TKStudio 内置 51 编译器 SDCC 参数详细讲解

SDCC (小型设备 C 编译器)是为 8 位微控制器开发的免费 C 编译器。尽管兼容多种不同体系结构，但 SDCC C 编译器更适合 8051 内核。
SDCC 是命令行固件开发工具，含预处理器、编译器、汇编器、链接器和优化器。安装文件中还捆绑了 SDCDB，类似于 gdb (GNU 调试器)的源码级调试器。无错的程序采用 SDCC 编译、链接后，生成一个 Intel 十六进制格式的加载模块。
SDCC 主要由以下各部分组成：

sdcc – 编译器
sdcpp – c 预处理器
asx8051 – 8051 汇编器

aslink.exe - 8051 连接器

sdcclib.exe - 库产生器

makebin.exe - 产生二进制代码器
packihx – Intel hex 转换器

当对 SDCC 工程进行编译，汇编以及连接源文件，输出文件如下所示：
*.asm - 由编译器产生的汇编源文件。
*.lst - 由汇编器产生的汇编列表文件。
*.rst - 由连接器产生的汇编列表文件，里面含有连接信息记录。
*.sym - 源文件的符号列表，由汇编器产生。
*.rel 或 *.o - 由汇编器产生的目标文件，供连接器来使用。
*.map - 读入模块的内存映射表，由连接器产生。
*.mem - 记录存储器使用的摘要。
*.ihx - intel hex 格式的文件（可以使用--out-fmts19 选项来选择 Motorola S19 输出格式）。
*.adb - 一个包含调试信息的中间文件，产生.cdb 文件必须依赖这些中间文件来实现（使用--debug 选项）。
*.cdb - 一个可选的包含调试信息的调试文件，在链接时使用选项--debug 就会产生这个文件。

*　－ 一个可选的包含调试信息的 AOMF 或者 AOMF51 文件（由选项--debug 产生）。整个目标模块格式是 OMF51 格式的子格式并且一般被第三方工具使用（调试器，模拟器）。
.dump -- 打印文件调试编译器本身（选项为 －dumpall）

命令行选项（在 TKStudio 中这些参数已经是用户可配置的部分，其配置非常简便，但了解各个参数的意义非常重要）

1 处理器选项

-mmcs51　产生 Intel MCS51 家族处理器代码，默认选项。

-mds390　产生 Dallas DS80C390 处理器代码。

-mds400　产生 Dallas DS80C400 处理器代码。

-mxa51　产生 Phillips XA51 处理器代码。

2 预处理选项

-I<path> 指定处理器查找头文件的路径。

-D<macro[=value]> 命令行里的宏定义，被传递到预处理器处理。

-M　告知预处理器去输出一个描述每个目标文件依赖性的规则。对于每个源文件预处理器输出将遵循一个规则,规则就是目标文件名是源文件并且依赖它里面的所有‘#include’文件，该规则可以使用一行，或者如果它很长，可以通过使用'\'来继续另一行。该规则列表将在标准输出中被打印而不是在 C 预处理程序。

-C 告知预处理器不要抛弃注释， 通常跟‘-E’参数一起使用。

-MM 如‘-M’， 但输出仅提及到用户头文件#include “file"，系统头文件#include <file>被忽略。

-Aquestion(answer) 如果它与像#if #question(answer)条件预处理语句一起进行测试，那么断言 answer 符合 question。‘-A-’关闭通常来描述目标机器的标准断言。

-Umacro 取消宏定义。在所有‘-D’选项之后, 跟在任何‘-include’和‘-imacros’选项之前，‘-U’选项才生效。

-dM 告知预处理器在预处理完成前，去输出一个有效的宏定义列表信息。

-dD 告知预处理器传递所有的宏定义到输出。

-dN 如‘-dD’， 除了宏的参数跟内容被忽略。仅仅‘#define name’被包含到输出里.

