SDCC Compiler User Guide

SDCC 2.5.1
$Date: 2005/07/12 17:44:51 $
$Revision: 1.121 $

Contents

Chapter 1

Introduction

1.1 About SDCC

SDCC (*S*mall *D*evice *C C*ompiler) is an open source, 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.), Dallas DS80C390 variants, Freescale (formerly Motorola) HC08 and Zilog Z80 based MCUs. 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, an open source 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 *open source* 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 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 (see section 8.2 for ANSI-Compliance).

- 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.
- bit and sbit types now consistently behave like the C99 _Bool type with respect to type conversion. The most common incompatibility resulting from this change is related to bit toggling idioms, e.g.:
  ```
  bit b;
  b = ~b; /* equivalent to b=1 instead of toggling b */
  b = !b; /* toggles b */
  ```
  In previous versions, both forms would have toggled the bit.

<pending: more incompatibilities?>

[1] Obviously this has pros and cons

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. A pdf version of this document is available at http://sdcc.sourceforge.net/doc/sdccman.pdf. 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 section 7.2 about filing feature requests.

Chapter 2

Installing SDCC

For most users it is sufficient to skip to either section 2.4.1 or section 2.4.9. 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

--disable-z80-port Excludes the z80 port

--disable-avr-port Excludes the AVR port

--disable-ds390-port Excludes the DS390 port

--disable-hc08-port Excludes the HC08 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.

Moreover the install path can be changed by defining DESTDIR:

```
make install DESTDIR=$(HOME)/sdcc.rpm/
```

Please note that DESTDIR must have a trailing slash!

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, it will be named something like sdcc.src.tar.gz http://sdcc.sourceforge.net/snap.php.
2. Bring up a command line terminal, such as xterm.
3. Unpack the file using a command like: **"tar -xvzf 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. Proceed with section 2.7.

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. Although a tabulator spacing of 8 is a sensible choice for programmers (it’s a power of 2 and allows to display 8/16 bit signed variables without loosing columns) the plan is to move towards using only spaces in the source.

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.

WARNING: Visual studio is very picky with line terminations; it expects the 0x0d, 0x0a DOS style line endings, not the 0x0a Unix style line endings. If you are getting a message such as "This makefile was not generated by Developer Studio etc. etc.” when opening the sdcc.dsw workspace or any of the *.dsp projects, then you need to convert the Unix style line endings to DOS style line endings. To do so you can use the ”unix2dos” utility freely available on the internet. Doug Hawkins reported in the sdcc-user list that this works:

C:\Programming\SDCC> unix2dos sdcc.dsw
C:\Programming\SDCC> for /R %I in (*.dsp) do @unix2dos "%I"

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 ZIP Package

1. Download the binary zip package from http://sdcc.sf.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.

2.4.9 Windows Install Using the Setup Program

Download the setup program *sdcc-x.y.z-setup.exe* for an official release from http://sf.net/project/showfiles.php?group_id=599 or a setup program for one of the snapshots *sdcc_yyyymmdd_setup.exe* from http://sdcc.sf.net/snap.php and execute it. A windows typical installer will guide you through the installation process.

2.5 Building the Documentation

If the necessary tools (LyX, LaTeX, LaTeX2HTML) are installed it is as easy as changing into the doc directory and typing **"make"** there. You're invited to make changes and additions to this manual (sdcc/doc/sdccman.lyx). Using LyX http://www.lyx.org as editor this is straightforward. Prebuilt documentation in html and pdf format is available from http://sdcc.sf.net/snap.php.

