MDT-IDE

使用手册

目录

第一部分　整合的开发环境

第一章　概要与安装

MDT-IDE 整合开发环境

MDT-IDE 是一个具有高效能，用于设计汇春 8 位 MCU 程序的整合开发环境。系统中的硬件及软件工具能够帮助客户使用 MDT MCU 芯片，快速且容易地开发应用程序。在 MDT-IDE 整合的开发环境中主要组件有 YS-Link，具有片内仿真功能的 MCU，它们提供了实时仿真功能和除错追踪功能。

1. 软件

MDT-IDE

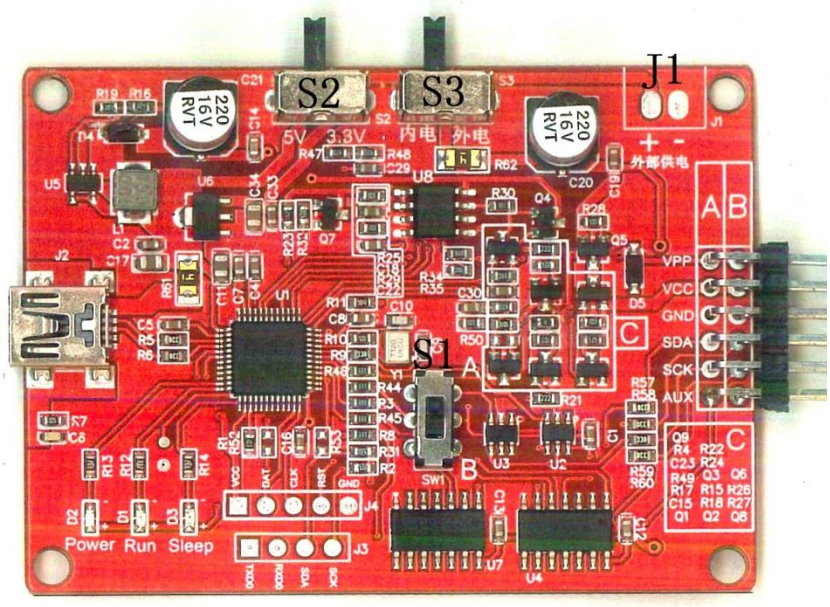
- 窗口架构的软件工具
- 原始程序层次的除错器
- 支持多个原始文件的工作平台
- 自带 C 编译器及汇编编译器

2. 硬件

YS-Link

- 安装使用容易，只需少数几根线与 MCU 相连
- 配合带片内调试功能的 MCU 实现断点、单步调试功能

下图为 YS-Link 接口图

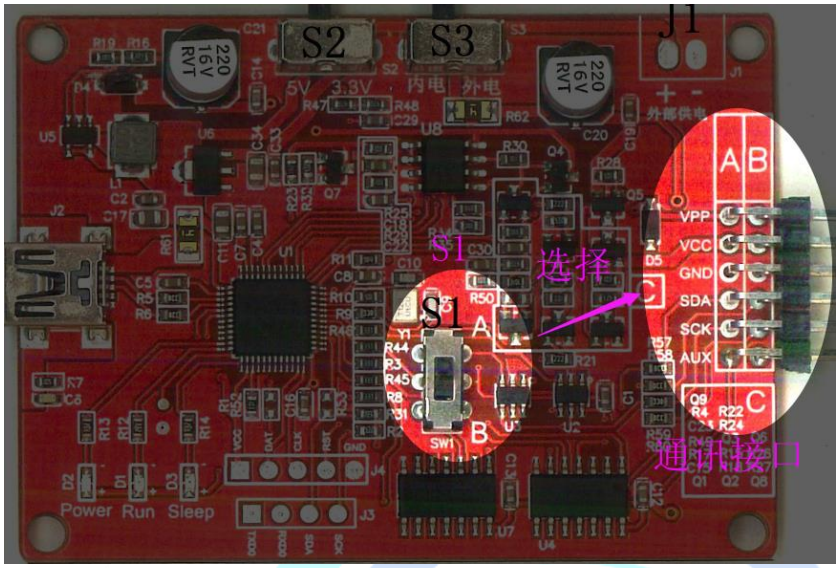


拔动开关功能使用说明：

S1:信号端口选择开关

当 S1 打到 A 端时，仿真信号（VPP,VCC,GND,SDA,SCK）经 A 边与目标 MCU 通讯；

当 S1 打到 B 端时，仿真信号（VPP,VCC,GND,SDA,SCK）经 B 边与目标 MCU 通讯；



注：

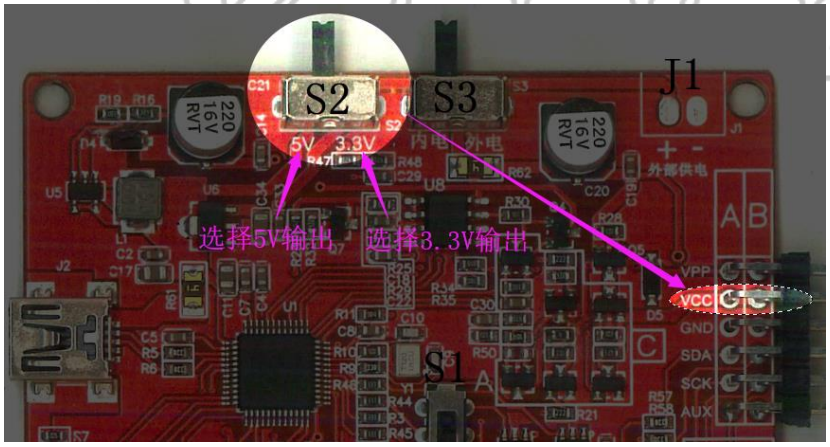
1、 选择 A 端时经过电平转换 IC 与目标 MCU 通讯，**绝大部分情况建议使用 A 端**；

2、 选择 B 端时经过缓冲器与目标 MCU 通讯，仅在 Datasheet 仿真信息中有特别指出的 MCU 选择使用；

S2:输出电压选择开关

S2 打到 5V 时，VCC 引脚对外输出 5V；

S2 打到 3.3V 时，VCC 引脚对外输出 3.3V；



注：

1、仅在 S3 选择仿真板（对外）供电时有效；

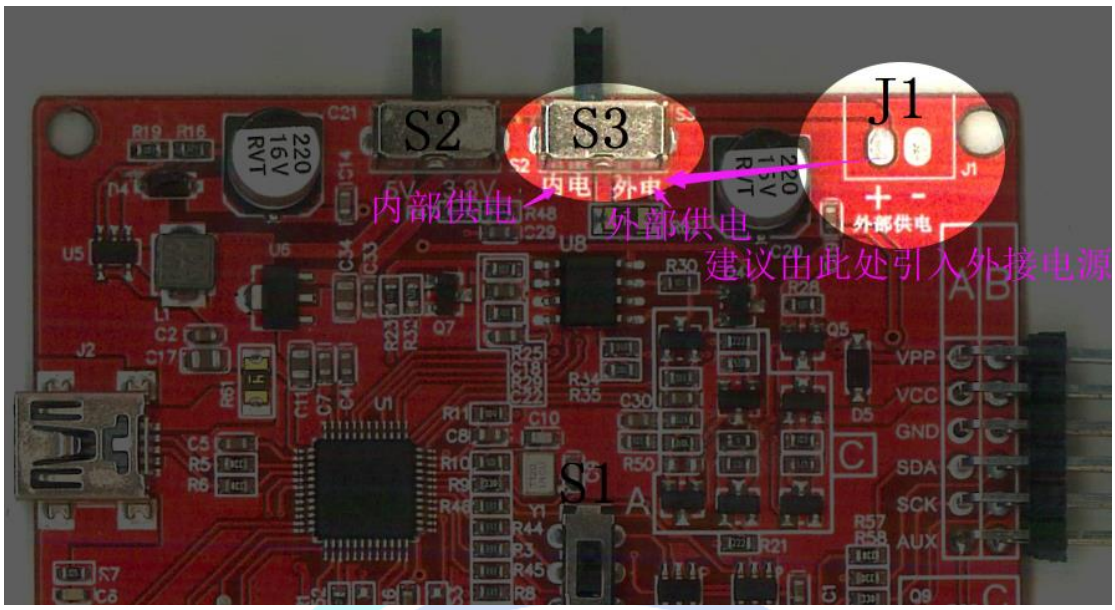
2、因 MTP 要求烧录电压为 4.5V-5.5V，所有 MTP 型 MCU 不建议使用 3.3V 电平烧录；

S3:目标板供电选择开关

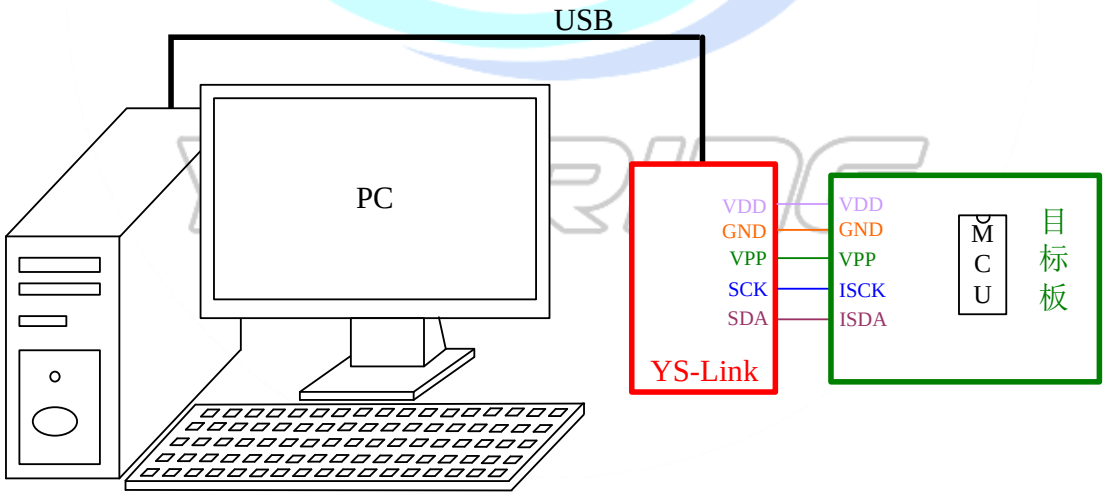
当 S3 打到“内电”时， VCC 电源由仿真板提供，输出电压由 S2 选择；

当 S3 打到“外电”时， VCC 电源需要外接供电，建议由 J1 外接电源，电压范围需符

合 MCU 的工作电压（见下图）；



3. 连接示意图



YS-Link 通过少量的接口与带片内仿真功能的 MCU 直接相联，可以达到仿真调试的目的，避免了仿真器与 MCU 差异带来的问题（比如仿真 OK，烧录 MCU 有问题）。

