数字信号处理器及其解决方案提供商

CodeCanvas 使用手册

版本V1.0

青岛本原微电子有限公司

目 录

1. CodeCanvas 概述

CodeCanvas 是针对中科本原公司 RV 系列处理器的简单易用集成开发环境(IDE)。该 IDE 基于 Eclipse™，采用最新一代成熟的代码生成工具，提供汇编、C/C++语言的编辑、编译和调试功能。

CodeCanvas 还为开发人员提供驱动器、服务和算法软件模块的高度集成插件支持。此类支持包括外设的驱动器支持、针对目标核专门优化的高性能函数库等。CodeCanvas 为用户提供了易于使用的开发工具，例如工具链、调试器、Flash 烧录工具。直观的 IDE 将引导开发人员完成应用开发流程的每个步骤，熟悉的工具和界面使入门变得简单。

1.1. 优势和特点

- 无需安装，解压即用
- 基于 Eclipse 的集成开发环境(IDE)
- 出色的开发与调试支持
- 出色的代码生成工具，包括编译器、汇编器、连接器和加载器
- 成熟、极致优化的高性能函数库
- 自带外设使用例程，轻松上手

1.2. 系统要求

Windows 10 Professional/Enterprise/ 64 位。

建议使用最低为 2 GHz 的单核处理器或最低 3.3 GHz 的双核处理器。

存储器(RAM)空间不低于 1 GB，建议采用 4 GB 存储器。

要求硬盘(HDD)空间不低于 2GB。

2. 创建工程

有两个创建工程的入口，一个是在第一次运行 eclipse 时，在左侧的工程浏览窗口，可以看到“CC Project”链接（如图 2-1），点击即可打开创建工程向导；另一个入口是点击菜单栏的“file”，然后选择“new”、“CC Project”，如图 2-2。

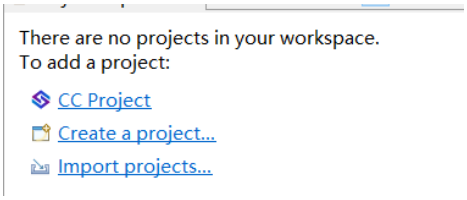


图 2-1 工程选项

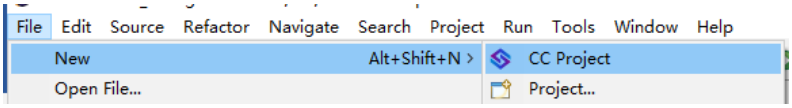


图 2-2 创建工程

如图 2-3，在弹出界面输入工程名，输出类型，工程语言，工具链类型，产品型号，然后点击“Finsh”按钮，IDE 将自动创建对应配置的工程。如果勾选了 with demo，生成工程时会生成外设使用相关代码.

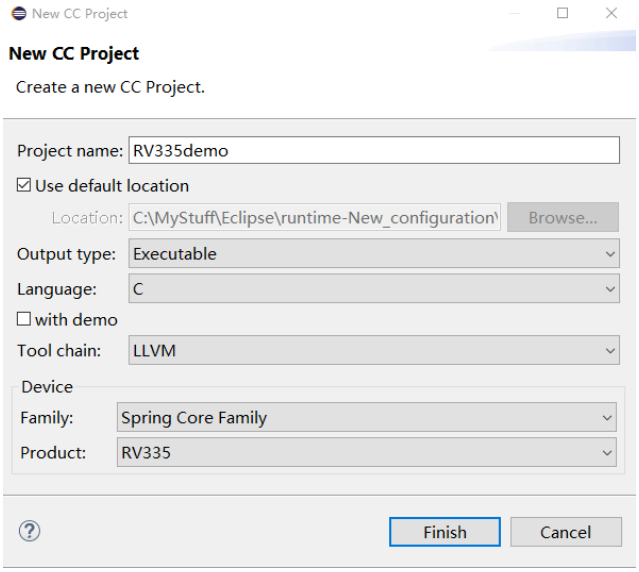


图 2-3 创建工程

新创建的工程将在工程浏览窗口看到，如图 2-4。

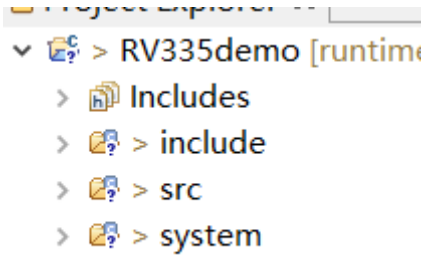


图 2-4 新工程

3. 编译工程

CodeCanvas 使用 Make 进行工程编译管理，在创建工程时，自动生成 makefile 文件，开发人员无需担心文件的添加、修改和删除，Make 采用增量编译，只对上一次编译后发生了改动的文件进行编译，未发生改动的文件不会被编译，节省了编译时间。

3.1. 编译配置

步骤一：先选中工程，然后点击鼠标右键，选择“Properties”。

步骤二：在弹出界面的左侧导航列表中，依次选择“C/C++ Build”、“Settings”。在右侧的“Tool Settings”选项卡，可以对编译器、汇编器、链接器的相关配置进行修改，如图 3-1。

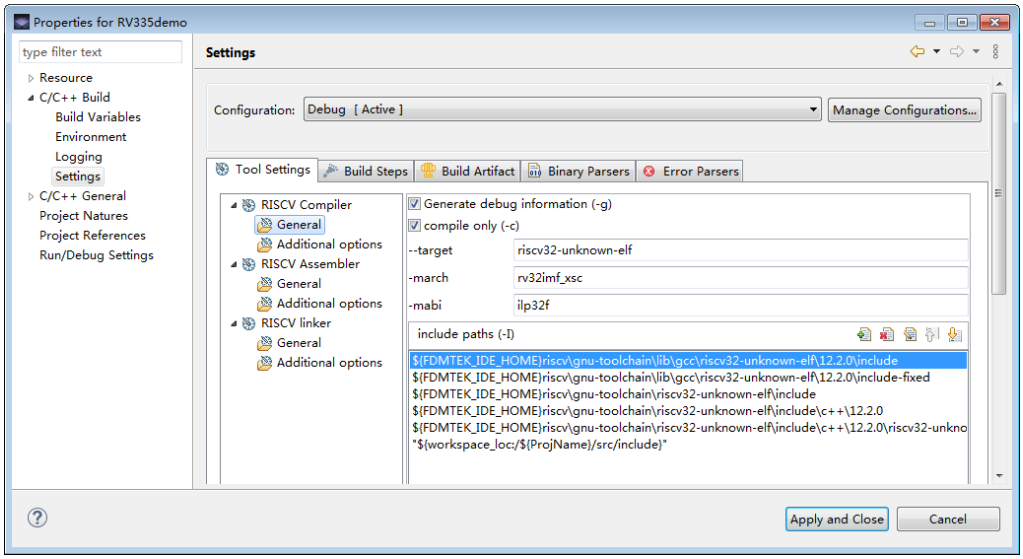


图 3-1 编译配置

如果希望在编译过程中链接静态库，需要在 linker 中进行配置。右键选择要配置的工程，依次点击 Properties => C/C++ Build => Settings => RISCV linker => General，打开链接器配置界面，如图 3-2 所示。在 Library search path 中配置静态库所在路径，在 Libraries 中配置静态库名字，注意静态库名称通常为 libxxx.a，我们配置时只写 xxx，比如我们要链接 libc.a 库，在这里配置时名字就只写 c。

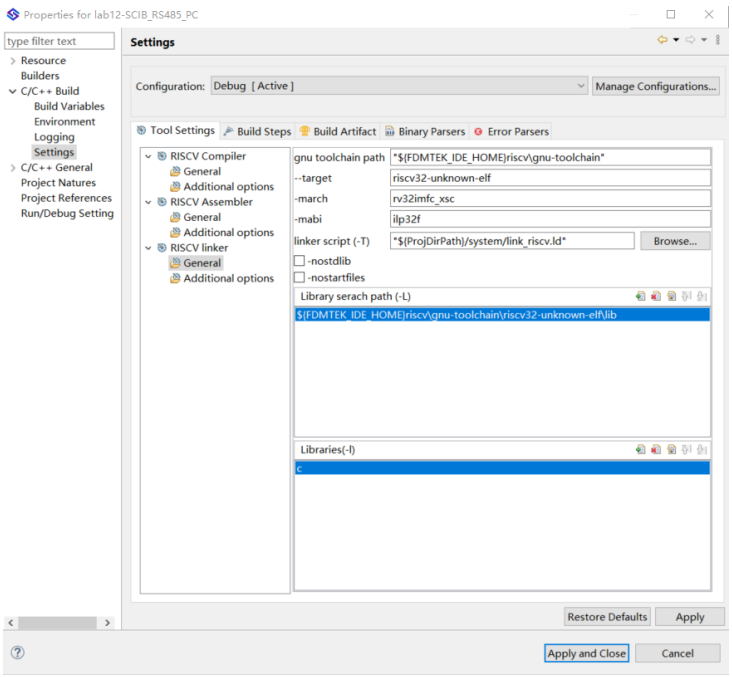


图 3-2 动态链接库

3.2. 编译（build）

IDE 有“Debug”和“Release”两种编译方案，“Debug”方案生成的目标文件包含调试信息，适合用于代码开发过程中的调试。“Release”方案生成的目标文件不含调试信息，因此体积相对小一些，适合用于发布目标文件。用户可以通过工具栏 图标右侧的三角形选择编译方案。

