

## 实验一 MDK-ARM 软件与 ARM 处理器基本编程

### 1. MDK-ARM 开发工具包简介

MDK-ARM（ARM 微控制器开发工具包）是 ARM 公司推荐的用于基于 ARM 处理器的微控制器的完整软件开发环境。MDK-ARM 由 Keil 公司（已被 ARM 公司收购）提供，利用了该公司先进的  $\mu$ Vision 集成开发环境，适用于基于 Cortex™-M、Cortex-R4、ARM7™ 和 ARM9™ 处理器的微控制器芯片开发。MDK-ARM 专为微控制器应用程序而设计，易于学习和使用，同时具有强大的功能，能满足大多数要求苛刻的嵌入式应用程序的需求。

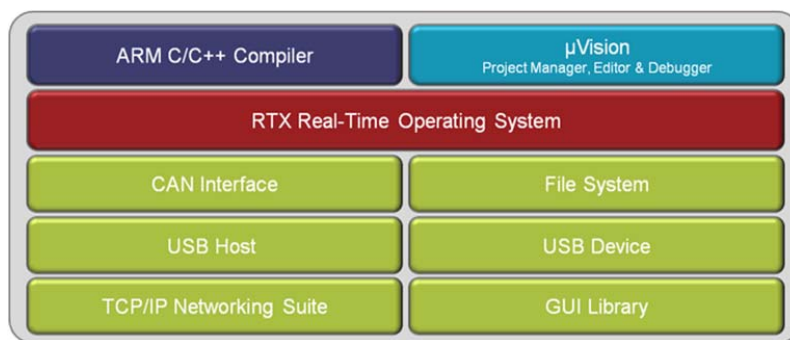


图 1 MDK-ARM 模块示意图

MDK-ARM 微控制器开发工具包的主要性能和特征包括：

- 完全支持 Cortex-M、Cortex-R4、ARM7 和 ARM9 微控制器芯片
- 集成行业领先的 ARM C/C++ 编译工具链

- 集成  $\mu$ Vision 集成开发环境，包括项目管理器、编辑器和调试器
- 集成 Keil RTX 确定性、空间占用小的实时操作系统（开源代码）
- 集成 TCP/IP 网络套件，提供多种协议和各种应用程序
- 为 USB 设备和 USB 主机堆栈配备标准驱动程序类
- ULINKpro 支持对正在运行的应用程序进行即时分析并记录执行的每条 Cortex-M 指令
- 可提供执行程序的完整的代码覆盖率信息
- 集成执行性能分析器和性能分析器支持程序优化
- 提供大量示例项目可帮助快速熟悉 MDK-ARM 强大的内置功能
- 符合 CMSIS Cortex 微控制器软件接口标准

MDK-ARM 具有四种版本：MDK-Lite、MDK 基础版、MDK 标准版和 MDK 专业版。所有版本都提供完整的 C/C++ 开发环境和调试支持，MDK 专业版主要包括了丰富的中间件库。其中，MDK-Lite 版提供免费下载，不需要序列号或许可证密钥，但是其所能开发的程序大小限制在 32KB 以内。下载网址在 <http://www.keil.com/arm/mdk.asp>。

## 2. $\mu$ Vision 集成开发环境简介

## μVision 集成开发环境（Integrated Development Environment, IDE）

基于窗口设计，是集项目管理、源代码编辑、编译汇编、程序调试和全功能仿真于一体的集成开发环境，可以帮助开发者快速创建嵌入式程序开始嵌入式软件开发。μVision IDE 的基本界面环境如图 2 所示。

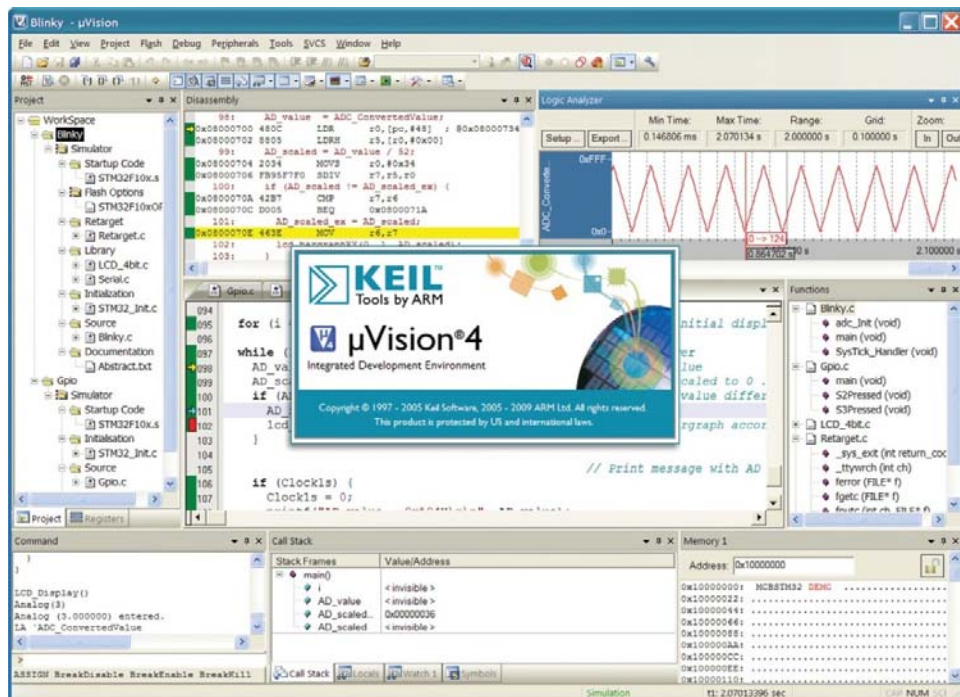


图 2 μVision IDE 软件环境界面

如图 3 所示，μVision IDE 集成有多种嵌入式程序开发所需的模块，包括项目管理器、源代码编辑器、Make 工具链、程序仿真调试器等，并能完成闪存 Flash 编程功能。以下，我们重点介绍其中的器件数据库、Make 工具链和程序仿真调试器。

- 器件数据库

μVision 器件数据库提供了一种便捷的方式来选择和配置 ARM 微控制器芯片与项目参数。它包括了预配的设置，因此可以使软件开发者完全集中于应用需求。此外，我们还可以添加自己的芯片器件，或者改变已有的设置。