下图例为 MDT10F684 的接口图：

MDT10F684	YS-Link
VDD	VDD
GND	GND
PA3	VPP
PA0	SCK

PA1	SDA

注：此信息均可在 MCU Datasheet 的“开发支持”中找到

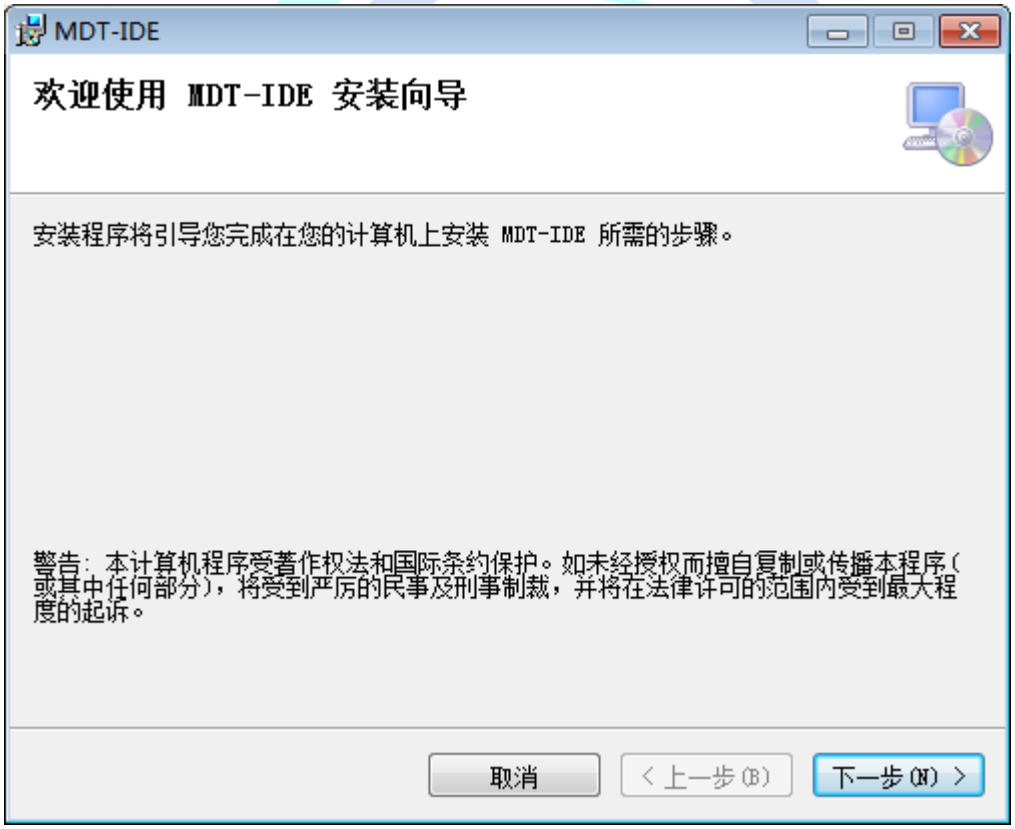
4. 系统配置

MDT-IDE 硬件需求非常低，能运行于目前大部分 PC 上，支持 Windows XP/7/8/10 操作系统。

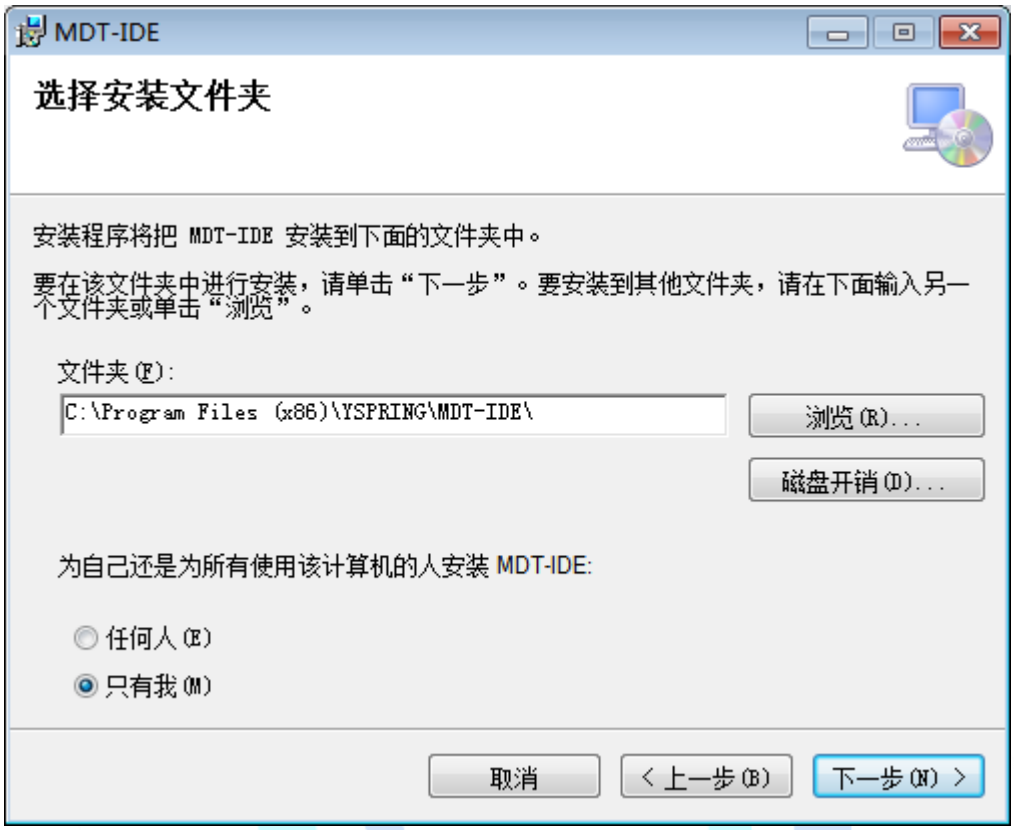
5. 软件安装

1、 双击安装图标 进行安装

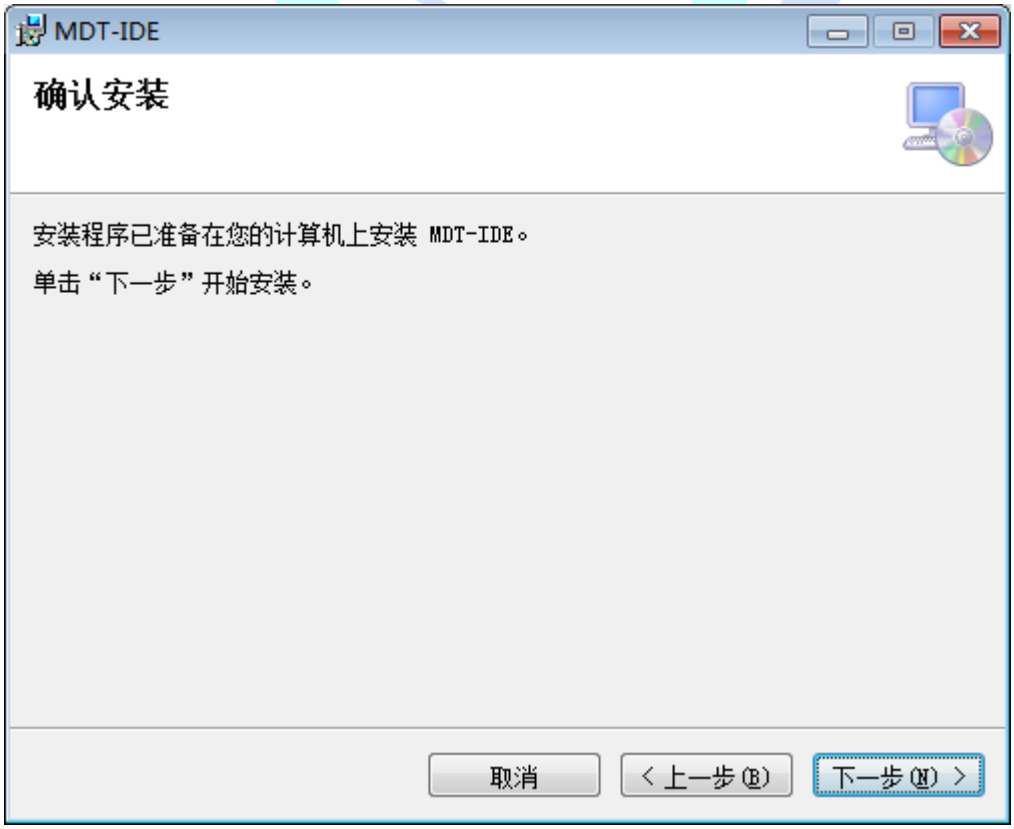
2、 显示如下安装向导界面，单击“下一步”



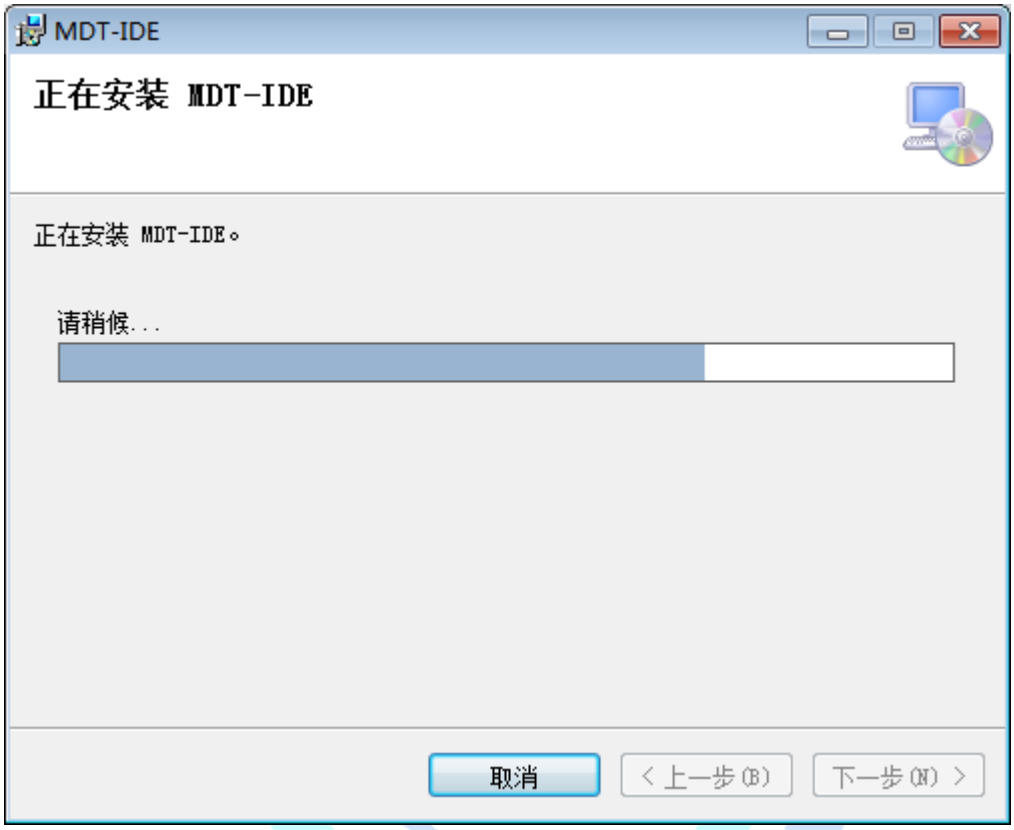
3、 如下图所示，单击“浏览”可以自定义设置安装地址，若用户不需要自定义设置安装文件夹，系统将默认如下图所示的安装地址。单击“下一步”，继续接下来的安装步骤。