-pedantic-parse-number 严格按照规定地分析数字，像 0xfe-LO_B(3)这种情况将被适当的分析并且宏 LO_B(3)得到扩展

注意：这功能不符合 C99 标准！

-Wp preprocessorOption[,preprocessorOption]... 传递 preprocessorOption 到 sdcpp 预处理器。

3 连接器选项

-L --lib-path <absolute path to additional libraries> 这个选项是指示连接器查找附加库的路径。路径必须是绝对的路径。

--xram-loc <Value> 外部 RAM 的起始地址, 默认值是 0。指定值可以是十六进制格式，也可以是十进制格式，例如：--xram-loc 0x8000 或者 --xram-loc 32768。

--code-loc <Value> 代码段的起始地址，默认值是 0。注意：当这选项被使用，那么中断向量表也要被重定位给出的地址。指定值可以是十六进制格式，也可以是十进制格式，如：

--xram-loc 0x8000 或 --xram-loc 32768。

--stack-loc <Value> 默认是堆栈分配在紧跟在数据段之后。使用这个选项，堆栈可以定位到 8051 内部存储空间的任何地方。指定值可以是十六进制格式，也可以是十进制格式，如：--stack-loc 0x20 或 --stack-loc 32。因为 sp 寄存器在压栈或者被调用的时候不断的增长， sp 初始化为一个字节长度的预先提供的值。这个被预提供的值应该不会覆盖任何其他的存储区域（寄存器组或数据段）和保证当前应用程序有足够的空间。选项--pack-iram（现在是默认设置）将重写这设置，因此你应该指定选项--no-pack-iram 去手动的设置堆栈。

--xstack-loc <Value> 默认是外部堆栈紧分配在紧跟 pdata 段之后，使用这个选项，xstack 可以定位到 8051 外部存储空间的任何地方。指定值可以是十六进制格式，也可以是十进制格式，如：--xstack-loc 0x8000 或 --stack-loc 32768. 那提供的值应该不覆盖任何其他存储空间（pdadta 或者 xdata 段）和保证当前应用程序有足够的空间。

--data-loc <Value> 定位内部存储数据段的起始地址，指定值可以是十六进制格式，也可以是十进制格式，如：--data-loc 0x20 或 --data-loc 32。（内部数据段的起始地址被默认设置为尽量低的存储地址，必须考虑到寄存器组跟 0x20 地址的位段情况，例如：如果寄存器 0 组跟 1 组被使用而没有位变量被使用，同时--data-loc 没被使用，那么数据段将被设置为 0x10）。

--idata-loc <Value> 设定 8051 内部存储器间接可寻址区的起始地址，默认值是 0x80。指定值可以是十六进制格式，也可以是十进制格式，如：--idata-loc 0x88 或 --idata-loc 136。

--bit-loc <Value> 设定 8051 内部存储器位可寻址区的起始位置，这里尚未实现。另一种可用的方法是直接传递给连接器： -Wl -bBSEG=<Value>。

--out-fmt-ihx 输出 intel 十六进制格式的最终目标代码。这里是默认的选项。

--out-fmt-s19 输出 Motorola S19 格式的最终目标代码。

--out-fmt-elf 输出 ELF 格式的最终目标代码。 (当前只支持 HC08 处理器)

-Wl linkOption[,linkOption]...传递 linkOption 到连接器。如果引导装载器使用选项如：-Wl -bCSEG=0x1000 将被设置为代码段的起始位置。

