第20章

# KEIL RealView Microcontroller

# Development Kit（RVMDK）使用入门

* 简介
* uVision使用入门
* 使用UART输出“Hello World”
* 测试示例程序
* 使用调试器
* 指令模拟器
* 修改向量表
* 使用中断实现的秒表示例程序

## 20.1 简介

有许多商业的开发平台可以用在CM3上，其中最流行的之一就是KEIL的RealView Microcontroller Development Kit（简称RealView MDK或RVMDK）。RVMDK的前身就是曾一度在8051开发业界享有盛誉的KEIL套件。RVMDK包含了很丰盛的组件：

* + uVision
  + 集成开发环境
  + 调试器
  + 模拟器
  + 由ARM提供的RealView工具链
    - C/C++编译器
    - 汇编器
    - 连接器
  + RTX实时内核
  + 为各单片机而设的详细启动代码（包含源代码）
  + 各种Flash的编程算法
  + 程序示例

（英蓓特还把RVMDK的帮助文件翻译成了中文，并包装成“中国版”的RVMDK——译者注）。

使用RVMDK来学习CM3，甚至不需要拥有CM3硬件——uVison环境包含了指令模拟器，使用它可以测试“纯粹”的CM3程序代码，对于学习和开发基于内核的系统软件都很有好处。

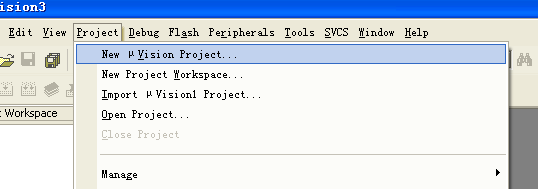
RVMDK还可以与GNU工具链一起使用。

可以从KEIL网站上获取免费的KEIL tool之演示版，也可以从http://www.realview.com.cn/ 处下载中文的相关资源。

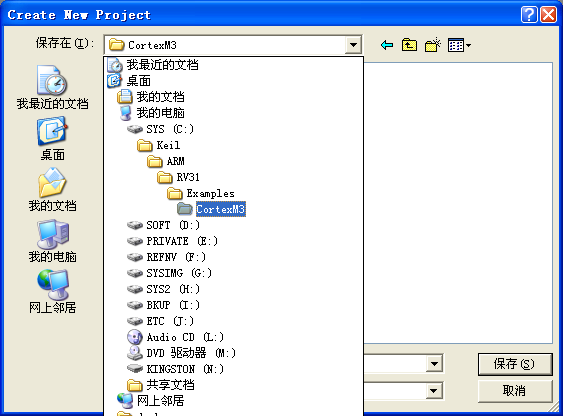
## 20.2 uVison使用入门

在RVMDK中附带了很多示例程序，包括Luminary Micro的Stellaris系列的单片机产品，也包括了ST的STM32系列的单片机产品。这些示例都使用了厂家提供的驱动程序库（固件库）。使用固件库可以免去写代码操作外设寄存器的任务。很容易通过修改示例程序来开发自己的应用程序，也可以自己从头设计工程，再摘抄一部分示例程序的代码。本章的示例基于RVMDK v3.03版，并且以Luminary Micro的LM3S811器件为蓝本（目前RVMDK已经出了3.20版，且作者写本书时Luminary Micro是唯一的CM3芯片供应商。目前ST也出品了STM32系列的CM3芯片，预计以后Atmel, TI, NXP也要提供CM3芯片——译者注）。

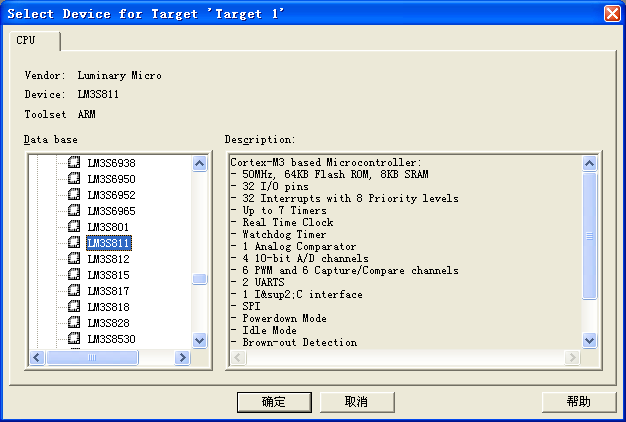
在安装了RVMDK后，就可以从开始菜单中启动uVision集成开发环境，并且它会打开一个为传统ARM处理器而写成的示例。我们可以关掉这个工程，并且通过选择”New Project”下拉菜单来新建一个工程：



