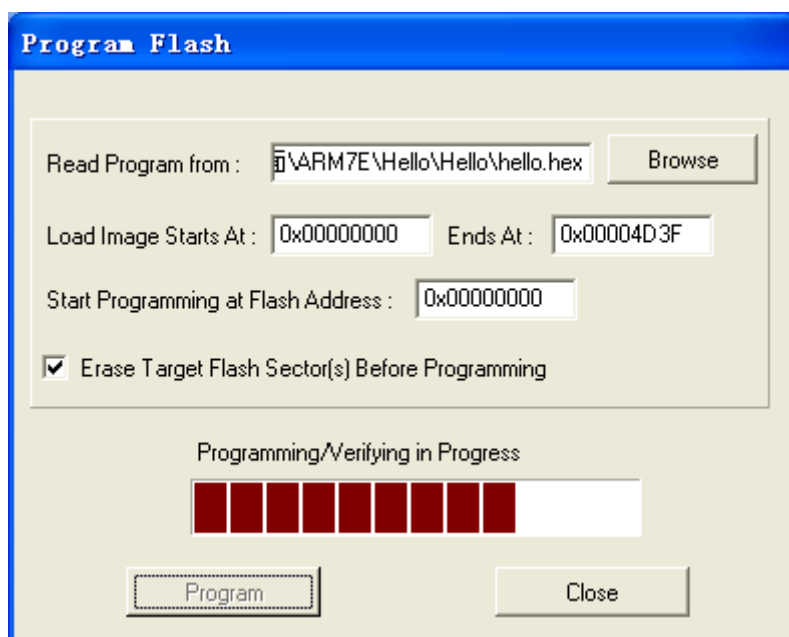


图 69, Program Flash

建议将“Erase Target Flash Sector Before Programming”前的勾打上。

注意：FlashProgram 支持.hex 文件，FlashP 支持.bin 文件，选择好 FLASH 烧写工具后请再在 ADS 里面设置输出的文件格式。

到此范例“Hello world!”全部完成。

这里简单总结一下整个过程：

- 1, 建立工程，添加和新建文件；
- 2, 设置工程，包括 RO, RW 地址，入口地址，Layout，输出调试结果等；
- 3, 编译除错，按照编译结果提示进行除错，直至编译通过；
- 4, Make，注意，如果文件是拷贝过来的话请先“Remove Object Code”，并更改相应输出结果（BIN 或 HEX 文件）的设置（格式、位置）；
- 5, 软件调试，可以直接跳过进入 JTAG 调试；
- 6, JTAG 调试，请选择好调试工具，如果是 BANYAN 需要先把 BANYAN 调试代理开启；
- 7, FLASH 烧写，不同的烧写软件支持不同的文件格式，如 FLASHP 对应 BIN 文件，FLASHPROGRAM 对应 HEX 文件，请根据不同软件生成不同可执行文件；
- 8, 复位、运行。

Enjoy!

附录一：ADS 开发环境详解

一、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。

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 链接器生成的是 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。

2 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。

3 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)] [-xref to
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** 也可能会产生 **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数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 **RO** 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。

-xref to 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 **-xref to** 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**，**-xref to**，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**，**-xref to**，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。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。

3. 添加为解析尚未解析的引用的库到库文件列表。

二、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。

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。

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。

附录二、AXD 运行调试工具条说明

AXD 运行调试工具条如图13 所示，调试观察窗口工具条如图14 所示，文件操作工具条如图1所示。

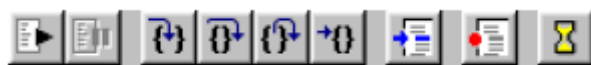


图1 运行调试工具条



全速运行(Go)



停止运行(Stop)