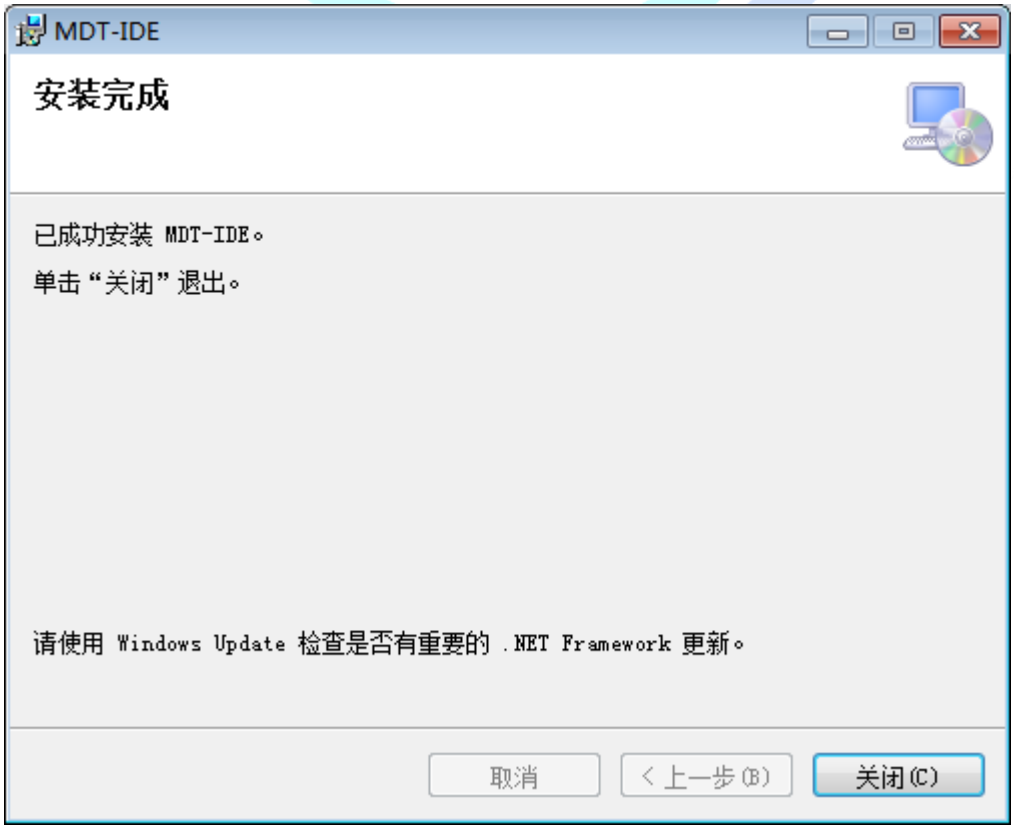
4、 出现如下界面，单击“下一步”开始软件安装



5、 软件开始安装



6、 软件安装完成，单击“关闭”，退出软件安装

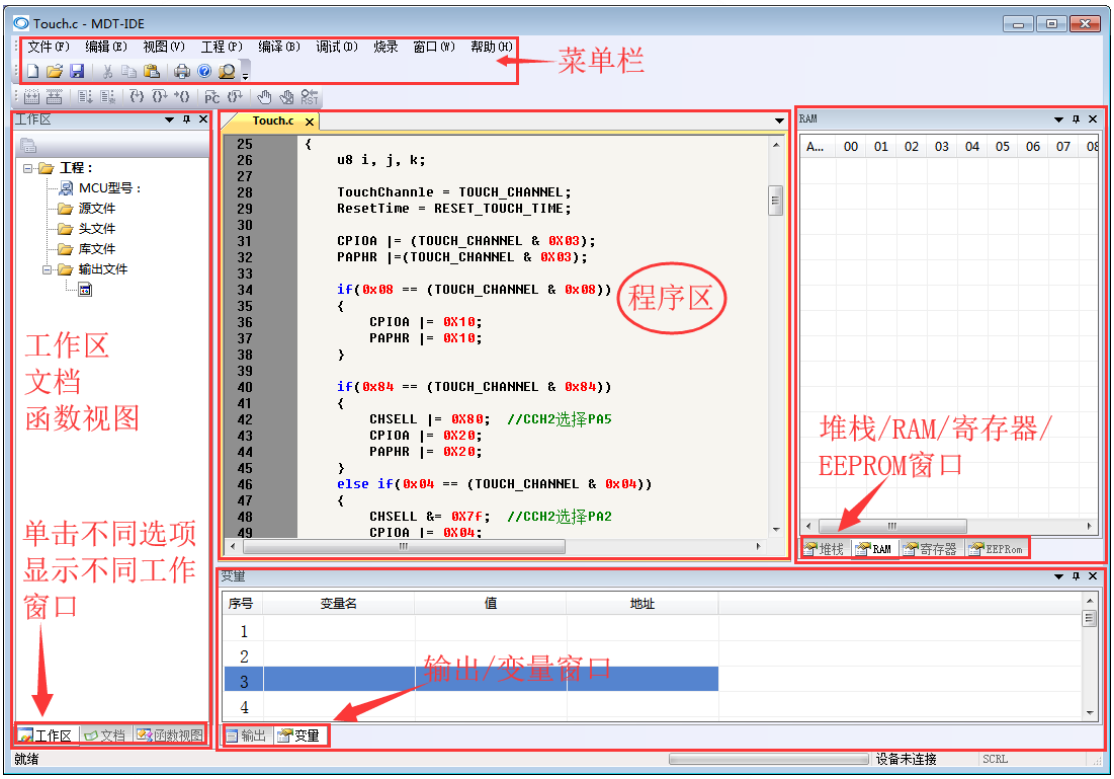


第二章　快速开始

本章简述如何快速使用 MDT-IDE 构建应用程序项目

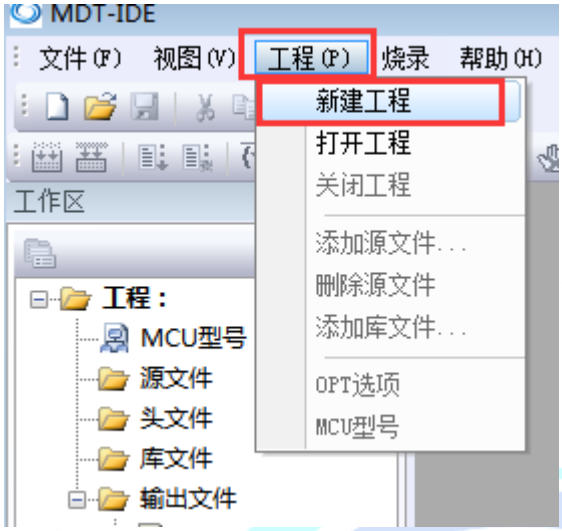
➢ 步骤一：打开 MDT-IDE 软件工具

- 双击图标 MDT-IDE 打开 MDT-IDE 软件，进入如下界面：

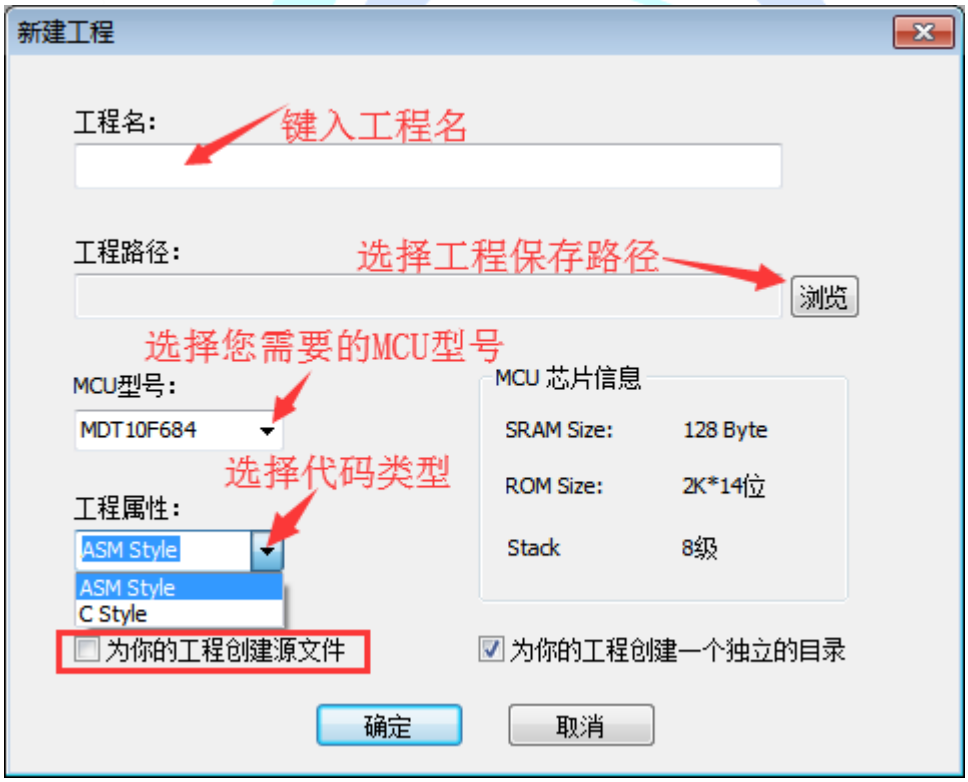


➢ 步骤二：新建工程

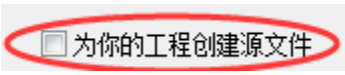
- 单击菜单“工程——新建工程”