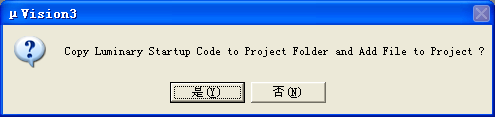
这里创建了一个名为CortexM3的文件夹：



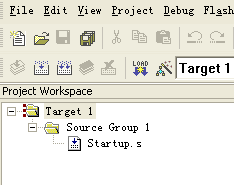
接下来为这个工程选择目标器件，在这里选择LM3S811



RVMDK会询问是否使用缺省的启动代码。这里我们选择Yes。



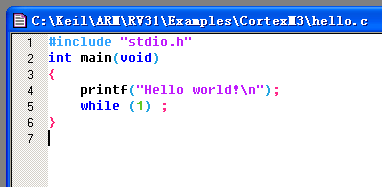
现在我们有了只含有一个源文件（Startup.s）的工程了：



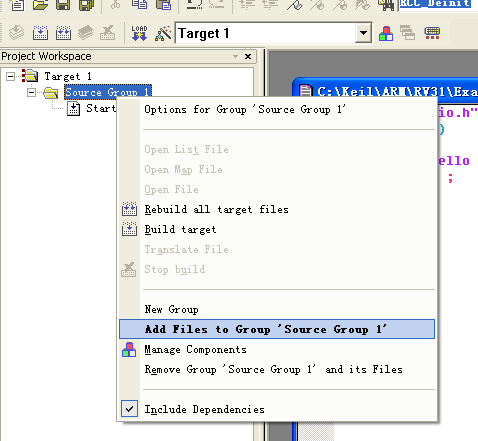
接下来，我们要创建一个含有main函数的C源程序。



这样就创建了一个文本文件。编辑它的内容，并且存储为hello.c：



现在，我们要把这个文件添加到Source Group 1中（右击Source Group 1）



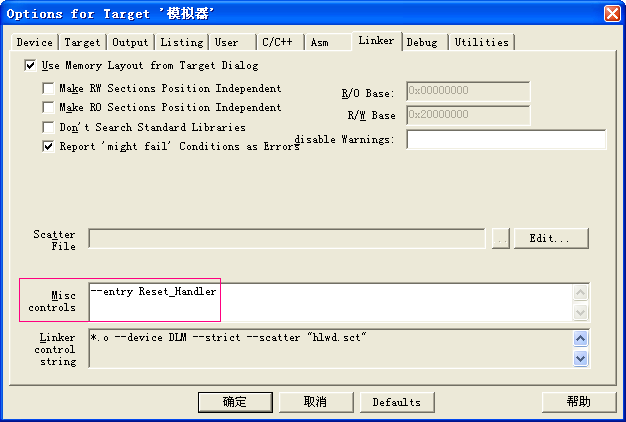
可以修改目标名“Target 1”和文件组名“Source Group 1”，以使它们更有意义。通过单击工程的工作区选中它们，过一会再单击即可修改（超过双击判定的最短间隔时间）：



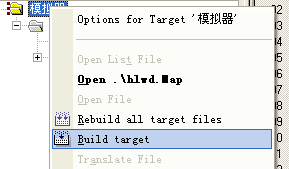
选择我们刚刚创建的hello.c并添加，则工程中就包含了2个源文件了：



我们还需要设置连接器以定义程序的入口点。通过在“Misc Controls box”中加入entry Reset\_Handler来实现。这个选项定义了程序的入口点为Reset\_Handler——它可以在Startup.s中找到（其实在RVMDK 3.20版本中，连接设置有了变化，此时不加也可以——译者注）。



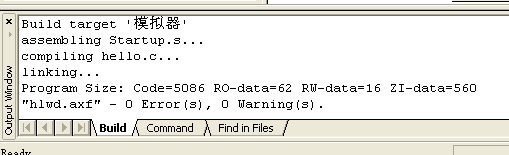
现在我们就可以编译了。如下所示：



也可以使用工具栏来方便地操作：



编译成功后，输出窗口如下显示：



## 20.3 使用UART输出“Hello World”

在上一个例子中，我们使用了C标准库中的printf函数。但是C标准库并不知道我们使用的硬件是什么，因此如果要“真实”地输出字符串（如通过UART输出），还必须添加一些代码。正如本书曾经提到的，为使输出送到实际的硬件，我们经常需要做所谓的“retargeting”工作。与Retargeting相关的函数除了可以用于输出文本外，还可以包含处理错误以及终结程序等其它功能。在本例，只介绍文本输出的功能。

在本例中，是打算把”Hello World”消息从LM3S811的UART0送出去，目标系统是Luminary Micro LM3S811评估板。板上晶振为6MHz，但单片机片内配有PLL模块，它把时钟频率上升到50MHz。波特率是115,200，并且使用PC上的超级终端程序来接收从UART发出的数据。

要对printf执行retarget处理，我们需要实现fputc函数。在下面的代码中，我们就创建了这个fputc函数，它又呼叫sendchar函数，而后者则操作UART输出字符：

#include "stdio.h"

#define CR 0x0D //回车符

#define LF 0x0A //换行符

void Uart0Init(void);

void SetClockFreq(void);

int sendchar(int ch);

// 若使用6MHz，则注释掉下一行

#define CLOCK50MHZ

// Register addresses

#define SYSCTRL\_RCC ((volatile unsigned long \*)(0x400FE060))

#define SYSCTRL\_RIS ((volatile unsigned long \*)(0x400FE050))

#define SYSCTRL\_RCGC1 ((volatile unsigned long \*)(0x400FE104))

#define SYSCTRL\_RCGC2 ((volatile unsigned long \*)(0x400FE108))

#define GPIOPA\_AFSEL ((volatile unsigned long \*)(0x40004420))

