SDCC Compiler User Guide

$Date: 2003/08/29 09:46:05 $

$Revision: 1.68 $

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 (8031, 8032, 8051, 8052, etc), Zilog Z80 based MCUs, and the Dallas DS80C390 variant. It can be retargetted for other microprocessors, support for Microchip PIC, Atmel AVR 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
- jump tables 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/snap.php. *Please note: the compiler will probably always be some steps ahead of this documentation*[1]**.**

1.2 Open Source

All packages used in this compiler system are *opensource* and *freeware*; source code for all the sub-packages (pre-processor, assemblers, linkers etc) 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

[1] Obviously this has pros and cons

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 *italic*.

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) which is not ANSI compliant
- the default directory for gcc-builds where include, library and documentation files are stored is now 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 --noregparms has been removed
- option --stack-after-data 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:

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

code banking support for mcs51
```

If you can think of some more, please see the chapter 6.2 about filing feature requests.

2 Installing SDCC

For most users it is sufficient to skip to either section 2.4.1 or section 2.4.8. More detailled instructions follow below.

2.1 Configure Options

The install paths, search paths and other options are defined when running 'configure'. The defaults can be overridden by:

--prefix see table below

--exec_prefix see table below

--bindir see table below

--datadir see table below

docdir environment variable, see table below

include_dir_suffix environment variable, see table below

lib_dir_suffix environment variable, see table below

sdccconf_h_dir_separator environment variable, either / or \\ makes sense here. This character will only be used in sdccconf.h; don't forget it's a C-header, therefore a double-backslash is needed there.

--disable-mcs51-port Excludes the Intel mcs51 port

--disable-gbz80-port Excludes the Gameboy gbz80 port

--z80-port Excludes the z80 port

--disable-avr-port Excludes the AVR port

--disable-ds390-port Excludes the DS390 port

--disable-pic-port Excludes the PIC port

--disable-xa51-port Excludes the XA51 port

--disable-ucsim Disables configuring and building of ucsim

--disable-device-lib-build Disables automatically building device libraries

--disable-packihx Disables building packihx

--enable-libgc Use the Bohem memory allocator. Lower runtime footprint.

Furthermore the environment variables CC, CFLAGS, ... the tools and their arguments can be influenced. Please see 'configure --help' and the man/info pages of 'configure' for details.

The names of the standard libraries STD_LIB, STD_INT_LIB, STD_LONG_LIB, STD_FP_LIB, STD_DS390_LIB, STD_XA51_LIB and the environment variables SDCC_DIR_NAME, SDCC_INCLUDE_NAME, SDCC_LIB_NAME are defined by 'configure' too. At the moment it's not possible to change the default settings (it was simply never required.

These configure options are compiled into the binaries, and can only be changed by rerunning 'configure' and recompiling SDCC. The configure options are written in *italics* to distinguish them from run time environment variables (see section search paths).

The settings for "Win32 builds" are used by the SDCC team to build the official Win32 binaries. The SDCC team uses Mingw32 to build the official Windows binaries, because it's

1. open source,
2. a gcc compiler and last but not least
3. the binaries can be built by cross compiling on Sourceforge's compile farm.

See the examples, how to pass the Win32 settings to 'configure'. The other Win32 builds using Borland, VC or whatever don't use 'configure', but a header file sdcc_vc_in.h is the same as sdccconf.h built by 'configure' for Win32.

These defaults are:

Variable	default	Win32 builds
PREFIX	/usr/local	\sdcc
EXEC_PREFIX	*$PREFIX*	*$PREFIX*
BINDIR	*$EXECPREFIX*/bin	*$EXECPREFIX*\bin
DATADIR	*$PREFIX*/share	*$PREFIX*
DOCDIR	*$DATADIR*/sdcc/doc	*$DATADIR*\doc
INCLUDE_DIR_SUFFIX	sdcc/include	include
LIB_DIR_SUFFIX	sdcc/lib	lib

'configure' also computes relative paths. This is needed for full relocatability of a binary package and to complete search paths (see section search paths below):

Variable (computed)	default	Win32 builds
BIN2DATA_DIR	../share	..
PREFIX2BIN_DIR	bin	bin
PREFIX2DATA_DIR	share/sdcc	

Examples:

```
./configure
./configure --prefix="/usr/bin" --datadir="/usr/share"
./configure --disable-avr-port --disable-xa51-port
```

To cross compile on linux for Mingw32 (see also 'sdcc/support/scripts/sdcc_mingw32'):

```
./configure \
CC="i586-mingw32msvc-gcc" CXX="i586-mingw32msvc-g++" \
RANLIB="i586-mingw32msvc-ranlib" \
STRIP="i586-mingw32msvc-strip" \
--prefix="/sdcc" \
--datadir="/sdcc" \
docdir="/sdcc/doc" \
include_dir_suffix="include" \
lib_dir_suffix="lib" \
sdccconf_h_dir_separator="\\\\" \
--disable-device-lib-build\
--disable-ucsim\
--host=i586-mingw32msvc --build=unknown-unknown-linux-gnu
```

To "cross"compile on Cygwin for Mingw32 (see also sdcc/support/scripts/sdcc_cygwin_mingw32):

```
./configure -C \
CFLAGS="-mno-cygwin -O2" \
LDFLAGS="-mno-cygwin" \
--prefix="/sdcc" \
--datadir="/sdcc" \
docdir="/sdcc/doc" \
include_dir_suffix="include" \
lib_dir_suffix="lib" \
sdccconf_h_dir_separator="\\\\" \
--disable-ucsim
```

'configure' is quite slow on Cygwin (at least on windows before Win2000/XP). The option '--C' turns on caching, which gives a little bit extra speed. However if options are changed, it can be necessary to delete the config.cache file.

2.2 Install paths

Description	Path	Default	Win32 builds
Binary files*	*$EXEC_PREFIX*	/usr/local/bin	\sdcc\bin
Include files	*$DATADIR/ $INCLUDE_DIR_SUFFIX*	/usr/local/share/sdcc/include	\sdcc\include
Library file**	*$DATADIR/$LIB_DIR_SUFFIX*	/usr/local/share/sdcc/lib	\sdcc\lib
Documentation	*$DOCDIR*	/usr/local/share/sdcc/doc	\sdcc\doc

*compiler, preprocessor, assembler, and linker
**the *model* is auto-appended by the compiler, e.g. small, large, z80, ds390 etc

The install paths can still be changed during 'make install' with e.g.:

```
make install prefix=$(HOME)/local/sdcc
```

Of course this doesn't change the search paths compiled into the binaries.

2.3 Search Paths

Some search paths or parts of them are determined by configure variables (in *italics*, see section above). Further search paths are determined by environment variables during runtime.
The paths searched when running the compiler are as follows (the first catch wins):

1. Binary files (preprocessor, assembler and linker)

Search path	default	Win32 builds
$SDCC_HOME/*$PPREFIX2BIN_DIR*	$SDCC_HOME/bin	$SDCC_HOME\bin
Path of argv[0] (if available)	Path of argv[0]	Path of argv[0]
$PATH	$PATH	$PATH

2. Include files

Search path	default	Win32 builds
--I dir	--I dir	--I dir
$SDCC_INCLUDE	$SDCC_INCLUDE	$SDCC_INCLUDE
$SDCC_HOME/ *$PREFIX2DATA_DIR/ $INCLUDE_DIR_SUFFIX*	$SDCC_ HOME/ share/sdcc/ include	$SDCC_HOME\include
path(argv[0])/ *$BIN2DATADIR/ $INCLUDE_DIR_SUFFIX*	path(argv[0])/ ../sdcc/include	path(argv[0])\..\include
$DATADIR/ $INCLUDE_DIR_SUFFIX	/usr/local/share/sdcc/ include	(not on Win32)

The option --nostdinc disables the last two search paths.

3. Library files

With the exception of "--L dir" the *model* is auto-appended by the compiler (e.g. small, large, z80, ds390 etc.).

Search path	default	Win32 builds
--L dir	--L dir	--L dir
$SDCC_LIB/ *<model>*	$SDCC_LIB/ *<model>*	$SDCC_LIB\ *<model>*
$SDCC_HOME/ *$PREFIX2DATA_DIR/ $LIB_DIR_SUFFIX/<model>*	$SDCC_HOME/ share/sdcc/ lib/*<model>*	$SDCC_HOME\lib\ *<model>*
path(argv[0])/ *$BIN2DATADIR/ $LIB_DIR_SUFFIX/<model>*	path(argv[0])/ ../sdcc/lib/*<model>*	path(argv[0])\ ..\lib*<model>*
$DATADIR/ $LIB_DIR_SUFFIX/<model>	/usr/local/share/sdcc/ lib/*<model>*	(not on Win32)

The option --nostdlib disables the last two search paths.

2.4 Building SDCC

2.4.1 Building SDCC on Linux

1. Download the source package either from the SDCC CVS repository or from the nightly snapshots http://sdcc.sourceforge.net/snap.php, it will be named something like sdcc.src.tar.gz.
2. Bring up a command line terminal, such as xterm.
3. Unpack the file using a command like: **"tar -xzf sdcc.src.tar.gz**", 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.

2.4.2 Building SDCC on OSX 2.x

Follow the instruction for Linux.

On OSX 2.x it was reported, that the default gcc (version 3.1 20020420 (prerelease)) fails to compile SDCC. Fortunately there's also gcc 2.9.x installed, which works fine. This compiler can be selected by running 'configure' with:

```
./configure CC=gcc2 CXX=g++2
```

2.4.3 Cross compiling SDCC on Linux for Windows

With the Mingw32 gcc cross compiler it's easy to compile SDCC for Win32. See section 'Configure Options'.

2.4.4 Building SDCC on Windows

With the exception of Cygwin the SDCC binaries uCsim and sdcdb can't be built on Windows. They use Unix-sockets, which are not available on Win32.

2.4.5 Building SDCC using Cygwin and Mingw32

For building and installing a Cygwin executable follow the instructions for Linux.

On Cygwin a "native" Win32-binary can be built, which will not need the Cygwin-DLL. For the necessary 'configure' options see section 'configure options' or the script 'sdcc/support/scripts/sdcc_cygwin_mingw32'.

In order to install Cygwin on Windows download setup.exe from www.cygwin.com http://www.cygwin.com/. Run it, set the "default text file type" to "unix" and download/install at least the following packages. Some packages are selected by default, others will be automatically selected because of dependencies with the manually selected packages. Never deselect these packages!

- flex
- bison
- gcc ; version 3.x is fine, no need to use the old 2.9x
- binutils ; selected with gcc
- make
- rxvt ; a nice console, which makes life much easier under windoze (see below)
- man ; not really needed for building SDCC, but you'll miss it sooner or later
- less ; not really needed for building SDCC, but you'll miss it sooner or later
- cvs ; only if you use CVS access

If you want to develop something you'll need:

- python ; for the regression tests
- gdb ; the gnu debugger, together with the nice GUI "insight"
- openssh ; to access the CF or commit changes
- autoconf and autoconf-devel ; if you want to fight with 'configure', don't use autoconf-stable!

rxvt is a nice console with history. Replace in your cygwin.bat the line

```
bash --login -i
```

with (one line):

```
rxvt -sl 1000 -fn "Lucida Console-12" -sr -cr red
     -bg black -fg white -geometry 100x65 -e bash --login
```

Text selected with the mouse is automatically copied to the clipboard, pasting works with shift-insert.

The other good tip is to make sure you have no //c/-style paths anywhere, use /cygdrive/c/ instead. Using // invokes a network lookup which is very slow. If you think ”cygdrive” is too long, you can change it with e.g.

```
mount -s -u -c /mnt
```

SDCC sources use the unix line ending LF. Life is much easier, if you store the source tree on a drive which is mounted in binary mode. And use an editor which can handle LF-only line endings. Make sure not to commit files with windows line endings. The tabulator spacing used in the project is 8.

2.4.6 Building SDCC Using Microsoft Visual C++ 6.0/NET (MSVC)

Download the source package either from the SDCC CVS repository or from the nightly snapshots http://sdcc.sourceforge.net/snap.php, it will be named something like sdcc.src.tgz. SDCC is distributed with all the projects, workspaces, and files you need to build it using Visual C++ 6.0/NET (except for sdcdb.exe which currently doesn't build under MSVC). The workspace name is 'sdcc.dsw'. Please note that as it is now, all the executables are created in a folder called sdcc\bin_vc. Once built you need to copy the executables from sdcc\bin_vc to sdcc\bin before running SDCC.

In order to build SDCC with MSVC you need win32 executables of bison.exe, flex.exe, and gawk.exe. One good place to get them is here http://unxutils.sourceforge.net

Download the file UnxUtils.zip. Now you have to install the utilities and setup MSVC so it can locate the required programs. Here there are two alternatives (choose one!):

1. The easy way:

   a) Extract UnxUtils.zip to your C:\ hard disk PRESERVING the original paths, otherwise bison won't work. (If you are using WinZip make certain that 'Use folder names' is selected)

   b) In the Visual C++ IDE click Tools, Options, select the Directory tab, in 'Show directories for:' select 'Executable files', and in the directories window add a new path: 'C:\user\local\wbin', click ok.

   (As a side effect, you get a bunch of Unix utilities that could be useful, such as diff and patch.)

2. A more compact way:

   This one avoids extracting a bunch of files you may not use, but requires some extra work:

   a) Create a directory were to put the tools needed, or use a directory already present. Say for example 'C:\util'.

   b) Extract 'bison.exe', 'bison.hairy', 'bison.simple', 'flex.exe', and gawk.exe to such directory WITHOUT preserving the original paths. (If you are using WinZip make certain that 'Use folder names' is not selected)

   c) Rename bison.exe to '_bison.exe'.

   d) Create a batch file 'bison.bat' in 'C:\util\' and add these lines:

```
set BISON_SIMPLE=C:\util\bison.simple
set BISON_HAIRY=C:\util\bison.hairy
_bison %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Steps 'c' and 'd' are needed because bison requires by default that the files 'bison.simple' and 'bison.hairy' reside in some weird Unix directory, '/usr/local/share/' I think. So it is necessary to tell bison where those files are located if they are not in such directory. That is the function of the environment variables BISON_SIMPLE and BISON_HAIRY.

e) In the Visual C++ IDE click Tools, Options, select the Directory tab, in 'Show directories for:' select 'Executable files', and in the directories window add a new path: 'c:\util', click ok. Note that you can use any other path instead of 'c:\util', even the path where the Visual C++ tools are, probably: 'C:\Program Files\Microsoft Visual Studio\Common\Tools'. So you don't have to execute step 'e' :)

That is it. Open 'sdcc.dsw' in Visual Studio, click 'build all', when it finishes copy the executables from sdcc\bin_vc to sdcc\bin, and you can compile using sdcc.