2.6 Reading the Documentation

Currently reading the document in pdf format is recommended, as for unknown reason the hyperlinks are working there whereas in the html version they are not[1].
You'll find the pdf version at http://sdcc.sf.net/doc/sdccman.pdf.
A html version should be online at http://sdcc.sf.net/doc/sdccman.html/index.html.
This documentation is in some aspects different from a commercial documentation:

- It tries to document SDCC for several processor architectures in one document (commercially these probably would be separate documents/products). This document currently matches SDCC for mcs51 and DS390 best and does give too few information about f.e. Z80, PIC14, PIC16 and HC08.
- There are many references pointing away from this documentation. Don't let this distract you. If there f.e. was a reference like http://www.opencores.org together with a statement "some processors which are targetted by SDCC can be implemented in a *f*ield *p*rogrammable *g*ate *a*rray" we expect you to have a quick look there and come back. If you read this you are on the right track.
- Some sections attribute more space to problems, restrictions and warnings than to the solution.
- The installation section and the section about the debugger is intimidating.
- There are still lots of typos and there are more different writing styles than pictures.

2.7 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 section 2.8 Install trouble-shooting 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 section 2.8 Install

[1] If you should know why please drop us a note

trouble-shooting 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 section 2.8 Install trouble-shooting section for suggestions). Use option **--print-search-dirs** to find exactly where SDCC is looking for the include and lib files.

2.8 Install Trouble-shooting

2.8.1 If 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.

2.8.2 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.

2.8.3 What the ”make” does

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

2.8.4 What the ”make install” command does.

This will install the compiler, other executables libraries and include files into the appropriate directories. See sections 2.2, 2.3 about install and search paths.
On most systems you will need super-user privileges to do this.

2.9 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.

2.9.1 sdcc - The Compiler

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

2.9.2 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.

2.9.3 as*xxxx*, aslink, link-*xxx* - 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.

2.9.4 s51 - The Simulator

S51 is a freeware, opensource simulator developed by Daniel Drotos. 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.

2.9.5 sdcdb - Source Level Debugger

Sdcdb is the companion source level debugger. More about sdcdb in section 5. The current version of the debugger uses Daniel's Simulator S51, but can be easily changed to use other simulators.

Chapter 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 or sourcefile.o - 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*). Both formats are documented in the documentation of 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. The format is documented in cdbfileformat.pdf
- sourcefile. - (no extension) An optional AOMF or AOMF51 file containing debug information (generated with option --debug). The (Intel) *a*bsolute *o*bject *m*odule *f*ormat is commonly used by third party tools (debuggers, simulators, emulators)
- sourcefile.dump* - Dump file to debug the compiler it self (generated with option --dumpall) (see section 3.2.9 and section 9.1 ”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.

3.1.4 Using sdcclib to Create and Manage Libraries

Alternatively, instead of having a .rel file for each entry on the library file as described in the preceding section, sdcclib can be used to embed all the modules belonging to such library in the library file itself. This results in a larger library file, but it greatly reduces the number of disk files accessed by the linker. Additionally, the packed library file contains an index of all include modules and symbols that significantly speeds up the linking process. To display a list of options supported by sdcclib type:

```
sdcclib -?
```

To create a new library file, start by compiling all the required modules. For example:

```
sdcc -c _divsint.c
sdcc -c _divuint.c
sdcc -c _modsint.c
sdcc -c _moduint.c
sdcc -c _mulint.c
```

This will create files _divsint.rel, _divuint.rel, _modsint.rel, _moduint.rel, and _mulint.rel. The next step is to add the .rel files to the library file:

sdcclib libint.lib _divsint.rel
sdcclib libint.lib _divuint.rel
sdcclib libint.lib _modsint.rel
sdcclib libint.lib _moduint.rel
sdcclib libint.lib _mulint.rel

If the file already exists in the library, it will be replaced. To see what modules and symbols are included in the library, options -s and -m are available. For example:

sdcclib -s libint.lib

```
_divsint.rel:
   __divsint_a_1_1
   __divsint_PARM_2
   __divsint
_divuint.rel:
   __divuint_a_1_1
   __divuint_PARM_2
   __divuint_reste_1_1
   __divuint_count_1_1
   __divuint
_modsint.rel:
   __modsint_a_1_1
   __modsint_PARM_2
   __modsint
_moduint.rel:
   __moduint_a_1_1
   __moduint_PARM_2
   __moduint_count_1_1
   __moduint
_mulint.rel:
   __mulint_PARM_2
   __mulint
```

