SDCC Compiler User Guide

16th July 2001

Contents

1 Introduction

1.1 About SDCC

SDCC is a Freeware, retargettable, optimizing ANSI-C compiler by **Sandeep Dutta** designed for 8 bit Microprocessors. The current version targets Intel MCS51 based Microprocessors(8051,8052, etc), Zilog Z80 based MCUs, and the Dallas DS80C390 variant. It can be retargetted for other microprocessors, support for PIC, AVR and 186 is under development. The entire source code for the compiler is distributed under GPL. SDCC uses ASXXXX & ASLINK, a Freeware, retargettable assembler & linker. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware effectively.

In addition to the MCU specific optimizations SDCC also does a host of standard optimizations like:

- global sub expression elimination,
- loop optimizations (loop invariant, strength reduction of induction variables and loop reversing),
- constant folding & propagation,
- copy propagation,
- dead code elimination
- jumptables for *switch* statements.

For the back-end SDCC uses a global register allocation scheme which should be well suited for other 8 bit MCUs.

The peep hole optimizer uses a rule based substitution mechanism which is MCU independent.

Supported data-types are:

- char (8 bits, 1 byte),
- short and int (16 bits, 2 bytes),
- long (32 bit, 4 bytes)
- float (4 byte IEEE).

The compiler also allows *inline assembler code* to be embedded anywhere in a function. In addition, routines developed in assembly can also be called.

SDCC also provides an option (–cyclomatic) to report the relative complexity of a function. These functions can then be further optimized, or hand coded in assembly if needed.

SDCC also comes with a companion source level debugger SDCDB, the debugger currently uses ucSim a freeware simulator for 8051 and other micro-controllers.

The latest version can be downloaded from `http://sdcc.sourceforge.net/`.

1.2 Open Source

All packages used in this compiler system are *opensource* and *freeware*; source code for all the sub-packages (asxxxx assembler/linker, pre-processor) is distributed with the package. This documentation is maintained using a freeware word processor (LyX).

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. In other words, you are welcome to use, share and improve this program. You are forbidden to forbid anyone else to use, share and improve what you give them. Help stamp out software-hoarding!

1.3 Typographic conventions

Throughout this manual, we will use the following convention. Commands you have to type in are printed in **"sans serif"**. Code samples are printed in `typewriter font.` Interesting items and new terms are printed in *italicised type.*

1.4 Compatibility with previous versions

This version has numerous bug fixes compared with the previous version. But we also introduced some incompatibilities with older versions. Not just for the fun of it, but to make the compiler more stable, efficient and ANSI compliant.

- short is now equivalent to int (16 bits), it used to be equivalent to char (8 bits)
- the default directory where include, library and documention files are stored is no in /usr/local/share

- char type parameters to vararg functions are casted to int unless explicitly casted, e.g.:
  ```
  char a=3;
  printf ("%d %c\n", a, (char)a);
  ```
  will push a as an int and as a char resp.
- option –regextend has been removed
- option –noreparms has been removed

<pending: more incompatibilities?>

1.5 System Requirements

What do you need before you start installation of SDCC? A computer, and a desire to compute. The preferred method of installation is to compile SDCC from source using GNU gcc and make. For Windows some pre-compiled binary distributions are available for your convenience. You should have some experience with command line tools and compiler use.

1.6 Other Resources

The SDCC home page at `http://sdcc.sourceforge.net/` is a great place to find distribution sets. You can also find links to the user mailing lists that offer help or discuss SDCC with other SDCC users. Web links to other SDCC related sites can also be found here. This document can be found in the DOC directory of the source package as a text or HTML file. Some of the other tools (simulator and assembler) included with SDCC contain their own documentation and can be found in the source distribution. If you want the latest unreleased software, the complete source package is available directly by anonymous CVS on cvs.sdcc.sourceforge.net.

1.7 Wishes for the future

There are (and always will be) some things that could be done. Here are some I can think of:

sdcc -c –model-large -o large _atoi.c (where large could be a different basename or a directory)

```
char KernelFunction3(char p) at 0x340;
```

If you can think of some more, please send them to the list.

<pending: And then of course a proper index-table>

2 Installation

2.1 Linux/Unix Installation

1. Download the source package, it will be named something like sdcc-2.x.x.tgz.
2. Bring up a command line terminal, such as xterm.
3. Unpack the file using a command like: **"tar -xzf sdcc-2.x.x.tgz**", this will create a sub-directory called sdcc with all of the sources.
4. Change directory into the main SDCC directory, for example type: **"cd sdcc**".
5. Type **"./configure**". This configures the package for compilation on your system.
6. Type **"make**". All of the source packages will compile, this can take a while.
7. Type **"make install"** as root. This copies the binary executables, the include files, the libraries and the documentation to the install directories.

2.2 Windows Installation

<pending: is this complete? where is borland, mingw>

For installation under Windows you first need to pick between a pre-compiled binary package, or installing the source package along with the Cygwin package. The binary package is the quickest to install, while the Cygwin package includes all of the open source power tools used to compile the complete SDCC source package in the Windows environment. If you are not familiar with the Unix command line environment, you may want to read the section on additional information for Windows users prior to your initial installation.

2.2.1 Windows Install Using a Binary Package

1. Download the binary package and unpack it using your favorite unpacking tool (gunzip, WinZip, etc). This should unpack to a group of sub-directories. An example directory structure after unpacking is: c:\usr\local\bin for the executables, c:\usr\local\share\sdcc\include and c:\usr\local\share\sdcc\lib for the include and libraries.
2. Adjust your environment PATH to include the location of the bin directory. For example, make a setsdcc.bat file with the following: set PATH=c:\usr\local\bin;%PATH%
3. When you compile with sdcc, you may need to specify the location of the lib and include folders. For example, sdcc -I c:\usr\local\share\sdcc\include -L c:\usr\local\share\sdcc\lib\small test.c

2.2.2 Windows Install Using Cygwin

1. Download and install the cygwin package from the redhat site`http://sources.redhat.com/cygwin/`. Currently, this involved downloading a small install program which then automates downloading and installing selected parts of the package (a large 80M byte sized dowload for the whole thing).
2. Bring up a Unix/Bash command line terminal from the Cygwin menu.
3. Follow the instructions in the preceding Linux/Unix installation section.

2.3 Testing out the SDCC Compiler

The first thing you should do after installing your SDCC compiler is to see if it runs. Type **"sdcc –version"** at the prompt, and the program should run and tell you the version. If it doesn't run, or gives a message about not finding sdcc program, then you need to check over your installation. Make sure that the sdcc bin directory is in your executable search path defined by the PATH environment setting (see the Trouble-shooting section for suggestions). Make sure that the sdcc program is in the bin folder, if not perhaps something did not install correctly.

SDCC binaries are commonly installed in a directory arrangement like this:

/usr/local/bin	Holds executables(sdcc, s51, aslink, ...)
/usr/local/share/sdcc/lib	Holds common C libraries
/usr/local/share/sdcc/include	Holds common C header files

Make sure the compiler works on a very simple example. Type in the following test.c program using your favorite editor:

```
int test(int t) {
return t+3;
}
```

Compile this using the following command: **"sdcc -c test.c".** If all goes well, the compiler will generate a test.asm and test.rel file. Congratulations, you've just compiled your first program with SDCC. We used the -c option to tell SDCC not to link the generated code, just to keep things simple for this step.

The next step is to try it with the linker. Type in **"sdcc test.c**". If all goes well the compiler will link with the libraries and produce a test.ihx output file. If this step fails (no test.ihx, and the linker generates warnings), then the problem is most likely that sdcc cannot find the /usr/local/share/sdcc/lib directory (see the Install trouble-shooting section for suggestions).

The final test is to ensure sdcc can use the standard header files and libraries. Edit test.c and change it to the following:

```
#include <string.h>
main() {
char str1[10];
strcpy(str1, "testing");
}
```

Compile this by typing **"sdcc test.c"**. This should generate a test.ihx output file, and it should give no warnings such as not finding the string.h file. If it cannot find the string.h file, then the problem is that sdcc cannot find the /usr/local/share/sdcc/include directory (see the Install trouble-shooting section for suggestions).

2.4 Install Trouble-shooting

2.4.1 SDCC cannot find libraries or header files.

The default installation assumes the libraries and header files are located at "/usr/local/share/sdcc/lib" and "/usr/local/share/sdcc/include". An alternative is to specify these locations as compiler options like this: **"sdcc -L /usr/local/sdcc/lib/small -I /usr/local/sdcc/include test.c"**.

2.4.2 SDCC does not compile correctly.

A thing to try is starting from scratch by unpacking the .tgz source package again in an empty directory. Confure it again and build like:

make 2&>1 | tee make.log

After this you can review the make.log file to locate the problem. Or a relevant part of this be attached to an email that could be helpful when requesting help from the mailing list.

2.4.3 What the "./configure" does

The "./configure" command is a script that analyzes your system and performs some configuration to ensure the source package compiles on your system. It will take a few minutes to run, and will compile a few tests to determine what compiler features are installed.

2.4.4 What the "make" does.

This runs the GNU make tool, which automatically compiles all the source packages into the final installed binary executables.

2.4.5 What the "make install" command does.

This will install the compiler, other executables and libraries in to the appropriate system directories. The default is to copy the executables to /usr/local/bin and the libraries and header files to /usr/local/share/sdcc/lib and /usr/local/share/sdcc/include.

2.5 Additional Information for Windows Users

<pending: is this up to date?>

The standard method of installing on a Unix system involves compiling the source package. This is easily done under Unix, but under Windows it can be a more difficult process. The Cygwin is a large package to download, and the compilation runs considerably slower under Windows due to the overhead of the Cygwin tool set. An alternative is to install a pre-compiled Windows binary package. There are various trade-offs between each of these methods.

The Cygwin package allows a Windows user to run a Unix command line interface (bash shell) and also implements a Unix like file system on top of Windows. Included are many of the famous GNU software development tools which can augment the SDCC compiler.This is great if you have some experience with Unix command line tools and file system conventions, if not you may find it easier to start by installing a binary Windows package. The binary packages work with the Windows file system conventions.

2.5.1 Getting started with Cygwin

SDCC is typically distributed as a tarred/gzipped file (.tgz). This is a packed file similar to a .zip file. Cygwin includes the tools you will need to unpack the SDCC distribution (tar and gzip). To unpack it, simply follow the instructions under the Linux/Unix install section. Before you do this you need to learn how to start a cygwin shell and some of the basic commands used to move files, change directory, run commands and so on. The change directory command is **“cd”**, the move command is **“mv”**. To print the current working directory, type **“pwd”**. To make a directory, use **“mkdir”**.