2.4.7 Building SDCC Using Borland

1. From the sdcc directory, run the command "make -f Makefile.bcc". This should regenerate all the .exe files in the bin directory except for sdcdb.exe (which currently doesn't build under Borland C++).

2. If you modify any source files and need to rebuild, be aware that the dependencies may not be correctly calculated. The safest option is to delete all .obj files and run the build again. From a Cygwin BASH prompt, this can easily be done with the command (be sure you are in the sdcc directory):

   find . \(-name '*.obj' -o -name '*.lib' -o -name '*.rul' \) -print -exec rm {} \;

   or on Windows NT/2000/XP from the command prompt with the command:

   del /s *.obj *.lib *.rul from the sdcc directory.

2.4.8 Windows Install Using a Binary Package

1. Download the binary package from http://sdcc.sourceforge.net/snap.php 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 the mingw32 package is: c:\sdcc\bin for the executables, c:\sdcc\include and c:\sdcc\lib for the include and libraries.

2. Adjust your environment variable PATH to include the location of the bin directory or start sdcc using the full path.

2.5 Building the Documentation

If the necessary tools are installed it is as easy as changing into the doc directory and typing **"make"** there. If you want to avoid installing the tools you will have some success with a bootable Knoppix CD http://www.knopper.net.

2.6 Testing 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 is commonly installed as described in section ”Install and search paths”

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

```
char test;

void main(void) {
    test=0;
}
```

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>

char str1[10];

void main(void) {
  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). Use option **--print-search-dirs** to find exactly where SDCC is looking for the include and lib files.

2.7 Install Trouble-shooting

2.7.1 SDCC does not build correctly.

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

./configure 2>&1 | tee configure.log

and build it like:

make 2>&1 | tee make.log

If anything goes wrong, you can review the log files to locate the problem. Or a relevant part of this can be attached to an email that could be helpful when requesting help from the mailing list.

2.7.2 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.

2.7.3 What the ”make” does.

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

2.7.4 What the ”make install” command does.

This will install the compiler, other executables libraries and include files into the appropriate directories. See section ”Install and Search PATHS”.
On most systems you will need super-user privileges to do this.

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 for gcc-builds:

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 (compress) Intel hex files.

In <installdir>/share/sdcc/include

- the include files

In <installdir>/share/sdcc/lib

- the subdirs src and small, large, z80, gbz80 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 - The 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 SDCC.

2.8.4 s51 - The 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 web site at: http://mazsola.iit.uni-miskolc.hu/~drdani/embedded/s51. It currently supports the core mcs51, the Dallas DS80C390 and the Phillips XA51 family.

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.mem - A file with a summary of the memory usage
- sourcefile.ihx - The load module in Intel hex format (you can select the Motorola S19 format with --out-fmt-s19. If you need another format you might want to use *objdump* or *srecord*)
- sourcefile.adb - An intermediate file containing debug information needed to create the .cdb file (with --debug)
- sourcefile.cdb - An optional file (with --debug) containing debug information
- sourcefile. - (no extension) An optional AOMF51 file containing debug information (with --debug). This format is commonly used by third party tools (debuggers, simulators, emulators)
- sourcefile.dump* - Dump file to debug the compiler it self (with --dumpall) (see section ”Anatomy of the compiler”).

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. The linker is invoked from sdcc using a script file with extension .lnk. You can view this file to troubleshoot linking problems such as those arising from missing libraries.

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 separate modules in separate 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 Intel MCS51 family of processors. This is the default processor target.

-mds390 Generate code for the Dallas DS80C390 processor.

-mds400 Generate code for the Dallas DS80C400 processor.

-mz80 Generate code for the Zilog Z80 family of processors.

-mgbz80 Generate code for the GameBoy Z80 processor.

-mavr Generate code for the Atmel AVR processor (In development, not complete). AVR users should probably have a look at avr-gcc http://savannah.nongnu.org/download/avr-libc/snapshots/.

-mpic14 Generate code for the Microchip PIC 14-bit processors (p16f84 and variants).

-mpic16 Generate code for the Microchip PIC 16-bit processors (p18f452 and variants).

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

-mxa51 Generate code for the Phillips XA51 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 preprocessor.

-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.

-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> By default the stack is placed after the data segment. Using this option the stack can be placed anywhere in the internal memory space of the 8051. The value entered can be in Hexadecimal or Decimal format, e.g. --stack-loc 0x20 or --stack-loc 32. Since the sp register is incremented before a push or call, the initial sp will be set to one byte prior the provided value. The provided value should not overlap any other memory areas such as used register banks or the data segment and with enough space for the current application.

--data-loc<Value> The start location of the internal ram data segment. The value entered can be in Hexadecimal or Decimal format, eg. --data-loc 0x20 or --data-loc 32. (By default, the start location of the internal ram data segment is set as low as possible in memory, taking into account the used register banks and the bit segment at address 0x20. For example if register banks 0 and 1 are used without bit variables, the data segment will be set, if --data-loc is not used, to location 0x10.)

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

--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.

--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.

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

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

--code-size<Value> Causes the linker to check if the code memory usage is within limits of the given value.

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 Z80 Options

--callee-saves-bc Force a called function to always save BC.

--no-std-crt0 When linking, skip the standard crt0.o object file. You must provide your own crt0.o for your system when linking.

3.2.7 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 jumptables. 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.

--nolabelopt Will not optimize labels (makes the dumpfiles more readable).

--no-xinit-opt Will not memcpy initialized data from code space into xdata space. This saves a few bytes in code space if you don't have initialized data.

3.2.8 Other Options

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

--c1mode reads the preprocessed source from standard input and compiles it. The file name for the assembler output must be specified using the -o option.

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

-o <path/file> The output path resp. file where everything will be placed. If the parameter is a path, it must have a trailing slash (or backslash for the Windows binaries) to be recognized as a path.

--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.

--callee-saves function1[,function2][,function3].... The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unnecessary 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 (function prologue & epilogue) 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 inter procedural 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 _mulint..., 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.

-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 .`'.

--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.

--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.

--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.

--no-c-code-in-asm Hides your ugly and inefficient c-code from the asm file, so you can always blame the compiler :).

--i-code-in-asm Include i-codes in the asm file. Sounds like noise but is most helpful for debugging the compiler itself.

--less-pedantic Disable some of the more pedantic warnings (jwk burps: please be more specific here, please!)

--print-search-dirs Display the directories in the compiler's search path

--vc Display errors and warnings using MSVC style, so you can use SDCC with visual studio.

--use-stdout Send errors and warnings to stdout instead of stderr.

3.2.9 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.