- 在弹出的新建工程面板输入工程名、路径及进行简单设定，如下图：

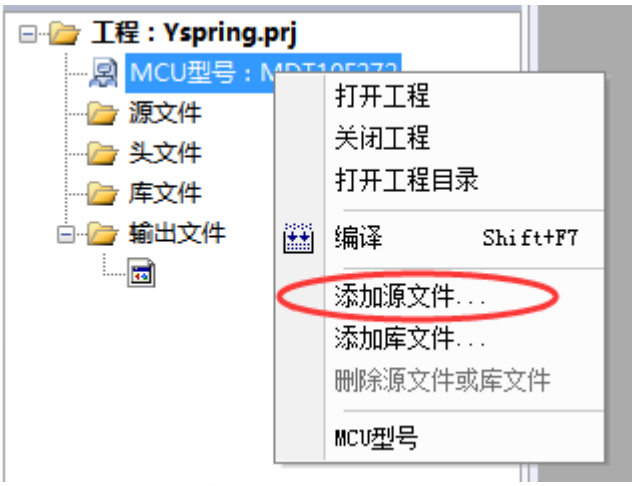


➢ 步骤三：添加文件

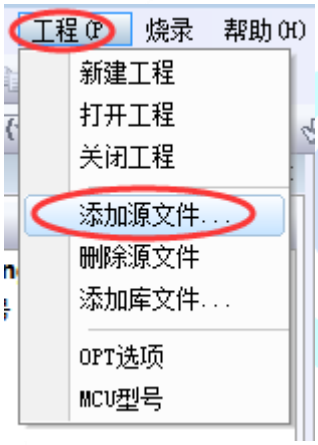
- 如果勾选了 

则无需再手动添加源文件；
- 如果没有勾选，则需要手动添加源文件，选择需要添加的主文件到工程，方法如下：

1、工作区窗口任意位置单击“右键——添加源文件”，如图：



2、在工程菜单添加源文件，如图：

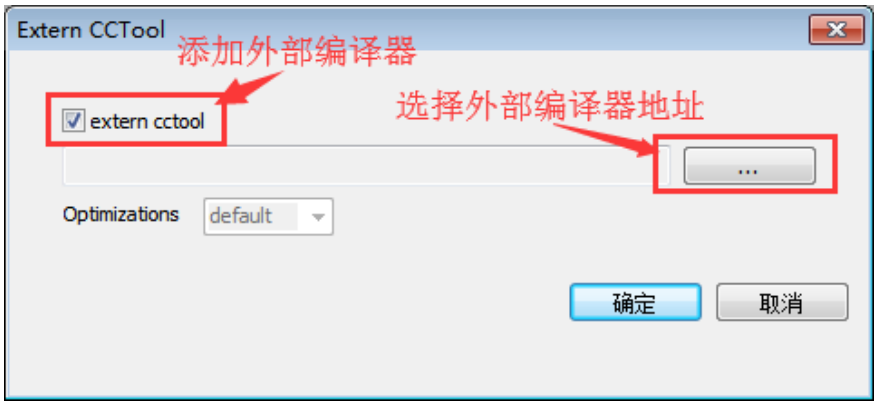


➢ 步骤四：添加外部编译器

当选择 C 语言编写程序时，建议使用第三方编译器，MDTIDE 可以很好支持第三方编译器，新建 C Type 工程时会自动弹出添加外部编译器窗口。

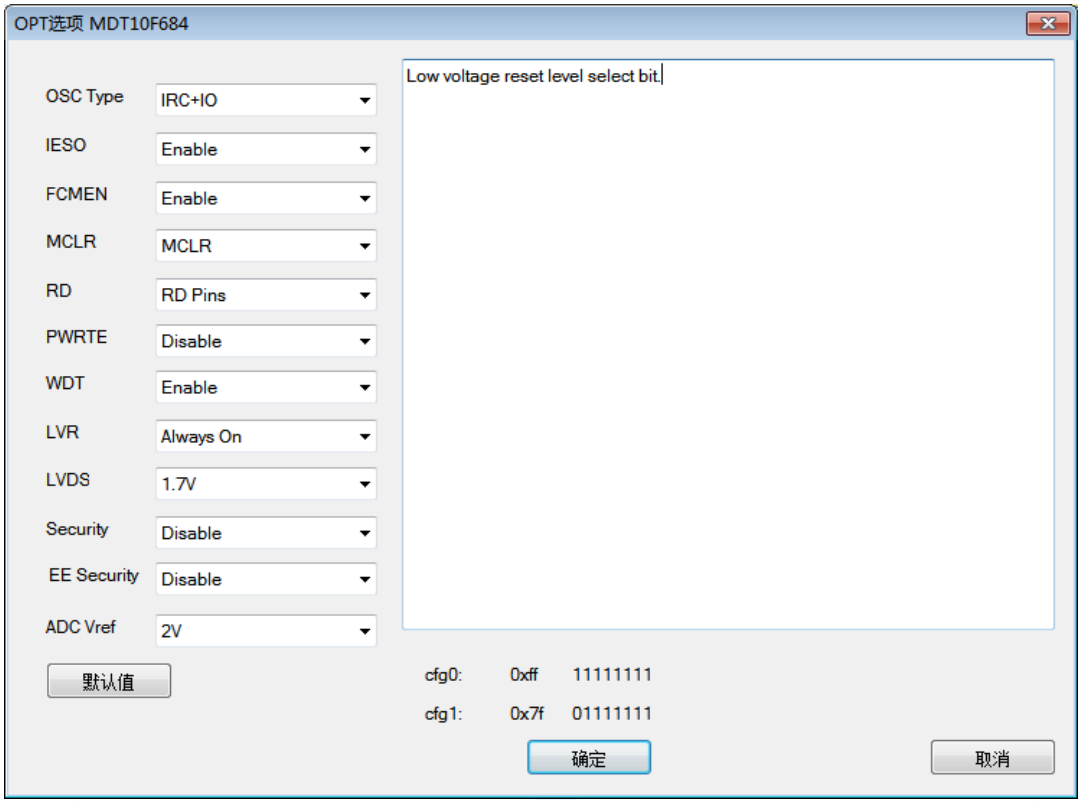
如下图界面所示，勾选“extern cctool”，单击 

找到外部编译器存放地址，添加外部编译器。



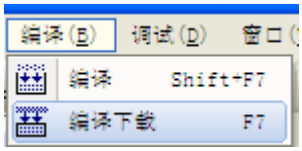
➢ 步骤五：设置 OPTION

- 不同型号的 MCU 有相应的配置项，需要开发仿真前设定好，下图例为 MDT10F684 配置项

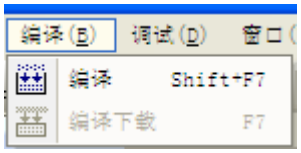


➢ 步骤六：编译下载

当 YS-Link 处于联机状态，可以使用“编译下载”菜单，把当前程序及 OPTION 下载到目标 MCU



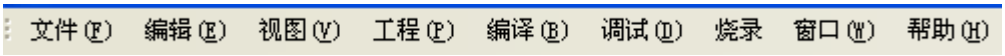
当 YS-Link 断开连接时“编译下载”菜单失效；



第三章　菜单说明

本章简单介绍 IDE 软件菜单功能，常见系统菜单不赘述。

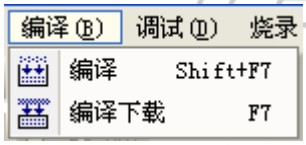
软件菜单视图如下：



1. 工程

本菜单主要对项目的操作，可以新建一个工程并添加删除源文件，并且可以随时通过本菜单修改 MCU 的配置项设置以及型号选择。

2. 编译



本菜单分为编译和编译下载，“编译”动作只会对文件进行编译，“编译下载”将完成编译下把代码下载到目标 MCU 的动作，仅当 YS-LINK 连接状态有效。

3. 调试

当程序成功下载到目标 MCU 后调试菜单功能将被激活，可以执行相应命令执行调试工作，在面板有相对应的按钮，如图：

下文第二部分将详细介绍仿真调试功能。

4. 烧录

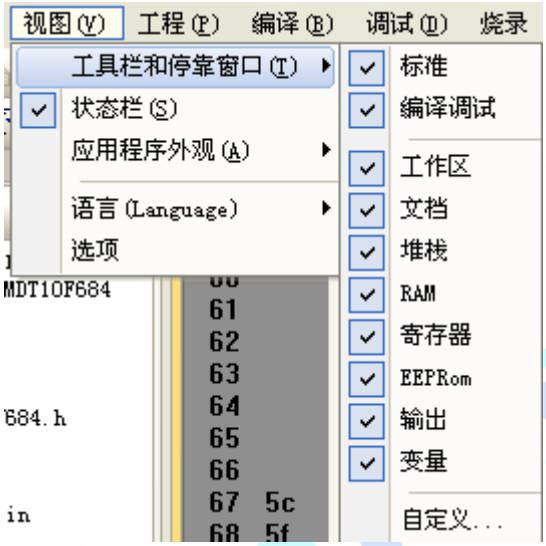
MDT-IDT 提供直接的编程功能，可以方便地使用 YS-Link 对 MCU 进行编程烧录，支持 PIC 的 HEX 档以及 MDT 的 BIN 文档的，点击烧录菜单后会弹出如下窗口：

在此面板下可以点击设置 OPTION 及查看校验信息。

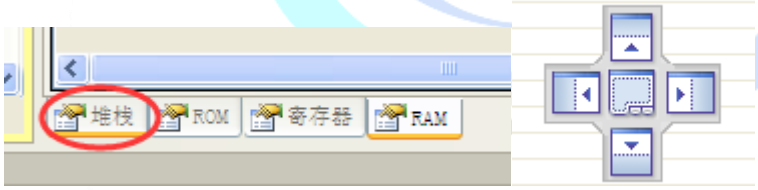
第四章　窗口视图

1. 开启窗口及调整布局

用户可以通过视图菜单选择需要用到的窗口，如下图：

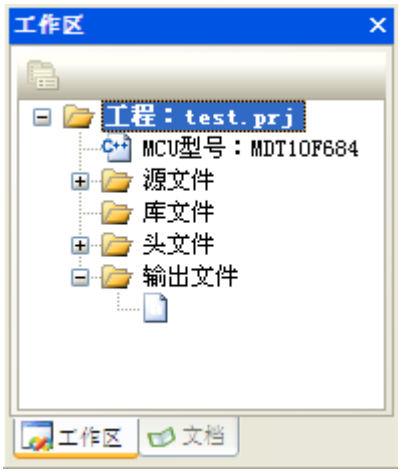


这些窗口都可以通过鼠标拖动标签改变布局：



2. 窗口介绍：

✓ 工作区

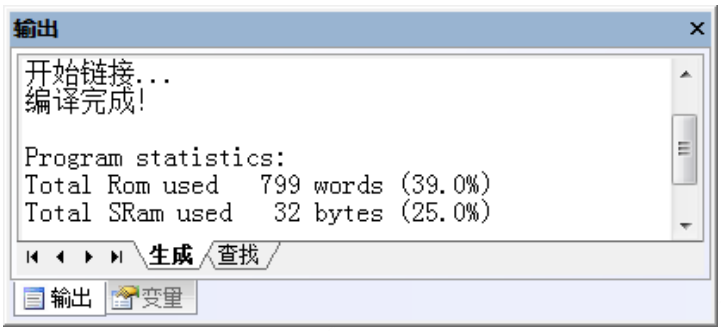


目录树形式展示当前仿真 MCU 型号及项目相关文档，并且可以在此处通过右键添加删除文件。

✓ **输出窗口**

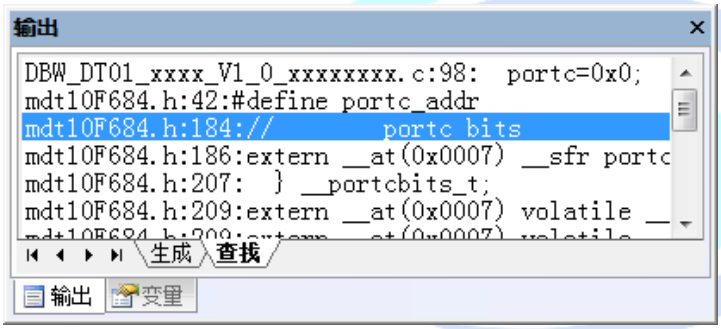
项目在编辑、编译、下载、调试的过程中会通过输出窗口输出相应的状态信息，下面简单介绍

1) 生成



编译及下载程序时将在此输出信息，**如果编译出错，可以通过双击错误信息定位**到程序出现错误的地方。

✓ **查找**



通过菜单“编辑——在文件中查找”，可以搜索文本内容，并在“输出——查找”窗口中列出查找到的内容，双击查找内容可快速定位。

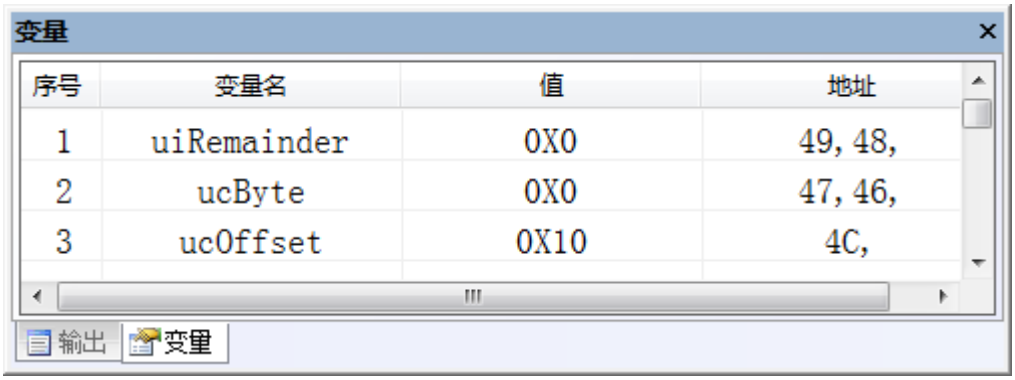
✓ **RAM 窗口**

RAM

ADR	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
@20	20	00	00	00	7B	7B	00	00	00	00	92	08	00	00	00	56
@30	88	00	8C	15	02	0B	07	28	47	88	A0	40	CC	B0	14	15
@40	20	0D	10	26	20	50	86	08	B1	B0	48	54	20	C1	56	03
@50	8B	65	05	40	69	35	20	09	2A	14	09	40	4A	81	0C	99
@60	46	04	AB	10	B5	13	91	53	B4	D5	92	02	46	A2	03	B4
@70	F0	8F	F0	14	F7	82	59	29	56	32	49	95	41	04	73	00
@A0	C8	73	C3	90	F6	F5	73	C3	90	32	00	05	31	0A	78	A6
@B0	BC	15	06	00	B2	80	B6	20	00	81	74	12	00	47	24	84

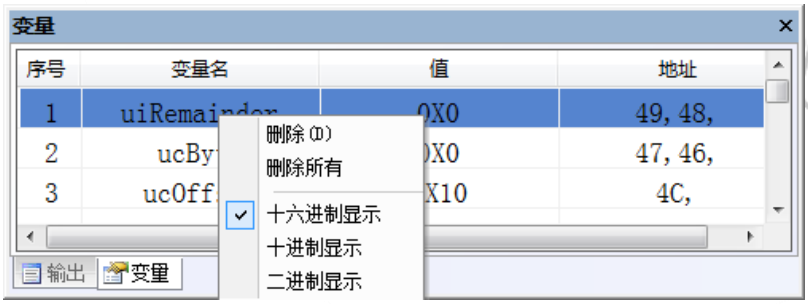
当调试状态，MCU 暂停时及单步执行时，RAM 窗口可以显示 MCU 整个用户 RAM 区数据，并且当**数值发生变化时会以红色字底显示。双击单元格可以编辑对应地址的数值。**

✓ **变量**

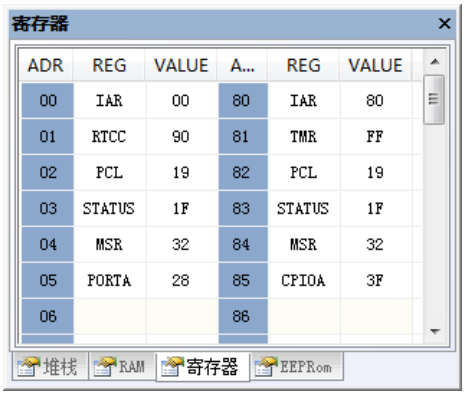


调试窗口可以察看变量（注：C 语言下面只能察看全局变量），通过双击“变量名”对应的单元格添加需察看的变量名，输入完成只要按 Enter 键或在其它位置单击鼠标即可结束输入并显示相应值（仅在调试暂停状态时显示）及相应内存地址。在调试暂停状态双击“值”对应单元格可以直接对变量进行编辑，并在完成编辑时将数据写入 MCU，以达到仿真时实时改变变量数值的目的。另外也**可以直接在编辑调试窗口选取需要察看的变量，右键菜单添加至变量窗口**，如图：

变量窗口点击右键可以删除相应行的变量察看或清除全部变量察看，也可以转换观察数值的数制，如图：

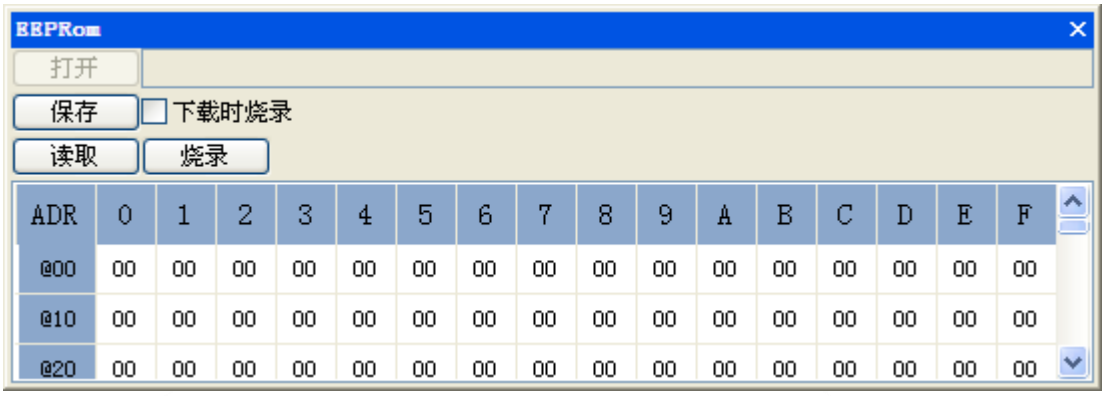


✓ **寄存器**



显示 MCU 特殊功能寄存器地址及数值，显示已对应 Datasheet 中寄存器 MAP。在调试状态下当数值发生变化时会在相应的单元格红底显示。双击单元格可以编辑对应寄存器的数值。

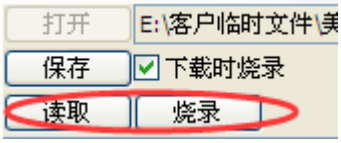
✓ **EEPROM**



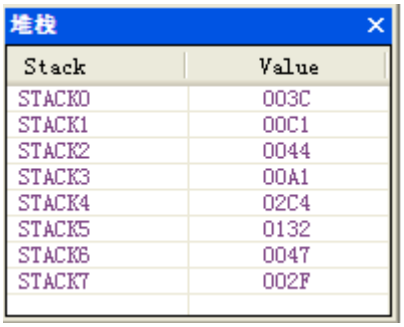
- 如下图，可打开事先保存的以“.epr”为后缀名的 EEPROM 数据文件，同样点保存可以把当前数据保存为以“.epr”为后缀名的 EEPROM 数据文件