要进行编译有两种方法，第一种方法，选中工程，点击鼠标右键，然后选择“Build Project”，如图所示。另一种方法是点击工具栏的 图标。

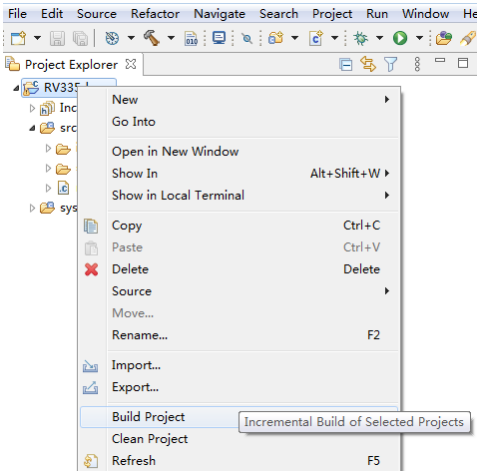


图 3-3 编译

如果编译成功，可以在工程浏览框中发现工程下多了 Binaries 目录，展开目录可以看到生成的目标文件。

3.3. 清空（clean）

如果要删除之前编译生成的所有目标代码文件，可以使用清空功能，有如下两种方法。

第一种：在工程浏览框中选中目标工程，然后点击鼠标右键，选择“Clean Project”。

第二种：在菜单栏，选择“Project”，然后选择“Clean…”，再在弹出的对话框中选择目标工程，然后点击下方的“Clean”按钮，如图所示。

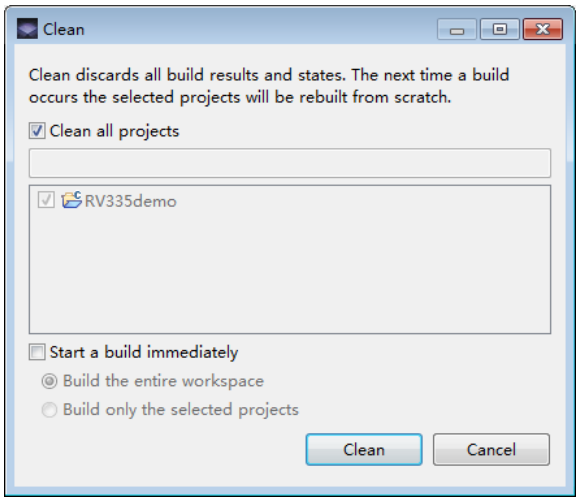


图 3-4 清除

4. 调试

在使用调试工程前，请确保目标工程已经生成了目标文件。

有三种进入调试的方法，下面依次讲解。

第一种方法：在工程浏览框中选择目标工程，然后点击鼠标右键，依次选择“Debug As”、“Debug Configurations”。

第二种方法：点击工具栏 图标右侧的三角形，然后选择“Debug Configurations”。

第三种方法：右键点击工程，选择 Debug as => Application with GDB and OpenOCD。

4.1. 创建调试配置

在前面弹出的窗口的左侧列表中，选中“Application with GDB and OpenOCD”，然后点击鼠标右键，再选择“New Configuration”，在右侧详情页面，可以对配置进行重命名，如下图 4-1。

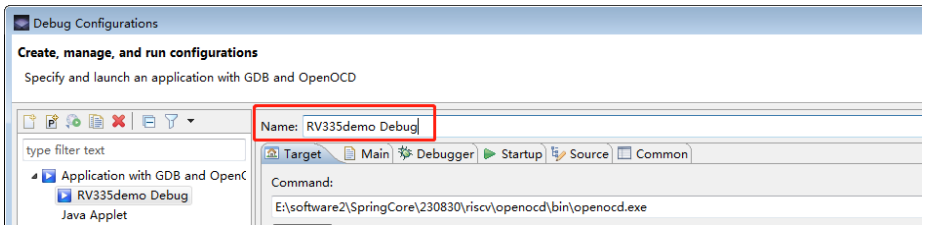


图 4-1 调试配置

4.2. 配置 target

CodeCanvas 会自动选择自带的 openOCD，如果没有特殊原因，开发人员无需修改该配置。

关于“Target（processor）”和“Board”选项，前者对应的 openOCD 配置文件只有核（core）的配置，“Board”选项相对于前者多了外设的配置。

4.3. 配置 Flash

在调试配置界面的 On-chip Flash 选项卡下，可以对 Flash 进行配置，如下图 4-2。具体配置参看第 7 章内容

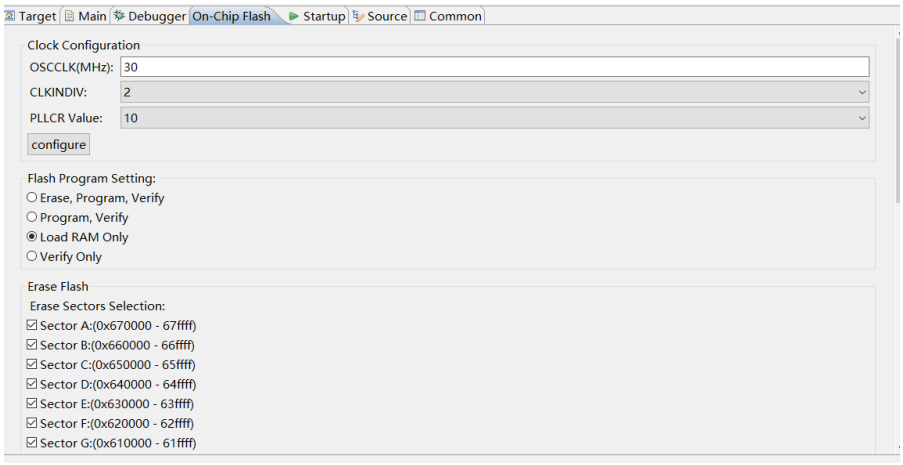


图 4-2

4.4. 设置被调试程序

在 Main 选项卡，不同创建调试配置的方法，该页默认显示的内容不同，有的创建方法会自动选择被调试工程及其目标文件，有的创建方法需要手动指定。如果未指定，则开发人员点击“Browse”按钮进行选择，如下图 4-3。