4 MCS51 选项

--model-small 为小模式程序产生代码，默认选项。

--model-medium 为中模式程序产生代码。

--model-large 为大模式程序产生代码。

--xstack 在 pdadta 区（通常是外部存储区的头 256 字节）使用一个伪造的堆栈来分配变量和传递参数。

--iram-size <Value> 引起连接器去检查是否内部存储器使用在给定的值之内。

--xram-size <Value> 引起连接器去检查是否外部存储器使用在给定的值之内。

--code-size <Value> 引起连接器去检查是否代码存储器使用在给定的值之内。

--stack-size <Value> 引起连接器去检查是否为堆栈分配了最少<Value>个字节。

--pack-iram 引起连接器去使用没有用到的寄存器组来存放数据变量跟数据包, idada 跟 stack。默认选项。

--no-pack-iram 引起链接器去使用老的方式去分配存储空间。

5 优化选项

--nogcse 不消除全局子表达式，当编译器创建不需要的大型堆或数据空间去存储编译时的临时文件，这个选项可能被使用。当这情况发生，将产生一个警告消息，编译器会指示出已经分配的额外字节数。这里推荐不使用此选项，#pragma nogcse 通常被用来关闭给定函数

里的全局子表达式消除。

--noinvariant 不作循环不变量优化，由于前一个选项的说明使这项可能被关闭。这里推荐不使用此选项，#pragma noinvariant 通常用来关闭给定函数的不变量优化。

--noinduction 将不会做循环感应优化。这里推荐不使用此选项，#pragma noinduction 通常用来关闭给定函数的循环感应优化。

--nojtbound 当 switch 语句使用跳表被实现，将不产生边界条件检查。这里推荐不使用此选项，#pragma nojtbound 通常用来关闭给定函数跳表边界检查。

--noloopreverse 将不做回路循环优化处理。

--nolabelopt 将不进行标签优化（使打印文件更加易读）

--no-xinit-opt 不会把已被初始化的数据从 code 空间复制到 xdata 空间。如果没有已初始化的数据，这样可以节省一些代码空间的字节数。

--nooverlay 编译器将不会覆盖任何函数的参数和局部变量。

--no-peep 禁用窥视孔优化的内置式规则。

--peep-file <filename> 这个选项可以用来使用附加规则来被窥视孔优化器使用。

--peep-asm 通过窥视孔优化器传递内嵌汇编代码。这可以使内嵌汇编代码造成一些意想不到的改变，使用这选项前，请仔细查看在源文件树的<target>/peeph.def 文件中的窥视孔优化规则定义。

--opt-code-speed 编译器优化代码生成实现快速的代码，可能是在牺牲代码大小

--opt-code-size 编译器优化代码生成实现简洁的代码，可能是在牺牲代码速度。

6 中间打印选项

下面的选项被提供给调试编译器使用。它们提供一种方法，打印出在编译的过程中编译器产生的不同平台上人为易读的中间代码。

--dumpraw 这个选项将引起编译器去打印中间代码到文件<source filename> .dumpraw 中。当一个函数的中间代码生成以后,中间代码就被转存到这个文件中去，而且在任何优化之前就已经做好了。

--dumpgcse 将产生一个消除全局子表达式后的中间代码文件<sourcefilename>.dumpgcse.。

--dumpdeadcode 将产生一个删除死代码后的中间代码文件<source filename>.dumpdeadcode。

--dumploop 将产生一个循环优化后的中间代码文件<source filename>.dumploop。

--dumprange 将产生分析符号生命期后的中间代码文件<source filename>.dumprange。

--dumlrange 将打印所有符号的生命期信息。

--dumpregassign 将产生一个分配寄存器后的中间代码文件<source filename>.dumprassgn。

--dumplrange 将打印临时符号变量的生命期信息。

--dumpall 将打印上面所提及到的所有中间代码文件信息。

7 其它选项

-c --compile-only 编译并汇编源文件，但不调用连接器。

--c1mode 从标准输入读取预处理了的源文件，并编译它。汇编器输出的文件名必须使用-o 选项指定。

-E 只有运行预处理器。预处理所有指定的 C 源文件跟输出结果到标准输出中。

-o <path/file> 输出路径名，这路径里的一切文件将被代替，或者被产生的输出文件的文件名。如果参数是路径，那么它尾部必须包含要“/”（或者“\\”在 windows 平台），被用来