- 当勾选“下载时烧录”，编译生成的烧录 BIN 文档将包含 EEPROM 数据，如果下载调试程序时也将 EEPROM 数据写入目标 MCU。

- 在调试模式的暂停状态下可以读取和写入目标 MCU 的 EEPROM 数据，方便调试使用。



✓ **堆栈**



显示 MCU 硬堆栈内容，不同的 MCU 型号可能有不同的堆栈深度。

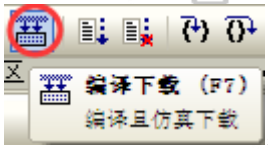
第二部分　仿真调试

本部分介绍调试工具的使用方法

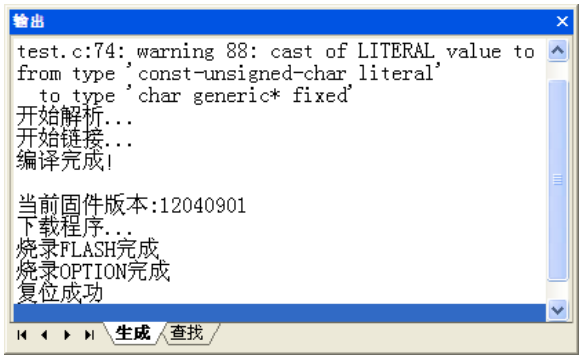
1. 调试菜单及面板

如上图，调试菜单包含了所有的调试指令，并且在后面标明了快捷键。

2. 进入调试状态

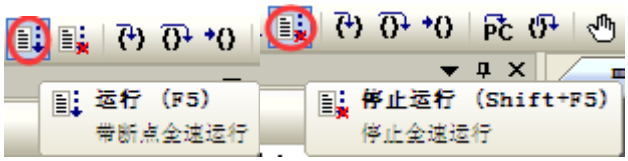


，在连接状态下通过编译下载按钮可以一键完成编译和下载程序的动作，下载过程软件下方状态栏将提示下载进度，下载完成后复位 MCU，进入调试状态，如图：



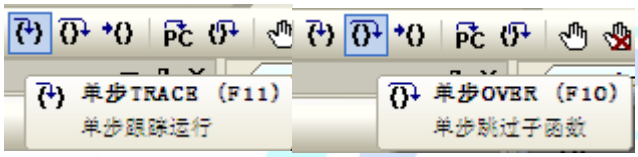
此时调试工具面板为激活状态。

3. 全速运行及停止



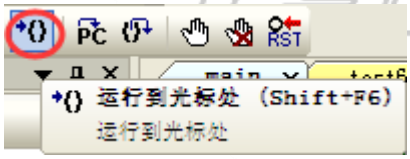
上图两按钮为全事运行及停止按钮，全速执行时遇到断点或停止命令时将停止运行，状态栏显示为“STOP”，如图：

4. 单步执行



单步 TRACB：单步执行，当遇到函数（子程序）时将进入函数（子程序）；
单步 OVER：单步执行，当遇到函数（子程序）时将子函数当作一条指令执行完成并停在下条指令；
注，如果 MCU 只支持一个断点（如：MDT10F684），且已经设置断点时，执行“单步 OVER”时遇到断点将不会暂停，需特别注意。

5. 执行到光标



此功能相当于设置了一个临时断点，只需在需要停止的地方的当前行任意位置单击鼠标，获取光标焦点后全速运行，当程序执行经过光标所在位置时自动停止。

```
    //============================
    //============================
    void main(void)
    {
2       device_init();
    //    while(1);
3       flagbit = 0x01;
6       CPIOA |= 0x01;
8       ADINS |= 0x0;
9       ADRESL = 0;
a       ADRESH = 0;
        ADS1   = 0x20;
```

```
19        //=========================
20        void main(void)
21        {
22    2       device_init();
23        //      while(1);
24    3       flagbit = 0x01;
25    6       CPIOA |= 0x01;
26    8       ADINS |= 0x01;
27    9       ADRESL = 0;
```

注，如果 MCU 只支持一个断点（如：MDT10F684），且已经设置断点时，执行“执行到光标”时遇到断点将不会暂停，需特别注意。

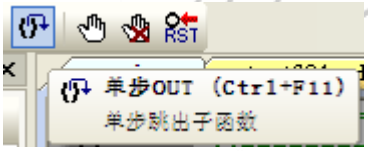
6. 单步 PC

此功能只在 C 环境下有效，通常一条 C 语句可能会对应多条汇编指令，为了方便观察 MCU 执行情况，可以使用单步 PC（PC 为程序指针）方式执行，当遇到多条汇编表达的 C 语句时将切换至汇编窗口，如图：

```
59    f6        CMSTA     = 0x07;

659                                   ;    .line   59; "main.c"    CMSTA     = 0x07;
660   0000f6  3a07    ldwi    0x07      LDWI    0x07
661   0000f7  1199    stwr    0x19      STWR    _CMSTA
```

7. 跳出函数



此功能只在 C 环境有效，当 PC 暂停在一子函数中时，可以通过执行此命令跳出当前子函数，如图：

```
53        void device_init(void)
54        {
55    f2          GIE = 0;
56
57    f4          PORTA     = 0x00;
58    f5          PORTC     = 0x00;
59    f6          CMSTA     = 0x07;
60
61    f8          CPIOA     = 0x19;
62    fb          CPIOC     = 0x00;
63    fc          PAPHR     = 0x10;
64    fe          OSCCON    = 0x61;
65   100  }
```

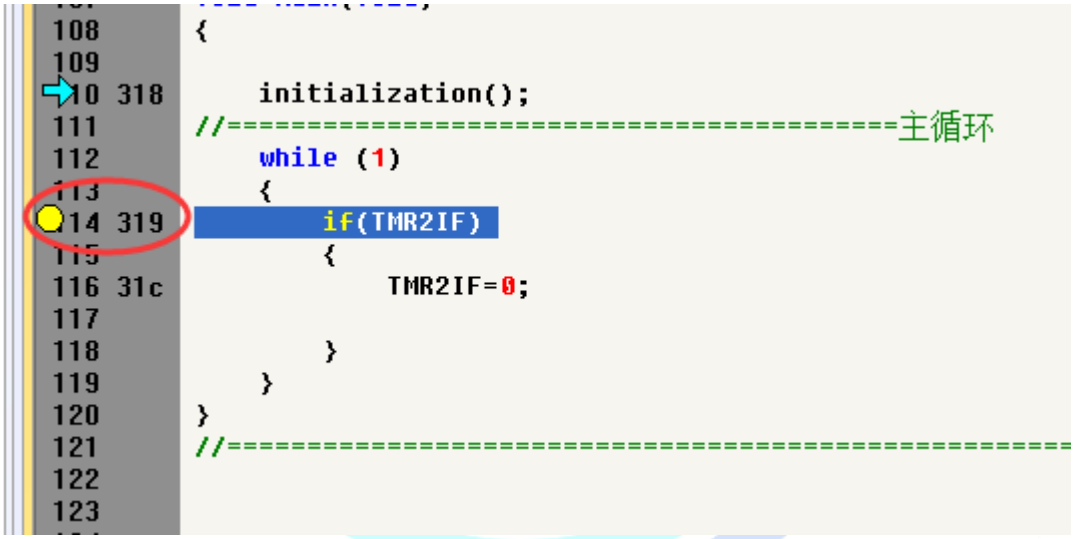
```
19        //=========================
20        void main(void)
21        {
22    2       device_init();
23        //      while(1);
24    3       flagbit = 0x01;
25    6       CPIOA |= 0x01;
26    8       ADINS |= 0x01;
27    9       ADRESL = 0;
28    a       ADRESH = 0;
29    c       ADS1   = 0x20;
30    f       ADS0   = 0x81;
31            while(1)
```

注：如果 MCU 只支持一个断点（如：MDT10F684），且已经设置断点时，执行“跳出函数”时遇到断点将不会暂停，需特别注意。

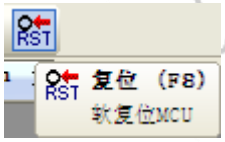
8. 断点

光标定位到需要设置断点的地方，点击添加“断点”可以添加或删除断点，点击“删除断点”可删除所有断点（当有多个断点时）。

另外**可以通过双击调试窗口的侧边栏进行添加删除断点操作（必须是有效语句）**，如图：

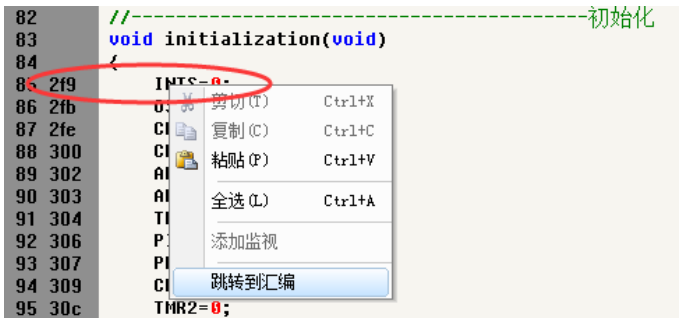


9. 复位



在任何状态单击复位按钮，MCU 将发生复位并且指针停在复位向量。

10. 跳转到汇编



在 C 环境下，可以在编辑调试窗口有效语句位置点右键跳转至相应汇编程序；

第三部分　汇编编译器

MDT-IDE 内置汇编编译器为汇春公司自主开发，支持 MDT 37 条汇编指令编译，支持丰富的伪指令。本章节详细介绍 MDT 汇编指令、伪指令及其用法。

1. 汇编指令一览表：

温馨提示：单击表中指令可快速定位到相应指令说明。

分类	指令	功能	操作	影响状态位
特殊操作指令	NOP	空操作	无	无
	CLRWT	清看门狗	0→WT	/TF, /PF
	SLEEP	进入睡眠模式	0→WT, stop OSC	/TF, /PF
	TMODE	将 W 寄存器值装入选项寄存器（81H）	W→81H	无
	CPIO R	设置端口方向寄存器（1 输入，0 输出）	W→CPIO r	无
数据移动指令	STWR R	将 W 寄存器内容送 R 寄存器	W→R	无
	LDR R,t	读寄存器 R,结果保存在 R(t=1)或者 W(t=0)	R→t	Z
	LDWI I	立即数送 W 寄存器	I→W	无
	SWAPR R,t	交换寄存器 R 的高低四位，结果保存在 R(t=1)或者 W(t=0)中	[R(0~3) R(4~7)]→t	无
加减（及判断）指令	INCR R,t	递增寄存器 R,结果保存在 R(t=1)或者 W(t=0)中	R + 1→t	Z
	INCRSZ R, t	递增寄存器 R,结果保存在 R(t=1)或者 W(t=0)中；如果结果等于 0 则跳过该指令接下来的指令	R + 1→t	无
	ADDWR R,t	W 寄存器与 R 寄存器相加，结果保存在 R(t=1)或者 W(t=0)中	W + R→t	C, HC, Z
	ADDWI I	W 寄存器与立即数 I 相加，结果保存到 W 中	PC+1→PC,W+I→W	C,HC,Z
	SUBWR R,t	R 寄存器减去 W 寄存器，结果保存在 R(t=1)或者 W(t=0)中	R A W→t (R+/W+1→t)	C, HC, Z
	SUBWI I	立即数 I 减去 W 寄存器，结果保存到 W 寄存器中	I-W→W	C,HC,Z
	DECR R,t	递减寄存器 R,结果保存在 R(t=1)或者 W(t=0)中	R A 1→t	Z
	DECRSZ R,t	递减寄存器 R, 结果保存在 R(t=1)或者 W(t=0)中；如果结果等于 0 则跳过该指令接下来的指令	R A 1→t	无