There are some basic differences between Unix and Windows file systems you should understand. When you type in directory paths, Unix and the Cygwin bash prompt uses forward slashes ’/’ between directories while Windows traditionally uses ’\’ backward slashes. So when you work at the Cygwin bash prompt, you will need to use the forward ’/’ slashes. Unix does not have a concept of drive letters, such as “c:“, instead all files systems attach and appear as directories.

2.5.2 Running SDCC as Native Compiled Executables

If you use the pre-compiled binaries, the install directories for the libraries and header files may need to be specified on the sdcc command line like this: **"sdcc -L c:\usr\local\sdcc\lib\small -I c:\usr\local\sdcc\include test.c"** if you are running outside of a Unix bash shell.

If you have successfully installed and compiled SDCC with the Cygwin package, it is possible to compile into native .exe files by using the additional makefiles included for this purpose. For example, with the Borland 32-bit compiler you would run **"make -f Makefile.bcc"**. A command line version of the Borland 32-bit compiler can be downloaded from the Inprise web site.

2.6 SDCC on Other Platforms

- **FreeBSD and other non-GNU Unixes** - Make sure the GNU make is installed as the default make tool.
- SDCC has been ported to run under a variety of operating systems and processors. If you can run GNU GCC/make then chances are good SDCC can be compiled and run on your system.

2.7 Advanced Install Options

The "configure" command has several options. The most commonly used option is –prefix=<directory name>, where <directory name> is the final location for the sdcc executables and libraries, (default location is /usr/local). The installation process will create the following directory structure under the <directory name> specified (if they do not already exist).

bin/ - binary exectables (add to PATH environment variable)
bin/share/
bin/share/sdcc/include/ - include header files
bin/share/sdcc/lib/
bin/share/sdcc/lib/small/ - Object & library files for small model library
bin/share/sdcc/lib/large/ - Object & library files for large model library
bin/share/sdcc/lib/ds390/ - Object & library files forDS80C390 library

The command **"./configure –prefix=/usr/local"** will configure the compiler to be installed in directory /usr/local.

2.8 Components of SDCC

SDCC is not just a compiler, but a collection of tools by various developers. These include linkers, assemblers, simulators and other components. Here is a summary of some of the components. Note that the included simulator and assembler have separate documentation which you can find in the source package in their respective directories. As SDCC grows to include support for other processors, other packages from various developers are included and may have their own sets of documentation.

You might want to look at the files which are installed in <installdir>. At the time of this writing, we find the following programs:

In <installdir>/bin:

- sdcc - The compiler.
- sdcpp - The C preprocessor.
- asx8051 - The assembler for 8051 type processors.

- as-z80, as-gbz80 - The Z80 and GameBoy Z80 assemblers.
- aslink -The linker for 8051 type processors.
- link-z80, link-gbz80 - The Z80 and GameBoy Z80 linkers.
- s51 - The ucSim 8051 simulator.
- sdcdb - The source debugger.
- packihx - A tool to pack Intel hex files.

In <installdir>/share/sdcc/include

- the include files

In <installdir>/share/sdcc/lib

- the sources of the runtime library and the subdirs small large and ds390 with the precompiled relocatables.

In <installdir>/share/sdcc/doc

- the documentation

As development for other processors proceeds, this list will expand to include executables to support processors like AVR, PIC, etc.

2.8.1 sdcc - The Compiler

This is the actual compiler, it in turn uses the c-preprocessor and invokes the assembler and linkage editor.

2.8.2 sdcpp (C-Preprocessor)

The preprocessor is a modified version of the GNU preprocessor. The C preprocessor is used to pull in #include sources, process #ifdef statements, #defines and so on.

2.8.3 asx8051, as-z80, as-gbz80, aslink, link-z80, link-gbz80 (The Assemblers and Linkage Editors)

This is retargettable assembler & linkage editor, it was developed by Alan Baldwin. John Hartman created the version for 8051, and I (Sandeep) have made some enhancements and bug fixes for it to work properly with the SDCC.

2.8.4 s51 - Simulator

S51 is a freeware, opensource simulator developed by Daniel Drotos (`mailto:drdani@mazsola.iit.uni-miskolc.hu`). The simulator is built as part of the build process. For more information visit Daniel's website at: `http://mazsola.iit.uni-miskolc.hu/~drdani/embedded/s51` .

2.8.5 sdcdb - Source Level Debugger

Sdcdb is the companion source level debugger. The current version of the debugger uses Daniel's Simulator S51, but can be easily changed to use other simulators.

3 Using SDCC

3.1 Compiling

3.1.1 Single Source File Projects

For single source file 8051 projects the process is very simple. Compile your programs with the following command **"sdcc sourcefile.c".** This will compile, assemble and link your source file. Output files are as follows

sourcefile.asm - Assembler source file created by the compiler
sourcefile.lst - Assembler listing file created by the Assembler
sourcefile.rst - Assembler listing file updated with linkedit information, created by linkage editor
sourcefile.sym - symbol listing for the sourcefile, created by the assembler
sourcefile.rel - Object file created by the assembler, input to Linkage editor
sourcefile.map - The memory map for the load module, created by the Linker
sourcefile.ihx - The load module in Intel hex format (you can select the Motorola S19 format with –out-fmt-s19)
sourcefile.cdb - An optional file (with –debug) containing debug information

3.1.2 Projects with Multiple Source Files

SDCC can compile only ONE file at a time. Let us for example assume that you have a project containing the following files:

foo1.c (contains some functions)
foo2.c (contains some more functions)
foomain.c (contains more functions and the function main)

The first two files will need to be compiled separately with the commands:

sdcc -c foo1.c
sdcc -c foo2.c

Then compile the source file containing the *main()* function and link the files together with the following command:

sdcc foomain.c foo1.rel foo2.rel

Alternatively, *foomain.c* can be separately compiled as well:

```
sdcc -c foomain.c
sdcc foomain.rel foo1.rel foo2.rel
```

The file containing the *main()* function MUST be the FIRST file specified in the command line, since the linkage editor processes file in the order they are presented to it.

3.1.3 Projects with Additional Libraries

Some reusable routines may be compiled into a library, see the documentation for the assembler and linkage editor (which are in <installdir>/share/sdcc/doc) for how to create a *.lib* library file. Libraries created in this manner can be included in the command line. Make sure you include the -L <library-path> option to tell the linker where to look for these files if they are not in the current directory. Here is an example, assuming you have the source file *foomain.c* and a library *foolib.lib* in the directory *mylib* (if that is not the same as your current project):

```
sdcc foomain.c foolib.lib -L mylib
```

Note here that *mylib* must be an absolute path name.

The most efficient way to use libraries is to keep seperate modules in seperate source files. The lib file now should name all the modules.rel files. For an example see the standard library file *libsdcc.lib* in the directory <installdir>/share/lib/small.

3.2 Command Line Options

3.2.1 Processor Selection Options

-mmcs51 Generate code for the MCS51 (8051) family of processors. This is the default processor target.

-mds390 Generate code for the DS80C390 processor.

-mz80 Generate code for the Z80 family of processors.

-mgbz80 Generate code for the GameBoy Z80 processor.

-mavr Generate code for the Atmel AVR processor(In development, not complete).

-mpic14 Generate code for the PIC 14-bit processors(In development, not complete).

-mtlcs900h Generate code for the Toshiba TLCS-900H processor(In development, not complete).

3.2.2 Preprocessor Options

-I<path> The additional location where the pre processor will look for <..h> or “..h” files.

-D<macro[=value]> Command line definition of macros. Passed to the pre processor.

-M Tell the preprocessor to output a rule suitable for make describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the files ‘#include’d in it. This rule may be a single line or may be continued with ‘\’-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program. ‘-M’ implies ‘-E’.

-C Tell the preprocessor not to discard comments. Used with the ‘-E’ option.

-MM Like ‘-M’ but the output mentions only the user header files included with ‘#include “file"’. System header files included with ‘#include <file>’ are omitted.

-Aquestion(answer) Assert the answer answer for question, in case it is tested with a preprocessor conditional such as ‘#if #question(answer)’. ‘-A-’ disables the standard assertions that normally describe the target machine.

-Aquestion (answer) Assert the answer answer for question, in case it is tested with a preprocessor conditional such as ‘#if #question(answer)’. ‘-A-’ disables the standard assertions that normally describe the target machine.

-Umacro Undefine macro macro. ‘-U’ options are evaluated after all ‘-D’ options, but before any ‘-include’ and ‘-imacros’ options.

-dM Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the ‘-E’ option.

-dD Tell the preprocessor to pass all macro definitions into the output, in their proper sequence in the rest of the output.

-dN Like ‘-dD’ except that the macro arguments and contents are omitted. Only ‘#define name’ is included in the output.

3.2.3 Linker Options

-L –lib-path <absolute path to additional libraries> This option is passed to the linkage editor’s additional libraries search path. The path name must be absolute. Additional library files may be specified in the command line. See section Compiling programs for more details.

–xram-loc<Value> The start location of the external ram, default value is 0. The value entered can be in Hexadecimal or Decimal format, e.g.: –xram-loc 0x8000 or –xram-loc 32768.

–code-loc<Value> The start location of the code segment, default value 0. Note when this option is used the interrupt vector table is also relocated to the given address. The value entered can be in Hexadecimal or Decimal format, e.g.: –code-loc 0x8000 or –code-loc 32768.

–stack-loc<Value> The initial value of the stack pointer. The default value of the stack pointer is 0x07 if only register bank 0 is used, if other register banks are used then the stack pointer is initialized to the location above the highest register bank used. eg. if register banks 1 & 2 are used the stack pointer will default to location 0x18. The value entered can be in Hexadecimal or Decimal format, eg. –stack-loc 0x20 or –stack-loc 32. If all four register banks are used the stack will be placed after the data segment (equivalent to –stack-after-data)

–stack-after-data This option will cause the stack to be located in the internal ram after the data segment.

–data-loc<Value> The start location of the internal ram data segment, the default value is 0x30.The value entered can be in Hexadecimal or Decimal format, eg. –data-loc 0x20 or –data-loc 32.

–idata-loc<Value> The start location of the indirectly addressable internal ram, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, eg. –idata-loc 0x88 or –idata-loc 136.