3.2.10 Redirecting output on Windows Shells

By default SDCC writes it's error messages to "standard error". To force all messages to "standard output" use --use-stdout. Aditionaly, if you happen to have visual studio installed in your windows machine, you can use it to compile your sources using a custom build and the SDCC --vc option. Something like this should work:

c:\sdcc\bin\sdcc.exe --vc --model-large -c $(InputPath)

3.3 Environment variables

SDCC recognizes the following environment variables:

SDCC_LEAVE_SIGNALS SDCC installs a signal handler to be able to delete temporary files after an user break (^C) or an exception. If this environment variable is set, SDCC won't install the signal handler in order to be able to debug SDCC.

TMP, TEMP, TMPDIR Path, where temporary files will be created. The order of the variables is the search order. In a standard *nix environment these variables are not set, and there's no need to set them. On Windows it's recommended to set one of them.

SDCC_HOME Path, see "2.3 Install and search paths".

SDCC_INCLUDE Path, see "2.3 Install and search paths".

SDCC_LIB Path, see "2.3 Install and search paths".

There are some more environment variables recognized by SDCC, but these are solely used for debugging purposes. They can change or disappear very quickly, and will never be documented.

3.4 MCS51/DS390 Storage Class Language Extensions

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

3.4.1 data

This is the **default** storage class for Small Memory model. Variables declared with this storage class will be allocated in the directly addressable portion of the internal RAM of a 8051, e.g.:

```
data unsigned char test_data;
```

Writing 0x01 to this variable generates the assembly code:

```
75*00 01   mov  _test_data,#0x01
```

3.4.2 xdata

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

```
xdata unsigned char test_xdata;
```

Writing 0x01 to this variable generates the assembly code:

```
90s00r00   mov  dptr,#_test_xdata
74 01      mov  a,#0x01
F0         movx @dptr,a
```

3.4.3 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 unsigned char test_idata;
```

Writing 0x01 to this variable generates the assembly code:

```
78r00       mov  r0,#_test_idata
76 01       mov  @r0,#0x01
```

3.4.4 pdata

Paged xdata access is currently not as straightforward as using the other addressing modes of a 8051. The following example writes 0x01 to the address pointed to. Please note, pdata access physically accesses xdata memory. The high byte of the address is determined by port P2 (or in case of some 8051 variants by a separate Special Function Register).

```
pdata unsigned char *test_pdata_ptr;

void main()
{
    test_pdata_ptr = (pdata *)0xfe;
    *test_pdata_ptr = 1;
}
```

Generates the assembly code:

```
75*01 FE   mov  _test_pdata_ptr,#0xFE
78 FE      mov  r0,#0xFE
74 01      mov  a,#0x01
F2         movx @r0,a
```

Be extremely carefull if you use pdata together with the --xstack option.

3.4.5 code

'Variables' declared with this storage class will be placed in the code memory:

```
code unsigned char test_code;
```

Read access to this variable generates the assembly code:

```
90s00r6F   mov dptr,#_test_code
E4         clr a
93         movc a,@a+dptr
```

3.4.6 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 test_bit;
```

Writing 1 to this variable generates the assembly code:

```
D2*00       setb _test_bit
```

3.4.7 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) */
```

3.4.8 Pointers to MCS51/DS390 specific memory spaces

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 uses (by default) generic pointers which can be used to point to any of the memory spaces.

Pointer declaration examples:

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

/* pointer physically in external 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 following is a generic pointer physically located in xdata space */
char * xdata p;
```

Well you get the idea.

All unqualified pointers are treated as 3-byte (4-byte for the ds390) *generic* pointers.

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.

3.5 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 0x7ffe unsigned int chksum;
```

In the above example the variable chksum will located at 0x7ffe and 0x7fff of the external ram. 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 good places to look for such overlaps.

In case of memory mapped I/O devices the keyword *volatile* should be used to tell the compiler that accesses might not be optimized away:

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

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. One possible use would be to write hardware portable code. For example, if you have a routine that uses one or more of the microcontroller I/O pins, and such pins are different for two different hardwares, you can declare the I/O pins in your routine using:

```
extern volatile bit SDI;
extern volatile bit SCLK;
extern volatile bit CPOL;

void DS1306_put(unsigned char value)
{
    unsigned char mask=0x80;

    while(mask)
    {
        SDI=(value & mask)?1:0;
        SCLK=!CPOL;
        SCLK=CPOL;
        mask/=2;
    }
}
```

Then, someplace in the code for the first hardware you would use

```
bit at 0x80 SDI;    /* I/O port 0, bit 0 */
bit at 0x81 SCLK;   /* I/O port 0, bit 1 */
bit CPOL;           /* This is a variable, let the linker allocate this one */
```

Similarly, for the second hardware you would use

```
bit at 0x83 SDI;    /* I/O port 0, bit 3 */
bit at 0x91 SCLK;   /* I/O port 1, bit 1 */
bit CPOL;           /* This is a variable, let the linker allocate this one */
```

and you can use the same hardware dependent routine without changes, as for example in a library. This is somehow similar to sbit, but only one absolute address has to be specified in the whole project.

3.6 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* 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 unsigned 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 should 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.

3.7 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
{
    ...
    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.

3.8 Interrupt Service Routines

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

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

The optional number following the *interrupt* keyword is the interrupt number this routine will service. When present, the compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. The *using* keyword can be 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. Furthermore nonreentrant functions should not be called from the

main program while the interrupt service routine might be active. If the interrupt service routines changes variables which are accessed by other functions these variables should be declared *volatile*.

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 8051/8052 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
5	Timer 2 (8052)	0x002B

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.
For some pitfalls see section 3.7about Overlaying and section 3.11about Functions using private banks.

3.9 Critical Functions

<TODO: this isn't implemented at all!>

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.*

3.10 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:

```
volatile data unsigned char counter;

void simpleInterrupt(void) interrupt 1
{
    counter++;
}

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

For an 8051 target, the generated simpleInterrupt looks like:

```
_simpleInterrupt:
    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 or reti in _naked function.
```

The #pragma directive EXCLUDE also allows to reduce pushing & popping the registers.

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 it is recommended that you stick to inline assembler.

3.11 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[2]).

(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

[2] 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

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 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.

3.12 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. Otherwise 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. See also the compiler option *--no-xinit-opt*.

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

An example acessing a C variable is in section 3.10.

3.14 Interfacing with Assembler Code

3.14.1 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 data / xdata memory (depending on the memory model).

3.14.2 Assembler Routine(non-reentrant)

In the following example the function c_func 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 variable 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

3.14.3 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.

3.15 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 optimizations are used. The following files contain the described routines, all of them can be found in <installdir>/share/sdcc/lib.

Function	Description
_mulint.c	16 bit multiplication
_divsint.c	signed 16 bit division (calls _divuint)
_divuint.c	unsigned 16 bit division
_modsint.c	signed 16 bit modulus (calls _moduint)
_moduint.c	unsigned 16 bit modulus
_mullong.c	32 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. Notice that you don't have to call this routines directly. The compiler will use them automatically every time a integer operation is required.

3.16 Floating Point Support

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

Function	Description
_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. Also notice that you don't have to call this routines directly. The compiler will use them automatically every time a floating point operation is required.

3.17 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 separate 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 recommended that the small model be used unless absolutely required.

3.18 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.

Note that the compiler does not generate any code to place the processor into 24 bitmode (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.

3.19 Pragmas

SDCC supports the following #pragma directives.