逻辑操作指令	ANDWR R,t	R 寄存器与 W 寄存器做“与”操作，结果保存在 R(t=1)或者 W(t=0)中	R ∩ W→t	Z
	ANDWI I	W 寄存器与立即数 I 做“与”操作，结果保存到 W 寄存器中	I ∩ W→W	Z
	IORWR R,t	R 寄存器与 W 寄存器做“或”操作，结果保存在 R(t=1)或者 W(t=0)中	R ∪ W→t	Z
	IORWI I	W 寄存器与立即数 I 做“或”操作，结果保存在 R(t=1)或者 W(t=0)中	I ∪ W→W	Z
	XORWR R,t	R 寄存器与 W 寄存器做“异或”操作，结果保存在 R(t=1)或者 W(t=0)中	R ⊕ W→t	Z
	XORWI I	W 寄存器与立即数 I 做“异或”操作，结果保存在 R(t=1)或者 W(t=0)中	I ⊕ W→W	Z
	COMR R,t	R 寄存器“取反”操作，结果保存在 R(t=1)或者 W(t=0)中	/R→t	Z
移位操作指令	RRR R,t	R 寄存器循环“右移”操作，结果保存在 R(t=1)或者 W(t=0)中	R(n)→R(n-1), C→R(7), R(0)→C	C
	RLR R,t	R 寄存器循环“左移”操作，结果保存在 R(t=1)或者 W(t=0)中	R(n)→r(n+1), C→R(0), R(7)→C	C
清除指令	CLRW	W 寄存器清 0	0→W	Z
	CLRR R	R 寄存器清 0	0→R	Z
位操作判断指令	BCR R,b	R 寄存器的第 b 位清 0	0→R(b)	无
	BSR R,b	R 寄存器的第 b 位置 1	1→R(b)	无
	BTSC R,b	如果 R 寄存器的第 b 位为 0，则跳过该指令接下来的指令	Skip if R(b)=0	无
	BTSS R,b	如果 R 寄存器的第 b 位为 1，则跳过该指令接下来的指令	Skip if R(b)=1	无
调用跳转及返回指令	LCALL N	在整个 2K 区域内的调用指令	N→PC, PC+1→Stack	无
	LJUMP N	在整个 2K 区域内的跳转指令	N→PC	无
	RTIW I	带立即数从子程序返回	Stack→PC, I→W	无
	RTFI	中断返回	Stack→PC,1→GIS	无
	RET	从子程序返回	Stack→PC	无

注：

W	：工作寄存器（累加器）	R	：寄存器	I	：立即数
WT	：看门狗定时器	B	：位	x	：没影响
TMODE	：TMODE 寄存器	t	：目标	N	：地址
CPIO	：IO 方向寄存器		=0：工作寄存器		
TF	：TIMER 溢出标志		=1：通用寄存器		
PF	：掉电标志	C	：进位标志位		
PC	：程序指针	HC	：辅助进位标志		
OSC	：系统时钟	Z	：0 标志		

2. 指令说明：

1) 特殊操作指令：

NOP (No Operation) 空操作指令
格式： NOP
操作数： 无
操作过程： 消耗单片机一个指令周期，无其它任何影响
影响状态位： 无
执行时间： 1 个指令周期
说明： 该指令一般用于程序运行进程中的延时 1 个指令周期
例： BCR PORTA,1 ；PORTA 口的第一位清零
NOP ；延时，使低电平稳定

CLRWT (Clear Watchdog timer) 看门狗清零指令
格式： CLRWT
操作数： 无
操作过程： 0 →WT
影响状态位： TF、PF
执行时间： 1 个指令周期
说明： 将看门狗清 0，使其不能计时溢出
例： CLRWT ；看门狗计数器清零

SLEEP (Sleep mode) 低功耗睡眠状态指令
格式： SLEEP
操作数： 无
操作过程： 0 →WT，Stop osc
影响状态位： TF、PF
执行时间： 1 个指令周期
说明： 芯片停止振荡，MDT 单片机进入低功耗模式，SLEEP 会清零 WDT 和预分频（如果预分频分配给 WDT 的话），并将 F3 的 PF 位清零，TF 位置 1。进入低功耗模式后，I/O 状态保持不变，WDT 清零后重新计时，一旦计时溢出即将 MDT 从 SLEEP 模式中唤醒（通过 RESET）

TMODE (Load W to TMODE register) 写 TMODE 寄存器指令
格式： TMODE
操作数： 6 位的立即数
操作过程： W →TMODE
影响状态位： 无
执行时间： 1 个指令周期
说明： 将 W 内容写入 TMODE 寄存器中
例： LDWI 27H ；W=27H
TMODE ；TMODE=W=27H
注：为了更好的兼容性，尽量不用此指令，请通过切 BANK 直接访问 TMODE 寄存器

CPIO (Control I/O port register) 写 IO 方向控制指令寄存器。

格式： CPIO R

操作数： 8 位的立即数

操作过程： W →CPIO

影响状态位： 无

执行时间： 1 个指令周期

说明： 将 W 内容写入 CPIO 寄存器中

例：
```
LDWI    54H                 ；W=54H
CPIO    PORTA               ；PORTA  CPIO=W=54H
```
注：为了更好的兼容性，尽量不用此指令，请通过切 BANK 直接访问方向寄存器

2) 数据移动指令：

STWR (Store W to register) 将 W 寄存器的内容传送至数据寄存器

格式： STWR R

操作数： R 为数据寄存器的低 7 位地址

操作过程： W→R

影响状态位： 无

执行时间： 1 个指令周期

说明： 将 W 中的内容传送到 R 寄存器中，W 中的内容保持不变

例：
```
LDWI    0dH             ；数据 0dH 送入 W 中
STWR    R               ；W→R
```

LDR (Load register) 将数据寄存器的内容传送至目标寄存器

格式： LDR R,t

操作数： R 为数据寄存器的低 7 位地址，t 为“0”或“1”

操作过程： R→t

t 为“0”时，送人 W 中；t 为“1”时，送入 R 中

影响状态位： Z

执行时间： 1 个指令周期

说明： 此指令可以通过 W 把数据寄存器的内容传送至它处，或对数据寄存“0”操作，如果传送的数据为“0”，则置位 Z 标志

例：
```
LDR     R1,0    ；(R1)→W
STWR    R2      ；W→(R2)
```

LDWI (Load immediate to W) 常数传送 W 指令

格式： LDWI i

操作数： i 为 8 位的立即数

操作过程： i→W

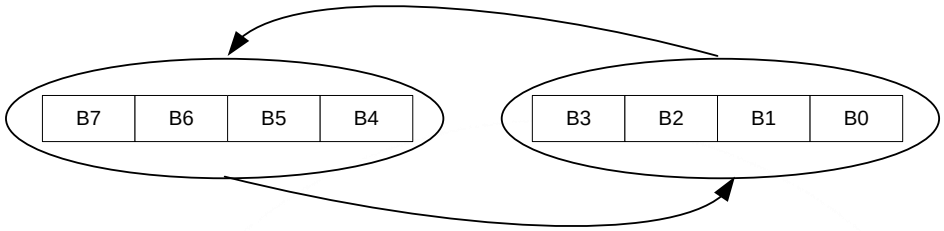
影响状态位： 无

执行时间： 1 个指令周期

说明： 将常数 i 送入 W 中，它不影响任何状态位

例：
```
LDWI    0B2H                ；W=0B2H
```
SWAPR (Swap halves register) 空操作指令

格式： SWAPR R，t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R(0→3) ←→R（4-7） →t
影响状态位： 无
执行时间： 1 个指令周期
说明： 将 R 寄存器内容的高 4 位（bit7~bit4）与低 4 位（bit3~bit0）相互交换，结果存放在 W（t=0）或 R（t=1）中，其交换过程如图 1-1 所示



例：
```
LDWI    0F3H    ；PORTA 口的第一位清零
STWR    09H     ；延时，使低电平稳定
SWAPR   09H,1   ；内容交换存入 R9 中，R9=3FH
```

3) 加减（及判断）指令：

INCR (Increment register) 寄存器加 1 指令
格式： INCR R,t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R+1→t
影响状态位： Z
执行时间： 1 个指令周期
说明： R 寄存器的内容加 1，存放在 W（t=0）或 R（t=1）中
例：
```
INCR    09H,1   ；R9+1→R9 中
```

INCRSZ (Increment register，skip if zero) 寄存器加 1，结果为零则跳转指令
格式： INCRSZ R,t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R+1→t
影响状态位： Z
执行时间： 1 个指令周期（不跳转时）或 2 个指令周期（跳转时）
说明： R 寄存器的内容加 1，存放在 W（t=0）或（t=1）中，同时它会根据 Z 标志位进行程序的分支跳转控制，如结果不为 0，Z≠0，程序按顺序执行，此时为 1 个指令周期，如结果为 0，Z=0，程序跳过下一条指令，继续按顺序执行，此时为 2 个指令周期。
例：
```
LOOP:
        INCRSZ  9,1     ；R9+1=R9
```

```
                JUMP        LOOP        ；判断加 1 后的结果，为 0
                                        ；则跳过此条指令，执行下
                                        ；一条空操作，不为 0，则
                                        ；跳到 LOOP 执行
```

ADDWR (Add W and register) 寄存器加法指令

格式： ADDWR R，t

操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0

操作过程： W+R→t

影响状态位： C、HC、Z

执行时间： 1 个指令周期

说明： R 寄存器的内容与 W 寄存器的内容相加，结果存放在 W（t=0）或 R（t=1）中

例：

```
ADDWR   9,0     ；R9+W→W
```

ADDWI (Add W and immediate) 立即数加法指令

格式： ADDWI i

操作数： i 为 8 位立即数

操作过程： W+i→W

影响状态位： C、HC、Z

执行时间： 1 个指令周期

说明： 立即数 i 与 W 寄存器的内容相加，结果存放在 W 中

例：

```
ADDWI   10      ；10+W→W
```

SUBWR (Subtract W from register) 寄存器减法指令

格式： SUBWR R,t

操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0

操作过程： R-W→t (R+/W+!→t)

影响状态位： C、HC、Z

执行时间： 1 个指令周期

说明： R 寄存器的内容减去 W 寄存器的内容，结果存放在 W（t=0）或 R（t=1）中

例：

```
CLRR    0FH     ；R0F=0
LDWI    1       ；W=01H
SUBWR   0FH,1   ；R0F-W=R0F 中，因有借位，故 C=0
```

SUBWI (Subtract W from immediate) 立即数减法指令

格式： SUBWI i

操作数： i 为 8 位立即数

操作过程： i-W→W

影响状态位： C、HC、Z

执行时间： 1 个指令周期

说明： 立即数 i 减去 W 寄存器的内容，结果存放在 W 中

例：

```
LDWI    10
SUBWI   10      ；10+W→W，W=0，故 Z=1，C=1
```

DECR (Decrement register) 寄存器减 1 指令

格式： DECR R,t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R-1→t
影响状态位： Z
执行时间： 1 个指令周期
说明： R 寄存器的内容减去 1，结果存放在 W（t=0）或（t=1）中
例： DECR 0DH,0 ；R0D-1→W

DECRSZ (Decrement register，skip if zero) 寄存器减 1，结果为零则跳转指令
格式： DECRSZ R,t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R11→t
影响状态位： Z
执行时间： 1 个指令周期（不跳转时）或 2 个指令周期（跳转时）
说明： R 寄存器的内容减 1，结果存放在 W（t=0）或 R（t=1）中，同时它会根据 Z 标志位进行程序的分支跳转控制，如结果不为 0，Z≠0，程序按顺序执行，此时为 1 个指令周期，如结果为 0，Z=0，程序跳过下一条指令，继续按顺序执行，此时为 2 个指令周期。
例： DECRSZ 09H,1 ；R9-1=R9,结果若为 0，则跳过 LDWI 指
；令，执行 LDR 指令，若不为 0，则顺
；序执行，即执行 LDWI 指令
LDWI 37H
LDR 11H

4) 逻辑操作指令：

ANDWR (AND W and register) 寄存器与指令
格式： ANDWR R,t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R∩W→t
影响状态位： Z
执行时间： 1 个指令周期
说明：将 R 寄存器的内容与 W 内容做逻辑与运算，结果存放在 W（t=0）或 R（t=1）中
例： ANDWR 09H,0 ；R9∩W→W

ANDWI (AND W and immediate) 常数与指令
格式： ANDWI i
操作数： i 为 8 位的立即数
操作过程： i∩W →W
影响状态位： Z
执行时间： 1 个指令周期
说明： 将常数 i 与 W 内容做与运算，结果存放在 W 中
例： LDWI 53H ；W=53H
ANDWI 74H ；W=53H∩74H=53H

IORWR (Inclu • OR W and register) 寄存器或指令

格式： IORWR R，t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R∪W→t
影响状态位： Z
执行时间： 1 个指令周期
说明：将 R 寄存器的内容与 W 内容做逻辑或运算，结果存放在 W（t=0）或 R（t=1）中
例： IORWR 09H,1 ；R9∪W→R9

IORWI (Inclu • OR W and immediate) 常数或指令
格式： IORWI i
操作数： i 为 8 位的立即数
操作过程： i∪W →W
影响状态位： Z
执行时间： 1 个指令周期
说明： 将常数 i 与 W 内容做或运算，结果存放在 W 中
例： LDWI 53H ；W=53H
ANDWI 74H ；W=53H∪74H=77H

XORWR (Exclu • OR W and immediate) 寄存器异或指令
格式： XORWR R，t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： R⊕W→t
影响状态位： Z
执行时间： 1 个指令周期
说明：将 R 寄存器的内容与 W 内容做逻辑异或运算，结果存放在 W（t=0）或 R（t=1）中
例： XORWR 09H，1 ；R9⊕W→R9