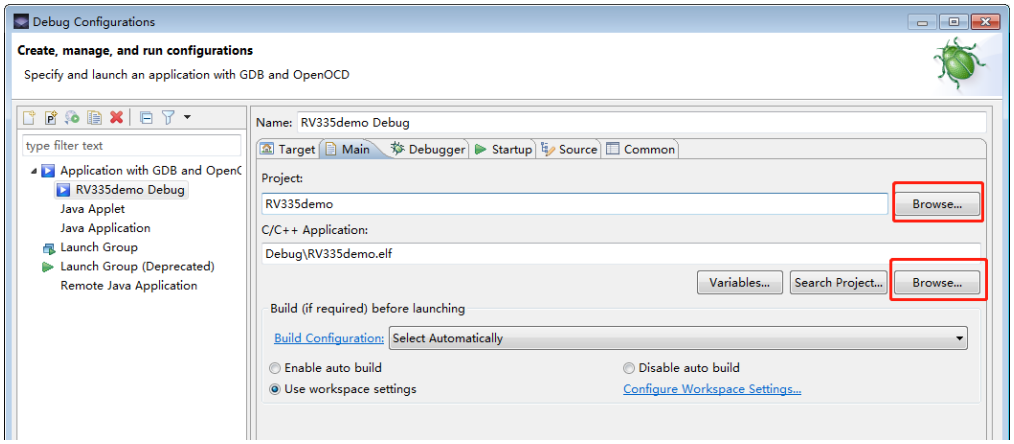


图 4-3 设置

4.5. 配置 gdb

在“Debugger”选项卡，可以设置调试器和 JTAG 信息，如果没有特殊原因，无需修改，使用默认值即可，如图 4-4。

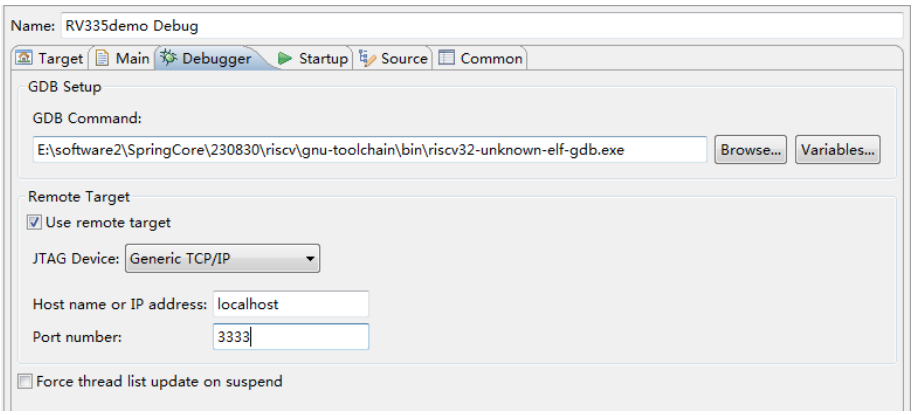


图 4-4 配置 gdb

4.6. 调试动作

4.6.1. 恢复（resume）

恢复挂起的 core，让其全速运行。如图 4-5 所示。

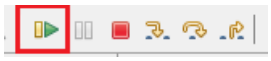


图 4-5 恢复

4.6.2. 挂起（suspend）

让全速运行的 core 挂起，暂停运行，可通过 resume 继续全速运行。如图 4-6 所示。

图 4-6 挂起

4.6.3. 终止（terminate）

终止程序运行。如图 4-7 所示。

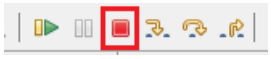


图 4-7 终止

4.6.4. 步入（step into）

单步调试，如果遇到函数调用，则会进入函数内部。如图 4-8 所示。

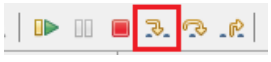


图 4-8 步入

4.6.5. 步过（step over）

单步调试，如果遇到函数调用，将其视为一条普通的 C 语句，直接运行，不会进入函数内部。如图 4-9 所示。

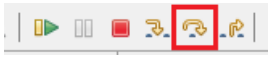


图 4-9 步过

4.6.6. 返回（step return）

从当前位置全速运行，直到函数调用者（caller）处。如图 4-10 所示。

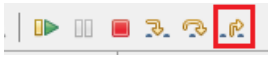


图 4-10 返回

4.7. 断点

"断点"是指在程序执行过程中设置的一个特殊的位置，用于暂停程序的执行，以便程序员可以检查程序的状态、变量的值和执行路径。当程序执行到断点位置时，程序会停止执行，进入调试模式。在调试模式下，程序员可以逐行执行代码，观察变量的值，查看函数的调用栈，以及进行其他调试操作。通过断点，程序员可以逐步跟踪程序的执行流程，观察变量的值的变化，判断程序的逻辑是否正确，找出程序中的错误并进行修复。断点是调试中的重要工具，对于复杂的程序调试和错误排查非常有帮助。

在 CodeCanvas 中，断点相关操纵主要包括设置断点，启用、禁用断点，删除断点等。

4.7.1. 创建断点

在编辑器中，双击行号即可在当前位置设置断点，创建断点后，编辑器在断点位置显示蓝色断点图标，如图 4-11 所示。

```
3  int add(int x, int y){
4      int ret = x + y;
5      return ret;
6  }
7
```

图 4-51 创建断点

4.7.2. 启用、禁用断点

在断点(Breakpoints)视图下，展示所有断点。默认情况下，打开 Debug 透视图后，断点视图在右侧展示，如果没有，可以通过 菜单栏 => Windows => Show View => Breakpoints 调出。通过勾选或取消勾选断点前的选择框可以启用或禁用相应断点。如图 4-12 所示。

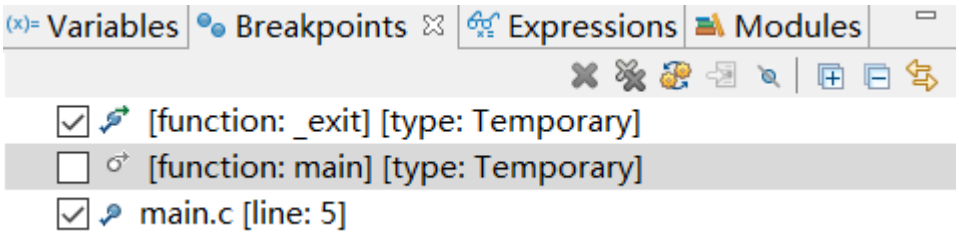


图 4-12 启用/禁用断点

4.7.3. 删除断点

在断点视图中，选中要删除的断点后，点击工具栏的叉号图标(Remove Selected Breakpoint)，即可删除相应断点。或者在编辑器中双击断点位置的行号，也可以删除断点。

在断点视图中，点击工具栏两个叉号的图标(Remove All Breakpoints)即可删除所有创建出的断点。

4.8. 调试视图

调试视图在通过 CodeCanvas 调试程序时起到了关键作用。它为程序员提供了一个集中管理和查看程序执行过程中的关键信息的界面。通过调试视图，程序员可以同时查看变量的值、调用栈、内存与寄存器信息等，从而帮助定位和解决程序中的错误。

常用的调试视图主要包括调用栈视图、反汇编视图、变量视图、表达式视图、内存视图和寄存器视图等。在 Debug 透视图下，可以通过 菜单栏 => Windows => Show View 调出所需要的调试视图。

4.8.1. 调用栈视图

调用栈视图用于显示程序的调用栈信息。调用栈是一个记录函数调用关系的数据结构，用于追踪程序执行过程中的函数调用和返回。在调试过程中，通过查看调用栈视图，程序员可以了解当前程序执行到哪个函数，以及该函数是被哪个函数调用的。

以 main 函数调用 add 函数为例，调用栈视图如图 4-13 所示。越处于外层的函数，在视图中的位置越在下方。其他视图默认展示当前代码所在函数的上下文信息，通过点击对应的函数，可以改变其他视图的展示信息对应当前选中的函数。

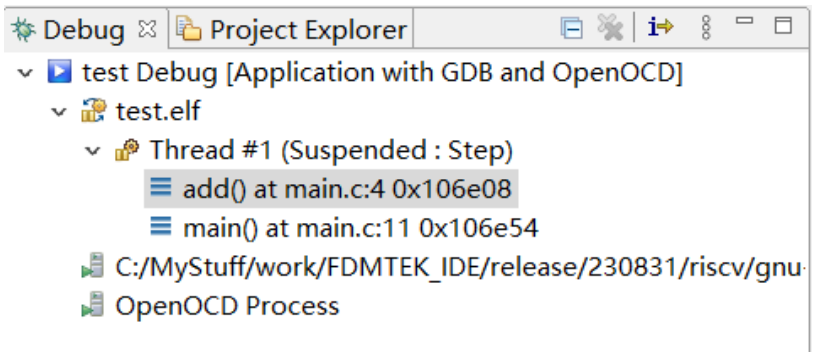


图 4-13 栈视图

4.8.2. 反汇编（Disassembly）视图

反汇编用于显示程序的汇编指令。通过查看反汇编视图，程序员可以了解程序的低级执行过程，分析程序的逻辑和性能。

如图 4-14 所示，反汇编视图主要包括以下几个部分:

1. 内存地址：每条汇编指令的内存地址显示在左侧，方便程序员定位到具体的指令。

2. 汇编指令：每条汇编指令显示为文本形式，表示程序中的机器码指令。

3. 符号名称：如果程序中的某个地址对应于已知的符号（如函数名、变量名），则反汇编视图可能会显示符号名称，方便程序员理解指令所在的上下文。

4. 源文件信息：标记当前汇编代码所对应的源文件内容，方便程序员理解当前程序的功能。

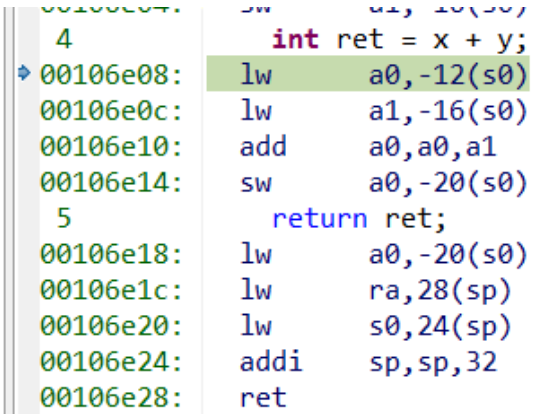


图 4-14 反汇编视图

反汇编视图的工具栏包含一些常用功能方便用户更好的使用反汇编相关功能，如图 4-15 所示。

1. 地址跳转：在文本框输入地址后敲击回车键，可以展示特定地址的反汇编信息。

2. 返回当前位置：在用户查看其他位置反汇编信息后，点击 Home 工具，可以使反汇编视图快速回到程序当前运行位置。

3. 链接源文件：切换是否要在反汇编视图中展示源文件内容。

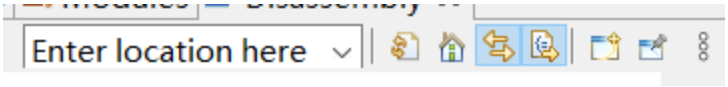


图 4-15 反汇编视图工具链

4.8.3. 变量（Variables）视图

变量视图用于显示程序中各个变量的当前值。通过变量视图，程序员可以实时监测、查看和修改变量的值，帮助调试和分析程序的状态。如图 4-16，在变量视图中，显示以下信息：