- SAVE - this will save all current options to the SAVE/RESTORE stack. See RESTORE.
- RESTORE - will restore saved options from the last save. SAVEs & RESTOREs can be nested. SDCC uses a SAVE/RESTORE stack: SAVE pushes current options to the stack, RESTORE pulls current options from the stack. See SAVE.
- NOGCSE - will stop global common 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 (dangerous).
- NOOVERLAY - the compiler will not overlay the parameters and local variables of a function.
- LESS_PEDANTIC - the compiler will not warn you anymore for obvious mistakes, you'r on your own now ;-(
- 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.
- NOIV - Do not generate interrupt vector table entries for all ISR functions defined after the pragma. This is useful in cases where the interrupt vector table must be defined manually, or when there is a secondary, manually defined interrupt vector table (e.g. for the autovector feature of the Cypress EZ-USB FX2).
- CALLEE-SAVES function1[,function2[,function3...]] - The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unnecessary register pushing & popping when calling small functions from larger functions. This option can be used to switch off the register saving convention for the function names specified. The compiler will not save registers when calling these functions, extra code need to be manually inserted 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 inter procedural analysis) may 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 in the 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.

3.20 Defines Created by the Compiler

The compiler creates the following #defines:

#define	**Description**
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	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 Debugging with SDCDB

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. Sdcdb is available on Unix platforms only.

4.1 Compiling for Debugging

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

4.2 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 .adb file. Then the linker creates the .cdb file from the individual .adb files 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.

4.3 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.

4.4 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.

4.5 Debugger Commands.

As mentioned 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. If you use a graphical user interface for the debugger you can skip the next sections.

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
```

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
```

continue

Continue program being debugged, after breakpoint.

finish

Execute till the end of the current function.

delete [n]

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

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.

step

Step program until it reaches a different source line.

next

Step program, proceeding through subroutine calls.

run

Start debugged program.

ptype variable

Print type information of the variable.

print variable

print value of variable.

file filename

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

frame

print information about current frame.

set srcmode

Toggle between C source & assembly source.

! 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.

quit.

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

4.6 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
;;
```

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 an 'unsigned 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 always 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 _moduint()), and the result will be casted to a char. 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 opportunities).

- Have a look at the assembly listing to get a "feeling" for the code generation.

5.1 Notes on MCS51 memory layout

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

- Bytes 00-1F - 32 bytes to hold up to 4 banks of the registers R0 to R7,
- Bytes 20-2F - 16 bytes to hold 128 bit variables and,
- Bytes 30-7F - 80 bytes for general purpose use.

Additionally some members of the MCS51 family may have up to 128 bytes of additional, indirectly addressable, internal RAM memory (*idata*). Furthermore, some chips may have some built in external memory (*xdata*) which should not be confused with the internal, directly addressable RAM memory (*data*). Sometimes this built in *xdata* memory has to be activated before using it (you can probably find this information on the datasheet of the microcontroller your are using).

Normally SDCC will only use the first bank of registers (register bank 0), 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

byte of allocated memory for variables. For example, if the first 2 banks of registers are used, and only four bytes are used for *data* variables, it will position the base of the internal stack at address 20 (0x14). 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 80 bytes for general purpose use. If any bit variables are used, the data variables will be placed after the byte holding the last bit variable. For example, if register banks 0 and 1 are used, and there are 9 bit variables (two bytes used), *data* variables will be placed starting at address 0x22. You can also use --data-loc to specify the start address of the *data* and --iram-size to specify the size of the total internal RAM (*data+idata*).

By default the 8051 linker will place the stack after the last byte of data variables. Option --stack-loc allows you to specify the start of the stack, i.e. you could start it after any data in the general purpose area. If your microcontroller has additional indirectly addressable internal RAM (*idata*) you can place the stack on it. You may also need to use --xdata-loc to set the start address of the external RAM (*xdata*) and --xram-size to specify its size. Same goes for the code memory, using --code-loc and --code-size. If in doubt, don't specify any options and see if the resulting memory layout is appropriate, then you can adjust it.

The 8051 linker generates two files with memory allocation information. The first, with extension .map shows all the variables and segments. The second with extension .mem shows the final memory layout. The linker will complaint either if memory segments overlap, there is not enough memory, or there is not enough space for stack. If you get any linking warnings and/or errors related to stack or segments allocation, take a look at either the .map or .mem files to find out what the problem is. The .mem file may even suggest a solution to the problem.

5.2 Tools included in the distribution

Name	Purpose	Directory
uCsim	Simulator for various architectures	sdcc/sim/ucsim
keil2sdcc.pl	header file conversion	sdcc/support/scripts
mh2h.c	header file conversion	sdcc/support/scripts
as-gbz80	Assembler	sdcc/bin
as-z80	Assembler	sdcc/bin
asx8051	Assembler	sdcc/bin
sdcdb	Simulator	sdcc/bin
aslink	Linker	sdcc/bin
link-z80	Linker	sdcc/bin
link-gbz80	Linker	sdcc/bin
packihx	ihx packer	sdcc/bin

5.3 Related open source tools

Name	Purpose	Where to get
gpsim	PIC simulator	http://www.dattalo.com/gnupic/gpsim.html
flP5	PIC programmer	http://digilander.libero.it/fbradasc/FLP5.html
srecord	Object file conversion, checksumming, ...	http://srecord.sourceforge.net/
objdump	Object file conversion, ...	Part of binutils (should be there anyway)
doxygen	Source code documentation system	http://www.doxygen.org
splint	Statically checks c sources	http://www.splint.org
ddd	Debugger, serves nicely as GUI to sdcdb (Unix only)	http://www.gnu.org/software/ddd/

5.4 Related documentation / recommended reading

Name	Subject / Title	Where to get
S. S. Muchnick	Advanced Compiler Design and Implementation	bookstore
c-refcard.pdf	C Reference Card, 2 pages	http://www.refcards.com/about/c.html
test_suite_spec.pdf	sdcc regression test	sdcc/doc
cdbfileformat.pdf	sdcc internal documentation	sdcc/doc

6 Support

SDCC has grown to be a large project. The compiler alone (without the preprocessor, assembler and linker) is well over 100,000 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.

The SDCC project is hosted on the sdcc sourceforge site at http://sourceforge.net/projects/sdcc. You'll find the complete set of mailing lists, forums, bug reporting system, patch submission system, download area and cvs code repository there.

6.1 Reporting Bugs

The recommended way of reporting bugs is using the infrastructure of the sourceforge site. You can follow the status of bug reports there and have an overview about the known bugs.

Bug reports are automatically forwarded to the developer mailing list and will be fixed ASAP. When reporting a bug, it is very useful to include a small test program (the smaller the better) 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. When reporting a bug please maker sure you:

1. Attach the code you are compiling with SDCC.
2. Specify the exact command you use to run SDCC, or attach your Makefile.
3. Specify the SDCC version (type "sdcc -v"), your platform, and operating system.
4. Provide an exact copy of any error message or incorrect output.
5. Put something meaningful in the subject of your message.

Please attempt to include these 5 important parts, as applicable, in all requests for support or when reporting any problems or bugs with SDCC. Though this will make your message lengthy, it will greatly improve your chance that SDCC users and developers will be able to help you. Some SDCC developers are frustrated by bug reports without code provided that they can use to reproduce and ultimately fix the problem, so please be sure to provide sample code if you are reporting a bug!

Please have a short check that you are using a recent version of SDCC and the bug is not yet known. This is the link for reporting bugs: http://sourceforge.net/tracker/?group_id=599&atid=100599.

6.2 Requesting Features

Like bug reports feature requests are forwarded to the developer mailing list. This is the link for requesting features: http://sourceforge.net/tracker/?group_id=599&atid=350599.

6.3 Getting Help

These links should take you directly to the Mailing lists http://sourceforge.net/mail/?group_id=599[3] and the Forums http://sourceforge.net/forum/?group_id=599, lists and forums are archived so if you are lucky someone already had a similar problem.

6.4 ChangeLog

You can follow the status of the cvs version of SDCC by watching the file ChangeLog http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/*checkout*/sdcc/sdcc/ChangeLog?rev=HEAD&content-type=text/plain in the cvs-repository.

[3] Traffic on sdcc-devel and sdcc-user is about 100 mails/month each not counting automated messages (mid 2003)

6.5 Release policy

Historically there often were long delays between official releases and the sourceforge download area tends to get not updated at all. Current excuses might refer to problems with live range analysis, but if this is fixed, the next problem rising is that another excuse will have to be found. Kidding aside, we have to get better there! On the other hand there are daily snapshots available at snap http://sdcc.sourceforge.net/snap.php, and you can always built the very last version (hopefully with many bugs fixed, and features added) from the source code available at Source http://sdcc.sourceforge.net/snap.php#Source.

6.6 Examples

You'll find some small examples in the directory sdcc/device/examples/

6.7 Quality control

The compiler is passed through nightly compile and build checks. The so called *regression tests* check that SDCC itself compiles flawlessly on several platforms and checks the quality of the code generated by SDCC by running the code through simulators. There is a separate document *test_suite.pdf* about this.

You'll find the test code in the directory *sdcc/support/regression*. You can run these tests manually by running make in this directory (or f.e. make test-mcs51 if you don't want to run the complete tests). The test code might also be interesting if you want to look for examples checking corner cases of SDCC or if you plan to submit patches.

The pic port uses a different set of regression tests, you'll find them in the directory *sdcc/src/regression*.

7 SDCC Technical Data

7.1 Optimizations

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

7.1.1 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.

7.1.2 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;
}
```