–out-fmt-ihx The linker output (final object code) is in Intel Hex format. (This is the default option).

–out-fmt-s19 The linker output (final object code) is in Motorola S19 format.

3.2.4 MCS51 Options

–model-large Generate code for Large model programs see section Memory Models for more details. If this option is used all source files in the project should be compiled with this option. In addition the standard library routines are compiled with small model, they will need to be recompiled.

–model-small Generate code for Small Model programs see section Memory Models for more details. This is the default model.

3.2.5 DS390 Options

–model-flat24 Generate 24-bit flat mode code. This is the one and only that the ds390 code generator supports right now and is default when using *-mds390*. See section Memory Models for more details.

–stack-10bit Generate code for the 10 bit stack mode of the Dallas DS80C390 part. This is the one and only that the ds390 code generator supports right now and is default when using *-mds390*. In this mode, the stack is located in the lower 1K of the internal RAM, which is mapped to 0x400000. Note that the support is incomplete, since it still uses a single byte as the stack pointer. This means that only the lower 256 bytes of the potential 1K stack space will actually be used. However, this does allow you to reclaim the precious 256 bytes of low RAM for use for the DATA and IDATA segments. The compiler will not generate any code to put the processor into 10 bit stack mode. It is important to ensure that the processor is in this mode before calling any re-entrant functions compiled with this option. In principle, this should work with the *–stack-auto* option, but that has not been tested. It is incompatible with the *–xstack* option. It also only makes sense if the processor is in 24 bit contiguous addressing mode (see the *–model-flat24 option*).

3.2.6 Optimization Options

–nogcse Will not do global subexpression elimination, this option may be used when the compiler creates undesirably large stack/data spaces to store compiler temporaries. A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It recommended that this option NOT be used, #pragma NOGCSE can be used to turn off global subexpression elimination for a given function only.

–noinvariant Will not do loop invariant optimizations, this may be turned off for reasons explained for the previous option. For more details of loop optimizations performed see section Loop Invariants.It recommended that this option NOT be used, #pragma NOINVARIANT can be used to turn off invariant optimizations for a given function only.

–noinduction Will not do loop induction optimizations, see section strength reduction for more details.It is recommended that this option is NOT used, #pragma NOINDUCTION can be used to turn off induction optimizations for a given function only.

–nojtbound Will not generate boundary condition check when switch statements are implemented using jump-tables. See section Switch Statements for more details. It is recommended that this option is NOT used, #pragma NOJTBOUND can be used to turn off boundary checking for jump tables for a given function only.

–noloopreverse Will not do loop reversal optimization.

3.2.7 Other Options

-c –compile-only will compile and assemble the source, but will not call the linkage editor.

-E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

–stack-auto All functions in the source file will be compiled as *reentrant*, i.e. the parameters and local variables will be allocated on the stack. see section Parameters and Local Variables for more details. If this option is used all source files in the project should be compiled with this option.

–xstack Uses a pseudo stack in the first 256 bytes in the external ram for allocating variables and passing parameters. See section on external stack for more details.

–callee-saves function1[,function2][,function3].... The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unneccessary register pushing & popping when calling small functions from larger functions. This option can be used to switch the register saving convention for the function names specified. The compiler will not save registers when calling these functions, no extra code will be generated at the entry & exit for these functions to save & restore the registers used by these functions, this can SUBSTANTIALLY reduce code & improve run time performance of the generated code. In the future the compiler (with interprocedural analysis) will be able to determine the appropriate scheme to use for each function call. DO NOT use this option for built-in functions such as _muluint..., if this option is used for a library function the appropriate library function needs to be recompiled with the same option. If the project consists of multiple source files then all the source file should be compiled with the same –callee-saves option string. Also see #pragma CALLEE-SAVES.

–debug When this option is used the compiler will generate debug information, that can be used with the SDCDB. The debug information is collected in a file with .cdb extension. For more information see documentation for SDCDB.

–regextend *This option is obsolete and isn't supported anymore.*

–noregparms *This option is obsolete and isn't supported anymore.*

–peep-file<filename> This option can be used to use additional rules to be used by the peep hole optimizer. See section Peep Hole optimizations for details on how to write these rules.

-S Stop after the stage of compilation proper; do not assemble. The output is an assembler code file for the input file specified.

-Wa_asmOption[,asmOption]... Pass the asmOption to the assembler.

-Wl_linkOption[,linkOption]... Pass the linkOption to the linker.

–int-long-reent Integer (16 bit) and long (32 bit) libraries have been compiled as reentrant. Note by default these libraries are compiled as non-reentrant. See section Installation for more details.

–cyclomatic This option will cause the compiler to generate an information message for each function in the source file. The message contains some *important* information about the function. The number of edges and nodes the compiler detected in the control flow graph of the function, and most importantly the *cyclomatic complexity* see section on Cyclomatic Complexity for more details.

–float-reent Floating point library is compiled as reentrant.See section Installation for more details.

–nooverlay The compiler will not overlay parameters and local variables of any function, see section Parameters and local variables for more details.

–main-return This option can be used when the code generated is called by a monitor program. The compiler will generate a 'ret' upon return from the 'main' function. The default option is to lock up i.e. generate a 'ljmp '.

–no-peep Disable peep-hole optimization.

–peep-asm Pass the inline assembler code through the peep hole optimizer. This can cause unexpected changes to inline assembler code, please go through the peephole optimizer rules defined in the source file tree '<target>/peeph.def' before using this option.

–iram-size<Value> Causes the linker to check if the interal ram usage is within limits of the given value.

–nostdincl This will prevent the compiler from passing on the default include path to the preprocessor.

–nostdlib This will prevent the compiler from passing on the default library path to the linker.

–verbose Shows the various actions the compiler is performing.

-V Shows the actual commands the compiler is executing.

3.2.8 Intermediate Dump Options

The following options are provided for the purpose of retargetting and debugging the compiler. These provided a means to dump the intermediate code (iCode) generated by the compiler in human readable form at various stages of the compilation process.

–dumpraw This option will cause the compiler to dump the intermediate code into a file of named *<source filename>.dumpraw* just after the intermediate code has been generated for a function, i.e. before any optimizations are done.

The basic blocks at this stage ordered in the depth first number, so they may not be in sequence of execution.

–dumpgcse Will create a dump of iCode's, after global subexpression elimination, into a file named *<source filename>.dumpgcse.*

–dumpdeadcode Will create a dump of iCode's, after deadcode elimination, into a file named *<source filename>.dumpdeadcode.*

–dumploop Will create a dump of iCode's, after loop optimizations, into a file named *<source filename>.dumploop.*

–dumprange Will create a dump of iCode's, after live range analysis, into a file named *<source filename>.dumprange.*

–dumlrange Will dump the life ranges for all symbols.

–dumpregassign Will create a dump of iCode's, after register assignment, into a file named *<source filename>.dumprassgn.*

–dumplrange Will create a dump of the live ranges of iTemp's

–dumpall Will cause all the above mentioned dumps to be created.

3.3 MCS51/DS390 Storage Class Language Extensions

In addition to the ANSI storage classes SDCC allows the following MCS51 specific storage classes.

3.3.1 xdata

Variables declared with this storage class will be placed in the extern RAM. This is the **default** storage class for Large Memory model, e.g.:

```
xdata unsigned char xduc;
```

3.3.2 data

This is the **default** storage class for Small Memory model. Variables declared with this storage class will be allocated in the internal RAM, e.g.:

```
data int iramdata;
```

3.3.3 idata

Variables declared with this storage class will be allocated into the indirectly addressable portion of the internal ram of a 8051, e.g.:

```
idata int idi;
```

3.3.4 bit

This is a data-type and a storage class specifier. When a variable is declared as a bit, it is allocated into the bit addressable memory of 8051, e.g.:

```
bit iFlag;
```

3.3.5 sfr / sbit

Like the bit keyword, *sfr / sbit* signifies both a data-type and storage class, they are used to describe the special function registers and special bit variables of a 8051, eg:

```
sfr at 0x80 P0; /* special function register P0 at location 0x80 */
sbit at 0xd7 CY; /* CY (Carry Flag) */
```

3.4 Pointers

SDCC allows (via language extensions) pointers to explicitly point to any of the memory spaces of the 8051. In addition to the explicit pointers, the compiler also allows a *_generic* class of pointers which can be used to point to any of the memory spaces.

Pointer declaration examples:

```
/* pointer physically in xternal ram pointing to object in internal ram
*/
data unsigned char * xdata p;

/* pointer physically in code rom pointing to data in xdata space */

xdata unsigned char * code p;

/* pointer physically in code space pointing to data in code space */

code unsigned char * code p;

/* the folowing is a generic pointer physically located in xdata space
*/
char * xdata p;
```

Well you get the idea.

For compatibility with the previous version of the compiler, the following syntax for pointer declaration is still supported but will disappear int the near future.

```
unsigned char _xdata *ucxdp; /* pointer to data in external
ram */
```

```
unsigned char _data  *ucdp ; /* pointer to data in internal
ram */
unsigned char _code  *uccp ; /* pointer to data in R/O code
space */
unsigned char _idata *uccp;  /* pointer to upper 128 bytes
of ram */
```

All unqualified pointers are treated as 3-byte (4-byte for the ds390) *generic* pointers. These type of pointers can also to be explicitly declared.

```
unsigned char _generic *ucgp;
```

The highest order byte of the *generic* pointers contains the data space information. Assembler support routines are called whenever data is stored or retrieved using *generic* pointers. These are useful for developing reusable library routines. Explicitly specifying the pointer type will generate the most efficient code. Pointers declared using a mixture of OLD and NEW style could have unpredictable results.

3.5 Parameters & Local Variables

Automatic (local) variables and parameters to functions can either be placed on the stack or in data-space. The default action of the compiler is to place these variables in the internal RAM (for small model) or external RAM (for Large model). This in fact makes them *static* so by default functions are non-reentrant.

They can be placed on the stack either by using the *–stack-auto* compiler option or by using the *reentrant* keyword in the function declaration, e.g.:

```
unsigned char foo(char i) reentrant
{
...
}
```

Since stack space on 8051 is limited, the *reentrant* keyword or the *–stack-auto* option should be used sparingly. Note that the reentrant keyword just means that the parameters & local variables will be allocated to the stack, it *does not* mean that the function is register bank independent.