鉴定为路径。Windows 用户请注意：如果路径包含空格，那么它必须使用双引号包括起来。尾部的反斜杆应该成双出现，为了防止最后的引用分号被解释出来。例如：-o ”F:\Projects\test3\output 1\\” 或者 反斜杆被放置在尾部引号的后面，例如：-o ”F:\Projects\test3\output 1”\。路径也可以使用斜杆作为目录分界符，如：-o ”F:/Projects/test3/output 1/”。

--stack-auto 所有源文件函数被编译为可重入函数，也就是，参数跟本地变量将被分配到堆栈上去。如果这选项被使用，工程里所有的源文件也该使用这选项被编译。

--callee-saves function1[,function2][,function3].... 默认情况下，编译器遵循由调用者保存寄存器的协定，然而当大函数调用比较小的函数的时候，该选项可能产生不必要的寄存器压栈出栈操作。该选项能够转变指定函数的寄存器保存协议。当调用这些函数时编译器将不保存寄存器，同时也没有为这些函数的入口跟出口产生附加的代码，来为这些函数写寄存器或者恢复寄存器时使用。这样可以充分地减少代码和改善执行已产生代码的运行时间。将来，编译器（有内部程序分析能力）能为每个函数调用确定适当的配置。千万不要为形如_mulint... 的 built-in 的函数使用这个选项，如果这个选项被用作库函数，适当的库函数也需要使用相同的选项被重新编译。如果这工程是由多个源文件组成，那么所有的源文件也该使用相同的 --callee-saves 选项被编译。

--debug 当该选项被使用时，编译器将产生调试信息。所有的调试信息将被保存到一个后缀名为.cdb 的文件中，并且能够被 SDCDB 工具使用。另外一个没个后缀名的文件将包含有关 AOMF 或者 AOMF51 格式的调试信息，一般情况下被第三方工具使用。

-S 在进行适当的编译以后停止，不作汇编处理。将会为指定的输入文件输出汇编代码。

--int-long-reent 整型（16 位）和长整型（32 位）库被编译为可重入的。注意默认情况下这些库被编译成非可重入的。

--cyclomatic 该选项将导致编译器为每个在源文件中的函数产生一个信息通知。该通知包含一些有关函数的重要信息。

--float reent 浮点类型库将被编译成可重入的。

--funsigned-char 默认的每种类型都是有符号的。有一些嵌入式环境，char 是默认无符号的。使用选项--funsigned-char，可以设置 char 为无符号。如果该选项被设置并且没有给出符号关键字(unsigned/signed)，字符类型将被认为是有符号的。该选项对其它类型没有影响。

--main-return 该选项主要用在当产生的代码被监视程序调用或者 main 例程包括一个无限循环。该选项产生的轻微的小量代码并且保存两个字节到堆栈中。从’main’返回将返回到调用 main 的函数中。默认的设置是死循环。例如：产生’sjmp .’代码。