#define UART0\_DATA ((volatile unsigned long \*)(0x4000C000))

#define UART0\_FLAG ((volatile unsigned long \*)(0x4000C018))

#define UART0\_IBRD ((volatile unsigned long \*)(0x4000C024))

#define UART0\_FBRD ((volatile unsigned long \*)(0x4000C028))

#define UART0\_LCRH ((volatile unsigned long \*)(0x4000C02C))

#define UART0\_CTRL ((volatile unsigned long \*)(0x4000C030))

#define UART0\_RIS ((volatile unsigned long \*)(0x4000C03C))

int main (void)

{

SetClockFreq(); // 建立时钟的配置 (50MHz/6MHz)

Uart0Init(); // 初始化UART0

printf ("Hello world!\n");

while (1);

}

void SetClockFreq(void)

{

#ifdef CLOCK50MHZ

// 置位BYPASS, 清除 USRSYSDIV 和 SYSDIV

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xF83FFFFF) | 0x800 ;

// 清零 OSCSRC, PWRDN 和 OEN

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xFFFFCFCF);

//修改 SYSDIV, 设置 USRSYSDIV 和 Crystal 位段的值

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xF87FFC3F) | 0x01C002C0;

// 等待PLLLRIS被置位

while ((\*SYSCTRL\_RIS & 0x40)==0); // wait until PLLLRIS is set

// 清除bypass

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xFFFFF7FF) ;

#else

// 置位 BYPASS, 清除 USRSYSDIV 和 SYSDIV

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xF83FFFFF) | 0x800 ;

#endif

return;

}

void Uart0Init(void)

{

\*SYSCTRL\_RCGC1 = \*SYSCTRL\_RCGC1 | 0x0003; // 使能 UART0 & UART1

// clock

\*SYSCTRL\_RCGC2 = \*SYSCTRL\_RCGC2 | 0x0001; // 使能 PORTA 时钟

\*UART0\_CTRL = 0; // 除能 UART

# ifdef CLOCK50MHZ

\*UART0\_IBRD = 27; // 以50MHz频率为基准编程波特率

\*UART0\_FBRD = 9;

# else

\*UART0\_IBRD = 3; // 以6MHz频率为基准编程波特率

\*UART0\_FBRD = 17;

# endif

\*UART0\_LCRH = 0x60; // 8 bit, 无奇偶

\*UART0\_CTRL = 0x301; // 使能 TX 和 RX, 以及 UART 使能

\*GPIOPA\_AFSEL = \*GPIOPA\_AFSEL | 0x3; // 把GPIO管脚交给UART0控制

return;

}

// 送给UART0一个字符（printf函数使用它来输出文字）

int sendchar (int ch)

{

if (ch == '\n')

{

while ((\*UART0\_FLAG & 0x8)); // 如果UART忙碌中则等待

\*UART0\_DATA = CR; // 输入附加的CR以使字符串被正确显示

}

while ((\*UART0\_FLAG & 0x8)); // 如果UART忙碌中则等待

return (\*UART0\_DATA = ch); // 输出数据

}

//文本输出的retargeting代码

int fputc(int ch, FILE \*f)

{

return (sendchar(ch));

}

代码中的SetupClockFreq()用于把系统时钟设置为50MHz。要注意的是这个函数是与具体的器件相关的。另外，还使用了条件编译来允许选择使用6MHz的原始频率。

UART的初始化是由Uart0Init()来执行的，它设置了波特率为115200，8个数据位，1个停止位，无奇偶校验，并且启用GPIO的第二功能，从而让GPIO控制器把管脚的控制权交给UART0。在使用UART和GPIO之前，必须先启用这两个模块。代码中的把SYSCTRL\_RCGC1和SYSCTRL\_RCGC2分别启用了这两个模块。

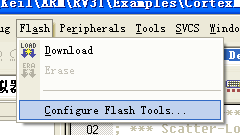
Retargeting的代码是由粗体的fputc()执行的。要注意的是，不能使用其它的名字，因为”fputc”是编译器预定义的用于字符输出的函数名。fputc()实际上只是个封皮，它直接调用sendchar()来做真实的工作。sendchar()除了输出一般的字符之外，还要在检测到”\n”时输出一个附加的CR，才能在超级中断上正常显示回车换行，否则将回到同一行的起始处，使先前输出的字符被新输出的字符覆盖掉。