Local variables can be assigned storage classes and absolute addresses, e.g.:

```
unsigned char foo() {
xdata unsigned char i;
bit bvar;
data at 0x31 unsiged char j;
...
}
```

In the above example the variable *i* will be allocated in the external ram, *bvar* in bit addressable space and *j* in internal ram. When compiled with *–stack-auto* or when a function is declared as *reentrant* this can only be done for static variables.

Parameters however are not allowed any storage class, (storage classes for parameters will be ignored), their allocation is governed by the memory model in use, and the reentrancy options.

3.6 Overlaying

For non-reentrant functions SDCC will try to reduce internal ram space usage by overlaying parameters and local variables of a function (if possible). Parameters and local variables of a function will be allocated to an overlayable segment if the function has *no other function calls and the function is non-reentrant and the memory model is small.* If an explicit storage class is specified for a local variable, it will NOT be overlayed.

Note that the compiler (not the linkage editor) makes the decision for overlaying the data items. Functions that are called from an interrupt service routine should be preceded by a #pragma NOOVERLAY if they are not reentrant.

Also note that the compiler does not do any processing of inline assembler code, so the compiler might incorrectly assign local variables and parameters of a function into the overlay segment if the inline assembler code calls other c-functions that might use the overlay. In that case the #pragma NOOVERLAY should be used.

Parameters and Local variables of functions that contain 16 or 32 bit multiplication or division will NOT be overlayed since these are implemented using external functions, e.g.:

```
#pragma SAVE
#pragma NOOVERLAY
void set_error(unsigned char errcd)
{
P3 = errcd;
}
#pragma RESTORE

void some_isr () interrupt 2 using 1
{
...
set_error(10);
...
}
```

In the above example the parameter *errcd* for the function *set_error* would be assigned to the overlayable segment if the #pragma NOOVERLAY was not present, this could cause unpredictable runtime behavior when called from an ISR. The #pragma NOOVERLAY ensures that the parameters and local variables for the function are NOT overlayed.

3.7 Interrupt Service Routines

SDCC allows interrupt service routines to be coded in C, with some extended keywords.

```
void timer_isr (void) interrupt 2 using 1
{
..
}
```

The number following the *interrupt* keyword is the interrupt number this routine will service. The compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. The *using* keyword is used to tell the compiler to use the specified register bank (8051 specific) when generating code for this function. Note that when some function is called from an interrupt service routine it should be preceded by a #pragma NOOVERLAY if it is not reentrant. A special note here, int (16 bit) and long (32 bit) integer division, multiplication & modulus operations are implemented using external support routines developed in ANSI-C, if an interrupt service routine needs to do any of these operations then the support routines (as mentioned in a following section) will have to be recompiled using the *–stack-auto* option and the source file will need to be compiled using the *–int-long-ren*t compiler option.

If you have multiple source files in your project, interrupt service routines can be present in any of them, but a prototype of the isr MUST be present or included in the file that contains the function *main*.

Interrupt Numbers and the corresponding address & descriptions for the Standard 8051 are listed below. SDCC will automatically adjust the interrupt vector table to the maximum interrupt number specified.

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000B
2	External 1	0x0013
3	Timer 1	0x001B
4	Serial	0x0023

If the interrupt service routine is defined without *using* a register bank or with register bank 0 (using 0), the compiler will save the registers used by itself on the stack upon entry and restore them at exit, however if such an interrupt service routine calls another function then the entire register bank will be saved on the stack. This scheme may be advantageous for small interrupt service routines which have low register usage.

If the interrupt service routine is defined to be using a specific register bank then only *a, b & dptr* are save and restored, if such an interrupt service routine calls another function (using another register bank) then the entire register bank of the called function will be saved on the stack. This scheme is recommended for larger interrupt service routines.

Calling other functions from an interrupt service routine is not recommended, avoid it if possible.

Also see the _naked modifier.

3.8 Critical Functions

A special keyword may be associated with a function declaring it as *critical*. SDCC will generate code to disable all interrupts upon entry to a critical function and enable them back before returning. Note that nesting critical functions may cause unpredictable results.

```
int foo () critical
{
...
...
}
```

The critical attribute maybe used with other attributes like *reentrant.*

3.9 Naked Functions

A special keyword may be associated with a function declaring it as *_naked.* The *_naked* function modifier attribute prevents the compiler from generating prologue and epilogue code for that function. This means that the user is entirely responsible for such things as saving any registers that may need to be preserved, selecting the proper register bank, generating the *return* instruction at the end, etc. Practically, this means that the contents of the function must be written in inline assembler. This is particularly useful for interrupt functions, which can have a large (and often unnecessary) prologue/epilogue. For example, compare the code generated by these two functions:

```
data unsigned char counter;
void simpleInterrupt(void) interrupt 1
{
counter++;
}

void nakedInterrupt(void) interrupt 2 _naked
{
_asm
inc     _counter
reti    ; MUST explicitly include ret in _naked function.
_endasm;
}
```

For an 8051 target, the generated simpleInterrupt looks like:

```
_simpleIterrupt:
push    acc
push    b
push    dpl
push    dph
push    psw
mov     psw,#0x00
inc     _counter
pop     psw
pop     dph
pop     dpl
pop     b
pop     acc
reti
```

whereas nakedInterrupt looks like:

```
_nakedInterrupt:
inc    _counter
reti   ; MUST explicitly include ret(i) in _naked function.
```

While there is nothing preventing you from writing C code inside a _naked function, there are many ways to shoot yourself in the foot doing this, and is is recommended that you stick to inline assembler.

3.10 Functions using private banks

The *using* attribute (which tells the compiler to use a register bank other than the default bank zero) should only be applied to *interrupt* functions (see note 1 below). This will in most circumstances make the generated ISR code more efficient since it will not have to save registers on the stack.

The *using* attribute will have no effect on the generated code for a *non-interrupt* function (but may occasionally be useful anyway[1]).
(pending: I don't think this has been done yet)

An *interrupt* function using a non-zero bank will assume that it can trash that register bank, and will not save it. Since high-priority interrupts can interrupt low-priority ones on the 8051 and friends, this means that if a high-priority ISR *using* a particular bank occurs while processing a low-priority ISR *using* the same bank, terrible and bad things can happen. To prevent this, no single register bank should be *used* by both a high priority and a low priority ISR. This is probably most easily done by having all high priority ISRs use one bank and all low priority ISRs use another. If you have an

[1]possible exception: if a function is called ONLY from 'interrupt' functions using a particular bank, it can be declared with the same 'using' attribute as the calling 'interrupt' functions. For instance, if you have several ISRs using bank one, and all of them call memcpy(), it might make sense to create a specialized version of memcpy() 'using 1', since this would prevent the ISR from having to save bank zero to the stack on entry and switch to bank zero before calling the function

ISR which can change priority at runtime, you're on your own: I suggest using the default bank zero and taking the small performance hit.

It is most efficient if your ISR calls no other functions. If your ISR must call other functions, it is most efficient if those functions use the same bank as the ISR (see note 1 below); the next best is if the called functions use bank zero. It is very inefficient to call a function using a different, non-zero bank from an ISR.

3.11 Absolute Addressing

Data items can be assigned an absolute address with the *at <address>* keyword, in addition to a storage class, e.g.:

```
xdata at 0x8000 unsigned char PORTA_8255 ;
```

In the above example the PORTA_8255 will be allocated to the location 0x8000 of the external ram. Note that this feature is provided to give the programmer access to *memory mapped* devices attached to the controller. The compiler does not actually reserve any space for variables declared in this way (they are implemented with an equate in the assembler). Thus it is left to the programmer to make sure there are no overlaps with other variables that are declared without the absolute address. The assembler listing file (.lst) and the linker output files (.rst) and (.map) are a good places to look for such overlaps.

Absolute address can be specified for variables in all storage classes, e.g.:

```
bit at 0x02 bvar;
```

The above example will allocate the variable at offset 0x02 in the bit-addressable space. There is no real advantage to assigning absolute addresses to variables in this manner, unless you want strict control over all the variables allocated.

3.12 Startup Code

The compiler inserts a call to the C routine *_sdcc__external__startup()* at the start of the CODE area. This routine is in the runtime library. By default this routine returns 0, if this routine returns a non-zero value, the static & global variable initialization will be skipped and the function main will be invoked Other wise static & global variables will be initialized before the function main is invoked. You could add a *_sdcc__external__startup()* routine to your program to override the default if you need to setup hardware or perform some other critical operation prior to static & global variable initialization.

3.13 Inline Assembler Code

SDCC allows the use of in-line assembler with a few restriction as regards labels. All labels defined within inline assembler code *has to be* of the form *nnnnn$* where nnnn is a number less than 100 (which implies a limit of utmost 100 inline assembler labels *per function*). It is strongly recommended that each assembly instruction (including labels) be placed in a separate line (as the example shows). When the *–peep-asm* command line option is used, the inline assembler code will be passed through the peephole optimizer. This might cause some unexpected changes in the inline assembler code. Please go throught the peephole optimizer rules defined in file *SDCCpeeph.def* carefully before using this option.

```
_asm
mov     b,#10
00001$:
djnz    b,00001$
_endasm ;
```

The inline assembler code can contain any valid code understood by the assembler, this includes any assembler directives and comment lines. The compiler does not do any validation of the code within the `_asm ... _endasm;` keyword pair.

Inline assembler code cannot reference any C-Labels, however it can reference labels defined by the inline assembler, e.g.:

```
foo() {
/* some c code */
_asm
; some assembler code
ljmp $0003
_endasm;
/* some more c code */
clabel:  /* inline assembler cannot reference this label */
_asm
$0003:  ;label (can be reference by inline assembler only)
_endasm ;
/* some more c code */
}
```

In other words inline assembly code can access labels defined in inline assembly within the scope of the funtion.

The same goes the other way, ie. labels defines in inline assembly CANNOT be accessed by C statements.

3.14 int(16 bit) and long (32 bit) Support

For signed & unsigned int (16 bit) and long (32 bit) variables, division, multiplication and modulus operations are implemented by support routines. These support routines are all developed in ANSI-C to facilitate porting to other MCUs, although some model specific assembler optimations are used. The following files contain the described routine, all of them can be found in <installdir>/share/sdcc/lib.