单步运行(Step In)，与Step 命令不同之处在于对函数调用语句，Step In 命令将进入该函数。



单步运行(Step)，每次执行一条语句，这时函数调用将被作为一条语句执行。



单步运行(Step Out)，执行完当前被调用的函数，停止在函数调用的下一条语句。



运行到光标(Run To Cursor)，运行程序直到当前光标所在行时停止。



设置断点(Toggle BreakPoint)



图2 调试观察窗口工具条



打开寄存器窗口(Processor Registers)



打开观察窗口(Processor Watch)



打开变量观察窗口(Context Variable)



打开存储器观察窗口(Memory)



打开反汇编窗口(Disassembly)



图3 文件操作工具条



加载调试文件(Load Image)



重新加载文件(Reload Current Image)。由于AXD 没有复位命令，所以通常使用Reload 实现复位(直接更改PC 寄存器为零也能实现复位)。

附录三、44B0板的存储空间分配和原理图

下图为44B0的存储空间地址分配图：

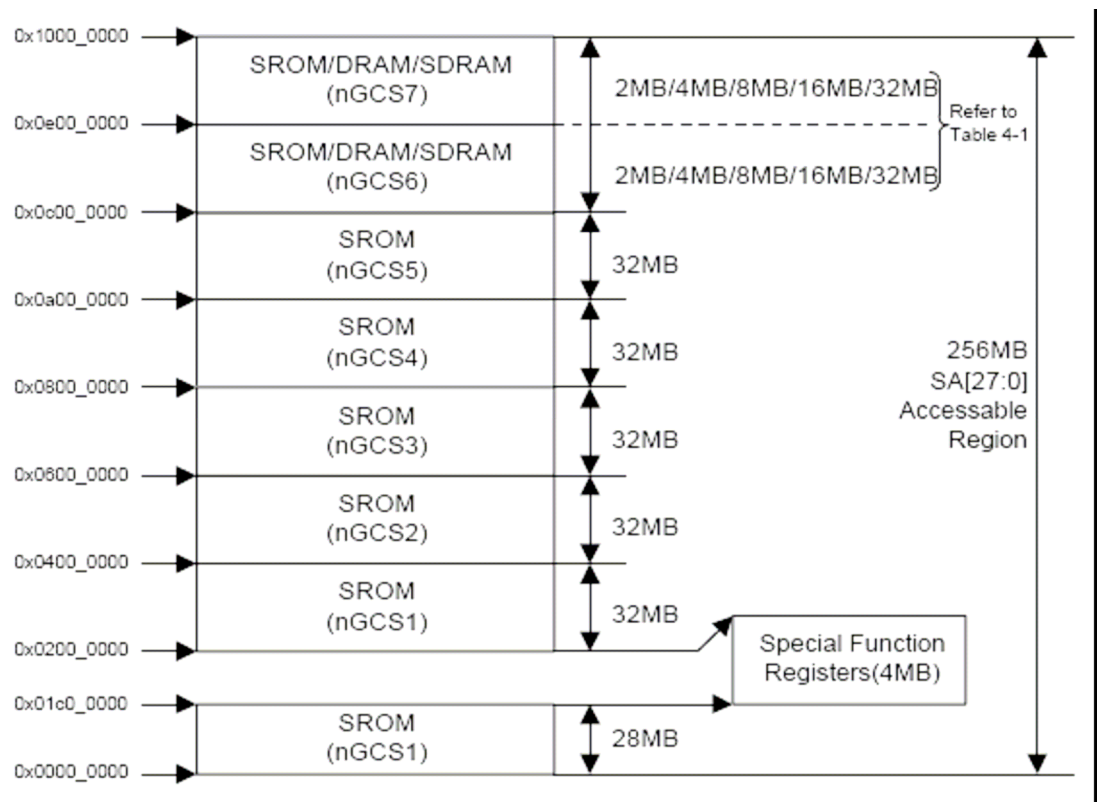


图 1， S3C44B0 复位后的存储器地址分配

本 44B0 板的 FLASH 型号为 SST39VF1601，地址空间从 0x00000000 到 0x00200000 共 16Mbit，即 2MB 字节，位于 Bank0。

本44B0板的SDRAM为HY57V641620，地址空间从0x0c000000到0x0c800000 共4Bank*1M*16bit，即8MB字节，位于Bank6。

附录四、参考资料

本手册仅介绍了一些基本的软件的操作方法和一些基础知识，如果需要很详尽的学习ADS和编程细节，有以下列表可供参考：

0, Getting Started.pdf: 快速上手；

This book provides an overview of the ADS tools and documentation.