1. 变量名称：每个变量在变量视图中以其名称显示，方便程序员识别和查找。

2. 变量类型：变量视图还显示每个变量的数据类型，例如整数、字符、数组等。

3. 变量值：变量视图显示每个变量的当前值。这些值可以是整数、浮点数、字符串等数据类型的实际值。

如果希望修改变量的值，可以点击对应的变量值，输入新值后敲击回车即可完成对变量值的修改。

点击选中变量可以展示多种格式的变量值。点击工具栏上  按钮，可以修改数据展示的默认格式和修改变量视图的布局方式。

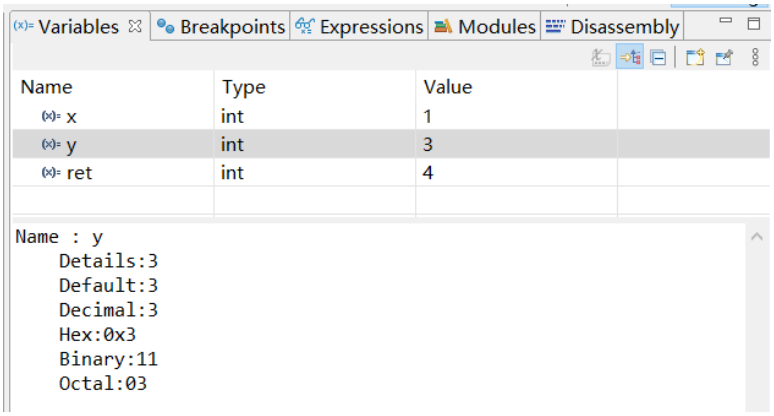


图 4-16 变量视图

4.8.4. 表达式（expression）视图

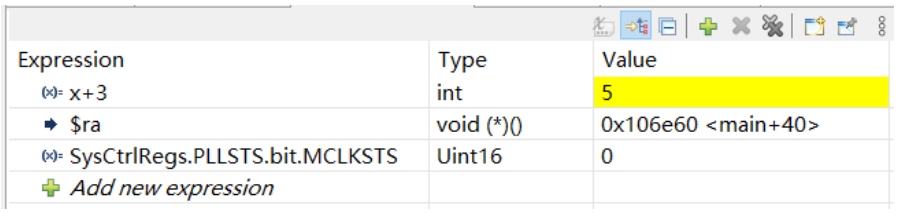
表达式视图用于评估和监视表达式的值。通过表达式视图，程序员可以输入和计算特定的表达式，并实时查看表达式的值，帮助调试和分析程序的状态。它对于动态计算和跟踪表达式的值非常有用，提供了更深入的调试和分析能力。

表达式视图如图 4-17 所示，包含三列，与变量视图类似，分别是表达式，类型，值。

通过点击 Add new expression 添加新的表达式，表达式可以是变量、常量、数学运算、函数调用、寄存器或内存等，输入完成后，敲击回车即可展示表达式的信息，在表达式的值发生变化时，会以黄色背景展示。

如果希望修改表达式的值，只需要点击对应表达式值，输入新之后敲击回车，即可完成对表达式值的修改。注意，涉及到数学运算的表达式无法修改值。

与变量视图类似，点击工具栏上 按钮，可以修改数据展示的默认格式和视图的布局方式。



Expression	Type	Value
x+3	int	5
$ra	void (*)()	0x106e60 <main+40>
SysCtrlRegs.PLLSTS.bit.MCLKSTS	Uint16	0
Add new expression		

图 4-17 表达式视图

4.8.5. 内存（memory）视图

内存视图帮助开发人员在调试程序时查看和修改内存中的数据。如图 4-18 所示，内存数据作为一组内存监视器(Monitor)显示。每个监视器都代表由其位置（称为基地址）指定的内存部分。每个内存监视器都可以采用不同的预定义数据格式（内存呈示）显示。调试器支持五种呈示类型，即十六进制（缺省）、ASCII、有符号整数、无符号整数和十六进制整数。创建监视器时，将自动显示缺省呈示。

通过左侧 Monitors 旁的绿色加号可以新建一个内存监视器，在弹出的对话框中输入要监看的内存基地址，点击 OK 后即可新建一个内存监视器以展示内存数据。

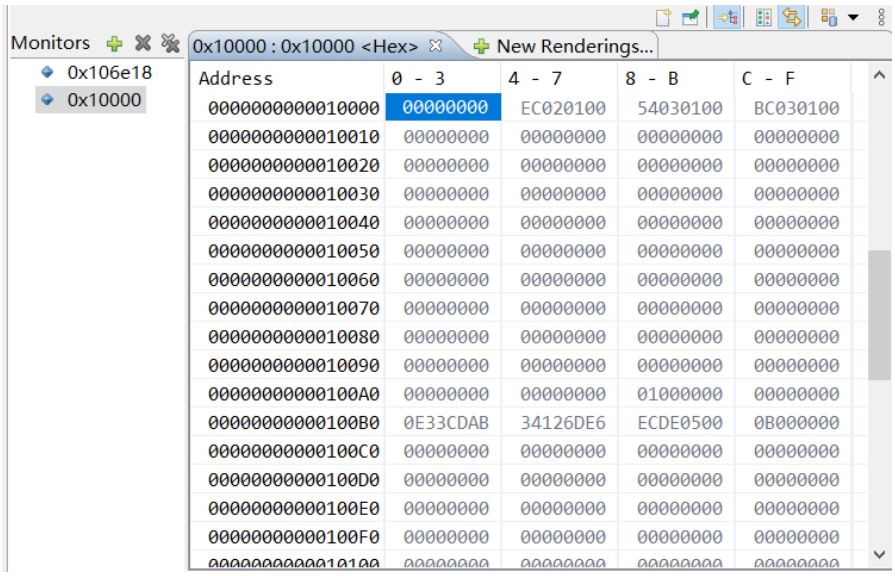


图 4-18 内存视图

通过点击上方的 New Renderings 可以选择展示其他格式的内存数据。如图 4-19 所示，选择所需格式后点击 Add Rendering(s)按钮即可添加新的展示格式。

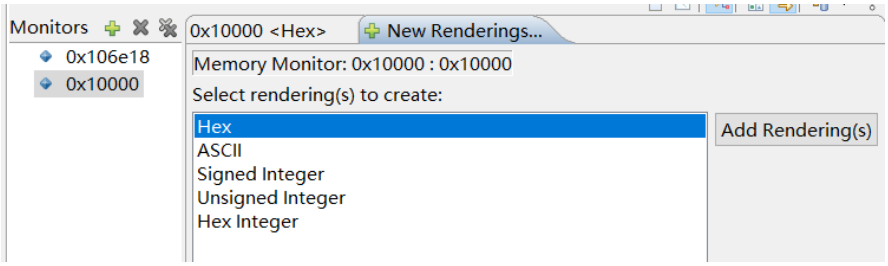


图 4-19 数据格式

在内存视图中点击右键，可以弹出上下文菜单。其中，可以通过 Format 调整内存每行数据量格式。通过 Add Watchpoint(C/C++)可以为指定内存位置添加观察点。

4.8.6. 寄存器（register）视图

寄存器视图是调试提供了一个界面来查看和修改内核的寄存器。在寄存器视图中，开发人员可以看到 CPU 或外设中的寄存器及其当前的值。通过查看寄存器的值，开发人员可以了解程序执行的当前状态，包括变量的值、函数调用的参数和返回值等。

寄存器视图如图 4-20 所示。共三列，包括寄存器名(或寄存器组名，寄存器位域名)，寄存器值和描述信息，点击选中寄存器可以在下方展示多种格式的值。发生变化的寄存器用黄色背景标注。如果要修改寄存器的值，可以点击相应寄存器的值，输入新值，敲击回车后即可完成对寄存器值的修改。

与变量和表达式视图类似，点击工具栏上 按钮，可以修改数据展示的默认格式和视图的布局方式。

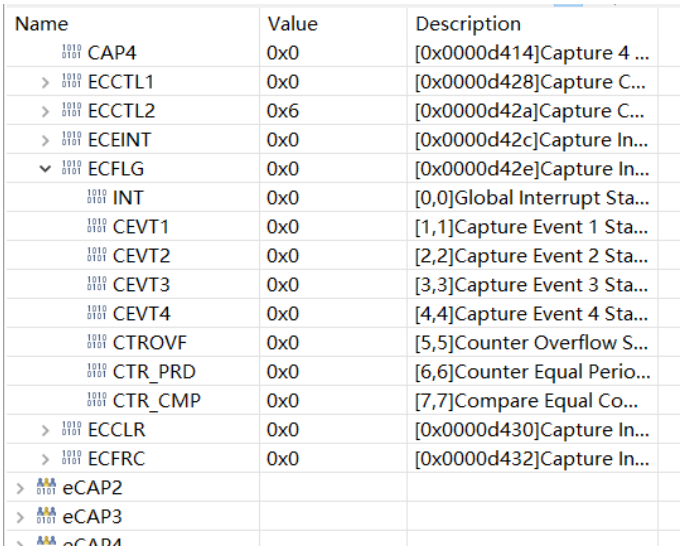


图 4-20 寄存器视图

5. 代码编辑

5.1. 编辑器（editor）

开发人员可以通过在“工程浏览”视图中，双击目标文件，在编辑器中会显示目标文件内容，开发人员可以在此对文件进行编辑，如图 5-1。

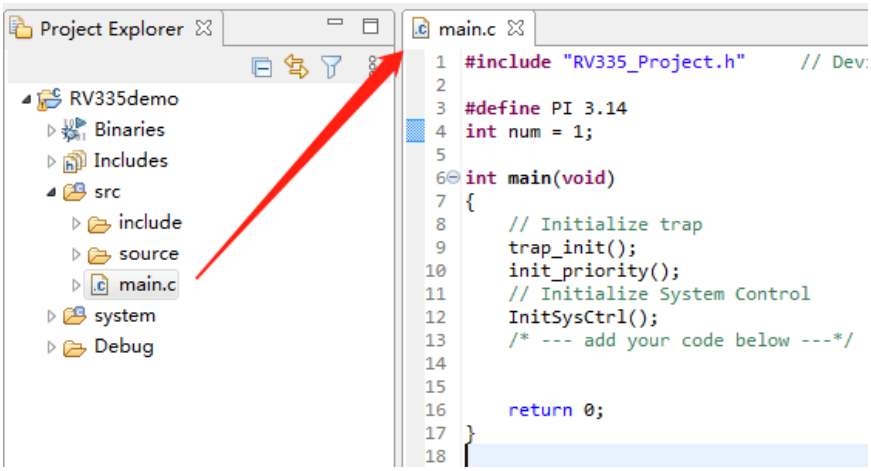


图 5-1 编辑器

5.1.1. 字体

点击菜单栏选项“Window”，然后选择“Preferences”，在弹出的对话框左侧列表中，依次选择“General”、“Appearance”、“Colors and Fonts”，再在右侧的列表中，依次选择“Basic”、“Text Font”，然后点击右侧的“Edit…”按钮，如图 5-2。