<pending: tabularise this>

_mulsint.c - signed 16 bit multiplication (calls _muluint)
_muluint.c - unsigned 16 bit multiplication
_divsint.c - signed 16 bit division (calls _divuint)
_divuint.c - unsigned 16 bit division
_modsint.c - signed 16 bit modulus (call _moduint)
_moduint.c - unsigned 16 bit modulus
_mulslong.c - signed 32 bit multiplication (calls _mululong)
_mululong.c - unsigned32 bit multiplication
_divslong.c - signed 32 division (calls _divulong)
_divulong.c - unsigned 32 division
_modslong.c - signed 32 bit modulus (calls _modulong)
_modulong.c - unsigned 32 bit modulus

Since they are compiled as *non-reentrant*, interrupt service routines should not do any of the above operations. If this is unavoidable then the above routines will need to be compiled with the *–stack-auto* option, after which the source program will have to be compiled with *–int-long-rent* option.

3.15 Floating Point Support

SDCC supports IEEE (single precision 4bytes) floating point numbers.The floating point support routines are derived from gcc's floatlib.c and consists of the following routines:

<pending: tabularise this>

_fsadd.c - add floating point numbers
_fssub.c - subtract floating point numbers
_fsdiv.c - divide floating point numbers
_fsmul.c - multiply floating point numbers
_fs2uchar.c - convert floating point to unsigned char
_fs2char.c - convert floating point to signed char
_fs2uint.c - convert floating point to unsigned int
_fs2int.c - convert floating point to signed int
_fs2ulong.c - convert floating point to unsigned long
_fs2long.c - convert floating point to signed long
_uchar2fs.c - convert unsigned char to floating point
_char2fs.c - convert char to floating point number
_uint2fs.c - convert unsigned int to floating point
_int2fs.c - convert int to floating point numbers
_ulong2fs.c - convert unsigned long to floating point number
_long2fs.c - convert long to floating point number

Note if all these routines are used simultaneously the data space might overflow. For serious floating point usage it is strongly recommended that the large model be used.

3.16 MCS51 Memory Models

SDCC allows two memory models for MCS51 code, small and large. Modules compiled with different memory models should *never* be combined together or the results would be unpredictable. The library routines supplied with the compiler are compiled as both small and large. The compiled library modules are contained in seperate directories as small and large so that you can link to either set.

When the large model is used all variables declared without a storage class will be allocated into the external ram, this includes all parameters and local variables (for non-reentrant functions). When the small model is used variables without storage class are allocated in the internal ram.

Judicious usage of the processor specific storage classes and the 'reentrant' function type will yield much more efficient code, than using the large model. Several optimizations are disabled when the program is compiled using the large model, it is therefore strongly recommdended that the small model be used unless absolutely required.

3.17 DS390 Memory Models

The only model supported is Flat 24. This generates code for the 24 bit contiguous addressing mode of the Dallas DS80C390 part. In this mode, up to four meg of external RAM or code space can be directly addressed. See the data sheets at www.dalsemi.com for further information on this part.

In older versions of the compiler, this option was used with the MCS51 code generator (*-mmcs51*). Now, however, the '390 has it's own code generator, selected by the *-mds390* switch.

Note that the compiler does not generate any code to place the processor into 24 bit-mode (although *tinibios* in the ds390 libraries will do that for you). If you don't use *tinibios*, the boot loader or similar code must ensure that the processor is in 24 bit contiguous addressing mode before calling the SDCC startup code.

Like the *–model-large* option, variables will by default be placed into the XDATA segment.

Segments may be placed anywhere in the 4 meg address space using the usual –*-loc options. Note that if any segments are located above 64K, the -r flag must be passed to the linker to generate the proper segment relocations, and the Intel HEX output format must be used. The -r flag can be passed to the linker by using the option *-Wl-r* on the sdcc command line. However, currently the linker can not handle code segments > 64k.

3.18 Defines Created by the Compiler

The compiler creates the following #defines.

- SDCC - this Symbol is always defined.
- SDCC_mcs51 or SDCC_ds390 or SDCC_z80, etc - depending on the model used (e.g.: -mds390)
- __mcs51 or __ds390 or __z80, etc - depending on the model used (e.g. -mz80)
- SDCC_STACK_AUTO - this symbol is defined when *–stack-auto* option is used.
- SDCC_MODEL_SMALL - when *–model-small* is used.
- SDCC_MODEL_LARGE - when *–model-large* is used.
- SDCC_USE_XSTACK - when *–xstack* option is used.
- SDCC_STACK_TENBIT - when *-mds390* is used
- SDCC_MODEL_FLAT24 - when *-mds390* is used

4 SDCC Technical Data

4.1 Optimizations

SDCC performs a host of standard optimizations in addition to some MCU specific optimizations.

4.1.1 Sub-expression Elimination

The compiler does local and global common subexpression elimination, e.g.:

```
i = x + y + 1;
j = x + y;
```

will be translated to

```
iTemp = x + y
i = iTemp + 1
j = iTemp
```

Some subexpressions are not as obvious as the above example, e.g.:

```
a->b[i].c = 10;
a->b[i].d = 11;
```

In this case the address arithmetic a->b[i] will be computed only once; the equivalent code in C would be.

```
iTemp = a->b[i];
iTemp.c = 10;
iTemp.d = 11;
```

The compiler will try to keep these temporary variables in registers.

4.1.2 Dead-Code Elimination

```
int global;
void f () {
int i;
i = 1;  /* dead store */
global = 1; /* dead store */
global = 2;
return;
global = 3; /* unreachable */
}
```

will be changed to

```
int global; void f ()
{
global = 2;
return;
}
```

4.1.3 Copy-Propagation

```
int f() {
int i, j;
i = 10;
j = i;
return j;
}
```

will be changed to

```
int f() {
int i,j;
i = 10;
j = 10;
return 10;
}
```

Note: the dead stores created by this copy propagation will be eliminated by dead-code elimination.

4.1.4 Loop Optimizations

Two types of loop optimizations are done by SDCC loop invariant lifting and strength reduction of loop induction variables. In addition to the strength reduction the optimizer marks the induction variables and the register allocator tries to keep the induction variables in registers for the duration of the loop. Because of this preference of the register allocator, loop induction optimization causes an increase in register pressure, which may cause unwanted spilling of other temporary variables into the stack / data space. The compiler will generate a warning message when it is forced to allocate extra space either on the stack or data space. If this extra space allocation is undesirable then induction optimization can be eliminated either for the entire source file (with –noinduction option) or for a given function only using #pragma NOINDUCTION.

Loop Invariant:

```
for (i = 0 ; i < 100 ; i ++)
f += k + l;
```

changed to

```
itemp = k + l;
for (i = 0; i < 100; i++)
f += itemp;
```

As mentioned previously some loop invariants are not as apparent, all static address computations are also moved out of the loop.

Strength Reduction, this optimization substitutes an expression by a cheaper expression:

```
for (i=0;i < 100; i++)
ar[i*5] = i*3;
```

changed to

```
itemp1 = 0;
itemp2 = 0;
for (i=0;i< 100;i++) {
ar[itemp1] = itemp2;
itemp1 += 5;
itemp2 += 3;
}
```

The more expensive multiplication is changed to a less expensive addition.

4.1.5 Loop Reversing

This optimization is done to reduce the overhead of checking loop boundaries for every iteration. Some simple loops can be reversed and implemented using a “decrement and jump if not zero” instruction. SDCC checks for the following criterion to determine if a loop is reversible (note: more sophisticated compilers use data-dependency analysis to make this determination, SDCC uses a more simple minded analysis).

- The ’for’ loop is of the form

  ```
  for (<symbol> = <expression> ; <sym> [< | <=] <expression> ; [<sym>++
  | <sym> += 1])
  <for body>
  ```

- The <for body> does not contain “continue” or ’break”.
- All goto’s are contained within the loop.
- No function calls within the loop.
- The loop control variable <sym> is not assigned any value within the loop
- The loop control variable does NOT participate in any arithmetic operation within the loop.
- There are NO switch statements in the loop.

Note djnz instruction can be used for 8-bit values *only*, therefore it is advantageous to declare loop control symbols as *char*. Ofcourse this may not be possible on all situations.

4.1.6 Algebraic Simplifications

SDCC does numerous algebraic simplifications, the following is a small sub-set of these optimizations.

```
i = j + 0 ; /* changed to */ i = j;
i /= 2; /* changed to */ i >>= 1;
i = j - j ; /* changed to */ i = 0;
i = j / 1 ; /* changed to */ i = j;
```

Note the subexpressions given above are generally introduced by macro expansions or as a result of copy/constant propagation.

4.1.7 'switch' Statements

SDCC changes switch statements to jump tables when the following conditions are true.

- The case labels are in numerical sequence, the labels need not be in order, and the starting number need not be one or zero.

```
switch(i) {                         switch (i) {
case 4:...                                  case 1:  ...
case 5:...                                  case 2:  ...
case 3:...                                  case 3:  ...
case 6:...                                  case 4:  ...
}                                   }
```

  Both the above switch statements will be implemented using a jump-table.

- The number of case labels is at least three, since it takes two conditional statements to handle the boundary conditions.
- The number of case labels is less than 84, since each label takes 3 bytes and a jump-table can be utmost 256 bytes long.

Switch statements which have gaps in the numeric sequence or those that have more that 84 case labels can be split into more than one switch statement for efficient code generation, e.g.:

```
switch (i) {
case 1:  ...
case 2:  ...
case 3:  ...
case 4:  ...
case 9:  ...
case 10:  ...
case 11:  ...
case 12:  ...
}
```

If the above switch statement is broken down into two switch statements

```
switch (i) {
case 1:  ...
case 2:  ...
case 3:  ...
case 4:  ...
}
```

and

```
switch (i) {
case 9:   ...
case 10:  ...
case 11:  ...
case 12: ...
}
```

then both the switch statements will be implemented using jump-tables whereas the unmodified switch statement will not be.

4.1.8 Bit-shifting Operations.

Bit shifting is one of the most frequently used operation in embedded programming. SDCC tries to implement bit-shift operations in the most efficient way possible, e.g.:

```
unsigned char i;
...
i>>= 4;
...
```

generates the following code:

```
mov a,_i
swap a
anl a,#0x0f
mov _i,a
```

In general SDCC will never setup a loop if the shift count is known. Another example:

```
unsigned int i;
...
i >>= 9;
...
```

will generate:

```
mov a,(_i + 1)
mov (_i + 1),#0x00
clr c
rrc a
mov _i,a
```

Note that SDCC stores numbers in little-endian format (i.e. lowest order first).