1, CodeWarriorIDEGuide.pdf: CodeWarrior IDE指南，详细介绍了关于IDE的各种操作；

This book provides user information for *the CodeWarrior IDE for the ARM Developer Suite*. It describes the major graphical user interface components of the CodeWarrior IDE, and provides information on ARM-specific features.

2, ADS_AsemblerGuide.pdf: ARM汇编指南；

This book provides tutorial and reference information for the ADS assemblers (armasm, the free-standing assembler, and inline assemblers in the C and C++ compilers). It describes the command-line options to the assembler, the pseudo-instructions and directives available to assembly language programmers, and the ARM, Thumb®, and *Vector Floating-point* (VFP) instruction sets.

3, ARM Architecture Reference Manual.pdf: ARM体系结构参考手册；

The purpose of this manual is to describe the ARM instruction set architecture, including its high code density Thumb subset, and two of its standard coprocessor extensions:

- The standard System Control coprocessor (coprocessor 15), which is used to control memory system components such as caches, write buffers, Memory Management Units, and Protection Units.
- The *Vector Floating-point* (VFP) architecture, which uses coprocessors 10 and 11 to supply a high-performance floating-point instruction set.

4, AXD and armsd Debuggers Guide.pdf: AXD和armsd调试器指南；

This book has two parts that describe the currently supported ARM debuggers:

- Part A describes the graphical user interface components of *ARM eXtended Debugger* (AXD). This is the most recent ARM debugger and is part of the *ARM Developer Suite* (ADS). Tutorial information is included to demonstrate the main features of AXD. If AXD is the only debugger you use, you can safely ignore Part B, but you might have to refer to the Appendixes, Glossary, and Index at the end of the book.
- Part B describes the *ARM Symbolic Debugger* (armsd).

5, Compilers and Libraries Guide.pdf: 编译器和库指南；

This book provides reference information for ADS. It describes the command-line options to the compilers. The book also gives reference material on the

ARM implementation of the C and C++ compilers and the C libraries.

6, Debug Target Guide.pdf: 目标调试指南;

This book provides reference information for the *ARM Developer Suite* (ADS). It describes:

- ARMulator®, the ARM processor simulator
- Semihosting SWIs, the means for your ARM programs to access facilities on your host computer.

7, Linker and Utilities Guide.pdf: 连接器和效用指南:

This book provides reference information for ADS. It describes the ommand-line options to the linker and other ARM tools in ADS.

8, Developer Guide.pdf: 开发者指南:

This book provides tutorial information on writing code targeted at the ARM family of processors.

附录五、使用技巧和注意事项

在使用过程中有一些需要注意的小事情和小技巧，下面分列一二：

一、Codewarrior使用小技巧：

1，打开已经存在的工程文件：

可以直接通过“File”->“Open”来打开也可以直接找到工程文件（*.mcp）然后双击（前提是已经安装ADS V1.2），有时候双击打开的时候会出现这样的错误提示：