XORWI (Exclu • OR W and immediate) 常数异或指令
格式： XORWI i
操作数： i 为 8 位的立即数
操作过程： i⊕W →W
影响状态位： Z
执行时间： 1 个指令周期
说明： 将常数 i 与 W 内容做异或运算，结果存放在 W 中
例： LDWI 53H ；W=53H
ANDWI 74H ；W=53H⊕74H=27H

COMR (Complement register) 寄存器取反指令
格式： COMR R，t
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0
操作过程： /R→t
影响状态位： Z
执行时间： 1 个指令周期
说明： 将 R 寄存器的内容做逻辑与运算，结果存放在 W（t=0）或 R（t=1）中

例： COMR 09H,0 ；R9 取反→W

5) 移位操作指令：

RRR (Rotate right register) 带进位的右移位指令

格式： RRR R,t

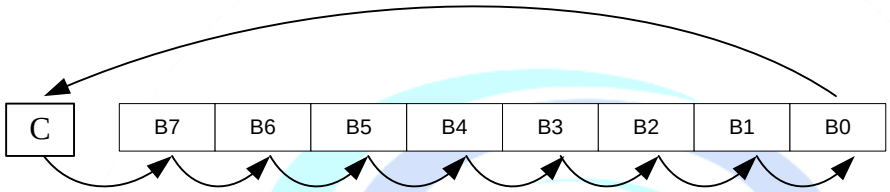
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0

操作过程： R（n）→R（n-1），R（0）→C，C→R(7)

影响状态位： C

执行时间： 1 个指令周期

说明：将 R 寄存器的内容和 C 一起做右移位操作，结果存放在 W（t=0）或 R（t=1）中，其移位过程如图 1-2 所示：



例：
```
LDWI    23H     ；W=B‘00100011’
STWR    09H     ；R9=W
BSR     03H,0   ；状态 C 清零
RRR     09H,1   ；R9 右移，R9=B‘00010001’，C=1
```

RLR (Rotate left register) 带进位的右移位指令

格式： RLR R，t

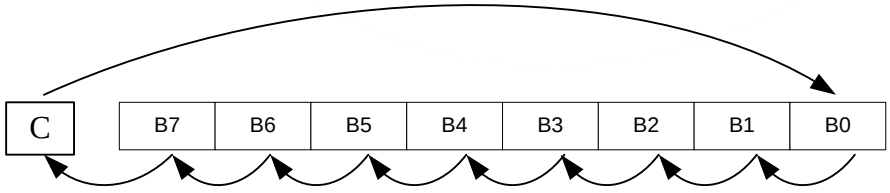
操作数： R 为数据寄存器的低 7 位地址，t 为 1 或 0

操作过程： R（n）→R（n+1），R（7）→C，C→R(0)

影响状态位： C

执行时间： 1 个指令周期

说明： 将 R 寄存器的内容和 C 一起做左移位操作，结果存放在 W(t=0)或 R(t=1)中，其移位过程如图 1-3 所示：



例：
```
LDWI    4AH     ；W=B‘01001010’
STWR    09H     ；R9=W
BSR     03H,0   ；状态 C 清零
RRR     09H,1   ；R9 左移，R9=B‘10010100’，C=1
```

6) 清除指令：

CLRW (Clear working register) 工作寄存器清零指令
格式： CLRW
操作数： R 为数据寄存器的低 7 位地址
操作过程： 0→W，1→Z
影响状态位： Z
执行时间： 1 个指令周期
说明： 将 W 寄存器清零，状态位 Z 置 1
例： CLRW ；W=0，Z=1

CLRR (Clear register) 通用寄存器清零指令
格式： CLRR
操作数： R 为数据寄存器的低 7 位地址
操作过程： 0→R，1→Z
影响状态位： Z
执行时间： 1 个指令周期
说明： 将 R 寄存器清零，状态位 Z 置 1
例： CLRR 08H ；R8=0，Z=1

7) 位操作判断指令：

BCR (Bit clear) 位清零指令
格式： BCR R,b
操作数： R 为数据寄存器的低 7 位地址，b 为数据位 0~7
操作过程： 0→R(b)
影响状态位： 无
执行时间： 1 个指令周期
说明： 将 R 寄存器的 b 位清零
例： BCR 09H,5 ；R9 的第 5 位清零

BSR (Bit set) 位置 1 指令
格式： BSR R,b
操作数： R 为数据寄存器的低 7 位地址，b 为数据位 0~7
操作过程： 1→R(b)
影响状态位： 无
执行时间： 1 个指令周期
说明： 将 R 寄存器的 b 位置 1
例： BCR 09H,5 ；R9 的第 5 位置 1

BTSC (Bit test，skip if clear) 位测试，为零则跳转指令

格式： BTSCR,b

操作数： R 为数据寄存器的低 7 位地址，b 为数据位 0~7

操作过程： 假如 R(b)=0，则跳转

影响状态位： 无

执行时间： 1 个指令周期（不跳转时）或 2 个指令周期（跳转时）

说明： 判断 R 寄存器的 b 位，如果为 0，则跳过下一条指令，为 2 个指令周期，如果不为 0，则按顺序执行，为一个指令周期

例：
```
BTSC    06H,3          ；判断 R6 的第 3 位，为 0，则
                       ；跳过下一条 JUMP 指令，执
                       ；行 INCR 指令，不为 0，则
                       ；按顺序
LJUMPLOOP              ；执行 LJUMP 指令
INCR    08H,1
```

BTSS (Bit test，skip if set) 位测试，为 1 则跳转指令

格式： BTSS R,b

操作数： R 为数据寄存器的低 7 位地址，b 为数据位 0~7

操作过程： 假如 R(b)=1，则跳转

影响状态位： 无

执行时间： 1 个指令周期（不跳转时）或 2 个指令周期（跳转时）

说明： 判断 R 寄存器的 b 位，如果为 1，则跳过下一条指令，为 2 个指令周期，如果不为 1，则按顺序执行，为一个指令周期

例：
```
BTSS    03H,2          ；判断状态寄存器的 Z 位，为 0
                       ；则跳过下一条 LDR 指令，执
                       ；行 DECR 指令，不为 0，则
                       ；按顺序执行 LDR 指令
LDR     06H,0
DECR    08H,1
```

8) 调用跳转返回指令：

LCALL (Call subroutine) 子程序长调用指令

格式： LCALL n

操作数： 11 位的立即数

操作过程： n →PC，PC+1→Stack

影响状态位： 无

执行时间： 2 个指令周期

说明： 将 PC 程序计数器加 1 后压入堆栈，将 11 位立即数 n 置入 PC，它允许在 2K 的程序存储空间内调用子程序，即子程序头可以放在四个页面的任何一页的任何位置，通常 n 是用子程序头的标号代替

例：
```
•
•
•
```

```
                LCALL       DELAY          ；不论 DELAY 地址在何处，程序直接跳
                                           ；转到 DELAY 处
                 •
                 •
        DELAY：  LDR         08H,F
                 •
```

LJUMP (Long JUMP to address) 无条件长跳转指令

格式： LJUMP n

操作数： 11 位的立即数

操作过程： n →PC

影响状态位： 无

执行时间： 2 个指令周期

说明： 将 11 位立即数 n 置入 PC，它允许在 2K 的程序存储空间内任意跳转，即程序可以跳转到存储空间的任何位置，通常 n 是用子程序头的标号代替

例：

```
                 •
                 •
                 •
                LJUMP        TEST           ；不论 TEST 地址在何处，程序直接跳
                                            ；转到 TEST 处
                 •
                 •
                 •
        TEST：  LDR          08H,F
                 •
                 •
```

RTWI (Return，place immediate to W) W 带常数返回指令

格式： RTWI i

操作数： 8 位的立即数

操作过程： Stack →PC，i→W

影响状态位： 无

执行时间： 2 个指令周期

说明： 由子程序带立即数返回，返回的立即数存放在 W 中

例：

```
RTWI        02H                 ；子程序返回，W=02H
```

RTFI (interruptReturn) 中断返回指令

格式： RTFI

操作数： 无

操作过程： Stack →PC

影响状态位： 无

执行时间： 2 个指令周期

说明： 退出中断服务程序，同时使能总中断

例：

```
RTFI                            ；退出中断并使能总中断
```

RET (Return) 子程序返回指令

格式: RET

操作数: 无

操作过程: Stack →PC

影响状态位: 无

执行时间: 2 个指令周期

说明: 退出当前子程序

例: RET ; 返回主程序

◆ **操作数格式:**

操作数支持十进制，十六进制和二进制，数据格式举例如下:

十进制：20、56

十六进制：20H、0x20、0xab、0abH

注：不区别大小写，字母开头的数字需加 0 作为开头，比如 0abH

二进制：10110100B、B'10001010'

3. 汇编伪指令

编译伪指令用来指导编译器如何在编译时产生目的码，编译伪指令可以依其行为细分如下。

1) 文档控制伪指令

语法:

INCLUDE file-name

说明

此伪指令会在编译时，将含入档 file-name 的内容，嵌入至目前的源代码档案，并被视为源代码。

范例:

INCLUDE MDT10F684.INC

INCLUDE macro.asm

在范例中，编译器将 MDT10F684.INC 包含进源代码中，将 macro.asm 内的源代码，嵌入至目前的源代码档案。

2) 程序伪指令

语法:

ORG expression

说明:

此伪指令会将 expression 的计算数值设定给编译器的地址计数器(location counter)，其后之程序代码和数据地址将根据 expression 所计算的偏移量做相对的调整。程序代码和数据偏移量与伪指令 ORG 所在的程序段的开始地址有关，程序段的属性会决定偏移量的实际值

(是绝对地址或相对地址)。
范例：

```
 ORG 008H
mov A, 1
```

在此范例中，指令 mov A, 1 的地址是在程序段的第 8 个地址。

语法：
END
说明：
此伪指令宣告程序的结束，因此最好不要在含入档(included file)中加入这个指令。避免编译器编译到此伪指令后就结束程序的编译流程，之后的指令及伪指令就不会被编译。

语法：
name EQU expression
说明：
程序员将 expression 指定给 name，伪指令 EQU 会制造一个新的数值符号、别名或文字符号(name)来代表 expression。数值符号是一个代表 16 位值的名称、别名则是另一个符号的名称、而文字符号则是代表一串字符组合的名称。name 必须是唯一的，就是之前未被定义过。expression 可以是一个整数、字符串常数、数学表达式或地址表达式。
范例：

```
accreg      EQU   5
bmove       EQU   accreg
```

在这个范例中， accreg 等于 5，而 bmove 相当于 accreg

语法：
#define name expression
说明：
程序员将 expression 指定给 name，伪指令#define 会制造一个新的数值符号、别名或文字符号(name)来代表 expression。数值符号是一个代表 16 位值的名称、别名则是另一个符号的名称、而文字符号则是代表一串字符组合的名称。name 必须是唯一的，之前未被定义过的。expression 可以是一个整数、字符串常数、数学表达式或地址表达式，或者一条指令。
范例：

```
#define   bank0   bcr   status,5
#define   led     porta,0
```

程序中使用：

```
bank0    ;代替 bcr   status,5 这条指令
bsr     led   ;led 为 PA0 口
```

语法：
banksel register
说明：
使用本指令，可以很方便切换到当前需要操作寄存器对应的 BANK，编译器会根据用户定义的寄存器地址并根据当前 MCU 型号，自动计算 BANK，同时自动生成对应的 BANK 选择指令。
注：register 定义地址时必须正确无误，此伪指令适用于特殊功能寄存器和用户寄存器。

范例：

```
PORTA     EQU     05H
CPIOA      EQU     85H
```

程序中使用：

```
banksel    porta     ;切换 bank 为 porta 对应的 bank
ldwi       0xf0
stwr       porta

banksel    cpioa     ;切换 bank 为 cpioa 对应的 bank
clrr       cpioa
```

3) 宏伪指令

宏伪指令是定义一个名称来代表一段原始表达式(指令及伪指令)，而在原始程序档案中可以重复使用此名称以取代这段表达式，也就是程序中所有用到此段表达式的地方皆可用此名称代替之。在编译时，编译器会自动将每一个宏伪指令的名称用宏伪指令所定义的表达式来取代。

在源文件的任何地方皆可定义宏伪指令，只要呼叫此宏伪指令的地方是在宏伪指令定义之后即可。

语法:

```
 nameMACRO [dummy-parameter [,...]]
statements
 endm
```

说明：

name 为宏名称，*dummy-parameter* 为形式参数，*statements* 宏内容。其中形式参数引用时只能为立即数。

范例：

定义宏：

```
EX1  MACRO   a,b        ;a,b 为形式参数
     LDWI    a
     STWR    TMR0
     STWR    TMR1L
     STWR    TMR1H
     LDWI    b
     STWR    CCPR1L
     STWR    CCPR1H
ENDM
```