在加入retargeting代码后，就可以重新编译了。

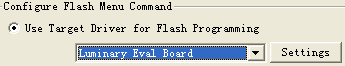
## 20.4 测试示例程序

如果有硬件设备，则可以把编译好的程序下载到Flash并且运行，步骤如下：

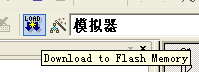
1. 配置flash下载选项：



1. 选择硬件平台



1. 接下来就可以下载程序到flash中了



1. 下载完后，程序将开始运行，在超级终端上应看到如下字样：

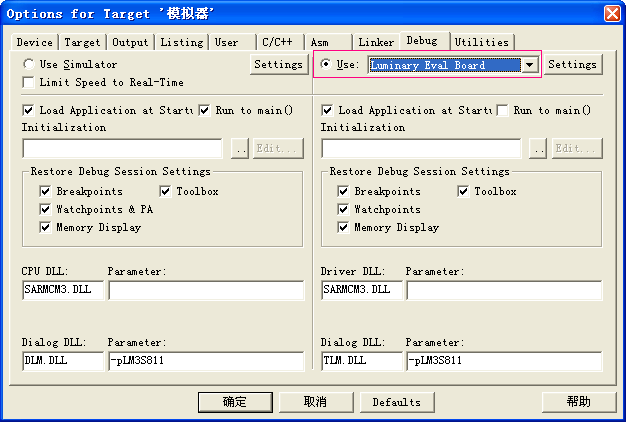


## 20.5 使用调试器

uVision中附带了调试器，这是一个可视化的源码级调试器。可以把它连接到目标板上（通过JTAG仿真器）。在设置时，单击“魔术棒”按钮，

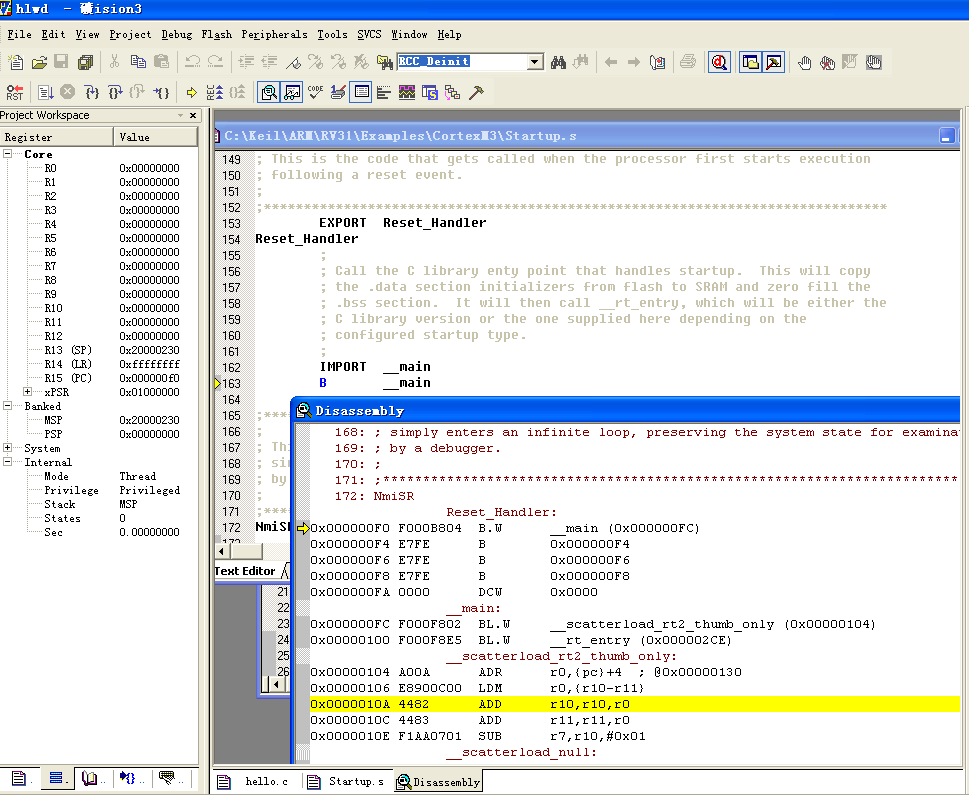


再在弹出的对话框中选择”Debug”选项卡，并且如下设置：

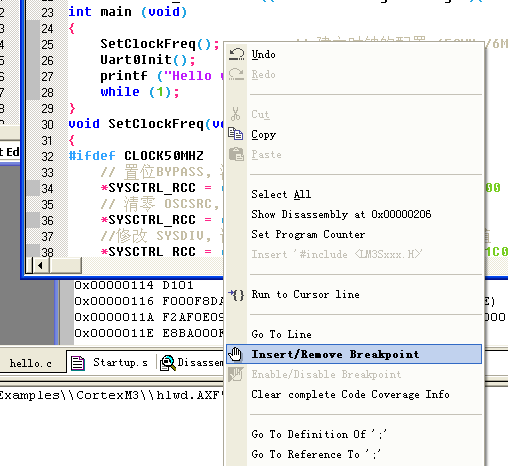


接下来就可以启动调试了。但要注意的是：如果板子已经在运行并且连上了超级终端，则需要关闭超级终端，断开USB电缆，并且在开始调试之前再重新连接。

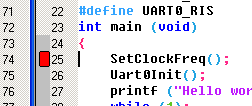


当调试器开始工作后，IDE会提供一个寄存器视图，显示当前寄存器的内容（通用寄存器和特殊功能寄存器都有）。还可以从源码级上看到程序当前的执行进度。从下图中我们可以看出刚开始执行时的情况——内核停止在Reset\_Handler上：

为了测试，我们可以在main()的入口处设置断点。通过右击代码并且在在菜单中执行“Insert/Remove Breakpoint”即可（也可以在调试选项中如下设置：。这样会在运行到main()时自动停止）。