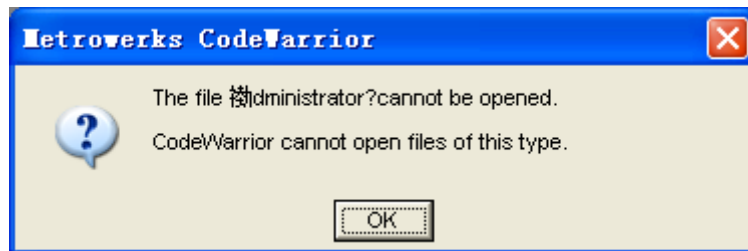


图1、错误提示

这表明该工程文件（*.mcp）的路径包含了中文，所以不能直接通过双击打开，这时候只能通过“File”->“Open”来打开。所以建议不要将工程放在中文路径下。

2，利用Codewarrior快速查找：

Codewarrior是一个很优秀的IDE，在Codewarrior里面查找函数原型，查找包含文件都非常方便，如下图所示：

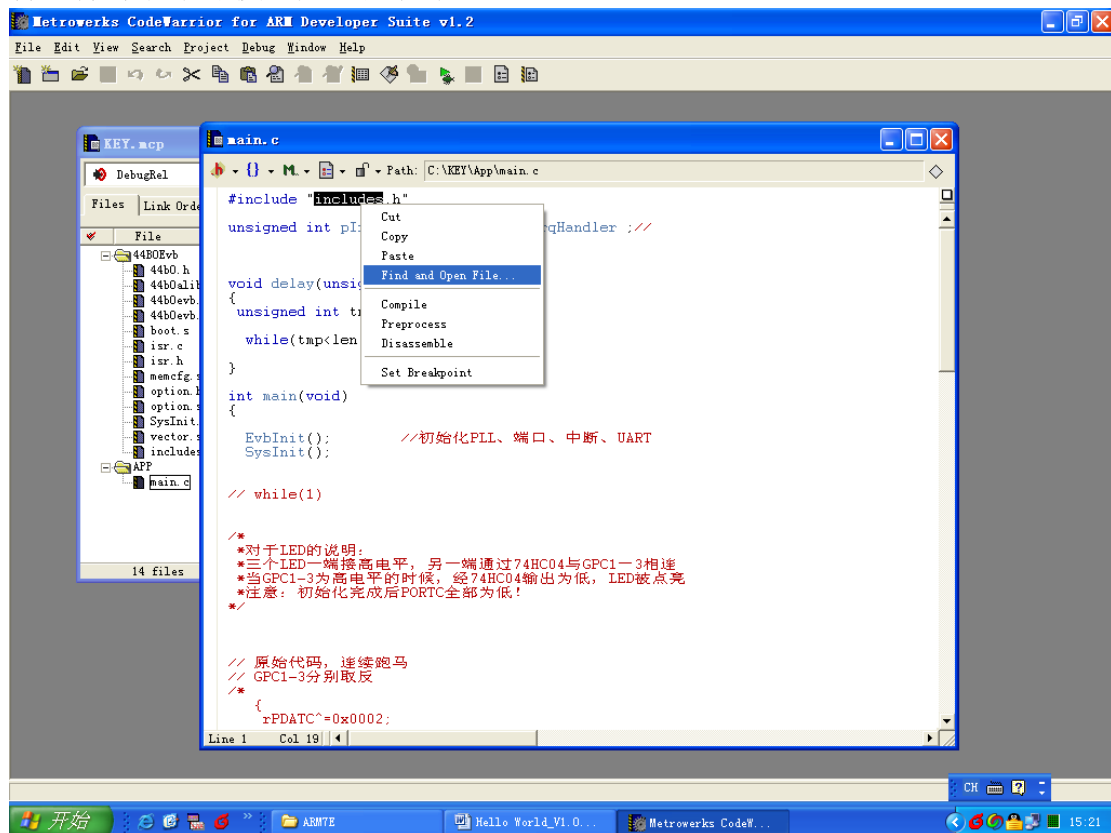


图2、查找文件

版本历史:

V1.1:

对V1.0中存在的部分小错误和描述不当的地方进行修改，正式发布版本，
2005-08完成

V1.0:

原始版本，2005-07完成



Powered by XiaoGuo's publishing Studio

QQ:8204136

Website: www.mcuzone.com

2005