If the source files are compiled using --debug, the corresponding debug information file .adb will be include in the library file as well. The library files created with sdcclib are plain text files, so they can be viewed with a text editor. It is not recomended to modify a library file created with sdcclib using a text editor, as there are file indexes numbers located accross the file used by the linker to quickly locate the required module to link. Once a .rel file (as well as a .adb file) is added to a library using sdcclib, it can be safely deleted, since all the information required for linking is embedded in the library file itself. Library files created using sdcclib are used as described in the preceding sections.

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.

-mhc08 Generate code for the Freescale/Motorola HC08 family of processors.

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

-mgbz80 Generate code for the GameBoy Z80 processor (Not actively maintained).

-mavr Generate code for the Atmel AVR processor (In development, not complete). AVR users should probably have a look at winavr http://sourceforge.net/projects/winavr or http://www.avrfreaks.net/index.php?name=PNphpBB2&file=index.

-mpic14 Generate code for the Microchip PIC 14-bit processors (p16f84 and variants. In development, not complete).

-mpic16 Generate code for the Microchip PIC 16-bit processors (p18f452 and variants. In development, not complete).

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

-mxa51 Generate code for the Phillips XA51 processor (Not maintained, 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.

-Wp preprocessorOption[,preprocessorOption]... Pass the preprocessorOption to the preprocessor `sdcpp`. SDCC uses an adapted version of the preprocessor cpp of the GNU Compiler Collection (gcc), if you need more dedicated options please refer to the documentation at http://www.gnu.org/software/gcc/onlinedocs/.

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. The **--pack-iram** option (which is now a default setting) will override this setting, so you should also specify the **--no-pack-iram** option if you need to manually place the stack.

--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 of the 8051, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, eg. --idata-loc 0x88 or --idata-loc 136.

--bit-loc <Value> The start location of the bit addressable internal ram of the 8051. This is *not* implemented yet. Instead an option can be passed directly to the linker: -Wl -bBSEG=<Value>.

--out-fmt-ihx The linker output (final object code) is in Intel Hex format. This is the default option. The format itself is documented in the documentation of srecord.

--out-fmt-s19 The linker output (final object code) is in Motorola S19 format. The format itself is documented in the documentation of srecord.

--out-fmt-elf The linker output (final object code) is in ELF format. (Currently only supported for the HC08 processors)

-Wl linkOption[,linkOption]... Pass the linkOption to the linker. See file sdcc/as/doc/asxhtm.html for more on linker options.

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 have to 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 3.17.1.2 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.

--stack-size <Value> Causes the linker to check if there is at minimum <Value> bytes for stack.

--pack-iram Causes the linker to use unused register banks for data variables and pack data, idata and stack together. This is the default now.

--no-pack-iram Causes the linker to use old style for allocating memory areas.

3.2.5 DS390 / DS400 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.

--protect-sp-update disable interrupts during ESP:SP updates.

--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*).

--stack-probe insert call to function __stack_probe at each function prologue.

--tini-libid <nnnn> LibraryID used in -mTININative.

--use-accelerator generate code for DS390 Arithmetic Accelerator.

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 (spill *loc*ations, sloc). A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It is 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 Loop Invariants in section 8.1.4. It is 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 8.1.7 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.

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

--no-peep Disable peep-hole optimization.

--peep-file <filename> This option can be used to use additional rules to be used by the peep hole optimizer. See section 8.1.12 Peep Hole optimizations for details on how to write these rules.

--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.

--opt-code-speed The compiler will optimize code generation towards fast code, possibly at the expense of code size.

--opt-code-size The compiler will optimize code generation towards compact code, possibly at the expense of code speed.