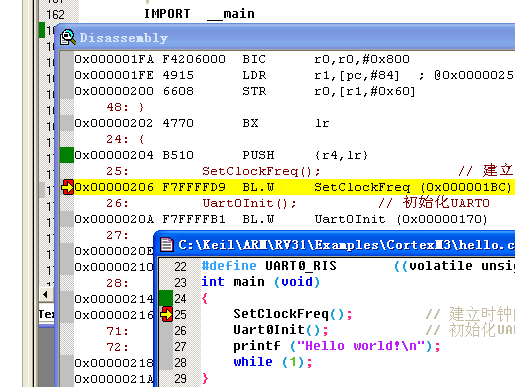
（也可以双击main()入口处“SetClockFreq()”前面的空白，或者是它的行号“25”来快速切换断点）。



上图的红色实心矩形表示断点已经设好。接下来运行程序：



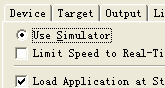
程序将停在断点处：



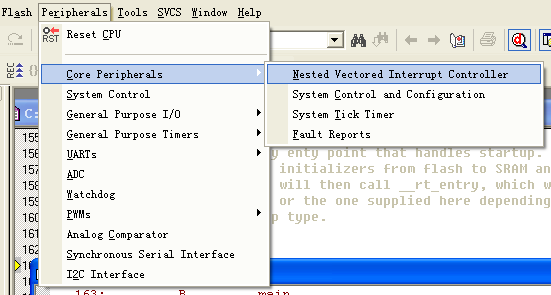
黄色小箭头表示停止后即将执行的语句。从图中还可以看到，反汇编窗口也对应地显示了这个断点所处的指令位置。译者使用的是模拟器，因此还能记录曾经执行过的语句。图中绿色小块就表示已经执行过的语句/指令。细心的读者可能会发现，“｛”竟然也被“执行”了（因为是绿色小块）！事实上，“｛”被编译后产生了汇编指令。看一下反汇编窗口——原来“｛”处的机器码是一条“PUSH ｛r4,lr｝”指令。

## 20.6 指令模拟器

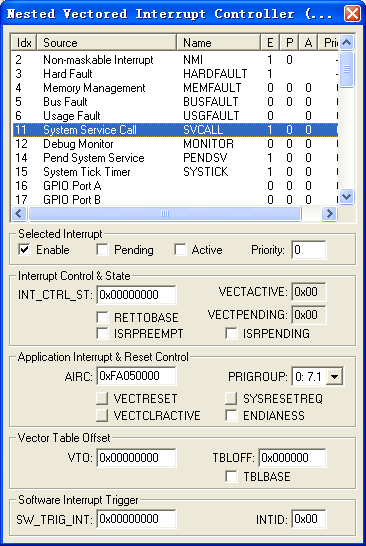
uVison IDE还附带了一个软件模拟器，它可以用来验证算法的各项性能指标，也是学习CM3的好帮手。若欲使用模拟器，只需在刚才的“Debug”选项里点中Simulator:



事实上，调试器还对片上外设包装了非常丰满和直观的可视化操作接口，模拟器也可以模拟它们。比如，欲查看NVIC的内部状态，可以通过如下菜单命令启动：

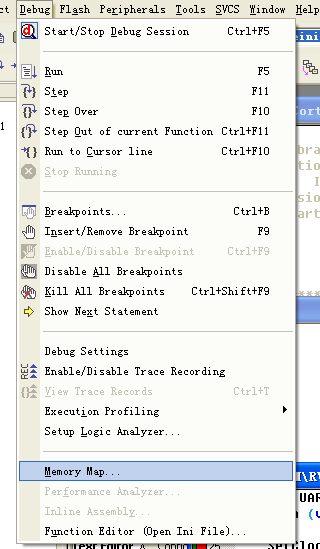


执行上图中的菜单命令后，将弹出如下对话框：

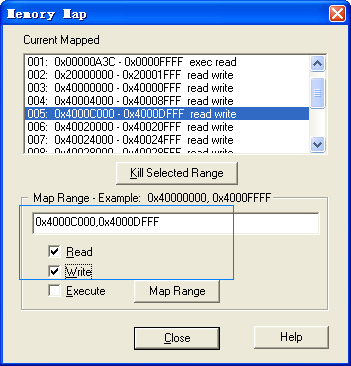


可以在它上面单击不同的异常来查看相关状态，上图中就查看了SVC异常的状态。

在使用模拟器时，如果被模拟的器件比较新，则有可能器件中某些地址范围模拟器没有来得及更新。这样在使用模拟器时，就会被判定成访问落空的地址范围，因此产生fault。比如，本例中，为了保证UART寄存器的地址范围被支持，可以使用如下的菜单命令检查：



执行后，弹出如下对话框：



本例中UART寄存器的地址范围是0x4000\_C000-0x4000\_DFFF。译者使用的RVMDK版本较新，已经加入了本地址范围。但是原著作者使用的RVMDK版本还没有加入，需要手工添加，语法如上图被框住的部分所示。RVMDK的调试器会自动合并“碎片”，对于译者使用的RVMDK，它会发现手工添加的内容与原有的重合，故而不会有任何影响。

------译者添加------

uVision IDE的调试功能非常强大，很多没有想到的功能它都有，简列如下：

* + 观察窗口，函数调用栈窗口，监视窗口等
  + 计算到目前为止已经经历的周期数
  + 计算每条指令被执行的频率（性能分析非常有用）
  + 逻辑分析仪
  + 源程序中的各种符号窗口（文件名，变量名等）
  + 存储器映射窗口