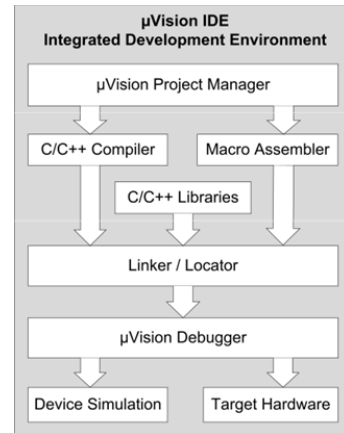


图 3 μVision IDE 嵌入式程序模块

- Make 工具链

针对 MDK-ARM 平台，μVision IDE 在 Make 工具链中集成了完整的 ARM 编译工具（之前也被称为 ARM RealView 编译工具），包括：

- armcc: ARM C/C++编译器，将 C/C++程序源代码翻译成可执行的机器语言，可以同时支持 ARM 和 Thumb 指令的生成，具有代码大小和性能优化功能，支持内嵌汇编器、内联函数、CPU 寄存器参数传递和可重入运行时类库，并支持 IEEE-754 兼容的单双精度浮点数据处理。
- MicroLib: Microlib 类库，专门针对基于 ARM 的嵌入式应用 C 语言程序开发的高度优化的库。与 ARM 编译工具链集成的标准

C 库相比，Microlib 提供了对于多种嵌入式系统需要的重要的代码大小优势。

- **armasm**: ARM 宏汇编器，将汇编语言程序翻译成可执行的机器语言，支持标准的宏处理和条件汇编控制。
- **armLink**: ARM 连接器，将多个将一个或多个由编译器或汇编器生成的目标文件外加库连接成为可执行文件或新的库文件，支持静态栈分析。
- **armar**: ARM 库管理工具，可以在标准格式 ar 库内有效管理和维护多个 ELF 目标文件集合。
- **fromelf**: ARM ELF 工具，可以帮助处理由编译器、汇编器和连接器生成的 ARM ELF 目标文件和映像文件。
- **μVision Debugger 调试器**

μVision Debugger 调试器完全集成于 μVision IDE 集成开发环境之中，可以实现以下功能：

- 对 C/C++源代码或汇编级代码进行反汇编，并在各种单步执行和视图模式下，随着程序执行将反汇编代码与源代码相对应。
- 多种断点设置方式，包括访问和复杂断点。

- 支持设置书签，便于快速查找和定义关键点。
- 查看和修改内存、变量和寄存器的值
- 列出包括栈变量在内的程序调用树
- 查看微控制器片上外设的状态
- 支持调试命令或类似 C 的脚本函数功能
- 执行分析功能可以显示执行时间，以及每条指令所需执行周期
- 为安全攸关应用测试进行代码覆盖率分析
- 多种分析工具、指令跟踪能力和窗口调整功能

μVision 调试器支持两种操作模式：仿真器（**Simulator**）模式和目标（**Target**）模式。

仿真器模式将 μVision 调试器配置成纯软件来精确的仿真目标系统，包括指令和大多数片上外设。在此模式下，我们可以在还没有硬件的情况下测试应用代码，便于我们能够快速开发可靠地嵌入式软件。仿真器模式支持：

- 不需要硬件环境在桌面上就可以进行软件测试
- 基于功能的早期软件调试来改善软件可靠性
- 支持硬件调试器无法实现的断点功能

- 理想的输入信号，没有硬件调试器的噪声
- 在信号处理算法过程中单步执行
- 检测出那些会损坏真实硬件外设的失效场景

目标模式将  $\mu$ Vision 与真实硬件连接起来，可以支持对目标开发板的调试，包括单步执行、设置断点和查看内存等有用的调试方法。本次实验主要使用仿真器仿真对软件进行调试，关于使用目标模式调试开发板在下一次实验中重点介绍。

### 3. 使用 $\mu$ Vision 集成开发环境

本小节，我们开始熟悉  $\mu$ Vision IDE 的软件界面。通过开始菜单启动 Keil  $\mu$ Vision。如图 4 所示， $\mu$ Vision IDE 是一个完全基于窗口的程序开发环境，支持窗口的重排、拖拽等窗口程序基本操作。与通常的 Windows 软件相似， $\mu$ Vision IDE 软件界面包括基本的菜单栏（Menu bar）、工具栏（Toolbars）以及状态栏（Status bar），此外还包括断点和书签设置、命令行提示与输入等功能。

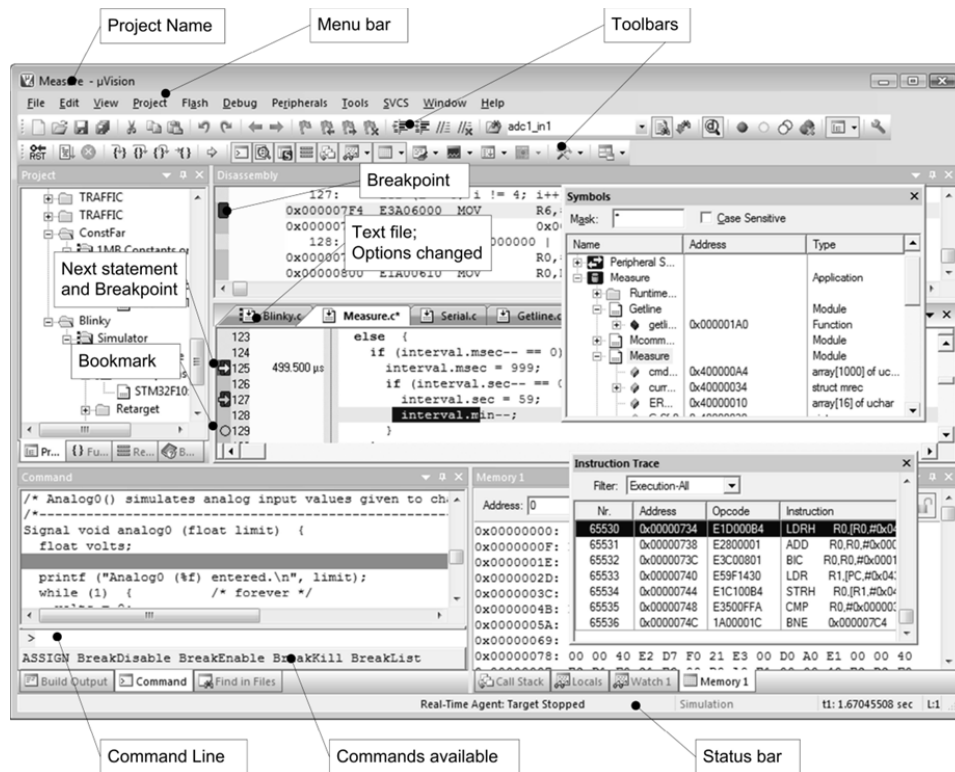


图 4 µVision IDE 软件界面指示图

为了便于理解，我们将 µVision IDE 中间的主显示区域定义划分为以下三个窗口区域，如图 5 所示。

- 项目窗口区域：位于屏幕显示的左侧部分，默认显示项目窗口、函数窗口、图书窗口和寄存器窗口。
- 编辑器窗口区域：位于屏幕显示的中间和右侧部分，可以修改源代码、查看性能和分析信息，并检查反汇编代码。