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 3.6 Parameters and Local Variables for more details. If this option is used all source files in the project should be compiled with this option. It automatically implies –int-long-reent and –float-reent.

--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. The debug information collected in a file with .cdb extension can be used with the SDCDB. For more information see documentation for SDCDB. Another file with no extension contains debug information in AOMF or AOMF51 format which is commonly used by third party tools.

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

--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.

--main-return This option can be used if the code generated is called by a monitor program or if the main routine includes an endless loop. This option might result in slightly smaller code and save two bytes of stack space. The return from the 'main' function will return to the function calling main. The default setting is to lock up i.e. generate a '`sjmp .`'.

--nostdinc 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 :)

--no-peep-comments Will not include peep-hole comments in the generated files.

--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!).

--disable-warning <nnnn> Disable specific warning with number <nnnn>.

--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.

-Wa asmOption[,asmOption]... Pass the asmOption to the assembler. See file sdcc/as/doc/asxhtm.html for assembler options.cd

--std-sdcc89 Generally follow the C89 standard, but allow SDCC features that conflict with the standard (default).

--std-c89 Follow the C89 standard and disable SDCC features that conflict with the standard.

--std-sdcc99 Generally follow the C99 standard, but allow SDCC features that conflict with the standard (incomplete support).

--std-c99 Follow the C99 standard and disable SDCC features that conflict with the standard (incomplete support).

more-pedantic Actually this is ***not*** a SDCC compiler option but if you want *more* warnings you can use a separate tool dedicated to syntax checking like splint http://www.splint.org. To make your source files parseable by splint you will have to include lint.h in your source file and add brackets around extended keywords (like "__at (0xab)" and "__interrupt (2)").
Splint has an excellent on line manual at http://www.splint.org/manual/ and it's capabilities go beyond pure syntax checking. You'll need to tell splint the location of SDCC's include files so a typical command line could look like this:
splint -I /usr/local/share/sdcc/include/mcs51/ myprogram.c

3.2.9 Intermediate Dump Options

The following options are provided for the purpose of retargetting and debugging the compiler. They provide a means to dump the intermediate code (iCode) generated by the compiler in human readable form at various stages of the compilation process. More on iCodes see chapter 9.1 "The anatomy of the compiler".

--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. Additionally, 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 section 2.2 " Install Paths".

SDCC_INCLUDE Path, see section 2.3 "Search Paths".

SDCC_LIB Path, see section 2.3 "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 Storage Class Language Extensions

3.4.1 MCS51/DS390 Storage Class Language Extensions

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

3.4.1.1 data / near

This is the **default** storage class for the Small Memory model (*data* and *near* can be used synonymously). 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
```

3.4.1.2 xdata / far

Variables declared with this storage class will be placed in the external RAM. This is the **default** storage class for the 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
```

3.4.1.3 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
```

Please note, the first 128 byte of idata physically access the same RAM as the data memory. The original 8051 had 128 byte idata memory, nowadays most devices have 256 byte idata memory. The stack is located in idata memory.

3.4.1.4 pdata

Paged xdata access is just as straightforward as using the other addressing modes of a 8051. It is typically located at the start of xdata and has a maximum size of 256 bytes. The following example writes 0x01 to the pdata variable. 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, see section 4.1).

```
pdata unsigned char test_pdata;
```

Writing 0x01 to this variable generates the assembly code:

```
78r00      mov r0,#_test_pdata
74 01      mov a,#0x01
F2         movx @r0,a
```

If the --xstack option is used the pdata memory area is followed by the xstack memory area and the sum of their sizes is limited to 256 bytes.

3.4.1.5 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
```

`char` indexed arrays of characters in code memory can be accessed efficiently:

```
code char test_array[] = {'c','h','e','a','p'};
```

Read access to this array using an 8-bit unsigned index generates the assembly code:

```
E5*00      mov a,_index
90s00r41   mov dptr,#_test_array
93         movc a,@a+dptr
```