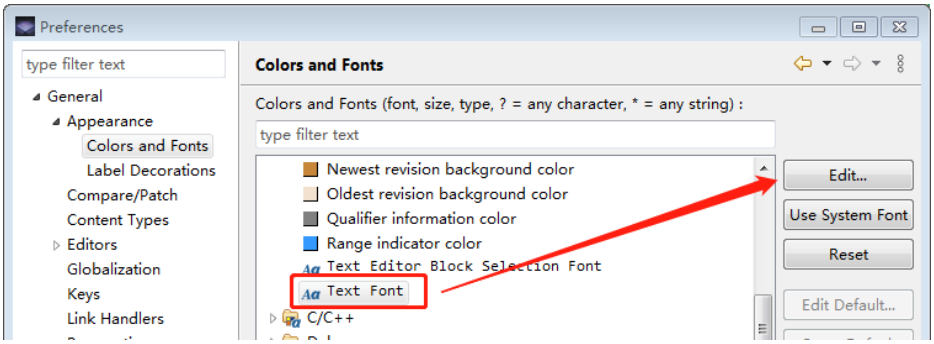


图 5-2 字体

在弹出界面可以对字体、字形和大小进行设置，如图 5-3，最后点击图 5-2 右下角的“Apply”按钮来应用修改。

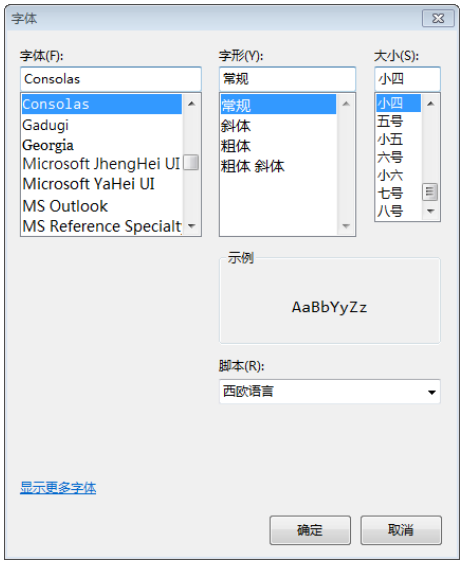


图 5-3 设置

5.1.2. 行号

点击菜单栏选项“Window”，然后选择“Preferences”，在弹出的对话框左侧列表中，依次选择“General”、“Editors”、“Text Editors”，然后在右侧配置项中，勾选“Show line numbers”则显示行号，反之则不显示行号，如图 5-4 和图 5-5 所示. 最后点击下方的“Apply and Close”按钮应用修改。

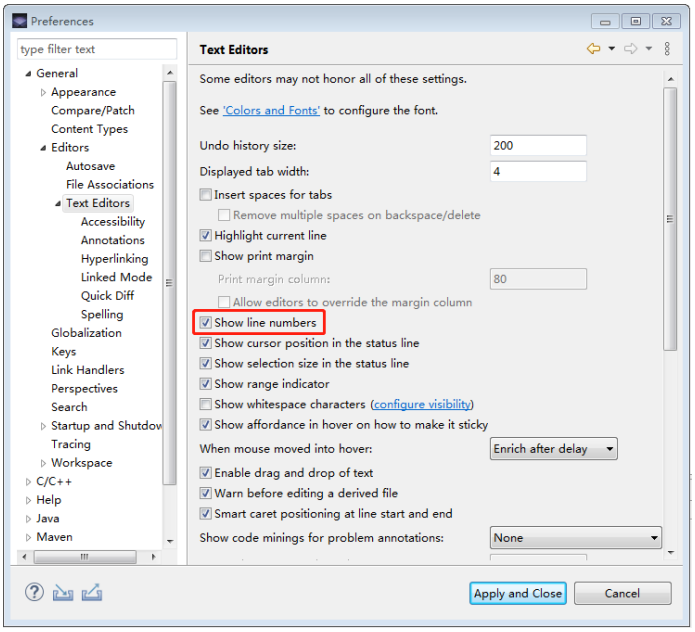


图 5-4 行号设置

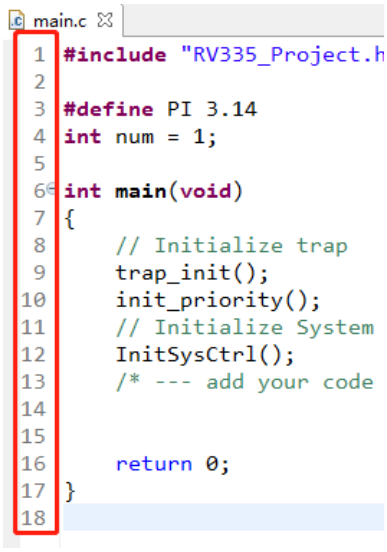


图 5-5 行号设置

5.1.3. 背景色

点击菜单栏选项“Window”，然后选择“Preferences”，在弹出的对话框左侧列表中，

依次选择“General”、“Editors”、“Text Editors”，然后在右侧配置项中，找到“Appearance color options”列表，在列表中选中“Background color”，然后点击右侧“Color：”旁边的颜色进行修改，如图 5-6，最后点击下方的“Apply and Close”按钮应用修改。

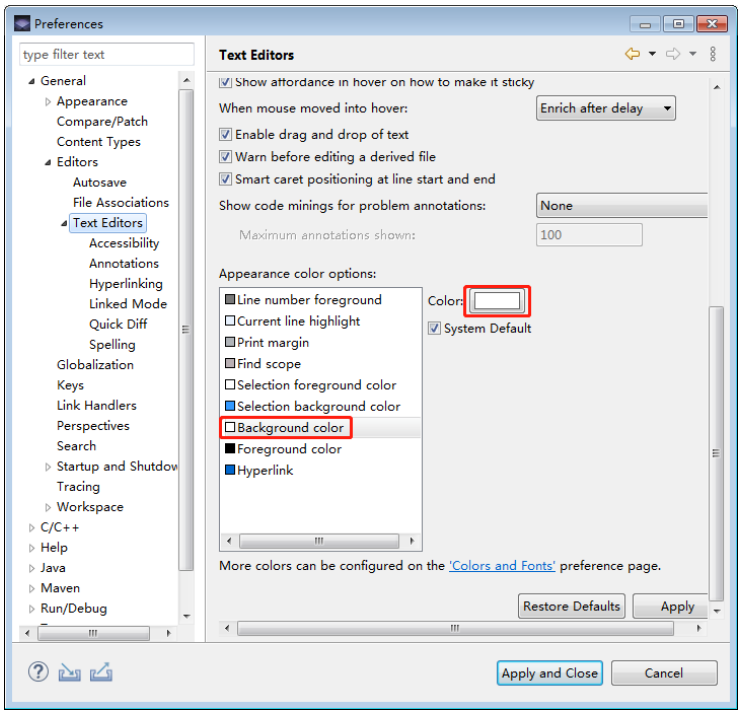


图 5-6 背景设置

5.1.4. 空白字符

点击菜单栏选项“Window”，然后选择“Preferences”，在弹出的对话框左侧列表中，依次选择“General”、“Editors”、“Text Editors”，然后在右侧配置项中，勾选“Show whitespace characters”则显示空白字符（如图 5-7），反之则不显示，最后点击下方的“Apply and Close”按钮应用修改，如图 5-8。