- 输出窗口区域：位于屏幕显示的下部，提供与调试、存储器、定义符号、调用栈、局部变量、命令、浏览信息以及文件查找结果等相关的信息。

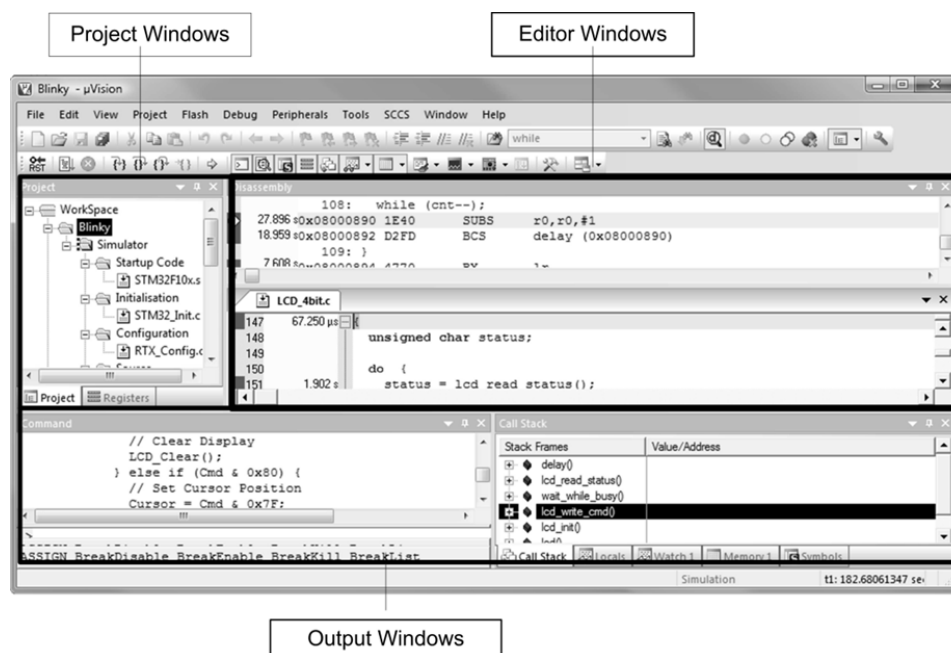


图 5 窗口区域划分示意图

窗口的位置和大小可以根据需要调整。必要时，我们可以通过调用 Window – Reset Current Layout 菜单，将窗口排布恢复到默认状态。

μVision IDE 有两种工作模式：构建（Build）模式和调试（Debug）模式。针对两种不同的模式，屏幕设置、工具栏设置以及项目选项的也相应不同。文件工具栏在两种模式下都会使能，而调试工具栏与构建工具栏分别在各自对应的模式下使能。所有按钮、图表和菜单按照工作模式而使能，当不可用时则显示为灰色。

构建模式为标准工作模式。此模式下，我们可以编写应用代码，配置项目，选择目标硬件和器件。同时，我们还将编译、汇编和连接源程序，修正可能的错误，最终生成可执行的目标文件。

调式模式下主要进行软件代码的仿真与调试。在调试模式下，我们也可以修改一些通用设置并编辑源代码文件，但是这些修改只能在返回构建模式后才能有效。只有那些与调试相关的设置才会立即生效。

$\mu$ Vision IDE 的大部分功能都可以通过调用相应的菜单或工具按钮来实现。这些菜单与工具栏的操作与大多数 Windows 程序很类似，我们将在后面的实验中逐步学习和熟悉。具体的说明请参见参考文献<sup>[1]</sup>。

#### 4. 创建嵌入式程序

创建一个新的嵌入式项目包括如下几个步骤：

##### 1) 新建项目文件

从菜单栏选择 **Project – New  $\mu$ Vision Project**，打开一个标准对话框，如图 6 所示。在对话框中为新建项目选择所在文件夹和项目文件名称，单击 **Save** 按钮。建议为每一个新建项目选择一个独立的文件夹，以便于项目文件管理。

单击 Save 按钮后， $\mu$ Vision 弹出 Select Device 对话框，需要我们选择项目的目标微控制器型号。Select Device 对话框中列出了  $\mu$ Vision 器件数据库中的所有微控制器芯片型号。在项目开发中，我们也可以通过 Project – Select Device for Target 菜单调出该对话框。为项目选择正确的微控制器芯片类型很重要，因为  $\mu$ Vision 将根据器件类型定制工具设置、外设以及相应的对话框。