4.1.9 Bit-rotation

A special case of the bit-shift operation is bit rotation, SDCC recognizes the following expression to be a left bit-rotation:

```
unsigned char i;
...
i = ((i << 1) | (i >> 7));
...
```

will generate the following code:

```
mov a,_i
rl a
mov _i,a
```

SDCC uses pattern matching on the parse tree to determine this operation.Variations of this case will also be recognized as bit-rotation, i.e.:

```
i = ((i >> 7) | (i << 1)); /* left-bit rotation */
```

4.1.10 Highest Order Bit

It is frequently required to obtain the highest order bit of an integral type (long, int, short or char types). SDCC recognizes the following expression to yield the highest order bit and generates optimized code for it, e.g.:

```
unsigned int gint;

foo () {
unsigned char hob;
...
hob = (gint >> 15) & 1;
..
}
```

will generate the following code:

```
61 ;  hob.c 7
000A E5*01              62          mov  a,(_gint + 1)
000C 33                 63          rlc  a
000D E4                 64          clr  a
000E 13                 65          rrc  a
000F F5*02              66          mov  _foo_hob_1_1,a
```

Variations of this case however will *not* be recognized. It is a standard C expression, so

I heartily recommend this be the only way to get the highest order bit, (it is portable). Of course it will be recognized even if it is embedded in other expressions, e.g.:

```
xyz = gint + ((gint >> 15) & 1);
```

will still be recognized.

4.1.11 Peep-hole Optimizer

The compiler uses a rule based, pattern matching and re-writing mechanism for peep-hole optimization. It is inspired by *copt* a peep-hole optimizer by Christopher W. Fraser (cwfraser@microsoft.com). A default set of rules are compiled into the compiler, additional rules may be added with the *–peep-file <filename>* option. The rule language is best illustrated with examples.

```
replace {
mov %1,a
mov a,%1
} by {
mov %1,a
}
```

The above rule will change the following assembly sequence:

```
mov r1,a
mov a,r1
```

to

```
mov r1,a
```

Note: All occurrences of a *%n* (pattern variable) must denote the same string. With the above rule, the assembly sequence:

```
mov r1,a
mov a,r2
```

will remain unmodified.

Other special case optimizations may be added by the user (via *–peep-file option*). E.g. some variants of the 8051 MCU allow only `ajmp` and `acall`. The following two rules will change all `ljmp` and `lcall` to `ajmp` and `acall`

```
replace { lcall %1 } by { acall %1 }
replace { ljmp %1 } by { ajmp %1 }
```

The *inline-assembler code* is also passed through the peep hole optimizer, thus the peephole optimizer can also be used as an assembly level macro expander. The rules themselves are MCU dependent whereas the rule language infra-structure is MCU independent. Peephole optimization rules for other MCU can be easily programmed using the rule language.

The syntax for a rule is as follows:

```
rule := replace [ restart ] '{' <assembly sequence> '\n'
'}' by '{' '\n'
<assembly sequence> '\n'
'}' [if <functionName> ] '\n'
```

<assembly sequence> := assembly instruction (each instruction including labels must be on a separate line).

The optimizer will apply to the rules one by one from the top in the sequence of their appearance, it will terminate when all rules are exhausted. If the 'restart' option is specified, then the optimizer will start matching the rules again from the top, this option for a rule is expensive (performance), it is intended to be used in situations where a transformation will trigger the same rule again. A good example of this the following rule:

```
replace restart {
pop %1
push %1 } by {
; nop
}
```

Note that the replace pattern cannot be a blank, but can be a comment line. Without the 'restart' option only the inner most 'pop' 'push' pair would be eliminated, i.e.:

```
pop ar1
pop ar2
push ar2
push ar1
```

would result in:

```
pop ar1
; nop
push ar1
```

with the restart option the rule will be applied again to the resulting code and then all the pop-push pairs will be eliminated to yield:

```
; nop
; nop
```

A conditional function can be attached to a rule. Attaching rules are somewhat more involved, let me illustrate this with an example.

```
replace {
ljmp %5
%2:
} by {
sjmp %5
%2:
} if labelInRange
```

The optimizer does a look-up of a function name table defined in function *callFuncByName* in the source file SDCCpeeph.c, with the name *labelInRange*. If it finds a corresponding entry the function is called. Note there can be no parameters specified for these functions, in this case the use of *%5* is crucial, since the function *labelInRange* expects to find the label in that particular variable (the hash table containing the variable bindings is passed as a parameter). If you want to code more such functions, take a close look at the function labelInRange and the calling mechanism in source file SDCCpeeph.c. I know this whole thing is a little kludgey, but maybe some day we will have some better means. If you are looking at this file, you will also see the default rules that are compiled into the compiler, you can add your own rules in the default set there if you get tired of specifying the –peep-file option.

4.2 Pragmas

SDCC supports the following #pragma directives. This directives are applicable only at a function level.

- SAVE - this will save all the current options.
- RESTORE - will restore the saved options from the last save. Note that SAVES & RESTOREs cannot be nested. SDCC uses the same buffer to save the options each time a SAVE is called.
- NOGCSE - will stop global subexpression elimination.
- NOINDUCTION - will stop loop induction optimizations.
- NOJTBOUND - will not generate code for boundary value checking, when switch statements are turned into jump-tables.
- NOOVERLAY - the compiler will not overlay the parameters and local variables of a function.
- NOLOOPREVERSE - Will not do loop reversal optimization

- EXCLUDE NONE | {acc[,b[,dpl[,dph]]] - The exclude pragma disables generation of pair of push/pop instruction in ISR function (using interrupt keyword). The directive should be placed immediately before the ISR function definition and it affects ALL ISR functions following it. To enable the normal register saving for ISR functions use #pragma EXCLUDE none.
- CALLEE-SAVES function1[,function2[,function3...]] - The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unneccessary register pushing & popping when calling small functions from larger functions. This option can be used to switch the register saving convention for the function names specified. The compiler will not save registers when calling these functions, extra code will be generated at the entry & exit for these functions to save & restore the registers used by these functions, this can SUBSTANTIALLY reduce code & improve run time performance of the generated code. In future the compiler (with interprocedural analysis) will be able to determine the appropriate scheme to use for each function call. If –callee-saves command line option is used, the function names specified in #pragma CALLEE-SAVES is appended to the list of functions specified inthe command line.

The pragma's are intended to be used to turn-off certain optimizations which might cause the compiler to generate extra stack / data space to store compiler generated temporary variables. This usually happens in large functions. Pragma directives should be used as shown in the following example, they are used to control options & optimizations for a given function; pragmas should be placed before and/or after a function, placing pragma's inside a function body could have unpredictable results.

```
#pragma SAVE /* save the current settings */
#pragma NOGCSE /* turnoff global subexpression elimination */
#pragma NOINDUCTION /* turn off induction optimizations */
int foo ()
{
...
/* large code */
...
}
#pragma RESTORE /* turn the optimizations back on */
```

The compiler will generate a warning message when extra space is allocated. It is strongly recommended that the SAVE and RESTORE pragma's be used when changing options for a function.

4.3 *<pending: this is messy and incomplete>* Library Routines

The following library routines are provided for your convenience.

stdio.h - Contains the following functions printf & sprintf these routines are developed by Martijn van Balen <balen@natlab.research.philips.com>.

%[flags][width][b|B|l|L]type

flags: - left justify output in specified field width
+ prefix output with +/- sign if output is signed type
space prefix output with a blank if it's a signed positive value
width: specifies minimum number of characters outputted for numbers
or strings.
- For numbers, spaces are added on the left when needed.
If width starts with a zero character, zeroes and used
instead of spaces.
- For strings, spaces are are added on the left or right (when
flag '-' is used) when needed.

b/B: byte argument (used by d, u, o, x, X)
l/L: long argument (used by d, u, o, x, X)
type: d decimal number
u unsigned decimal number
o unsigned octal number
x unsigned hexadecimal number (0-9, a-f)
X unsigned hexadecimal number (0-9, A-F)
c character
s string (generic pointer)
p generic pointer (I:data/idata, C:code, X:xdata, P:paged)
f float (still to be implemented)

Also contains a very simple version of printf (printf_small). This simplified version of printf supports only the following formats.

format	output type	argument-type
%d	decimal	short/int
%ld	decimal	long
%hd	decimal	char
%x	hexadecimal	short/int
%lx	hexadecimal	long
%hx	hexadecimal	char
%o	octal	short/int
%lo	octal	long
%ho	octal	char
%c	character	char
%s	character	_generic pointer

The routine is very stack intesive, –stack-after-data parameter should be used when using this routine, the routine also takes about 1K of code space. It also expects an external function named putchar(char) to be present (this can be changed). When using the %s format the string / pointer should be cast to a generic pointer. eg.

printf_small("my str %s, my int %d\n",(char _generic *)mystr,myint);

- stdarg.h - contains definition for the following macros to be used for variable parameter list, note that a function can have a variable parameter list if and only if it is 'reentrant' va_list, va_start, va_arg, va_end.

- setjmp.h - contains defintion for ANSI setjmp & longjmp routines. Note in this case setjmp & longjmp can be used between functions executing within the same register bank, if long jmp is executed from a function that is using a different register bank from the function issuing the setjmp function, the results may be unpredictable. The jump buffer requires 3 bytes of data (the stack pointer & a 16 byte return address), and can be placed in any address space.
- stdlib.h - contains the following functions.

  atoi, atol.
- string.h - contains the following functions.

  strcpy, strncpy, strcat, strncat, strcmp, strncmp, strchr, strrchr, strspn, strcspn, strpbrk, strstr, strlen, strtok, memcpy, memcmp, memset.
- ctype.h - contains the following routines.

  iscntrl, isdigit, isgraph, islower, isupper, isprint, ispunct, isspace, isxdigit, isalnum, isalpha.
- malloc.h - The malloc routines are developed by Dmitry S. Obukhov (dso@usa.net). These routines will allocate memory from the external ram. Here is a description on how to use them (as described by the author).

  ```
  //Example:
  //    #define DYNAMIC_MEMORY_SIZE 0x2000
  //    .....
  //   unsigned char xdata dynamic_memory_pool[DYNAMIC_MEMORY_SIZE];
  //    unsigned char xdata * current_buffer;
  //    .....
  //    void main(void)
  //    {
  //        ...
  //      init_dynamic_memory(dynamic_memory_pool,DYNAMIC_MEMORY_SIZE);
  //        //Now it's possible to use malloc.
  //        ...
  //        current_buffer = malloc(0x100);
  //
  ```
- serial.h - Serial IO routines are also developed by Dmitry S. Obukhov (dso@usa.net). These routines are interrupt driven with a 256 byte circular buffer, they also expect external ram to be present. Please see documentation in file SDCCDIR/sdcc51lib/serial.c. Note the header file “serial.h” MUST be included in the file containing the ’main’ function.
- ser.h - Alternate serial routine provided by Wolfgang Esslinger <wolfgang@WiredMinds.com> these routines are more compact and faster. Please see documentation in file SDCCDIR/sdcc51lib/ser.c

- ser_ir.h - Another alternate set of serial routines provided by Josef Wolf <jw@raven.inka.de>, these routines do not use the external ram.
- reg51.h - contains register definitions for a standard 8051
- float.h - contains min, max and other floating point related stuff.