7.1.3 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.

7.1.4 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.

7.1.5 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.

7.1.6 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.

7.1.7 '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) {
   case 4:  ...
   case 5:  ...
   case 3:  ...
   case 6:  ...
}
```

```
switch (i) {
    case 0:  ...
    case 1:  ...
    case 2:  ...
    case 3:  ...
}
```

  Both the above switch statements will be implemented using a jump-table. The example to the right side is slightly more efficient as the check for the lower boundary of the jump-table is not needed.

- 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. You might also consider dummy cases 0 and 5 to 8 in this example. The pragma NOJTBOUND can be used to turn off checking the *j*ump *t*able *bound*aries.

7.1.8 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).

7.1.9 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 */
```

7.1.10 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 23                   63          rl    a
000D 54 01                64          anl   a,#0x01
000F F5*02                65          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.

7.1.11 Peephole 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. An example of this (not a good one, it has side effects) is 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. Currently implemented are *labelInRange, labelRefCount, labelIsReturnOnly, operandsNotSame, xramMovcOption, 24bitMode, portIsDS390, 24bitModeAndPortDS390* and *notVolatile*.

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 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.

7.2 Library Routines

<pending: this is messy and incomplete>

1. Compiler support routines (_gptrget, _mulint etc)
2. Stdclib functions (puts, printf, strcat etc)
3. Math functions (sin, pow, sqrt etc)

Libraries included in SDCC should have a license at least as liberal as the GNU Lesser General Public License *LGPL*.

7.3 External Stack

The external stack (--xstack option) 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.

7.4 ANSI-Compliance

Deviations from the compliance:

- 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) /* 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()' */
```

7.5 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.

7.6 Other Processors

7.6.1 MCS51 variants

MCS51 processors are available from many vendors and come in many different flavours. While they might differ considerably in respect to Special Function Registers the core MCS51 is usually not modified or is kept compatible.

pdata access by SFR

With the upcome of devices with internal xdata and flash memory devices using port P2 as dedicated I/O port is becoming more popular. Switching the high byte for pdata access which was formerly done by port P2 is then achieved by a Special Function Register. In well-established MCS51 tradition the address of this *sfr* is where the chip designers decided to put it. As pdata addressing is used in the startup code for the initialization of xdata variables a separate startup code should be used as described in section 3.12.

Other Features available by SFR

Some MCS51 variants offer features like Double DPTR, multiple DPTR, decrementing DPTR, 16x16 Multiply. These are currently not used for the MCS51 port. If you absolutely need them you can fall back to inline assembly or submit a patch to SDCC.

7.6.2 The Z80 and gbz80 port

SDCC can target both the Zilog and the Nintendo Gameboy's Z80-like gbz80. The Z80 port is passed through the same *regressions tests* as MCS51 and DS390 ports, so floating point support, support for long variables and bitfield support is fine.

As always, the code is the authoritative 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 variables. IY is currently unused. 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.

7.7 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.

8 Compiler internals

8.1 The anatomy of the compiler

This is an excerpt from an article published in Circuit Cellar Magazine in august 2000. It's a little outdated (the compiler is much more efficient now and user/developer friendly), but pretty well exposes the guts of it all.

The current version of SDCC can generate code for Intel 8051 and Z80 MCU. It is fairly easy to retarget for other 8-bit MCU. Here we take a look at some of the internals of the compiler.

Parsing Parsing the input source file and creating an AST (Annotated Syntax Tree). This phase also involves propagating types (annotating each node of the parse tree with type information) and semantic analysis. There are some MCU specific parsing rules. For example the storage classes, the extended storage classes are MCU specific while there may be a xdata storage class for 8051 there is no such storage class for z80 or Atmel AVR. SDCC allows MCU specific storage class extensions, i.e. xdata will be treated as a storage class specifier when parsing 8051 C code but will be treated as a C identifier when parsing z80 or ATMEL AVR C code.

Generating iCode Intermediate code generation. In this phase the AST is broken down into three-operand form (iCode). These three operand forms are represented as doubly linked lists. ICode is the term given to the intermediate form generated by the compiler. ICode example section shows some examples of iCode generated for some simple C source functions.

Optimizations. Bulk of the target independent optimizations is performed in this phase. The optimizations include constant propagation, common sub-expression elimination, loop invariant code movement, strength reduction of loop induction variables and dead-code elimination.

Live range analysis During intermediate code generation phase, the compiler assumes the target machine has infinite number of registers and generates a lot of temporary variables. The live range computation determines the lifetime of each of these compiler-generated temporaries. A picture speaks a thousand words. ICode example sections show the live range annotations for each of the operand. It is important to note here, each iCode is assigned a number in the order of its execution in the function. The live ranges are computed in terms of these numbers. The from number is the number of the iCode which first defines the operand and the to number signifies the iCode which uses this operand last.

Register Allocation The register allocation determines the type and number of registers needed by each operand. In most MCUs only a few registers can be used for indirect addressing. In case of 8051 for example the registers R0 & R1 can be used to indirectly address the internal ram and DPTR to indirectly address the external ram. The compiler will try to allocate the appropriate register to pointer variables if it can. ICode example section shows the operands annotated with the registers assigned to them. The compiler will try to keep operands in registers as much as possible; there are several schemes the compiler uses to do achieve this. When the compiler runs out of registers the compiler will check to see if there are any live operands which is not used or defined in the current basic block being processed, if there are any found then it will push that operand and use the registers in this block, the operand will then be popped at the end of the basic block.

There are other MCU specific considerations in this phase. Some MCUs have an accumulator; very short-lived operands could be assigned to the accumulator instead of general-purpose register.

Code generation Figure II gives a table of iCode operations supported by the compiler. The code generation involves translating these operations into corresponding assembly code for the processor. This sounds overly simple but that is the essence of code generation. Some of the iCode operations are generated on a MCU specific manner for example, the z80 port does not use registers to pass parameters so the SEND and RECV iCode operations will not be generated, and it also does not support JUMPTABLES.
<Where is Figure II ?>

ICode Example This section shows some details of iCode. The example C code does not do anything useful; it is used as an example to illustrate the intermediate code generated by the compiler.