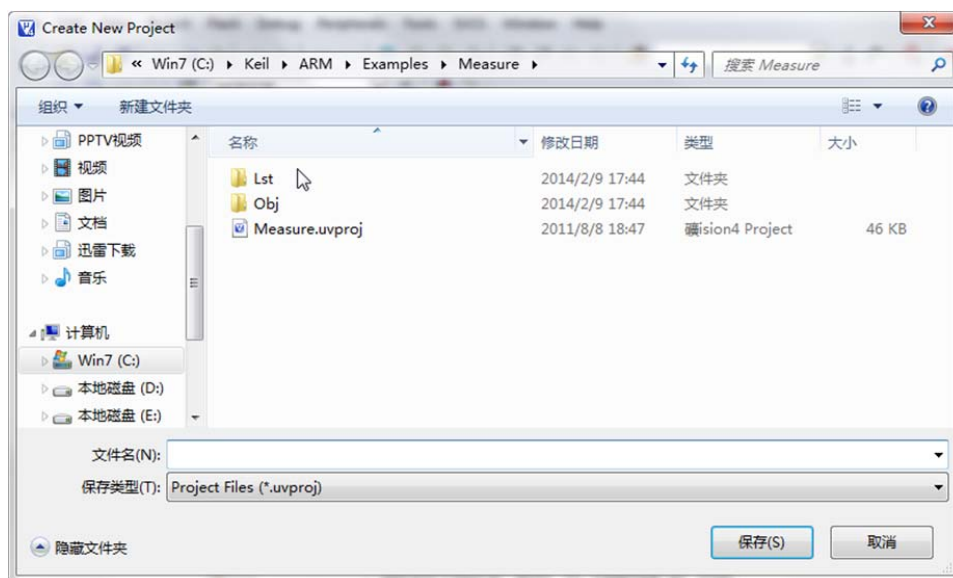


图 6 新建工程示意图

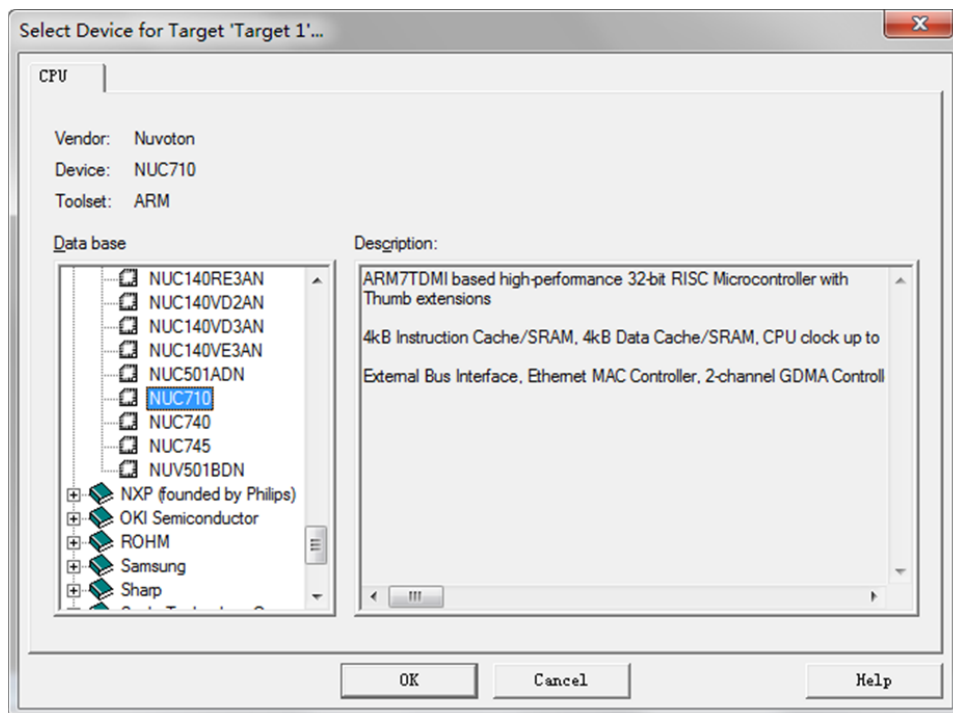


图 7 选择微控制器型号

在 Select Device 对话框点击 OK 后， $\mu$ Vision 会弹出选择拷贝启动代码的对话框。所有的嵌入式程序都需要微控制器初始化和启动的代码来完成与芯片、开发板等相关的硬件配置。 $\mu$ Vision 中集成了器件数据库中所列出的绝大多数芯片器件的启动代码。通常， $\mu$ Vision 将根据器件类型自动选择并拷贝相应的启动代码到项目文件夹中。此处，只需选择 Yes 即可。

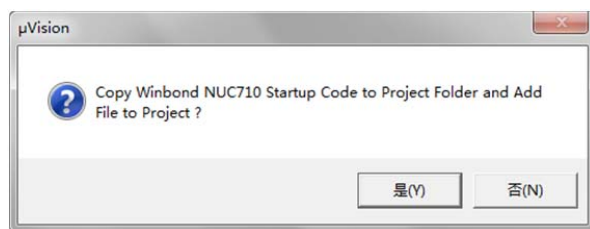


图 8 是否添加启动代码到工程

## 2) 使用项目窗口

当我们成功的创建新项目后，项目窗口中会显示该项目的 targets、groups 和 files。默认情况下，target 的名称是 Target 1，group 的名称是 Source Group 1。我们可以看到，启动代码文件已经被添加到工程中。



图 9 启动代码添加到工程

## 3) 新建源文件

使用 **File** 工具栏或选择 **File – New** 菜单创建一个新的源文件。在编辑器窗口中，我们可以为新建源文件输入源代码。 $\mu$ Vision 支持根据文件后缀名的语法彩色高亮功能。为了尽快使用该功能，我们在输入源代码前先将空文件保存。选择 **File** 工具栏的保存按钮或使用 **File – Save** 菜单，将源文件保存为 **main.c**。

我们采用课程教材中 45 页应用示例 2.1 的 FIR 滤波器设计作为 C 源代码输入 **main.c** 文件。C 源代码如下：

```
#define N    6
int main(void)
{
    unsigned char c[6] = {1,2,3,4,5,6};
    unsigned char x[6] = {4,8,9,3,5,1};
    unsigned char i,f;

    for (i=0, f=0; i<N; i++)
    {
        f = f + c[i]*x[i];
    }

    return 0;
}
```

#### 4) 添加源文件到工程

源文件虽然保存在项目目录下，但是由于没有被添加到当前项目中，也不会被编译。我们需要在项目窗口中，用鼠标右键单击 **Source Group 1**，在弹出菜单中选择 **Add File to Group**，按照提示一步步将新建源代码文件添加到当前工程中。