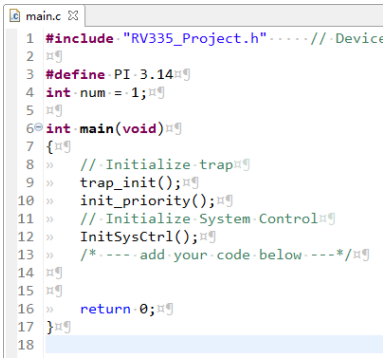


图 5-7 空白字符

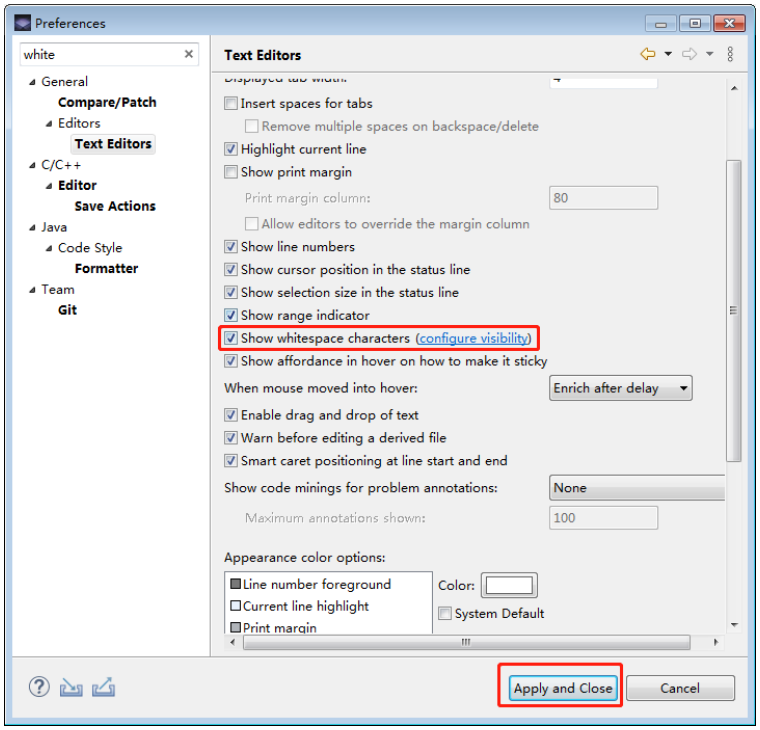


图 5-8 空白字符

5.2. 透视图（Perspective）

Eclipse 的透视图提供了一种灵活的方式来组织和管理工作空间，以适应不同的开发任务和工作流程。通过自定义透视图的布局、工具栏和快捷键，可以提高开发效率并提供更好的用户体验。

透视图在 eclipse 的右上角展示，开发人员可以通过点击这些透视图图标来实现透视图切换，如图 5-9，也可以点击左侧的图标，在弹框中选择其他透视图。

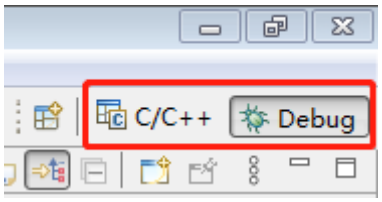


图 5-9 透视图

5.3. 开发视图

5.3.1. 工程浏览器（project explorer）

开发人员可以在工程浏览器中查看工程及工程内的文件，可以实现文件的添加、修改、删除、重命名等操作，也支持 ctrl + c 的文件复制和 ctrl + v 的文件粘贴等快捷键。

通过点击图标（如图 5-10），开启编辑器中文件在工程浏览器中的自动定位。

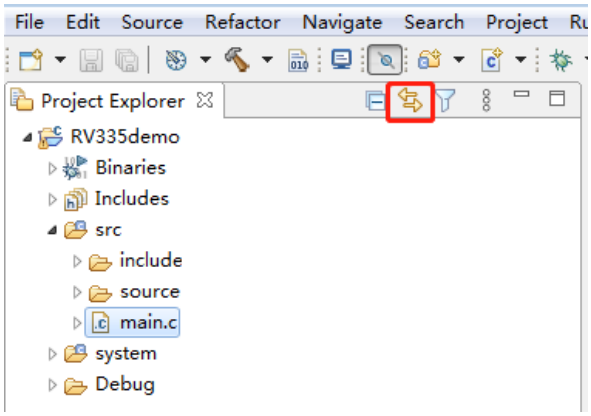


图 5-10 工程浏览器

5.3.2. 大纲（outline）视图

在打开 C 文件后，在右侧的大纲视图中，会显示当前文件中定义的宏、全局变量、函数，开发人员通过点击大纲视图中的元素，可以快速定位到元素位置，如图 5-11。

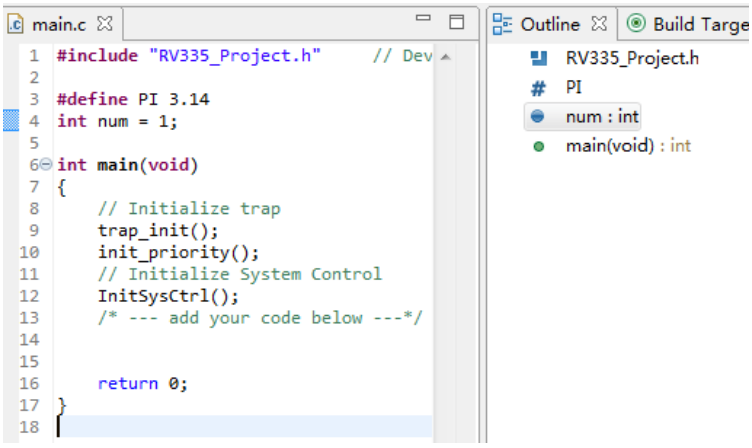


图 5-11 大纲视图

6. TI2RV 转换工具

TI2RV 转换工具用于将 TI 平台的 C 代码转换为 FDM320RV335 上可以运行的代码，基于文本替换，为源代码到源代码的转换。

6.1. 目录结构

TI2RV 的目录结构如图 6-1 所示，共包含四个文件，其中:

1. rules.txt 为转换规则文件。
2. ti2rv.jar 为转换程序 jar 包。
3. ti2rvjat.bat 为对 ti2rv.jar 的封装。
4. ti2rv-all.bat 为批量转换脚本，会对当前目录以及子目录下所有 C 文件进行转换。
5. 规则说明.xlsx 提供 TI2RV 转换规则的说明以供编写 rules.txt 时参考。

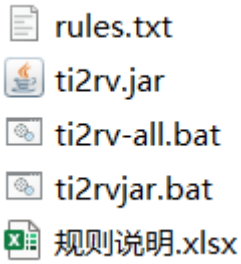


图 6-1 目录结构

6.2. 使用说明

1. 编写 rules.txt 转换规则。规则文件由一组或多组大括号包围的逻辑处理块组成，每个处理块内描述了如何对源文件进行处理，主要包含以下内容:
   a) op: 操作类型。如 del(删除), add(新增), rpl(替换)等。
   b) obj: 执行位置。如 glbl_var(全局变量), func_dcl(函数声明), blck_sntn(块内)等。
   c) times: 执行操作的次数。包括 once(只处理第一次匹配的内容), infi(无限次)。
   d) pat: 匹配的原始内容，支持正则表达式。
   e) to: 对于替换操作，要替换为的内容。

详细的规则说明，参考 规则说明.xlsx 表格文件

2. 运行转换程序，可以采用以下两种方式:

- ti2rv.jar -v -c <要处理的原始文件> -r <规则文件> -t <生成的新文件>
- 直接运行 ti2rv-all.bat 脚本, 自动递归对当前目录下所有 C 文件执行转换, 转换规则文件使用当前目录下的 rules.txt。

7. Flash

IDE 提供 Flash 相关操作，包括擦除、烧录、验证以及安全域相关操作等。

7.1. Flash 操作入口

IDE 提供了两个 Flash 操作的入口，分别在菜单栏 => Tools => On-Chip Flash 和 Debug Configuration 中的 On-Chip Flash 标签页下. 两种入口打开后界面内容是一样的. 如果是通过 Debug Configuration 打开的页面，调试或运行时会自动根据在 Flash 界面进行的配置对 Flash 进行相应的操作.

7.2. 时钟配置

如图 7-1，输入 OSCCLK，选择 CLKINDIV 和 PLLCR Value 后，点击 configure 按钮，稍等便可以完成时钟配置，配置成功后，会弹出 Config Clock Success 提示对话框.

图 7-1 时钟配置

7.3. Flash 操作设置

如图 7-2，通过 Debug Configuration 打开 Flash 界面，可以选择 Flash 操作的序列，分别为 Erase, Program, Verify (先擦除，再烧录，最后验证); Program, Verify(先烧录后验证); Load RAM Only(不进行任何 Flash 相关操作); Verify Only(仅进行验证).

在这里进行的配置会与后面设置的 flash 具体操作内容结合起来，共同决定在进行 Debug 或 Run 时如何对 flash 进行操作.

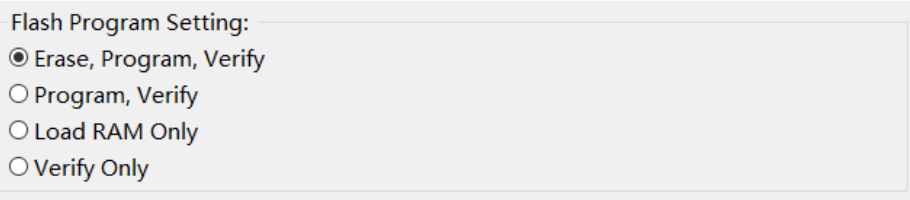


图 7-2 操作配置

7.4. 擦除

如图 7-3 所示，在 Erase Flash 区域可以擦除 flash 扇区，勾选要擦除的扇区后点击 Erase Flash 按钮后稍等即可擦除指定的扇区，擦除成功后会弹出 success 对话框.