调用宏：

```
EX1  0x20,0x30
```

4) EEPROM 数值预设

程序中可以通过 DE 伪指令定义 EEPROM 数据（仅限于带 EEPROM 的 MCU），定义 EEPROM

数据，必须用 ORG 指定特殊的地址 0x2100 作为起始地址。

举例：

```
ORG     0x2100
DE  12,22,50,80,55,23,24,25
DE  15,22,50,85,55,23,24,20
DE  0x12,22H,50,80,55,23,24,25
```

说明：

DE 后面跟的数值为单字节，使用半角逗号分隔，个数不限（直到总数达到最大值或者遇到换行）。换行后需重新以 DE 开头。数值格式支持：十进制，十六进制，二进制，可混合使用。

第四部分　YSDCC 编译器使用说明

1. YSDCC 的 C 语言开发工具的语言特点

YSDCC 的 C 语言按 ANSI C 来定义，并进行了 C 语言的扩展，本文不对 C 语言语法作赘述。需指出的是 YSDCC 不支持函数的递归调用，这是因为 MDT 单片机内的堆栈大小是由硬件决定的，资源有限，所以不支持递归调用。它的数据也遵从标准 C 的数据结构，YSDCC 的数据结构是以数据类型的形式出现的。YSDCC 编译器支持的数据类型有无符号字符（unsigned char）、有符号字符（signed char）、无符号整形（unsigned int）、有符号整形（signed int）、无符号长整型（unsigned long）、有符号长整型（signed long）、浮点（float）和指针类型等，需要注意的是，YSDCC 支持的多字节数据都采用低字节在前，高字节在后的原则，即一个多字节数，比如 int 型，在内存单元中存储顺序为低位字节存储在地址低的存储单元中，高位字节存储在地址高的存储单元中，程序员在用 union 定义变量时一定要注意这一特点。

2. 位变量的声明和使用

常常在程序中要定义一些作为标志使用的位变量，在 C 程序中是怎么实现的呢？有两种方式来可以达到这个目的。

1) 第一种：

（1）定义一个联合（union）类型 FLAG（可以任意取自己喜欢的名称）：

```
typedef union
{
struct
    {
unsigned    b0    :1;
unsigned    b1    :1;
unsigned    b2    :1;
unsigned    b3    :1;
unsigned    b4    :1;
unsigned    b5    :1;
unsigned    b6    :1;
unsigned    b7    :1;
    } bit;

unsigned char byte;
}  FLAG
```

为什么要定义成联合体呢？就是为了初始化方便。我们也可以仅仅定义一个位变量的结构体，但是在初始化的时候需要对结构体的每一个成员进行初始化，在位变量比较多的时候，这样做就显得很麻烦。

（2）然后我们就可以这样定义标志位：

```
        FLAG   F_STS;                //定义一个联合体的变量
        #define     FS_AN5AN2     F_STS.bit.b0
        #define     FS_2          F_STS.bit.b2
……
```

（3）在程序中这样使用：

```
    F_STS.byte=0;                //整个结构体初始化为 0
```

如果不定义成联合体的话，那么就必须对每个位变量进行初始化，形如：

```
    FS_AN5AN2=0;
    FS_2=0;
……
```

使用：

```
if (FS_2)
    {
……
    }
else
    {
……
    }
```

2）第二种：

不定义为联合体，使用绝对地址的方式定义

```
struct
    {
unsigned    b0    :1;
unsigned    b1    :1;
unsigned    b2    :1;
unsigned    b3    :1;
unsigned    b4    :1;
unsigned    b5    :1;
unsigned    b6    :1;
unsigned    b7    :1;
    } bit;
volatile __at (0x20) struct bitF_STS;
volatile __at (0x20) unsigned char byte;
```

定义位变量：

```
    #define    FS_AN5AN2    F_STS. b0
    #define    FS_2         F_STS. b2
……
```

初始化、使用跟第一种方式一致；

3. 绝对地址变量的定义

在某些特殊情况下，程序员可能不需要 YSDCC 动态分配某些特殊的全局或静态的局部变量，而使用__at(address)结构定义全局或静态的局部变量，例如：

```
__at (0x20) unsigned charvar1;
volatile__at (0x20) unsigned charvar2; //使用 volatile 关键字，可避免被编译器优化
```

定义了一个称为 var1 的变量，使其定位于单元 0x20，但是需要程序员注意的是，编译器并不保留任何存储空间，而只是将此变量分配到此地址，在编译时，编译器也不能给出任何的警告或错误提示，因此程序员在用绝对地址定义变量时，应该自己查看编译器产生的映象文件或符号表，必须保证绝对变量被分配了唯一的地址，尤其是当此绝对地址正好落在了 YSDCC 分配的自动变量区时，查错是很困难的。

对于 SFR，必须以绝对地址进行定义（一般情况下，在头文件中已经定义）。绝对地址的定义方法如下：

```
volatile __at (0x00) unsigned char IAR;
volatile __at (0x01) unsigned char RTCC;
volatile __at (0x02) unsigned char PCL;
volatile __at (0x03) unsigned char STATUS;
……
```

注：如果通过上述方式定义 SFR，不能通过指针方式操作 SFR；建议尽量使用编译器自带定义文件，文件地址参考为安装目录：\Program Files\YSPRING\MDT-IDE\lib\header；

4、const 类型限定符

const 类型限定符用来通知编译器一个目标具有常数值，不能被改变，被 const 定义的常量被放在 ROM 中，例如：

```
unsigned char const var1[]  ={"yspring"};
unsigned char const var2[]  ={0x00,0x01,0x02,0x03};
```

这两种定义都是合法的，但若通过指针访问这些数组变量，必须将指针定义为常数字符指针才能访问。例如某函数声明为：

```
void func1(const unsigned char  *ptr);
```

则调用常数数组的方法为：

```
void func2(void)
{
    ……
    func1((const unsigned char  *)var1);
    ……
}
```

这一点程序员一定要认真对待，以免编译运行错误很难查找故障。

5、可变型变量 volatile 类型限定符

可变型变量 volatile 类型限定符用来通知编译器某一变量不能保证在连续访问的条件下，其值不被改变，例如所有的与 I/O 口有关的变量在编译器自带的头文件中都是被声明为 volatile 类型，如下所示：

```
volatile __at (0x05) unsigned char PORTA;
```

需要程序员注意的是有可能在中断时被改变的变量应该被定义为 volatile 类型，尤其是编译时选择全局优化级别较高时，定义为 volatile 可以禁止编译器对此变量进行优化，这能够防止编译器进行程序优化时，将认为明显多余的可变型变量删除。

4. volatile 关键字的使用

volatile 关键字是 C 语言里标准的关键字，但却可能常常被忽略使用。然而 C 语言在单片机的应用中常常与硬件打交道，不可不知道其作用及用法，下面简单介绍 volatile 关键字作用。

volatile 影响编译器编译的结果,指出 volatile 变量是随时可能发生变化的，与 volatile 变量有关的运算，不要进行编译优化，以免出错，（YSDCC 在生成汇编代码时会进行编译优化，加 volatile 关键字的变量有关的运算，将不进行编译优化）。

例如，如果对 IO 取反操作，让其输出高低电平：

```
PORTA=0xff;
PORTA=0x00;
```

因为定义头文件时已经使用了 volatile 来声明 PORTA，编译器就会老老实实把上面两条语句全部编译出来。

下面一个典型案例：

```
unsigned char abc;

//中断程序
void Intr(void) __interrupt
{
    if(TIF)
    {
        TIF=0;
        abc++;
    }

}
//主函数
Main
{
    initialization(); //初始化并开了定时中断
    abc=0x00;
    while(abc<100);
    .
    .
    .
}
```

这个例子会发现程序一直在 while 循环里出不来，虽然中断里 abc 有在累加，原因是编译器不会去检查中断里对 abc 的操作，只判断到 abc=0 这条语句，所以对 while 里的判断作了优化，只编译生成了一条原地运行的语句“ljump $”，**如果要实现类似上面的程序，切记使用 volatile 关键字**：volatileunsigned char abc;只有这样定义后，程序才会把 while 后面的判断语句不加优化地编译出来。

5. YSDCC 语言和汇编语言的混合编程

一般情况下，主程序都是用 C 语言编写，C 语言与汇编语言最大的区别就在于汇编程序执行效率较高些，因为 C 语言首先要用 C 编译器生成汇编代码，在不少情况下，C 编译器生成的汇编代码不如用手工生成的汇编代码效率高。在 YSDCC 中，可以用两种办法在 C 程序中嵌入汇编程序，使用__asm　　__endasm 及__asm__在 C 语言中直接嵌入汇编代码.

1、嵌入汇编语句序列：__asm 和__endasm 指令分别用于标示嵌入汇编程序块的开头和结尾，汇编语句的写法遵循汇编编程规范；__asm、__endasm 语句对，以分号结尾；

2、嵌入单条语句：__asm__用于将单条汇编指令嵌入到编译器生成的代码中；

如下所示：

```
void func1(void)
{
    __asm__ (“BANKSEL   PORTA”); //因为不清楚当前寄存器 BANK 处于什么状态，
    __asm__  (“bcr   PORTA”);//所以在使用该语句时，必须再加入一条选择 BANK 的语
```

句

```
        __asm
BANKSEL PORTA
                BSR     PORTA,1
                NOP
                NOP
                BCR     PORTA,1
        __endasm;

        __asm__ (″nop″);
    }
```

由__asm 和__endasm 括起来的语句对中允许使用标号，标号是以下划线“_”打头的字母数字组合，同时还可以使用 C 语言中定义的变量，但此**变量在汇编语句中需要加下划线“_”**；如果是**编译器头文件中已经定义的特殊功能寄存器名称则无需加下划线**，例如：

```
void func2(void)
{
    unsigned char cnt;
    unsigned char t_count;
    for (cnt=0;cnt<10;cnt++)
    {
        t_count=20;
        __asm

        _L00001:
BANKSEL  PORTA
                BSR         PORTA,2
                NOP
                NOP
                BCR         PORTA,2
                BANKSEL     _t_count
                DECRSZ      _t_count,1
                LJUMP       _L00001

        __endasm;
    }
 }
```

特别要注意的地方：因编译器不对汇编指令做任何操作，因此要用户控制汇编指令中的寄存器及变量的 BANK，用宏 BANKSEL 来实现这个功能。

6. 中断定义

void interrupt_identifier (void) __interrupt interrupt_number

1、可在 ISR 内部写、并可在 ISR 外部访问的每个全局变量必须被声明为 volatile，以确保优化器不会删除与该变量相关的指令。

2、以非原子(non-atomic)方式使用数据时(如，访问 16 位/32 位变量)应禁止中断。当对变量的访问为原子方式时，处理器无法中断(带有 ISR)对存储器的数据存取。

3、避免在 ISR 内部调用函数。

例如，定义 TMR0 中断

```
void int_tmr(void) __interrupt 0
{
T0IF=0;
TMR0+=56;
PORTC^=0x04;
}
```

7. 注意事项

使用 YSDCC 时，为了能顺利使用以及更有效的利用资源，应注意以下几点：

（1） 尽量使用无符号数和字节变量。

（2） 函数中尽量使用全局变量，不建议使用局部变量，一可以提高程序效率，二可以防止出错。**在函数发生嵌套时，不同层次之间不能同时使用局部变量**，因为编译器无法分辨函数的嵌套，有可能使两层之间的（同名或不同名）局部变量分配到同一 RAM 地址，造成程序错误，此情况必须使用全局变量避免错误；

（3） **声明变量时的初始化编译器不报错，但实际不起作用，需要额外进行赋值**如：unsigned char abc=0;并不会使 abc=0。

（4） 对于有一定汇编经验的人在开始使用 YSDCC 时，应多注意观看经编译后产生的汇编源代码，并应经常观看经正确编译连接后产生的映象文件（.lst 文件），在该文件中详细列出了分配给变量和代码的地址和生成代码的大小等信息。使用者可了解代码是否优化，变量分配是否合理，堆栈是否溢出等，从而写出高效简洁的 C 源代码。

（5） YSDCC 在好多情况下，不支持类型强制转换，即在类型不匹配时须查验编译后的汇编代码，看是否正确,尤其是对指针操作的时候一定要注意。

（6） 对某位变量自操作时，比如求反，不可以直接简写，例如： !flag;
编译后不能正确产生代码，而须写成：flag = !flag;

（7） 若有某一代码很短的函数被多个函数经常调用，最好将其定义为宏，因为若函数代码很短时，由于被调函数和调用函数不在同一代码页所产生的附加代码可能都会超过函数代码本身的长度。

（8） 当程序中有 16 位数表格，且表长度的两倍超过 255 时（如：const u16 g_usaCRCTable[256] = {0x0000,0x1021,0x2042，……}，256*2 远大于 255），

表指针必须定义大于 8 位位宽的变量类型，如 unsigned short。