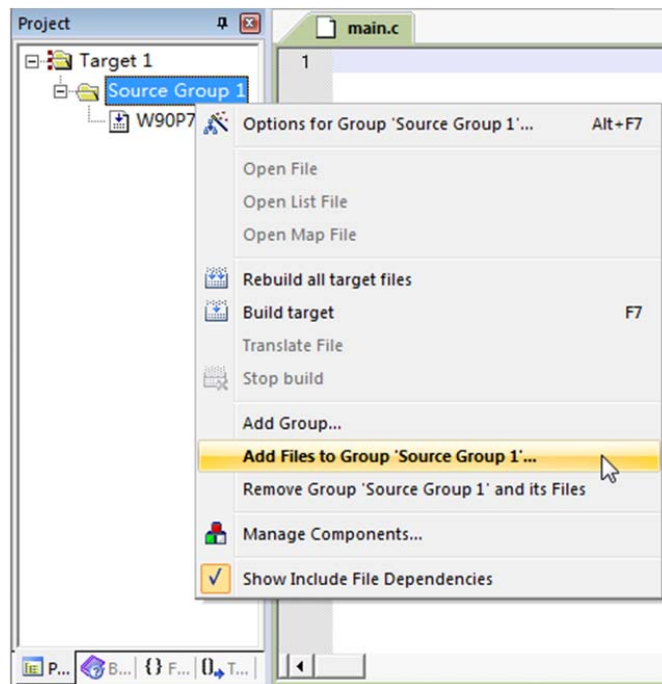


图 10 添加源文件到工程示意图

## 5) 设置目标选项

通过 Build 工具条或从 Project 菜单打开 Options for Target 对话框。

在对话框中，我们可以更改目标器件类型，设置目标选项，并配置开发工具链。请按照如图 11 所示设置 Target，其余使用默认设置。

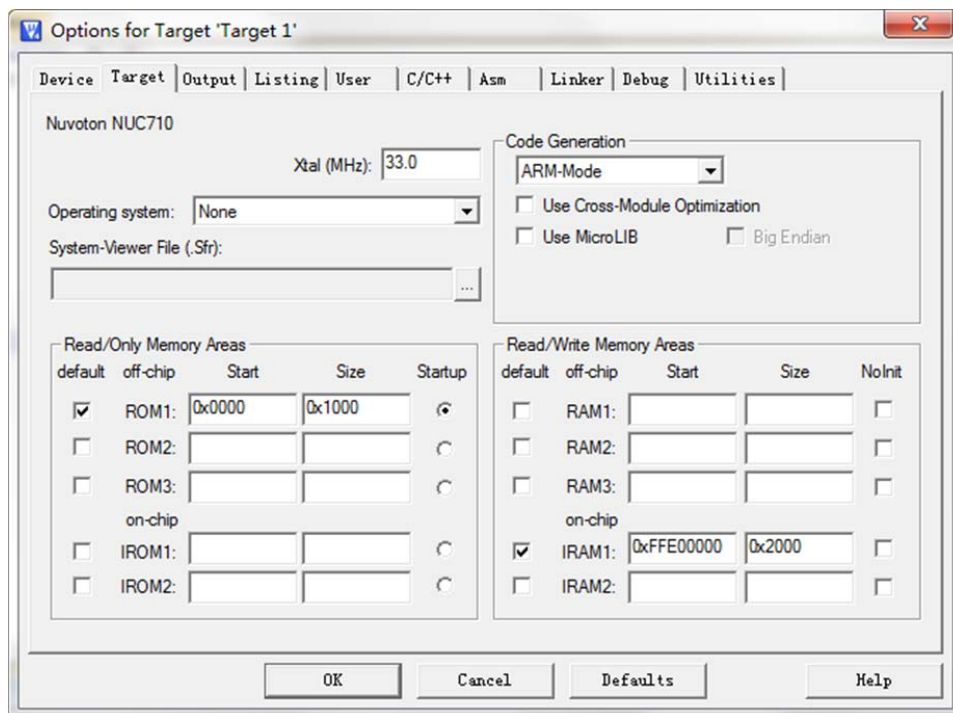





图 11 设置目标选项

## 6) 项目 Build

项目 Build 的过程即是源代码编译、汇编、连接并最终生成目标文件的过程。Build 过程可以通过以下三个工具按钮操作实现。

 Translate File - 编译或汇编当前源文件。

 Build Target - 编译和汇编所有被修改过的文件，并连接项目。

 Rebuild - 编译和汇编所有文件，无论被修改与否，并连接项目。

在汇编、编译和连接的过程中， $\mu$ Vision 会在 Build 输出窗口显示错误和警告，如图 12 所示。对于提示的错误或警告，我们可以双击该信



息跳转到源代码的对应行查找错误或警告原因。当项目成功完成所有 build 过程， $\mu$ Vision 会显示 **0 Error(s), 0 Warnings(s)**。同时，Build 输出窗口还会显示程序代码和数据的大小。

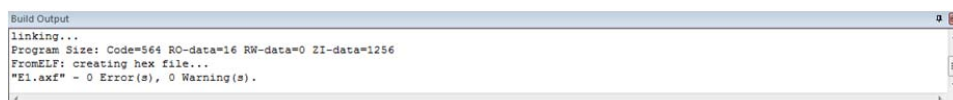


图 12  $\mu$ Vision 错误警告显示窗口

我们还可以通过在 Options for Target – Output 中设置在 build 过程中自动生成 HEX 文件，以用于 Flash 存储器的编程。

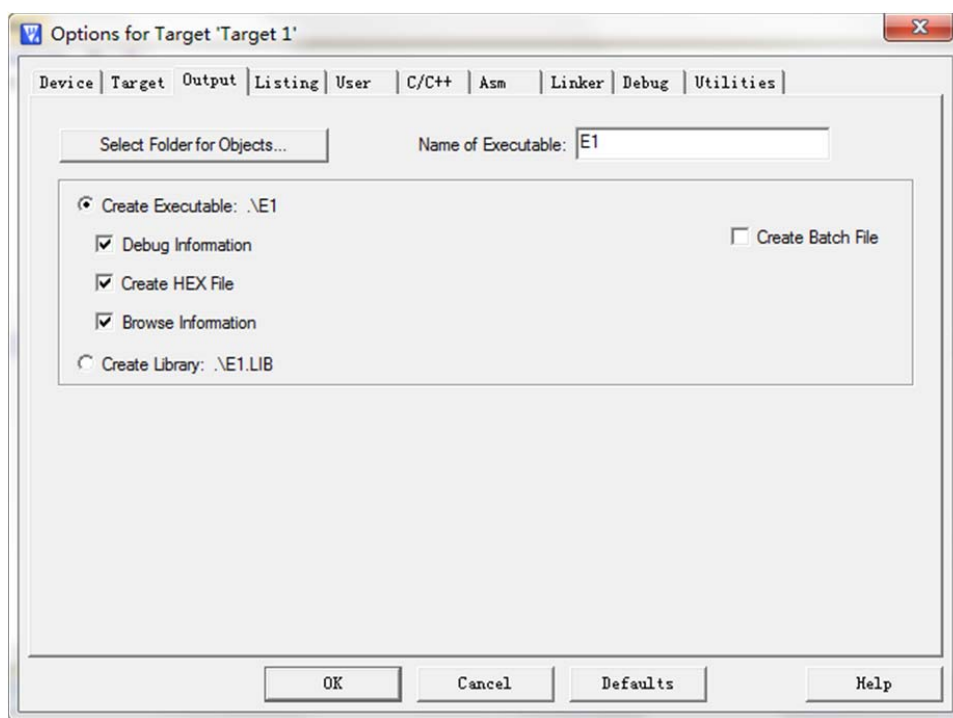


图 13 选择 HEX 文件输出

注：HEX 全称 Intel HEX，是一种文件格式，用以对可编程微控制器、EPROMs 和其他类似芯片传送二进制信息。通常，编译器或汇编器将程序源代码转换成机器码并输出到 HEX 文件。该 HEX 文件随后被编程器导入，用来将机器码“烧写”或传送到目标系统或 ROM 中装载并执行。