--nostdinc 该选项将阻止编译器传递默认的 include 路径到预处理器

--nostdlib 该选项将阻止编译器传递默认的库路径到连接器。

--verbose 显示编译器使用的各种选项。

-V 显示编译器实际运行的各种命令。

--no-c-code-in-asm 在 asm 文件中隐藏丑陋的和低效的 C 语言代码。

--fverbose-asm 在产生的 asm 文件里包含代码产生器和窥孔注释。

--no-peep-comments 在产生的 asm 文件里不包括窥孔注释，即使--fverbose-asm 选项被指定。

--i-code-in-asm 在 asm 文件中包括 i-codes。主要用来帮助调试编译器本身。

--less pedantic 禁止一些更严重的警告。

--disable-warning <nnnn> 编译器将不会做对警告编号为<nnnn>作出任何警告提示。

-print-search-dirs 显示编译器的搜索目录。

--vc 使用 MSVC 的样式显示错误和警告，因此能够使 SDCC 和 visual studio IDE 一起使

用。

--use-stdout 发送错误和警告到 stdout，来代替 stderr。

-Wa asmOption[,asmOption]... 传递 asm 选项到汇编器。

--std-sdcc89 C 语言程序将遵循 C89 标准，但允许 SDCC 中标准有冲突的特性。

--std-c89 C 语言程序将遵循 C89 标准，并且禁止 SDCC 中标准有冲突的特性。

--std-sdcc99 C 语言程序将遵循 C99 标准，但允许 SDCC 中标准有冲突的特性。

--std-c99 C 语言程序将遵循 C99 标准，并且禁止 SDCC 中标准有冲突的特性。

--codeseg <Name> 使用<name>（最多 8 个字节）做为代码段名，默认情况下为 CSEG。

--constseg <Name>使用<name>（最多 8 个字节）做为常量段名，默认情况下为 CONST。

--fdollars-in-identifiers 允许’$’作为标识字符。

--more-pedantic 实际上它不是 SDCC 编译器的选项但如果想要更多的警告信息，可以使用其它分离的工具来进行语法检查。像 splint http://www.splint.org。

--short-is-8bits short 类型将作为 8 位来对待。

1.编译器引入的宏定义

编译器引入下面的宏定义：

#define Description

SDCC 总是被定义。从 2.5.6 版本以后版本号作为一个整数（ex.256）

SDCC_mcs51 或者 SDCC_ds390 依赖于模型的使用

SDCC_STACK_AUTO 当--stack-auto 选项被使用

SDCC_MODEL_SMALL 当--model-small 被使用

SDCC_MODEL_MEDIUM 当--model-medium 被使用

SDCC_MODEL_LARGE 当--model-large 被使用

SDCC_USE_XSTACK 当--xstack option 被使用

SDCC_STACK_TENBIT 当-mds390 被使用

SDCC_MODEL_FLAT24 当-mds390 被使用

2.pragma 预处理命令

Pragma 用来打开或者关闭某些编译器选项。一部分 Pragma 跟相应的命令行选项密切相关。Pragma 应该放在一个函数之前或之后，把它放在函数体内部将发生不可预知的结果。

SDCC 支持下面的#pragma 指示：

save –保存当前选项到 save/restore stack

restore –恢复最后保存的选项。saves & restores 能够被嵌套使用。SDCC 使用一个 save/restore 堆栈：save 将当前选项压入堆栈，而 restore 从堆栈中弹出当前选项。

如下所示：

#pragma save

callee_saves function1[,function2[,function3...]] –默认情况下，编译器遵循由调用者保存寄存器的协定，然而当大函数调用比较小的函数的时候，该选项可能产生不必要的寄存器压栈出栈操作。该选项能够转变指定函数的寄存器保存协议。当调用这些函数时编译器将不保存寄存器，额外的代码需要手工的插入到这些函数体的开始和结尾部分，用来保存和恢复这些函数使用的寄存器。这样可以充分地减少代码和改善执行已产生代码的运行时间。将来，编

译器（有内部程序分析能力）能为每个函数调用确定适当的配置。

exclude none | {acc[,b[,dpl[,dph]]] - exclude pragma 禁止在中断服务例程中生成 push/pop 指令。该指示应该放在 ISR 函数的定义前面，并且它将影响所有在它后面的 ISR 函数。

less_pedantic- 编译器将不会对明显的错误进行警告。更明确地，下面这些警告将被禁止：
comparison is always [true/false] due to limited range of data type (94);
overflow in implicit constant conversion (158);
the (in)famous] conditional flow changed by optimizer: so said EVELYN the modified DOG(110);
function '[function name]' must return value (59).
甚至不重要的警告也被关闭了，如下：
constant value '[]', out of range (81);
[left/right] shifting more than size of object changed to zero(116);
unreachable code (126);
integer overflow in expression_r_r (165);
unmatched #pragma save and #pragma restore (170);
comparison of 'signed chsar' with 'unsigned char' requires promotion to int (185);
ISO C90 does not support flexible array members (187);
extended stack by [number] bytes for compiler temp(s) :in function '[function name]': [](114);
function '[function name]', # edges [number] , # nodes [number] ,cyclomatic complexity [number] (121).