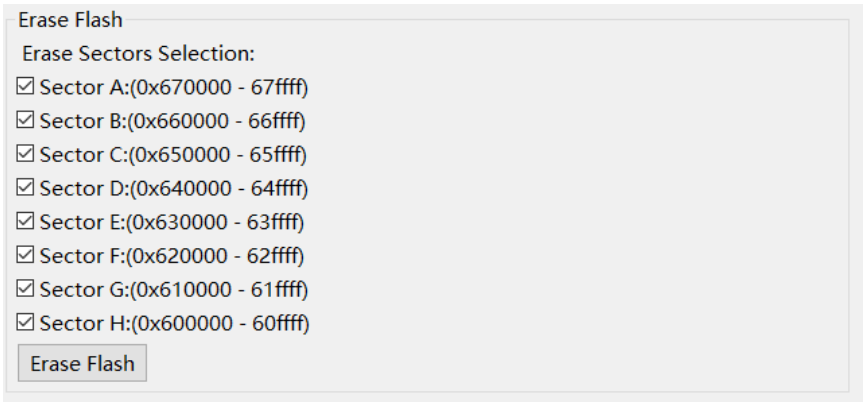


图 7-3 擦除

7.5. 烧录与验证

如图 7-4 所示，在此区域可以对 flash 进行烧录和验证. 通过 select 按钮选择程序，填写要烧录的地址，然后便可以通过 Program Flash 进行烧录或通过 Verify 按钮进行验证. 如果选择的程序是 elf 文件，则无需填写烧录地址.

图 7-4 烧录与验证

7.6. 安全域操作

如图 7-5 所示，在 Code Security 中可以对 Flash 安全域进行操作. 在未上锁状态，可以在 keys 中填写好新密码后，通过 Program Password 按钮设置新密码，设置新密码后会自动上锁，也可以通过 Lock 按钮手动上锁. 在上锁状态下，可以在 keys 中填写好密码后通过 Unlock 进行解锁操作

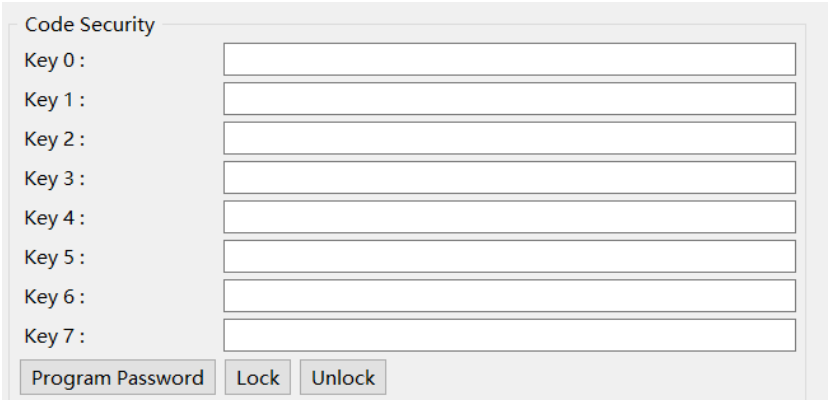


图 7-5 安全域

7.7. 频率测试

如图 7-6 所示，配置 GPIO 引脚后，点击 Start Frequency Test 按钮，即可进行频率测试.

图 7-6 频率测试

7.8. 校验和

如图 7-7，点击 Calculate Checksum 按钮，即可在上面文本框中生成 Flash 和 OTP 各自的校验和

图 7-7 校验和

7.9. Flash Boot

在程序调试完善后，如果希望固化到 Flash 中，并且板卡上电后就自动执行其中的程序，需要将 IDE 工程模板中 system 文件夹下的 fdm_start.s 和 link_riscv.ld 两个文件用 fdm_start.s_flash 和 link_riscv.ld_flash 分别替换掉，替换完成后，重新编译程序并烧录到 flash 中即可。

8. 其他工具

8.1. Elf2bin

Elf2bin 工具用于将 elf 文件转换为 boot 使用 bin 格式文件. 打开工具的入口在 IDE 菜单栏 => Tools => Elf2Bin Tool. 打开后界面如图 8-1 所示.

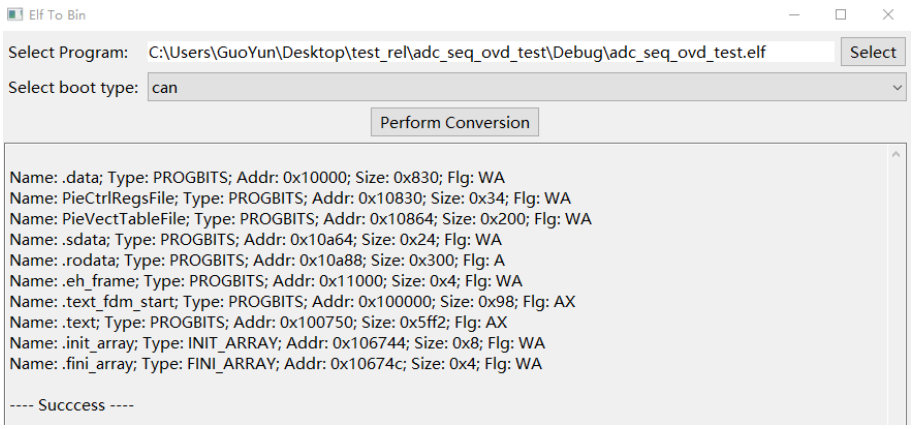


图 8-1 转换工具

首先点击 Select 按钮选择要转换的 elf 文件, 然后选择 boot 的类型, 接着点击 Perform Conversion 按钮, 即可在下方文本框中获取到转换的结果.

8.2. 连接测试

连接测试工具用于检查 IDE 与仿真器和板卡的连接是否可用. 工具的入口在 IDE 菜单栏 => Tools => Test Connection. 打开后界面如图 8-2 所示. 首先选择仿真器型号和连接目标型号, 对于 RV335, interface 选择 jlink.cfg, target 选择 sc001.cfg. 然后点击 Test Connection, 即可在下面的文本框中展示连接测试结果.

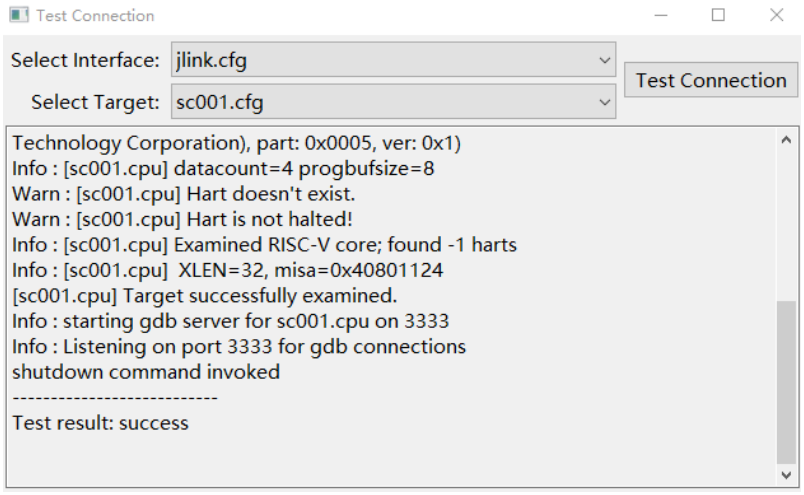


图 8-2 连接测试

9. 附录 A：Intrinsic

9.1. 定点函数

9.1.1. __builtin_riscv_asr64

- 函数原型

```
int64_t __builtin_riscv_asr64(int64_t rs1, int rs2);
```

- 汇编指令

```
ASR64 rd, rs1, rs2
```

- 功能描述

该函数进行 64 比特整数算术右移位操作，高位以符号位填充，移位长度由 rs2 指定。

- 函数参数

参数	类型	说明
rs1	int64_t	被移位整型数据。
rs2	int	移位长度，取值范围 0~63。

- 返回值

参数	类型	说明
return val	int64_t	移位后结果。

9.1.2. __builtin_riscv_dmac

- 函数原型

```
int64_t __builtin_riscv_dmac(int rs1, int rs2, int rs3);
```

- 汇编指令

```
DMAC rd, rs1, rs2 [init] [out]
```

- 功能描述

该指令将整型数据 rs1、rs2 分割成高低 16bit 两部分，进行对应乘加运算。

- 函数参数

参数	类型	说明
rs1	int	乘累加数据 1
rs2	int	乘累加数据 2。
rs3	整型字面值	取值范围 0~3，rs3[0]（最低位）如果为 1，则会重置 MR0、MR1 寄存器（添加 init 标记）。rs3[1]如果为 1，则表示获取乘累加结果（添加 out 标记）。