限于篇幅，本书不能详细地讲述RVMDK的方方面面。但好消息是市面上新出了一本专门讲解RVMDK的书，书名为《ARM开发工具RealView MDK使用入门》，作者：李宁。北京航空航天大学出版社出版。

## 20.7 修改向量表

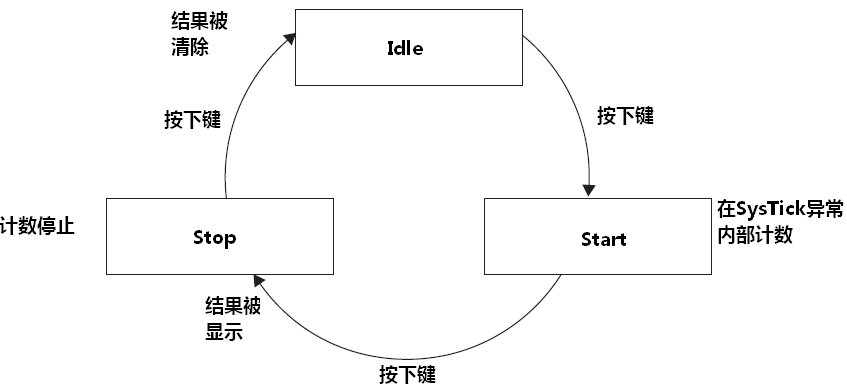
在上例中，向量表在Startup.s中已预先定义，这是由开发工具提供的向量表模板，它只包含了必备的MSP初始值、复位向量、NMI向量以及硬fault向量。但是我们往往还要响应其它中断，因此要添加其它的向量，或者需要把预先提供的向量也改成自己的。在这些场合下，就要更改Startup.s的代码，把向量表中对应的位置写成我们提供的服务例程名。

但问题又来了，ISR不是在Startup.s中实现，而是在其它文件实现的。那么怎么让startup.s汇编出的代码startup.o在连接时知晓ISR的地址呢？这时，就要使用IMPORT指示字。IMPORT后面跟随函数名或变量名，作用相当于C中的extern关键字，指出这些全局符号是在其它源文件中定义的。下一小节作为本书的谢幕，就提供了一个例子演示了IMPORT的使用。

## 20.8 使用中断实现的秒表示例程序

终于到了最后一节了，来看压轴好戏吧！

在这一小节中，将给出了一个最完整的示例——秒表程序示例。它使用了SysTick异常和UART0中断。秒表程序内部是以状态机的方式实现的，其状态转换图如下：



在上例的基础上，我们使用PC来通过UART以控制秒表程序的执行。为简化示例代码，我们使用固定的50MHz主频。时间测量上，由SysTick提供时基——它以100Hz的频率给出异常请求。本例中，SysTick以内核的50MHz时钟运行(FCLK)，每次响应SysTick中断时，如果秒表在走，则把计数器加1——自增TickCounter变量。

因为使用UART显示文字是个很耗费时间的工作，因此不再使用以前的查询方式，而转用中断来实现（这也是编程基本功），而对于秒表数值的格式化则在mani()中完成（线程模式下）。程序中的主状态机由UART服务例程启动状态转换——每收到一个字符转换一次。

使用上例的创建步骤，我们再创建一个名为stopwatch的工程。这次添加的代码是stopwatch.c：

#include "stdio.h"

#define CR 0x0D // Carriage return

#define LF 0x0A // Linefeed

void Uart0Init(void);

void SysTickInit(void);

void SetClockFreq(void);

void DisplayTime(void);

void PrintValue(int value);

int sendchar(int ch);

int getkey(void);

void Uart0Handler(void);

void SysTickHandler(void);

// 寄存器地址

#define SYSCTRL\_RCC ((volatile unsigned long \*)(0x400FE060))

#define SYSCTRL\_RIS ((volatile unsigned long \*)(0x400FE050))

#define SYSCTRL\_RCGC1 ((volatile unsigned long \*)(0x400FE104))

#define SYSCTRL\_RCGC2 ((volatile unsigned long \*)(0x400FE108))

#define GPIOPA\_AFSEL ((volatile unsigned long \*)(0x40004420))

#define UART0\_DATA ((volatile unsigned long \*)(0x4000C000))

#define UART0\_FLAG ((volatile unsigned long \*)(0x4000C018))

#define UART0\_IBRD ((volatile unsigned long \*)(0x4000C024))

#define UART0\_FBRD ((volatile unsigned long \*)(0x4000C028))

#define UART0\_LCRH ((volatile unsigned long \*)(0x4000C02C))

#define UART0\_CTRL ((volatile unsigned long \*)(0x4000C030))

#define UART0\_IM ((volatile unsigned long \*)(0x4000C038))

#define UART0\_RIS ((volatile unsigned long \*)(0x4000C03C))

#define UART0\_ICR ((volatile unsigned long \*)(0x4000C044))

#define NVIC\_IRQ\_EN0 ((volatile unsigned long \*)(0xE000E100))