disable_warning <nnnn> -编译器将不会做对警告编号为<nnnn>作出任何警告提示。
nogcse –停止全局公共子表达式的消除
noinduction – 停止循环归纳优化器
noinvariant –不做循环不变式的最优化
noiv – 禁止产生中断矢量表。当需要手工定义中断向量表，该选项非常有用。
nojtbound –当 switch 语句转变为跳表时，将不生成边界值检查代码。
noloopreverse –不做循环回路优化
nooverlay – 编译器将不会覆盖函数的参数和局部变量。
stackauto- 函数将被编译为可重入的，函数参数和局部变量将被分配在堆栈中。
opt_code_speed – 编译器在生成代码时将做速度方面的优化，很可能要以牺牲代码大小为代价
opt_code_size –编译器将为生成的简洁的代码做优化，很可能要以牺牲执行速度为代价。不过该选项对代码生成只有一点点影响。
opt_code_balanced –编译器会对产生的代码做大小和速度两方面的考虑，在这两者之间找平衡点（默认选项）
std_sdcc89 –C 语言程序将遵循 C89 标准，但允许 SDCC 中标准有冲突的特性
std_c89 –C 语言程序将遵循 C89 标准，并且禁止 SDCC 中标准有冲突的特性
std_sdcc99 –C 语言程序将遵循 C99 标准，但允许 SDCC 中标准有冲突的特性
std_c99 –C 语言程序将遵循 C99 标准，并且禁止 SDCC 中标准有冲突的特性
codeseg <name>- 使用<name>（最多 8 个字节）作为代码段名

constseg <name>- 使用<name>（最多 8 个字节）作为常量段名

pedantic_parse_number (+ | -) -严格按照规定地分析数字，像 0xfe-LO_B(3)这种情况将被适当的分析，并且宏 LO_B(3)得到扩展。默认情况下该选项是关闭的。

下面介绍一个怎样使用这个 pragma 例子：

程序清单 3 1 pedantic_parse_number 宏使用例子

```
#pragma pedantic_parse_number +
#define LO_B(x) ((x) & 0xff)
unsigned char foo(void)
{
unsigned char c=0xfe-LO_B(3);
return c;
}
```

注意：该功能与 C 标准不一致

preproc_asm (+ | -) –_asm _endasm 语句块的预处理开或者关功能。默认情况下是开的。通过这个 prama 来定义多行汇编代码。这将防止预处理器改变汇编代码要求的格式。

下面是介绍是怎样使用这个 pragma 的例子：

程序清单 3 2 preproc_asm 宏使用例子

```
#pragma preproc_asm -
#define MYDELAY _asm
nop ;my assembly comment...
nop
nop
_endasm
#pragma preproc_asm +
void foo (void)
{
...
MYDELAY;
...
}
```

sdcc_hash (+ | -) –允许在宏定义中有"naked"无用信息，例如：

```
#define DIR_LO(x) #(x & 0xff)
```

默认情况下是关闭的。

下面介绍怎样使用这个 pragma 的例子。

程序清单 3 3 sdcc_hash 宏使用例子

```
#pragma preproc_asm +
#pragma sdcc_hash +
#define ROMCALL(x) \
mov R6_B3, #(x & 0xff) \
mov R7_B3, #((x > > 8) & 0xff) \
lcall __romcall

...
```

```
_asm
ROMCALL(72)
_endasm;
...
```

一些 pragmas 被规定为用来打开或者关闭某项优化，这样可能导致编译器产生额外的堆栈或者数据空间来存放编译器产生的临时变量。在比较大的函数中经常发生。Pragma 指示应该要象下面的例子那样被使用，它们被用来在一个给定的函数中控制选项和优化。

程序清单 3 4 pragma 的使用

```
#pragma save
#pragma nogcse
#pragma noinduction
int foo ()
{
...

...
}
#pragma restore
```

当额外的空间被分配时，编译器将产生警告信息。当改变函数的选项时，这里强烈推荐使用 save 和 restore pragma。