3.4.1.6 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
```

The bit addressable memory consists of 128 bits which are located from 0x20 to 0x2f in data memory.
Apart from this 8051 specific storage class most architectures support ANSI-C bitfields[1]. In accordance with ISO/IEC 9899 bits and bitfields without an explicit signed modifier are implemented as unsigned.

3.4.1.7 sfr / sfr16 / sfr32 / sbit

Like the bit keyword, *sfr / sfr16 / sfr32 / sbit* signify both a data-type and storage class, they are used to describe the *s*pecial *f*unction *r*egisters and *s*pecial *bit* variables of a 8051, eg:

```
sfr at 0x80 P0;  /* special function register P0 at location 0x80 */
/* 16 bit special function register combination for timer 0 */
/* with the high byte at location 0x8C and the low byte at location 0x8A */
sfr16 at 0x8C8A TMR0;
sbit at 0xd7 CY; /* CY (Carry Flag) */
```

Special function registers which are located on an address dividable by 8 are bit-addressable, an *sbit* addresses a specific bit within these sfr.
16 Bit and 32 bit special function register combinations which require a certain access order are better not declared using *sfr16* or *sfr32*. Allthough SDCC usually accesses them Least Significant Byte (LSB) first, this is not guaranteed.

[1] Not really meant as examples, but nevertheless showing what bitfields are about: device/include/mc68hc908qy.h and support/regression/tests/bitfields.c

3.4.1.8 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;

/* the following is a function pointer physically located in data space */
char (* data fp)(void);
```

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.

3.4.1.9 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, see also section 3.11 Startup-Code).

Normally SDCC will only use the first bank of registers (register bank 0), but it is possible to specify that other banks of registers (keyword *using*) 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 in unused register banks and 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 from address 0x10 to 0x20 and continue 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 (i)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 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 complain 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.

3.4.2 Z80/Z180 Storage Class Language Extensions

3.4.2.1 sfr (in/out to 8-bit addresses)

The Z80 family has separate address spaces for memory and *i*nput/*o*utput memory. I/O memory is accessed with special instructions, e.g.:

```
sfr at 0x78 IoPort;  /* define a var in I/O space at 78h called IoPort */
```

Writing 0x01 to this variable generates the assembly code:

```
3E 01      ld a,#0x01
D3 78      out (_IoPort),a
```

3.4.2.2 banked sfr (in/out to 16-bit addresses)

The keyword *banked* is used to support 16 bit addresses in I/O memory e.g.:

```
sfr banked at 0x123 IoPort;
```

Writing 0x01 to this variable generates the assembly code:

```
01 23 01   ld bc,#_IoPort
3E 01      ld a,#0x01
ED 79      out (c),a
```

3.4.2.3 sfr (in0/out0 to 8 bit addresses on Z180/HD64180)

The compiler option --portmode=180 (80) and a compiler #pragma portmode=z180 (z80) is used to turn on (off) the Z180/HD64180 port addressing instructions `in0/out0` instead of `in/out`. If you include the file z180.h this will be set automatically.

3.4.3 HC08 Storage Class Language Extensions

3.4.3.1 data

The data storage class declares a variable that resides in the first 256 bytes of memory (the direct page). The HC08 is most efficient at accessing variables (especially pointers) stored here.

3.4.3.2 xdata

The xdata storage class declares a variable that can reside anywhere in memory. This is the default if no storage class is specified.

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 be located at 0x7ffe and 0x7fff of the external ram. The compiler does *not* 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. Variables with an absolute address are *not* initialized.