```
1. xdata int * p;
2. int gint;
3. /* This function does nothing useful.  It is used
4.    for the purpose of explaining iCode */
5. short function (data int *x)
6. {
7.   short i=10; /* dead initialization eliminated */
8.   short sum=10; /* dead initialization eliminated */
9.   short mul;
10.  int j ;
11.  while (*x) *x++ = *p++;
12.    sum = 0 ;
13.  mul = 0;
14.  /* compiler detects i,j to be induction variables */
15.  for (i = 0, j = 10 ; i < 10 ; i++, j--) {
16.    sum += i;
17.    mul += i * 3; /* this multiplication remains */
18.    gint += j * 3;/* this multiplication changed to addition */
19.  }
20.  return sum+mul;
21. }
```

In addition to the operands each iCode contains information about the filename and line it corresponds to in the source file. The first field in the listing should be interpreted as follows:

Filename(linenumber: iCode Execution sequence number : ICode hash table key : loop depth of the iCode).

Then follows the human readable form of the ICode operation. Each operand of this triplet form can be of three basic types a) compiler generated temporary b) user defined variable c) a constant value. Note that local variables and parameters are replaced by compiler generated temporaries. Live ranges are computed only for temporaries (i.e. live ranges are not computed for global variables). Registers are allocated for temporaries only. Operands are formatted in the following manner:

Operand Name [lr live-from : live-to] { type information } [registers allocated].

As mentioned earlier the live ranges are computed in terms of the execution sequence number of the iCodes, for example

the iTemp0 is live from (i.e. first defined in iCode with execution sequence number 3, and is last used in the iCode with sequence number 5). For induction variables such as iTemp21 the live range computation extends the lifetime from the start to the end of the loop.

The register allocator used the live range information to allocate registers, the same registers may be used for different temporaries if their live ranges do not overlap, for example r0 is allocated to both iTemp6 and to iTemp17 since their live ranges do not overlap. In addition the allocator also takes into consideration the type and usage of a temporary, for example itemp6 is a pointer to near space and is used as to fetch data from (i.e. used in GET_VALUE_AT_ADDRESS) so it is allocated a pointer registers (r0). Some short lived temporaries are allocated to special registers which have meaning to the code generator e.g. iTemp13 is allocated to a pseudo register CC which tells the back end that the temporary is used only for a conditional jump the code generation makes use of this information to optimize a compare and jump ICode.

There are several loop optimizations performed by the compiler. It can detect induction variables iTemp21(i) and iTemp23(j). Also note the compiler does selective strength reduction, i.e. the multiplication of an induction variable in line 18 (gint = j * 3) is changed to addition, a new temporary iTemp17 is allocated and assigned a initial value, a constant 3 is then added for each iteration of the loop. The compiler does not change the multiplication in line 17 however since the processor does support an 8 * 8 bit multiplication.

Note the dead code elimination optimization eliminated the dead assignments in line 7 & 8 to I and sum respectively.

```
Sample.c (5:1:0:0) _entry($9) :
Sample.c(5:2:1:0) proc _function [lr0:0]{function short}
```

```
Sample.c(11:3:2:0) iTemp0 [lr3:5]{_near * int}[r2] = recv
Sample.c(11:4:53:0) preHeaderLbl0($11) :
Sample.c(11:5:55:0) iTemp6 [lr5:16]{_near * int}[r0] := iTemp0 [lr3:5]{_near * int}[r2]
Sample.c(11:6:5:1) _whilecontinue_0($1) :
Sample.c(11:7:7:1) iTemp4 [lr7:8]{int}[r2 r3] = @[iTemp6 [lr5:16]{_near * int}[r0]]
Sample.c(11:8:8:1) if iTemp4 [lr7:8]{int}[r2 r3] == 0 goto _whilebreak_0($3)
Sample.c(11:9:14:1) iTemp7 [lr9:13]{_far * int}[DPTR] := _p [lr0:0]{_far * int}
Sample.c(11:10:15:1) _p [lr0:0]{_far * int} = _p [lr0:0]{_far * int} + 0x2 {short}
Sample.c(11:13:18:1) iTemp10 [lr13:14]{int}[r2 r3] = @[iTemp7 [lr9:13]{_far * int}[DPTR]]
Sample.c(11:14:19:1) *(iTemp6 [lr5:16]{_near * int}[r0]) := iTemp10 [lr13:14]{int}[r2 r3]
Sample.c(11:15:12:1) iTemp6 [lr5:16]{_near * int}[r0] = iTemp6 [lr5:16]{_near * int}[r0] + 0x2 {short}
Sample.c(11:16:20:1) goto _whilecontinue_0($1)
Sample.c(11:17:21:0)_whilebreak_0($3) :
Sample.c(12:18:22:0) iTemp2 [lr18:40]{short}[r2] := 0x0 {short}
Sample.c(13:19:23:0) iTemp11 [lr19:40]{short}[r3] := 0x0 {short}
Sample.c(15:20:54:0)preHeaderLbl1($13) :
Sample.c(15:21:56:0) iTemp21 [lr21:38]{short}[r4] := 0x0 {short}
Sample.c(15:22:57:0) iTemp23 [lr22:38]{int}[r5 r6] := 0xa {int}
Sample.c(15:23:58:0) iTemp17 [lr23:38]{int}[r7 r0] := 0x1e {int}
Sample.c(15:24:26:1)_forcond_0($4) :
Sample.c(15:25:27:1) iTemp13 [lr25:26]{char}[CC] = iTemp21 [lr21:38]{short}[r4] < 0xa {short}
Sample.c(15:26:28:1) if iTemp13 [lr25:26]{char}[CC] == 0 goto _forbreak_0($7)
Sample.c(16:27:31:1) iTemp2 [lr18:40]{short}[r2] = iTemp2 [lr18:40]{short}[r2] + ITemp21 [lr21:38]{short}[r4]
Sample.c(17:29:33:1) iTemp15 [lr29:30]{short}[r1] = iTemp21 [lr21:38]{short}[r4] * 0x3 {short}
Sample.c(17:30:34:1) iTemp11 [lr19:40]{short}[r3] = iTemp11 [lr19:40]{short}[r3] + iTemp15 [lr29:30]{short}[r1]
Sample.c(18:32:36:1:1) iTemp17 [lr23:38]{int}[r7 r0]= iTemp17 [lr23:38]{int}[r7 r0]- 0x3 {short}
Sample.c(18:33:37:1) _gint [lr0:0]{int} = _gint [lr0:0]{int} + iTemp17 [lr23:38]{int}[r7 r0]
Sample.c(15:36:42:1) iTemp21 [lr21:38]{short}[r4] = iTemp21 [lr21:38]{short}[r4] + 0x1 {short}
Sample.c(15:37:45:1) iTemp23 [lr22:38]{int}[r5 r6]= iTemp23 [lr22:38]{int}[r5 r6]- 0x1 {short}
Sample.c(19:38:47:1) goto _forcond_0($4)
Sample.c(19:39:48:0)_forbreak_0($7) :
Sample.c(20:40:49:0) iTemp24 [lr40:41]{short}[DPTR] = iTemp2 [lr18:40]{short}[r2] + ITemp11 [lr19:40]{short}[r3]
Sample.c(20:41:50:0) ret iTemp24 [lr40:41]{short}
Sample.c(20:42:51:0)_return($8) :
Sample.c(20:43:52:0) eproc _function [lr0:0]{ ia0 re0 rm0}{function short}
```

Finally the code generated for this function:

```
.area DSEG (DATA)
_p::
  .ds 2
_gint::
  .ds 2
; sample.c 5
; ——————————————————————————-
; function function
; ——————————————————————————-
_function:
; iTemp0 [lr3:5]{_near * int}[r2] = recv
  mov r2,dpl
; iTemp6 [lr5:16]{_near * int}[r0] := iTemp0 [lr3:5]{_near * int}[r2]
  mov ar0,r2
;_whilecontinue_0($1) :
00101$:
; iTemp4 [lr7:8]{int}[r2 r3] = @[iTemp6 [lr5:16]{_near * int}[r0]]
; if iTemp4 [lr7:8]{int}[r2 r3] == 0 goto _whilebreak_0($3)
  mov ar2,@r0
  inc r0
  mov ar3,@r0
  dec r0
  mov a,r2
  orl a,r3
  jz 00103$
00114$:
; iTemp7 [lr9:13]{_far * int}[DPTR] := _p [lr0:0]{_far * int}
  mov dpl,_p
  mov dph,(_p + 1)
; _p [lr0:0]{_far * int} = _p [lr0:0]{_far * int} + 0x2 {short}
```

```
  mov a,#0x02
  add a,_p
  mov _p,a
  clr a
  addc a,(_p + 1)
  mov (_p + 1),a
; iTemp10 [lr13:14]{int}[r2 r3] = @[iTemp7 [lr9:13]{_far * int}[DPTR]]
  movx a,@dptr
  mov r2,a
  inc dptr
  movx a,@dptr
  mov r3,a
; *(iTemp6 [lr5:16]{_near * int}[r0]) := iTemp10 [lr13:14]{int}[r2 r3]
  mov @r0,ar2
  inc r0
  mov @r0,ar3
; iTemp6 [lr5:16]{_near * int}[r0] =
; iTemp6 [lr5:16]{_near * int}[r0] +
; 0x2 {short}
  inc r0
; goto _whilecontinue_0($1)
  sjmp 00101$
; _whilebreak_0($3) :
00103$:
; iTemp2 [lr18:40]{short}[r2] := 0x0 {short}
  mov r2,#0x00
; iTemp11 [lr19:40]{short}[r3] := 0x0 {short}
  mov r3,#0x00
; iTemp21 [lr21:38]{short}[r4] := 0x0 {short}
  mov r4,#0x00
; iTemp23 [lr22:38]{int}[r5 r6] := 0xa {int}
  mov r5,#0x0A
  mov r6,#0x00
; iTemp17 [lr23:38]{int}[r7 r0] := 0x1e {int}
  mov r7,#0x1E
  mov r0,#0x00
; _forcond_0($4) :
00104$:
; iTemp13 [lr25:26]{char}[CC] = iTemp21 [lr21:38]{short}[r4] < 0xa {short}
; if iTemp13 [lr25:26]{char}[CC] == 0 goto _forbreak_0($7)
  clr c
  mov a,r4
  xrl a,#0x80
  subb a,#0x8a
  jnc 00107$
00115$:
; iTemp2 [lr18:40]{short}[r2] = iTemp2 [lr18:40]{short}[r2] +
; iTemp21 [lr21:38]{short}[r4]
  mov a,r4
  add a,r2
  mov r2,a
; iTemp15 [lr29:30]{short}[r1] = iTemp21 [lr21:38]{short}[r4] * 0x3 {short}
  mov b,#0x03
  mov a,r4
  mul ab
  mov r1,a
; iTemp11 [lr19:40]{short}[r3] = iTemp11 [lr19:40]{short}[r3] +
; iTemp15 [lr29:30]{short}[r1]
  add a,r3
  mov r3,a
; iTemp17 [lr23:38]{int}[r7 r0]= iTemp17 [lr23:38]{int}[r7 r0]- 0x3 {short}
  mov a,r7
  add a,#0xfd
  mov r7,a
  mov a,r0
  addc a,#0xff
  mov r0,a
; _gint [lr0:0]{int} = _gint [lr0:0]{int} + iTemp17 [lr23:38]{int}[r7 r0]
  mov a,r7
```

```
  add a,_gint
  mov _gint,a
  mov a,r0
  addc a,(_gint + 1)
  mov (_gint + 1),a
; iTemp21 [lr21:38]{short}[r4] = iTemp21 [lr21:38]{short}[r4] + 0x1 {short}
  inc r4
; iTemp23 [lr22:38]{int}[r5 r6]= iTemp23 [lr22:38]{int}[r5 r6]- 0x1 {short}
  dec r5
  cjne r5,#0xff,00104$
  dec r6
; goto _forcond_0($4)
  sjmp 00104$
; _forbreak_0($7) :
00107$:
; ret iTemp24 [lr40:41]{short}
  mov a,r3
  add a,r2
  mov dpl,a
; _return($8) :
00108$:
  ret
```

8.2 A few words about basic block successors, predecessors and dominators

Successors are basic blocks that might execute after this basic block.
Predecessors are basic blocks that might execute before reaching this basic block.
Dominators are basic blocks that WILL execute before reaching this basic block.

[basic block 1]
if (something)
  [basic block 2]
else
  [basic block 3]
[basic block 4]

a) succList of [BB2] = [BB4], of [BB3] = [BB4], of [BB1] = [BB2,BB3]
b) predList of [BB2] = [BB1], of [BB3] = [BB1], of [BB4] = [BB2,BB3]
c) domVect of [BB4] = BB1 ... here we are not sure if BB2 or BB3 was executed but we are SURE that BB1 was executed.

9 Acknowledgments

http://sdcc.sourceforge.net#Who

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
All product names mentioned herein may be trademarks of their respective companies.

10 Alphabetical index

To avoid confusion, the installation and building options for sdcc itself (chapter 2) are not part of the index.

Index