## 5. 程序调试

下面我们学习使用  $\mu$ Vision 调试器仿真调试之前编写的程序。如前所述， $\mu$ Vision 调试器可以配置成仿真器模式或目标调试器模式。本次实验，我们先学习仿真器调试模式，目标调试器模式在下一次结合硬件开发板的实验中学习。通过 Build 工具条打开 Options for Target 对话框，切换到 Debug 标签，按照图 14 所示设置调试器为仿真器模式。

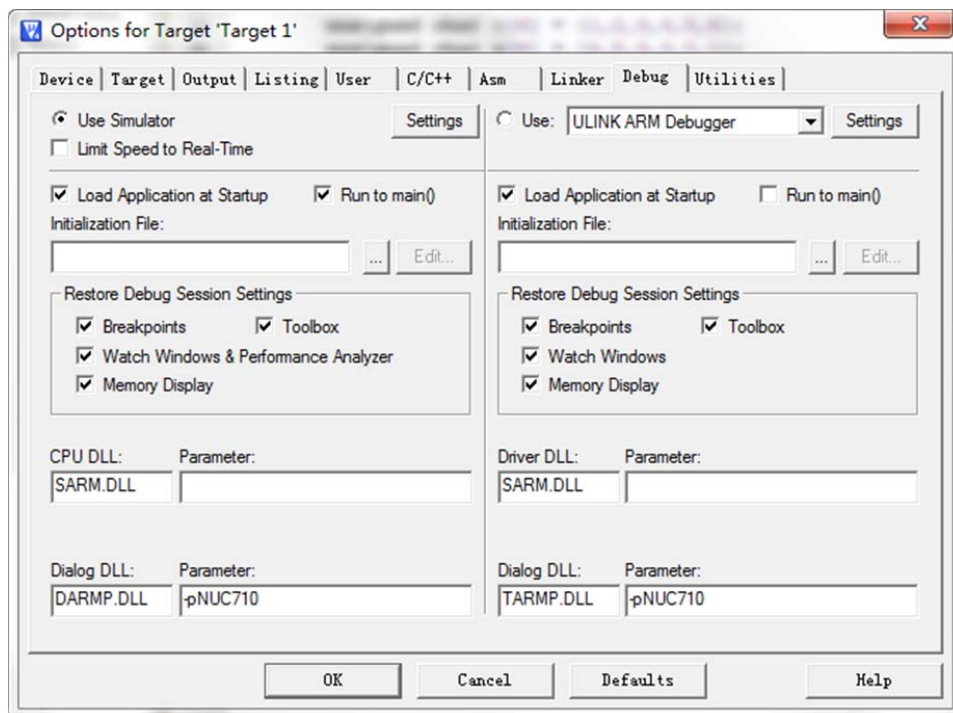



图 14  $\mu$ Vision 仿真器模式设置窗口

从 Debug 工具条选择 Start/Stop Debug Session 命令按钮进入调试模式。 $\mu$ Vision IDE 会装载应用，执行启动代码，并按照配置一直执行到 main 函数入口处暂停。当程序执行暂停后， $\mu$ Vision IDE 打开文本编辑器窗口、反汇编窗口以及寄存器窗口、栈调用窗口等调试输出窗口。 $\mu$ Vision IDE 进入调式模式后的界面如图 15 所示。

左边是 Register Window，实时显示处理器寄存器中的值。

右边是 Disassembly Window 和 Text Editor Window。其中，上边 Disassembly Window，显示 C 语言等高级语言反汇编后的汇编代码；下边的 Text Editor Window，对源程序进行修改或编辑。

右下方是 Call Stack Window，查看程序中的变量在栈中的变化情况。

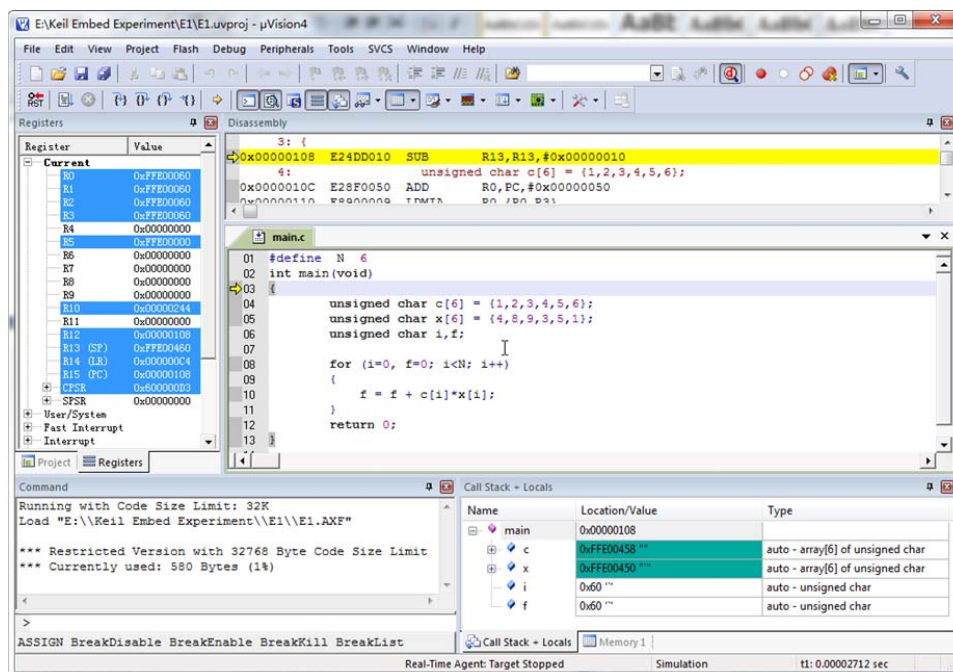


图 15  $\mu$ Vision 进入调式模式后的界面


$\mu$ Vision IDE 进入调试模式后，我们可以通过操作 Debug 工具栏或 Debug 菜单的调试命令来控制代码的执行。Debug 工具栏如图 16 所示：





图 16 Debug 工具栏


控制代码执行的常用命令解释如下：


 Run 命令：启动执行程序


 Stop 命令：暂停执行程序

 Reset CPU 命令：PC 指针复位




 Step 命令：在程序中单步执行中进入函数内部

 **Step Over 命令**：在程序中单步执行中将函数看做是一条指令

 **Step Out 命令**：从当前函数中跳出进入上一级程序

 **执行到光标行命令**：执行程序到当前光标所在代码行

我们还可以通过双击编辑器窗口代码行前面的空白处或 **Debug - Insert/Remove Breakpoint** 插入断点，辅助控制程序的执行。图 17-19 中，代码行前面的红点就是插入的断点，同样也可以双击红色断点取消插入的断点，程序中允许插入多个断点。