// 全局变量

volatile int CurrState; // 状态机

volatile unsigned long TickCounter; // 秒表当前值

volatile int KeyReceived; // 指示用户按下了键

volatile int userinput ; // 用户按下的键

#define IDLE\_STATE 0 // 状态的定义

#define RUN\_STATE 1

#define STOP\_STATE 2

int main (void)

{

int CurrStateLocal; // 局部变量

// 初始化全局变量

CurrState = 0;

KeyReceived = 0;

// 初始化硬件

SetClockFreq(); // 设置时钟

Uart0Init();

SysTickInit();

printf ("Stop Watch\n");

while (1)

{

CurrStateLocal = CurrState; // 建立一个局部的复本

// 因为SysTick ISR随时可能修改它的值

switch (CurrStateLocal) {

case (IDLE\_STATE):

printf ("\nPress any key to start\n");

break;

case (RUN\_STATE):

printf ("\nPress any key to stop\n");

break;

case (STOP\_STATE):

printf ("\nPress any key to clear\n");

break;

default:

CurrState = IDLE\_STATE;

break;

} // end of switch

while (KeyReceived == 0)

{

if (CurrState==RUN\_STATE)

{

DisplayTime();

}

}; // 等待用户输入

if (CurrStateLocal==STOP\_STATE)

{

TickCounter=0;

DisplayTime(); //显示，以指示结果被清

}

else if (CurrStateLocal==RUN\_STATE)

{

DisplayTime(); // 显示结果

}

if (KeyReceived!=0) KeyReceived=0;

}; // end of while loop

} // end of main

void SetClockFreq(void)

{

// Set BYPASS, clear USRSYSDIV and SYSDIV

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xF83FFFFF) | 0x800 ;

// Clr OSCSRC, PWRDN and OEN

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xFFFFCFCF);

// 修改 SYSDIV, 设置 USRSYSDIV 和 Crystal 的值

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xF87FFC3F) | 0x01C002C0;

// 等待直到PLLRIS置位

while ((\*SYSCTRL\_RIS & 0x40)==0); // 等待直到PLLLRIS 置位

// 清除bypass

\*SYSCTRL\_RCC = (\*SYSCTRL\_RCC & 0xFFFFF7FF) ;

return;

}

// UART0 初始化

void Uart0Init(void)

{

\*SYSCTRL\_RCGC1 = \*SYSCTRL\_RCGC1 | 0x0003; // 使能 UART0 & UART1 时钟

\*SYSCTRL\_RCGC2 = \*SYSCTRL\_RCGC2 | 0x0001; // 使能 PORTA 时钟

\*UART0\_CTRL = 0; // 除能 UART

\*UART0\_IBRD = 27; // 基于50MHz编程波特率

\*UART0\_FBRD = 9;

\*UART0\_LCRH = 0x60; // 8 bit, 无奇偶

\*UART0\_CTRL = 0x301; // 使能 TX 和 RX, 并使能 UART

\*UART0\_IM = 0x10; // 使能 UART 接收中断

\*GPIOPA\_AFSEL = \*GPIOPA\_AFSEL | 0x3; // 让UART0控制GPIO管脚

\*NVIC\_IRQ\_EN0 = (0x1<<5); // 在NVIC中使能UART中断

return;

}

// SYSTICK 初始化

void SysTickInit(void)

{

#define NVIC\_STCSR ((volatile unsigned long \*)(0xE000E010))

#define NVIC\_RELOAD ((volatile unsigned long \*)(0xE000E014))

#define NVIC\_CURRVAL ((volatile unsigned long \*)(0xE000E018))

#define NVIC\_CALVAL ((volatile unsigned long \*)(0xE000E01C))

\*NVIC\_STCSR = 0; // 除能 SYSTICK

\*NVIC\_RELOAD = 499999; // 基于50MHz主频的100Hz装载值

\*NVIC\_CURRVAL = 0; // 清除当前值

\*NVIC\_STCSR = 0x7; // 使能SYSTICK，使能中断，使用内核时钟

return;

}

// SYSTICK 异常服务例程

void SysTickHandler(void)

{

if (CurrState==RUN\_STATE) {

TickCounter++;

}

return;

}

// UART0 RX 中断服务例程

void Uart0Handler(void)

{

userinput = getkey();

// 表示收到了按键请求

KeyReceived++;

// 释放UART请求

\*UART0\_ICR = 0x10;

// 状态机转换

switch (CurrState)

{

case (IDLE\_STATE):

CurrState = RUN\_STATE;

break;

case (RUN\_STATE):

CurrState = STOP\_STATE;

break;

case (STOP\_STATE):

CurrState = IDLE\_STATE;

break;

default:

CurrState = IDLE\_STATE;

break;

} // end of switch

return;

}

// 显示时间值

void DisplayTime(void)

{

unsigned long TickCounterCopy;

unsigned long TmpValue;

sendchar(CR);

TickCounterCopy = TickCounter; // 建立一个局部的复本

// 因为SysTick ISR随时可能修改它的值

TmpValue = TickCounterCopy / 6000; // 分钟

PrintValue(TmpValue);

TickCounterCopy = TickCounterCopy - (TmpValue \* 6000);

TmpValue = TickCounterCopy / 100; // 秒

sendchar(':');