All library routines are compiled as –model-small, they are all non-reentrant, if you plan to use the large model or want to make these routines reentrant, then they will have to be recompiled with the appropriate compiler option.

Have not had time to do the more involved routines like printf, will get to them shortly.

4.4 Interfacing with Assembly Routines

4.4.1 Global Registers used for Parameter Passing

The compiler always uses the global registers *DPL,DPH,B* and *ACC* to pass the first parameter to a routine. The second parameter onwards is either allocated on the stack (for reentrant routines or if –stack-auto is used) or in the internal / external ram (depending on the memory model).

4.4.2 Assembler Routine(non-reentrant)

In the following example the function cfunc calls an assembler routine asm_func, which takes two parameters.

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j)
{
return asm_func(i,j);
}

int main()
{
return c_func(10,9);
}
```

The corresponding assembler function is:

```
.globl _asm_func_PARM_2
.globl _asm_func
.area OSEG
_asm_func_PARM_2:
.ds 1
.area CSEG
```

```
_asm_func:
mov a,dpl
add a,_asm_func_PARM_2
mov dpl,a
mov dpl,#0x00
ret
```

Note here that the return values are placed in 'dpl' - One byte return value, 'dpl' LSB & 'dph' MSB for two byte values. 'dpl', 'dph' and 'b' for three byte values (generic pointers) and 'dpl','dph','b' & 'acc' for four byte values.

The parameter naming convention is _<function_name>_PARM_<n>, where n is the parameter number starting from 1, and counting from the left. The first parameter is passed in "dpl" for One bye parameter, "dptr" if two bytes, "b,dptr" for three bytes and "acc,b,dptr" for four bytes, the varible name for the second parameter will be _<function_name>_PARM_2.

Assemble the assembler routine with the following command:

asx8051 -losg asmfunc.asm

Then compile and link the assembler routine to the C source file with the following command:

sdcc cfunc.c asmfunc.rel

4.4.3 Assembler Routine(reentrant)

In this case the second parameter onwards will be passed on the stack, the parameters are pushed from right to left i.e. after the call the left most parameter will be on the top of the stack. Here is an example:

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j) reentrant
{
return asm_func(i,j);
}

int main()
{
return c_func(10,9);
}
```

The corresponding assembler routine is:

```
.globl _asm_func
```

```
_asm_func:
push _bp
mov _bp,sp
mov r2,dpl
mov a,_bp
clr c
add a,#0xfd
mov r0,a
add a,#0xfc
mov r1,a
mov a,@r0
add a,r2
mov dpl,a
mov dph,#0x00
mov sp,_bp
pop _bp
ret
```

The compiling and linking procedure remains the same, however note the extra entry & exit linkage required for the assembler code, _bp is the stack frame pointer and is used to compute the offset into the stack for parameters and local variables.

4.5 External Stack

The external stack is located at the start of the external ram segment, and is 256 bytes in size. When –xstack option is used to compile the program, the parameters and local variables of all reentrant functions are allocated in this area. This option is provided for programs with large stack space requirements. When used with the –stack-auto option, all parameters and local variables are allocated on the external stack (note support libraries will need to be recompiled with the same options).

The compiler outputs the higher order address byte of the external ram segment into PORT P2, therefore when using the External Stack option, this port MAY NOT be used by the application program.

4.6 ANSI-Compliance

Deviations from the compliancy.

- functions are not always reentrant.
- structures cannot be assigned values directly, cannot be passed as function parameters or assigned to each other and cannot be a return value from a function, e.g.:

  ```
  struct s { ... };
  struct s s1, s2;
  ```

```
foo()
{
...
s1 = s2 ; /* is invalid in SDCC although allowed in ANSI */
...
}
struct s foo1 (struct s parms) /* is invalid in SDCC although allowed
in ANSI */
{
struct s rets;
...
return rets;/* is invalid in SDCC although allowed in ANSI */
}
```

- ’long long’ (64 bit integers) not supported.
- ’double’ precision floating point not supported.
- No support for setjmp and longjmp (for now).
- Old K&R style function declarations are NOT allowed.

```
foo(i,j) /* this old style of function declarations */
int i,j; /* are valid in ANSI but not valid in SDCC */
{
...
}
```

- functions declared as pointers must be dereferenced during the call.

```
int (*foo)();
...
/* has to be called like this */
(*foo)(); /* ansi standard allows calls to be made like 'foo()'
*/
```

4.7 Cyclomatic Complexity

Cyclomatic complexity of a function is defined as the number of independent paths the program can take during execution of the function. This is an important number since it defines the number test cases you have to generate to validate the function. The accepted industry standard for complexity number is 10, if the cyclomatic complexity reported by SDCC exceeds 10 you should think about simplification of the function logic. Note that the complexity level is not related to the number of lines of code in a function. Large functions can have low complexity, and small functions can have large complexity levels.

SDCC uses the following formula to compute the complexity:

complexity = (number of edges in control flow graph) - (number of nodes in control flow graph) + 2;

Having said that the industry standard is 10, you should be aware that in some cases it be may unavoidable to have a complexity level of less than 10. For example if you have switch statement with more than 10 case labels, each case label adds one to the complexity level. The complexity level is by no means an absolute measure of the algorithmic complexity of the function, it does however provide a good starting point for which functions you might look at for further optimization.

5 TIPS

Here are a few guidelines that will help the compiler generate more efficient code, some of the tips are specific to this compiler others are generally good programming practice.

- Use the smallest data type to represent your data-value. If it is known in advance that the value is going to be less than 256 then use a ’char’ instead of a ’short’ or ’int’.
- Use unsigned when it is known in advance that the value is not going to be negative. This helps especially if you are doing division or multiplication.
- NEVER jump into a LOOP.
- Declare the variables to be local whenever possible, especially loop control variables (induction).
- Since the compiler does not do implicit integral promotion, the programmer should do an explicit cast when integral promotion is required.
- Reducing the size of division, multiplication & modulus operations can reduce code size substantially. Take the following code for example.

  ```
  foobar(unsigned int p1, unsigned char ch)
  {
  unsigned char ch1 = p1 % ch ;
  ....
  }
  ```

  For the modulus operation the variable ch will be promoted to unsigned int first then the modulus operation will be performed (this will lead to a call to support routine _muduint()), and the result will be casted to an int. If the code is changed to

```
foobar(unsigned int p1, unsigned char ch)
{
unsigned char ch1 = (unsigned char)p1 % ch ;
....
}
```

It would substantially reduce the code generated (future versions of the compiler will be smart enough to detect such optimization oppurtunities).

5.1 Notes on MCS51 memory layout

The 8051 family of micro controller have a minimum of 128 bytes of internal memory which is structured as follows

- Bytes 00-1F - 32 bytes to hold up to 4 banks of the registers R7 to R7
- Bytes 20-2F - 16 bytes to hold 128 bit variables and
- Bytes 30-7F - 60 bytes for general purpose use.

Normally the SDCC compiler will only utilise the first bank of registers, but it is possible to specify that other banks of registers should be used in interrupt routines. By default, the compiler will place the stack after the last bank of used registers, i.e. if the first 2 banks of registers are used, it will position the base of the internal stack at address 16 (0X10). This implies that as the stack grows, it will use up the remaining register banks, and the 16 bytes used by the 128 bit variables, and 60 bytes for general purpose use.

By default, the compiler uses the 60 general purpose bytes to hold "near data". The compiler/optimiser may also declare some Local Variables in this area to hold local data.

If any of the 128 bit variables are used, or near data is being used then care needs to be taken to ensure that the stack does not grow so much that it starts to over write either your bit variables or "near data". There is no runtime checking to prevent this from happening.

The amount of stack being used is affected by the use of the "internal stack" to save registers before a subroutine call is made (–stack-auto will declare parameters and local variables on the stack) and the number of nested subroutines.

If you detect that the stack is over writing you data, then the following can be done. –xstack will cause an external stack to be used for saving registers and (if –stack-auto is being used) storing parameters and local variables. However this will produce more code which will be slower to execute.

–stack-loc will allow you specify the start of the stack, i.e. you could start it after any data in the general purpose area. However this may waste the memory not used by the register banks and if the size of the "near data" increases, it may creep into the bottom of the stack.

–stack-after-data, similar to the –stack-loc, but it automatically places the stack after the end of the "near data". Again this could waste any spare register space.

–data-loc allows you to specify the start address of the near data. This could be used to move the "near data" further away from the stack giving it more room to grow. This will only work if no bit variables are being used and the stack can grow to use the bit variable space.

Conclusion.

If you find that the stack is over writing your bit variables or "near data" then the approach which best utilised the internal memory is to position the "near data" after the last bank of used registers or, if you use bit variables, after the last bit variable by using the –data-loc, e.g. if two register banks are being used and no bit variables, –data-loc 16, and use the –stack-after-data option.

If bit variables are being used, another method would be to try and squeeze the data area in the unused register banks if it will fit, and start the stack after the last bit variable.

6 Retargetting for other MCUs.

The issues for retargetting the compiler are far too numerous to be covered by this document. What follows is a brief description of each of the seven phases of the compiler and its MCU dependency.

- Parsing the source and building the annotated parse tree. This phase is largely MCU independent (except for the language extensions). Syntax & semantic checks are also done in this phase, along with some initial optimizations like back patching labels and the pattern matching optimizations like bit-rotation etc.

- The second phase involves generating an intermediate code which can be easy manipulated during the later phases. This phase is entirely MCU independent. The intermediate code generation assumes the target machine has unlimited number of registers, and designates them with the name iTemp. The compiler can be made to dump a human readable form of the code generated by using the –dumpraw option.