选择 **Debug** 按钮 进入 **Debug** 模式后，源程序从图 17 中的黄色箭头处开始执行。当点击 **Debug** 工具栏上的  按钮时，源程序会单步向下执行，执行过后的源程序前会变成绿色，如图 18 所示；在源程序没有执行到断点之前点击 **Debug** 工具栏上的 **Run** 按钮，程序会自动执行到红色的断点处，如图 19 所示。

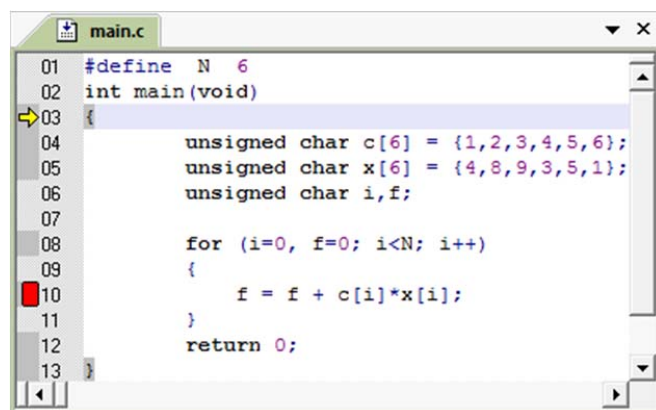
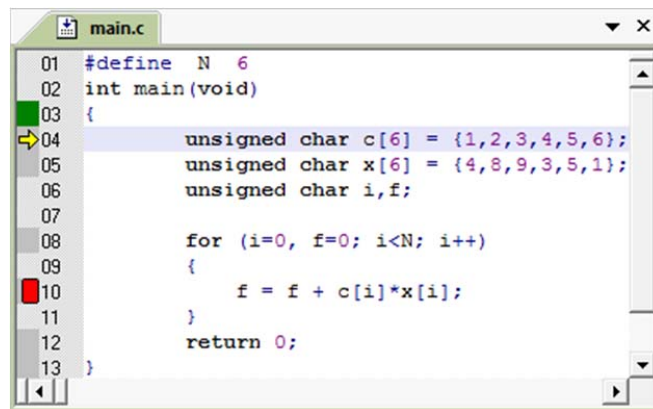
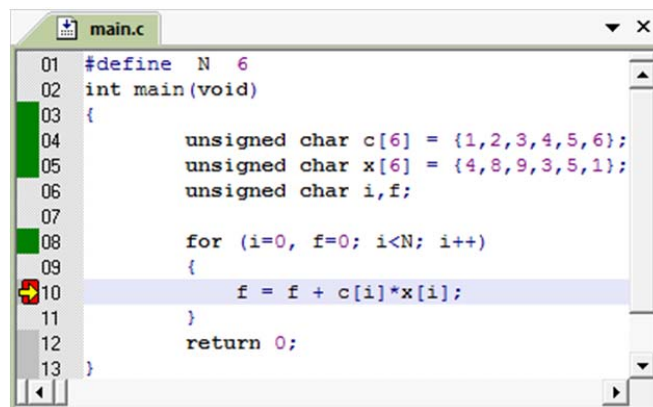


图 17 程序开始单步调试



```
01 #define N 6
02 int main(void)
03 {
04     unsigned char c[6] = {1,2,3,4,5,6};
05     unsigned char x[6] = {4,8,9,3,5,1};
06     unsigned char i,f;
07
08     for (i=0, f=0; i<N; i++)
09     {
10         f = f + c[i]*x[i];
11     }
12     return 0;
13 }
```

图 18 程序单步调试后



```
01 #define N 6
02 int main(void)
03 {
04     unsigned char c[6] = {1,2,3,4,5,6};
05     unsigned char x[6] = {4,8,9,3,5,1};
06     unsigned char i,f;
07
08     for (i=0, f=0; i<N; i++)
09     {
10         f = f + c[i]*x[i];
11     }
12     return 0;
13 }
```

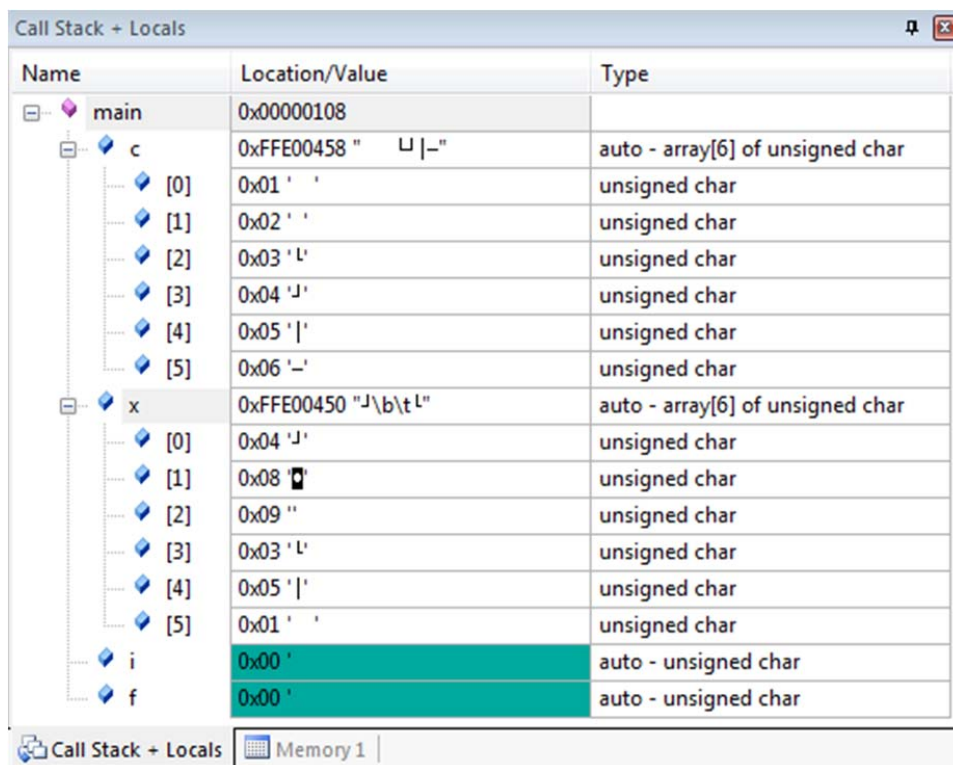
图 19 程序执行到断点处

程序在执行过程中，相应工作模式下的寄存器的值会发生变化，图 20 中显示了微控制器寄存器的值。可以通过双击寄存器的值来改变其值或使用 F2 快捷键选择相应的寄存器后在改变其值。