PrintValue(TmpValue);

TmpValue = TickCounterCopy - (TmpValue \* 100);

sendchar(':');

PrintValue(TmpValue); // 1/100秒

sendchar(' ');

sendchar(' ');

return;

}

// 显示10进制数值

void PrintValue(int value)

{

printf ("%d", value);

return;

}

// 往UART0送出一个字符（使用printf来输出数据）

int sendchar (int ch)

{

if (ch == '\n')

{

while ((\*UART0\_FLAG & 0x20)); // 如果TXFIFO满则等待

\*UART0\_DATA = CR; // 输出附加的CR，以在超级终端上得到正确的显示

}

while ((\*UART0\_FLAG & 0x20)); // 如果TXFIFO满则等待

return (\*UART0\_DATA = ch); // 输出数据

}

// 获取用户输入

int getkey (void)

{

// 从串口读取字节

while (\*UART0\_FLAG & 0x10); // 如果Rx FIFO空则等待

return (\*UART0\_DATA);

}

// retarget输出

int fputc(int ch, FILE \*f)

{

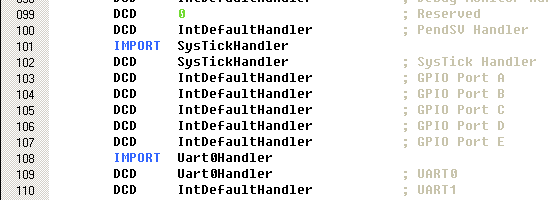
return (sendchar(ch));

}

为使用中断，本例中的UART初始化代码略有改动。在使用中断前，既要设置UART中断掩蔽寄存器，又要设置NVIC来打开对应的外中断。对于SysTick，因为它是NVIC内建的，每个CM3芯片都一样，所以初始化代码也是通用的。

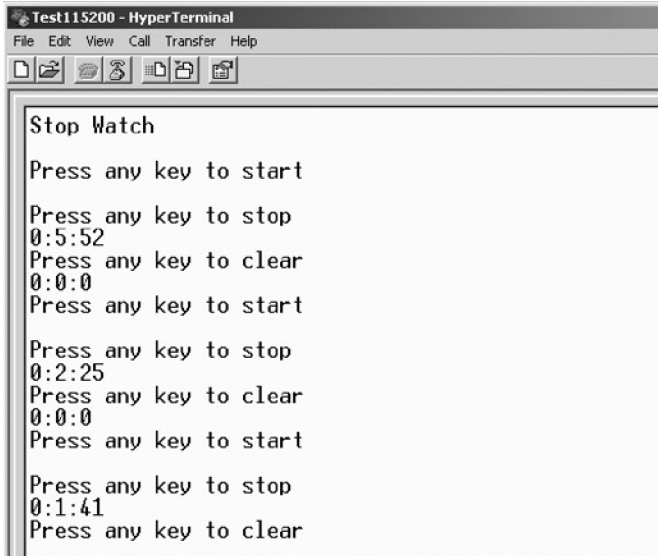
此外，还添加了若干个函数，包括UART和SysTick服务例程、显示函数、以及SysTick初始化的函数。根据外设的不同设计，中断服务例程可能要手工清除中断标志位，也可能由硬件清零。在本例中，是通过UART0\_ICR来手工清除的。

为了让startup.s能认出我们新添加的两个中断服务例程，需要如下修改startup.s



注意IMPORT指示字的使用。它们后面跟着的函数名是由其它源文件实现的函数。有了它，汇编器就知道了这个情况，从而相应地处理。

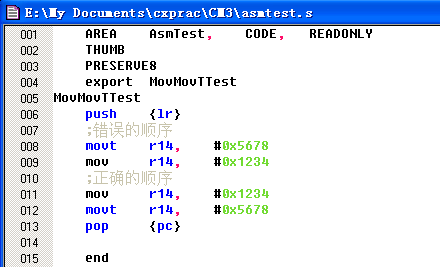
本程序的一次运行情况（亦称为一个实例）如下图所示：



注意：如果使用了虚拟的COM口，则有可能无法正常使用（因为此时按键无法送至目标板），这是虚拟COM口驱动程序的一个bug导致的。如果遇到这种情况，可能要在另一台没有安装过RVMDK的PC上测试这个示例程序。

以下内容由译者添加，对学习第4章很有用。

也可以使用和添加C源文件相同的方式，来添加汇编源文件。只不过汇编源文件的扩展名是.s。下面给也出一个汇编源文件的示例：



这里练习了mov和movt指令（还刻意演示了push/pop），为在C程序中调用”MovMovTTest”，需要先在某个.h或在使用该函数的.c文件中长明该函数：

void MovMovTTest(void);

如果写一个接受参数的函数，方法类似，但是要使用ARM的调用标准，如：有下面的汇编函数：

Add3

add r0, r0, r1

add r0, r0, r2

bx lr

则对应的C声明为：

int Add3(int a, int b, int c); //计算a+b+c

这种例子虽然看起来很低等，但是只是为了抛砖引玉。读者可以用上面例子所演示的骨架，去练习第4章的各种指令；也可以试着把本书中的汇编子程序包装成可以由C调用的函数。