- This phase does the bulk of the standard optimizations and is also MCU independent. This phase can be broken down into several sub-phases:

  Break down intermediate code (iCode) into basic blocks.
  Do control flow & data flow analysis on the basic blocks.
  Do local common subexpression elimination, then global subexpression elimination
  Dead code elimination
  Loop optimizations
  If loop optimizations caused any changes then do 'global subexpression elimination' and 'dead code elimination' again.

- This phase determines the live-ranges; by live range I mean those iTemp variables defined by the compiler that still survive after all the optimizations. Live range analysis is essential for register allocation, since these computation determines which of these iTemps will be assigned to registers, and for how long.
- Phase five is register allocation. There are two parts to this process.

  The first part I call 'register packing' (for lack of a better term). In this case several MCU specific expression folding is done to reduce register pressure.

  The second part is more MCU independent and deals with allocating registers to the remaining live ranges. A lot of MCU specific code does creep into this phase because of the limited number of index registers available in the 8051.
- The Code generation phase is (unhappily), entirely MCU dependent and very little (if any at all) of this code can be reused for other MCU. However the scheme for allocating a homogenized assembler operand for each iCode operand may be reused.
- As mentioned in the optimization section the peep-hole optimizer is rule based system, which can reprogrammed for other MCUs.

7 SDCDB - Source Level Debugger

SDCC is distributed with a source level debugger. The debugger uses a command line interface, the command repertoire of the debugger has been kept as close to gdb (the GNU debugger) as possible. The configuration and build process is part of the standard compiler installation, which also builds and installs the debugger in the target directory specified during configuration. The debugger allows you debug BOTH at the C source and at the ASM source level.

7.1 Compiling for Debugging

The debug option must be specified for all files for which debug information is to be generated. The complier generates a .cdb file for each of these files. The linker updates the .cdb file with the address information. This .cdb is used by the debugger.

7.2 How the Debugger Works

When the –debug option is specified the compiler generates extra symbol information some of which are put into the the assembler source and some are put into the .cdb file, the linker updates the .cdb file with the address information for the symbols. The debugger reads the symbolic information generated by the compiler & the address information generated by the linker. It uses the SIMULATOR (Daniel's S51) to execute the program, the program execution is controlled by the debugger. When a command is issued for the debugger, it translates it into appropriate commands for the simulator.

7.3 Starting the Debugger

The debugger can be started using the following command line. (Assume the file you are debugging has the file name foo).

```
sdcdb foo
```

The debugger will look for the following files.

- foo.c - the source file.
- foo.cdb - the debugger symbol information file.
- foo.ihx - the intel hex format object file.

7.4 Command Line Options.

- –directory=<source file directory> this option can used to specify the directory search list. The debugger will look into the directory list specified for source, cdb & ihx files. The items in the directory list must be separated by ':', e.g. if the source files can be in the directories /home/src1 and /home/src2, the –directory option should be –directory=/home/src1:/home/src2. Note there can be no spaces in the option.
- -cd <directory> - change to the <directory>.
- -fullname - used by GUI front ends.
- -cpu <cpu-type> - this argument is passed to the simulator please see the simulator docs for details.
- -X <Clock frequency > this options is passed to the simulator please see the simulator docs for details.
- -s <serial port file> passed to simulator see the simulator docs for details.
- -S <serial in,out> passed to simulator see the simulator docs for details.

7.5 Debugger Commands.

As mention earlier the command interface for the debugger has been deliberately kept as close the GNU debugger gdb, as possible. This will help the integration with existing graphical user interfaces (like ddd, xxgdb or xemacs) existing for the GNU debugger.

7.5.1 break [line | file:line | function | file:function]

Set breakpoint at specified line or function:

```
sdcdb>break 100
sdcdb>break foo.c:100
sdcdb>break funcfoo
sdcdb>break foo.c:funcfoo
```

7.5.2 clear [line | file:line | function | file:function]

Clear breakpoint at specified line or function:

```
sdcdb>clear 100
sdcdb>clear foo.c:100
sdcdb>clear funcfoo
sdcdb>clear foo.c:funcfoo
```

7.5.3 continue

Continue program being debugged, after breakpoint.

7.5.4 finish

Execute till the end of the current function.

7.5.5 delete [n]

Delete breakpoint number 'n'. If used without any option clear ALL user defined break points.

7.5.6 info [break | stack | frame | registers]

- info break - list all breakpoints
- info stack - show the function call stack.
- info frame - show information about the current execution frame.
- info registers - show content of all registers.

7.5.7 step

Step program until it reaches a different source line.

7.5.8 next

Step program, proceeding through subroutine calls.

7.5.9 run

Start debugged program.

7.5.10 ptype variable

Print type information of the variable.

7.5.11 print variable

print value of variable.

7.5.12 file filename

load the given file name. Note this is an alternate method of loading file for debugging.

7.5.13 frame

print information about current frame.

7.5.14 set srcmode

Toggle between C source & assembly source.

7.5.15 ! simulator command

Send the string following '!' to the simulator, the simulator response is displayed. Note the debugger does not interpret the command being sent to the simulator, so if a command like 'go' is sent the debugger can loose its execution context and may display incorrect values.

7.5.16 quit.

"Watch me now. Iam going Down. My name is Bobby Brown"

7.6 Interfacing with XEmacs.

Two files (in emacs lisp) are provided for the interfacing with XEmacs, sdcdb.el and sdcdbsrc.el. These two files can be found in the $(prefix)/bin directory after the installation is complete. These files need to be loaded into XEmacs for the interface to work. This can be done at XEmacs startup time by inserting the following into your '.xemacs' file (which can be found in your HOME directory):

```
(load-file sdcdbsrc.el)
```

.xemacs is a lisp file so the () around the command is REQUIRED. The files can also be loaded dynamically while XEmacs is running, set the environment variable 'EMACSLOADPATH' to the installation bin directory (<installdir>/bin), then enter the following command ESC-x load-file sdcdbsrc. To start the interface enter the following command:

ESC-x sdcdbsrc

You will prompted to enter the file name to be debugged.

The command line options that are passed to the simulator directly are bound to default values in the file sdcdbsrc.el. The variables are listed below, these values maybe changed as required.

- sdcdbsrc-cpu-type '51
- sdcdbsrc-frequency '11059200
- sdcdbsrc-serial nil

The following is a list of key mapping for the debugger interface.

```
;; Current Listing ::
;;key               binding                      Comment
;;---               -------                      -------
;;
;; n              sdcdb-next-from-src          SDCDB next command
;; b              sdcdb-back-from-src          SDCDB back command
;; c              sdcdb-cont-from-src          SDCDB continue command
;; s              sdcdb-step-from-src          SDCDB step command
;; ?               sdcdb-whatis-c-sexp          SDCDB ptypecommand for
data at
;;                                              buffer point
;; x              sdcdbsrc-delete              SDCDB Delete all breakpoints
if no arg
;;                                             given or delete arg (C-u
arg x)
;; m              sdcdbsrc-frame               SDCDB Display current
frame if no arg,
;;                                              given or display frame
arg
;;                                              buffer point
;; !               sdcdbsrc-goto-sdcdb          Goto the SDCDB output
buffer
;; p              sdcdb-print-c-sexp           SDCDB print command for
data at
;;                                              buffer point
;; g              sdcdbsrc-goto-sdcdb          Goto the SDCDB output
buffer
;; t              sdcdbsrc-mode                Toggles Sdcdbsrc mode
(turns it off)
;;
;; C-c C-f        sdcdb-finish-from-src        SDCDB finish command
```

```
;;
;; C-x SPC         sdcdb-break                  Set break for line with
point
;; ESC t           sdcdbsrc-mode                Toggle Sdcdbsrc mode

;; ESC m           sdcdbsrc-srcmode             Toggle list mode
;;
```

8 Other Processors

8.1 The Z80 and gbz80 port

SDCC can target both the Zilog Z80 and the Nintendo Gameboy's Z80-like gbz80. The port is incomplete - long support is incomplete (mul, div and mod are unimplimented), and both float and bitfield support is missing. Apart from that the code generated is correct.

As always, the code is the authoritave reference - see z80/ralloc.c and z80/gen.c. The stack frame is similar to that generated by the IAR Z80 compiler. IX is used as the base pointer, HL is used as a temporary register, and BC and DE are available for holding varibles. IY is currently unusued. Return values are stored in HL. One bad side effect of using IX as the base pointer is that a functions stack frame is limited to 127 bytes - this will be fixed in a later version.

9 Support

SDCC has grown to be a large project. The compiler alone (without the preprocessor, assembler and linker) is about 40,000 lines of code (blank stripped). The open source nature of this project is a key to its continued growth and support. You gain the benefit and support of many active software developers and end users. Is SDCC perfect? No, that's why we need your help. The developers take pride in fixing reported bugs. You can help by reporting the bugs and helping other SDCC users. There are lots of ways to contribute, and we encourage you to take part in making SDCC a great software package.

9.1 Reporting Bugs

Send an email to the mailing list at 'user-sdcc@sdcc.sourceforge.net' or 'devel-sdcc@sdcc.sourceforge.net'. Bugs will be fixed ASAP. When reporting a bug, it is very useful to include a small test program which reproduces the problem. If you can isolate the problem by looking at the generated assembly code, this can be very helpful. Compiling your program with the –dumpall option can sometimes be useful in locating optimization problems.

10 Acknowledgments

Sandeep Dutta (sandeep.dutta@usa.net) - SDCC, the compiler, MCS51 code generator, Debugger, AVR port
Alan Baldwin (baldwin@shop-pdp.kent.edu) - Initial version of ASXXXX & ASLINK.
John Hartman (jhartman@compuserve.com) - Porting ASXXX & ASLINK for 8051
Dmitry S. Obukhov (dso@usa.net) - malloc & serial i/o routines.
Daniel Drotos (drdani@mazsola.iit.uni-miskolc.hu) - for his Freeware simulator
Malini Dutta(malini_dutta@hotmail.com) - my wife for her patience and support.
Unknown - for the GNU C - preprocessor.
Michael Hope - The Z80 and Z80GB port, 186 development
Kevin Vigor - The DS390 port.
Johan Knol - Lots of fixes and enhancements, DS390/TINI libs.
Scott Datallo - The PIC port.

Thanks to all the other volunteer developers who have helped with coding, testing, web-page creation, distribution sets, etc. You know who you are :-)

This document was initially written by Sandeep Dutta

Index