Registers	
Register	Value
<b>Current</b>	
R0	0xFFE00060
R1	0xFFE00060
R2	0xFFE00060
R3	0xFFE00060
R4	0x00000000
R5	0xFFE00000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000244
R11	0x00000000
R12	0x00000108
R13 (SP)	0xFFE00460
R14 (LR)	0x000000C4
R15 (PC)	0x00000108
+ CPSR	0x600000D3
+ SPSR	0x00000000
+ User/System	
+ Fast Interrupt	
+ Interrupt	
+ <b>Supervisor</b>	
+ Abort	
+ Undefined	
- Internal	
PC \$	0x00000108
Mode	Supervisor
States	895
Sec	0.00002712

图 20 寄存器窗口

可以在 Call Stack Window 中查看源程序中变量的变化情况，图 21 反映的是源程序执行到图 17 中的断点处时，栈中变量的变化情况。此时，数组 c[6]和 x[6]，变量 i 和 f 的初始值都被压入到了栈中，实现对数组和变量的初始化操作。



Name	Location/Value	Type
main	0x00000108	
c	0xFFE00458 "  -"	auto - array[6] of unsigned char
[0]	0x01 ' '	unsigned char
[1]	0x02 ' '	unsigned char
[2]	0x03 'L'	unsigned char
[3]	0x04 'J'	unsigned char
[4]	0x05 ' '	unsigned char
[5]	0x06 '-'	unsigned char
x	0xFFE00450 "J\b\tL"	auto - array[6] of unsigned char
[0]	0x04 'J'	unsigned char
[1]	0x08 'b'	unsigned char
[2]	0x09 'L'	unsigned char
[3]	0x03 'L'	unsigned char
[4]	0x05 ' '	unsigned char
[5]	0x01 ' '	unsigned char
i	0x00 '0'	auto - unsigned char
f	0x00 '0'	auto - unsigned char

图 21 栈显示窗口

在 Keil MDK 中还有一些很强大的工具，比如逻辑分析仪、代码覆盖率分析工具等，下面对这两种工具做一下简单介绍。

**逻辑分析仪：**在 keil MDK 中软件逻辑分析仪很强的功能，可以分析数字信号，模拟化的信号，CPU 的总线(UART、IIC 等一切有输出的管脚)，提供调试函数机制，用于产生自定义的信号，如 Sin，三角波、噪声信号等，这些都可以定义。

**代码覆盖率分析：**代码覆盖率分析工具能够提供几乎不受限制的跟踪信息流服务，为程序代码提供完整的代码覆盖率分析。代码覆盖率



标识每个已执行的指令，从而确保对程序进行彻底测试，这是对完整的软件验证和认证的基本要求。代码覆盖率标识已执行的代码，可有助于确保彻底测试应用程序。

6. 创建汇编源代码项目

创建汇编源代码程序与 C 语言源程序类似，但有两个不同点：在新建项目这一步不需要添加启动代码；新建汇编源文件时，汇编源文件后缀名为.s 而不是.c。

汇编源程序如下：

```
AREA FIR, CODE, READONLY ;声明代码段
CODE32 ;声明程序为 32 位 ARM 指令
ENTRY ;声明程序入口
MOV r0, #0
MOV r8, #0
ADR r2, N
LDR r1, [r2]
MOV r2, #0
ADR r3, C
ADR r5, X
; loop body
loop
    LDR r4, [r3, r8]
    LDR r6, [r5, r8]
    MUL r9, r4, r6
    ADD r2, r2, r9
    ADD r8, r8, #4
    ADD r0, r0, #1
    ; test for exit
    CMP r0, r1
    BLT loop
    B .
ALIGN
N DCD &0a
C DCD &01, &02, &03, &04, &05, &06, &07, &08, &09, &0a
X DCD &01, &02, &03, &04, &05, &06, &07, &08, &09, &0a
END
```

注：在 keil 中编写汇编源程序时，汇编指令不能顶格写。汇编语言程序每行分为 4 个区域，从左到右，分别是标号域、助记符、操作数、注释域，其中标号域是根据需要决定有否，而注释域是可有可无的，一个训练有素的程序设计人员，总是会很好地使用注释域。

将汇编源程序 Translation-Bulid 后，选择 Debug 按钮进入调试模式。通过单步执行可以很容易地查看到寄存器值的变化，如图 22 和图 23 所示：

图 22 是刚进入调试模式时寄存器的值，图 23 是单步执行到语句”MOV r2, #0”时寄存器的值，可以看出寄存器 r1、r2、r15 的值都发生了变化，同理可以查看其它汇编语句执行时，寄存器值的变化情况。

其中，寄存器 r2 存储的是标签 N 所在存储空间的首地址。将地址 0x40 填入 Memory Window 中的 Address 文本框中，可以看到内存中为 0x40 的地址空间已经被写入了我们初始化的数据 0x0a，如图 24 所示，同理也可以查看其它地址空间中初始化的数据。

Register	Value
<b>Current</b>	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000000
+ CPSR	0x000000D3
+ SPSR	0x00000000
+ User/System	
+ Fast Interrupt	
+ Interrupt	
+ Supervisor	
+ Abort	
+ Undefined	
- Internal	
PC \$	0x00000000
Mode	Supervisor
States	0
Sec	0.00000000

图 22 寄存器窗口 1

Register	Value
<b>Current</b>	
R0	0x00000000
R1	0x0000000A
R2	0x00000040
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000010
+ CPSR	0x000000D3
+ SPSR	0x00000000
+ User/System	
+ Fast Interrupt	
+ Interrupt	
+ Supervisor	
+ Abort	
+ Undefined	
- Internal	
PC \$	0x00000010
Mode	Supervisor
States	6
Sec	0.00000018

图 23 寄存器窗口 2

Memory 1	
Address: 0x40	
0x00000040:	0A 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
0x00000050:	04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
0x00000060:	08 00 00 00 09 00 00 00 0A 00 00 00 01 00 00 00
0x00000070:	02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00
0x00000080:	06 00 00 00 07 00 00 00 08 00 00 00 09 00 00 00
0x00000090:	0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000A0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000B0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 24 内存显示窗口

**参考文献:**

- [1] Getting Started, Creating Applications with  $\mu$ Vision<sup>®</sup> 4
- [2] 嵌入式计算系统设计原理, 沃尔夫, 李仁发, 机械工业出版社, 2009 年 6 月.