- 返回值

参数	类型	说明
return val	int64_t	饱和处理后的乘累加结果。

9.1.3. __builtin_riscv_flip

- 函数原型

```
int __builtin_riscv_flip(int rs1);
```

- 汇编指令

```
FLIP rd, rs1
```

- 功能描述

该函数进行对传入参数的低 16 位进行比特翻转操作，高位保持不变。

- 函数参数

参数	类型	说明
rs1	int	被翻转数据

- 返回值

参数	类型	说明
return val	int	翻转后数据。

9.1.4. __builtin_riscv_get_mr

- 函数原型

```
int __builtin_riscv_get_mr(int imm);
```

- 汇编指令

```
MV.X.MR rd, #imm1
```

- 功能描述

获取指定寄存器 MR 的数据。

- 函数参数

参数	类型	说明
imm	整型字面值	有效值{0,1}。0 对应 MR0，1 对应 MR1

- 返回值

参数	类型	说明
return val	int	指定寄存器 MR 的数据。

9.1.5. __builtin_riscv_get_mr

- 函数原型

```
void __builtin_riscv_set_mr(int imm, int rs1);
```

- 汇编指令

```
MV.MR.X #imm1, rs1
```

- 功能描述

给指 MR 定寄存器赋值。

- 函数参数

参数	类型	说明
imm	整型字面值	有效值{0,1}。0 对应 MR0，1 对应 MR1。
rs1	int	目标值。

- 返回值

无。

9.1.6. __builtin_riscv_lsl64

- 函数原型

```
int64_t __builtin_riscv_lsl64(int64_t rs1, int rs2);
```

- 汇编指令

```
LSL64 rd, rs1, rs2
```

- 功能描述

进行 64 比特寄存器对逻辑左移位操作，低位以 0 填充，移位长度由 rs2 指定。

- 函数参数

参数	类型	说明
rs1	int64_t	将被移位数据。
rs2	int	移位范围为 0~63。

- 返回值

参数	类型	说明
return val	int	移位后结果。

9.1.7. __builtin_riscv_lsr64

- 函数原型

```
int64_t __builtin_riscv_lsr64(int64_t rs1, int rs2);
```

- 汇编指令

```
LSR64 rd, rs1, rs2
```

- 功能描述

进行 64 比特寄存器对逻辑右移位操作，高位以 0 填充，移位长度由 rs2 指定。

- 函数参数

参数	类型	说明
rs1	int64_t	将被移位数据
rs2	int	移位范围为 0~63。

- 返回值

参数	类型	说明
return val	int	移位后结果。

9.1.8. __builtin_riscv_max

- 函数原型

```
int __builtin_riscv_max(int rs1, int rs2);
```

- 汇编指令

```
MAX rd, rs1, rs2
```

- 功能描述

该函数比较两个有符号整数，返回其中较大者。

- 函数参数

参数	类型	说明
rs1	int	有符号整数 1
rs2	int	有符号整数 2

- 返回值

参数	类型	说明
return val	int	两个数中的较大者。

9.1.9. __builtin_riscv_min

- 函数原型

```
int __builtin_riscv_min(int rs1, int rs2);
```

- 汇编指令

```
MIN rd, rs1, rs2
```

- 功能描述

该函数比较两个有符号整数，返回其中较小者。

- 函数参数

参数	类型	说明
rs1	int	有符号整数 1
rs2	int	有符号整数 2

- 返回值

参数	类型	说明
return val	int	两个数中的较小者。

9.1.10. __builtin_riscv_mpya

- 函数原型

```
int __builtin_riscv_mpya(int rs1, int rs2, int imm);
```

- 汇编指令

```
MPYA rd, rs1, rs2 [init] [out]
```

- 功能描述

该函数进行并行的乘累加运算。

- 函数参数

参数	类型	说明
rs1	int	乘累加参数 1
rs2	int	乘累加参数 2
imm	整数字面值	取值范围 0~3，rs3[0]（最低位）如果为 1，则会重置 MR0（等价于 init 标记）。rs3[1]如果为 1，则表示获取乘累加结果（等价于 out 标记）。

- 返回值

参数	类型	说明
return val	int	饱和处理后的乘累加结果。要获取结果，rs3[1]需要设置为 1，即添加 out 标记。

9.1.11.__builtin_riscv_mpys

- 函数原型

```
int __builtin_riscv_mpys(int rs1, int rs2, int imm);
```

- 汇编指令

```
MPYS rd, rs1, rs2 [init] [out]
```

- 功能描述

该函数进行并行的乘累减运算。

- 函数参数

参数	类型	说明
rs1	int	乘累减参数 1
rs2	int	乘累减参数 2
imm	整数字面值	取值范围 0~3，rs3[0]（最低位）如果为 1，则会重置 MR0（等价于 init 标记）。rs3[1]如果为 1，则表示获取乘累加结果（等价于 out 标记）。

- 返回值

参数	类型	说明
return val	int	饱和处理后的乘累减结果。要获取结果，rs3[1]需要设置为 1，即添加 out 标记。

9.1.12.__builtin_riscv_qmpya

- 函数原型

```
int __builtin_riscv_qmpya(int rs1, int rs2, int imm);
```

- 汇编指令

```
QMPYA rd, rs1, rs2 [init] [out]
```

- 功能描述

该函数进行并行的乘累加运算，其中乘法结果取乘积的高 32bit。

- 函数参数

参数	类型	说明
rs1	int	乘累加参数 1
rs2	int	乘累加参数 2
imm	整数字面值	取值范围 0~3，rs3[0]（最低位）如果为 1，则会重置 MR0（等价于 init 标记）。rs3[1]如果为 1，则表示获取乘累加结果（等价于 out 标记）。

- 返回值

参数	类型	说明
return val	int	饱和处理后的乘累加结果。要获取结果，rs3[1]需要设置为 1，即添加 out 标记。

9.1.13. __builtin_riscv_qmpys

- 函数原型

```
int __builtin_riscv_qmpys(int rs1, int rs2, int imm);
```

- 汇编指令

```
QMPYS rd, rs1, rs2 [init] [out]
```

- 功能描述

该函数进行并行的乘累减运算，其中乘法结果取乘积的高 32bit。

- 函数参数

参数	类型	说明
rs1	int	乘累减参数 1
rs2	int	乘累减参数 2
imm	整数字面值	取值范围 0~3，rs3[0]（最低位）如果为 1，则会重置 MR0（等价于 init 标记）。rs3[1]如果为 1，则表示获取乘累加结果（等价于 out 标记）。

- 返回值

参数	类型	说明
return val	int	饱和处理后的乘累减结果。要获取结果，rs3[1]需要设置为1，即添加 out 标记。

9.1.14.__builtin_riscv_restore

- 函数原型

```
void __builtin_riscv_restore();
```

- 汇编指令

```
RESTORE
```

- 功能描述

该函数快速从 shadow 寄存器加载到架构寄存器，是__builtin_riscv_save()函数的逆操作。

- 函数参数

无。

- 返回值

无。

9.1.15.__builtin_riscv_save

- 函数原型

```
void __builtin_riscv_save ();
```

- 汇编指令

```
SAVE
```

- 功能描述

该函数快速将架构寄存器保存到 shadow 寄存器。对 X 寄存器做硬件现场保护，对 F 寄存器做软件保护，其它如 CSR 寄存器为软件保护。

- 函数参数

无。

- 返回值

无。

9.1.16.__builtin_riscv_rol

- 函数原型

```
int __builtin_riscv_rol(int rs1, int rs2);
```

- 汇编指令

```
ROL rd, rs1, rs2
```

- 功能描述

函数对 rs1 进行循环左移位操作，移位长度由 rs2 指定。

- 函数参数

参数	类型	说明
rs1	int	将被循环左移数据。
rs2	int	左移位数，有效值范围 0～31。

- 返回值

参数	类型	说明
return val	int	循环左移后的数据。

9.1.17.__builtin_riscv_ror

- 函数原型

```
int __builtin_riscv_ror(int rs1, int rs2);
```

- 汇编指令

```
ROR rd, rs1, rs2
```

- 功能描述

函数对 rs1 进行循环右移位操作，移位长度由 rs2 指定。

- 函数参数

参数	类型	说明
rs1	int	将被循环右移数据。
rs2	int	右移位数，有效值范围 0～31。

- 返回值

参数	类型	说明
return val	int	循环右移后的数据。

9.1.18.__builtin_riscv_sadd

- 函数原型

```
int __builtin_riscv_sadd(int rs1, int rs2);
```

- 汇编指令

```
SADD rd, rs1, rs2
```

- 功能描述

函数将两个有符号数整型数据相加，结果做饱和处理。

- 函数参数

参数	类型	说明
rs1	int	相加参数 1
rs2	int	相加参数 2

- 返回值

参数	类型	说明
return val	int	饱和处理后的相加结果。

9.1.19.__builtin_riscv_ssub

- 函数原型

```
int __builtin_riscv_ssub(int rs1, int rs2);
```

- 汇编指令

```
SSUB rd, rs1, rs2
```

- 功能描述

函数将两个有符号数整型数据相减，结果做饱和处理。

- 函数参数

参数	类型	说明
rs1	int	相减参数 1
rs2	int	相减参数 2

- 返回值

参数	类型	说明
return val	int	饱和处理后的相减结果。

9.2. 单浮点函数

9.2.1. __builtin_riscv_fracf32

- 函数原型

```
float __builtin_riscv_fracf32(float rs1);
```

- 汇编指令

```
FRACF32 rd, rs1
```

- 功能描述

返回浮点数据的小数部分。

- 函数参数

参数	类型	说明
rs1	float	将被截取数据

- 返回值

参数	类型	说明
return val	float	小数部分。