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

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

For some architectures (mcs51) array accesses are more efficient if an (xdata/far) array starts at a block (256 byte) boundary (section 3.12.1 has an example).
Absolute addresses 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 MOSI;    /* master out, slave in */
extern volatile bit MISO;    /* master in, slave out */
extern volatile bit MCLK;    /* master clock */

/* Input and Output of a byte on a 3-wire serial bus.
   If needed adapt polarity of clock, polarity of data and bit order
 */
unsigned char spi_io(unsigned char out_byte)
{
    unsigned char i=8;
    do {
        MOSI = out_byte & 0x80;
        out_byte <<= 1;
        MCLK = 1;
        /* _asm nop _endasm; */        /* for slow peripherals */
        if(MISO)
            out_byte += 1;
        MCLK = 0;
    } while(--i);
    return out_byte;
}
```

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

```
bit at 0x80 MOSI;    /* I/O port 0, bit 0 */
bit at 0x81 MISO;    /* I/O port 0, bit 1 */
bit at 0x82 MCLK;    /* I/O port 0, bit 2 */
```

Similarly, for the second hardware you would use

```
bit at 0x83 MOSI;    /* I/O port 0, bit 3 */
bit at 0x91 MISO;    /* I/O port 1, bit 1 */
bit at 0x92 MCLK;    /* I/O port 1, bit 2 */
```

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 similar to *static* so by default functions are non-reentrant.

They can be placed on the stack by using the *--stack-auto* option, by using *#pragma stackauto* 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 interrupt service routine. The #pragma nooverlay ensures that the parameters and local variables for the function are NOT overlayed.

3.8 Interrupt Service Routines

3.8.1 General Information

SDCC allows *i*nterrupt *s*ervice *r*outines 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. 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*. The optional *using* keyword can be used to tell the compiler to use the specified register bank (8051 specific) when generating code for this function.

Interrupt service routines open the door for some very interesting bugs:
If an interrupt service routine changes variables which are accessed by other functions these variables have to be declared *volatile*.

If the access to these variables is not *atomic* (i.e. the processor needs more than one instruction for the access and could be interrupted while accessing the variable) the interrupt must be disabled during the access to avoid inconsistent data. Access to 16 or 32 bit variables is obviously not atomic on 8 bit CPUs and should be protected by disabling interrupts. You're not automatically on the safe side if you use 8 bit variables though. We need an example here: f.e. on the 8051 the harmless looking "`flags |= 0x80;`" is not atomic if `flags` resides in xdata. Setting "`flags |= 0x40;`" from within an interrupt routine might get lost if the interrupt occurs at the wrong time. "`counter += 8;`" is not atomic on the 8051 even if `counter` is located in data memory. Bugs like these are hard to reproduce and can cause a lot of trouble.

The return address and the registers used in the interrupt service routine are saved on the stack so there must be sufficient stack space. If there isn't variables or registers (or even the return address itself) will be corrupted. This *stack overflow* is most likely to happen if the interrupt occurs during the "deepest" subroutine when the stack is already in use for f.e. many return addresses.

A special note here, int (16 bit) and long (32 bit) integer division, multiplication & modulus and floating-point 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-reent* compiler option.

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

Also see section 3.7 about Overlaying and section 3.10 about Functions using private register banks.

3.8.2 MCS51/DS390 Interrupt Service Routines

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* & psw are saved 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.

3.8.3 HC08 Interrupt Service Routines

Since the number of interrupts available is chip specific and the interrupt vector table always ends at the last byte of memory, the interrupt numbers corresponds to the interrupt vectors in reverse order of address. For example, interrupt 1 will use the interrupt vector at 0xfffc, interrupt 2 will use the interrupt vector at 0xfffa, and so on. However, interrupt 0 (the reset vector at 0xfffe) is not redefinable in this way; instead see section 3.11 for details on customizing startup.

3.8.4 Z80 Interrupt Service Routines

The Z80 uses several different methods for determining the correct interrupt vector depending on the hardware implementation. Therefore, SDCC ignores the optional interrupt number and does not attempt to generate an interrupt vector table.

By default, SDCC generates code for a maskable interrupt, which uses an RETI instruction to return from the interrupt. To write an interrupt handler for the non-maskable interrupt, which needs an RETN instruction instead, add the *critical* keyword:

```
void nmi_isr (void) critical interrupt
{
    ...
}
```

3.9 Enabling and Disabling Interrupts

3.9.1 Critical Functions and Critical Statements

A special keyword may be associated with a block or a function declaring it as *critical*. SDCC will generate code to disable all interrupts upon entry to a critical function and restore the interrupt enable to the previous state before returning. Nesting critical functions will need one additional byte on the stack for each call.

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

The critical attribute maybe used with other attributes like *reentrant.*
The keyword *critical* may also be used to disable interrupts more locally:

```
critical{ i++; }
```

More than one statement could have been included in the block.

3.9.2 Enabling and Disabling Interrupts directly

Interrupts can also be disabled and enabled directly (8051):

```
EA = 0;            or:          EA_SAVE = EA;
...                             EA = 0;
EA = 1;                         ...
                                EA = EA_SAVE;
```

On other architectures which have seperate opcodes for enabling and disabling interrupts you might want to make use of defines with inline assembly (HC08):

```
#define CLI _asm  cli  _endasm;
#define SEI _asm  sei  _endasm;
...
```

Note: it is sometimes sufficient to disable only a specific interrupt source like f.e. a timer or serial interrupt by manipulating an *interrupt mask* register.

Usually the time during which interrupts are disabled should be kept as short as possible. This minimizes both *interrupt latency* (the time between the occurrence of the interrupt and the execution of the first code in the interrupt routine) and *interrupt jitter* (the difference between the shortest and the longest interrupt latency). These really are something different, f.e. a serial interrupt has to be served before its buffer overruns so it cares for the maximum interrupt latency, whereas it does not care about jitter. On a loudspeaker driven via a digital to analog converter which is fed by an interrupt a latency of a few milliseconds might be tolerable, whereas a much smaller jitter will be very audible.

You can reenable interrupts within an interrupt routine and on some architectures you can make use of two (or more) levels of *interrupt priorities*. On some architectures which don't support interrupt priorities these can be implemented by manipulating the interrupt mask and reenabling interrupts within the interrupt routine. Check there is sufficient space on the stack and don't add complexity unless you have to.

3.9.3 Semaphore locking (mcs51/ds390)

Some architectures (mcs51/ds390) have an atomic bit test and clear instruction. These type of instructions are typically used in preemptive multitasking systems, where a routine f.e. claims the use of a data structure ('acquires a lock on it'), makes some modifications and then releases the lock when the data structure is consistent again. The instruction may also be used if interrupt and non-interrupt code have to compete for a resource. With the atomic bit test and clear instruction interrupts don't have to be disabled for the locking operation.

SDCC generates this instruction if the source follows this pattern:

```
volatile bit resource_is_free;

if (resource_is_free)
  {
    resource_is_free=0;
    ...
    resource_is_free=1;
  }
```

Note, mcs51 and ds390 support only an atomic bit test and *clear* instruction (as opposed to atomic bit test and *set).*

3.10 Functions using private register banks (mcs51/ds390)

Some architectures have support for quickly changing register sets. SDCC supports this feature with the *using* attribute (which tells the compiler to use a register bank other than the default bank zero). It should only be applied to *interrupt* functions (see footnote 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 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.

3.11 Startup Code

3.11.1 MCS51/DS390 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. On some mcs51 variants xdata memory has to be explicitly enabled before it can be accessed or if the watchdog needs to be disabled, this is the place to do it. The startup code clears all internal data memory, 256 bytes by default, but from 0 to n-1 if *--iram-sizen* is used. (recommended for Chipcon CC1010).

See also the compiler option *--no-xinit-opt* and section 4.1 about MCS51-variants.

3.11.2 HC08 Startup Code

The HC08 startup code follows the same scheme as the MCS51 startup code.

3.11.3 Z80 Startup Code

On the Z80 the startup code is inserted by linking with crt0.o which is generated from sdcc/device/lib/z80/crt0.s. If you need a different startup code you can use the compiler option *--no-std-crt0* and provide your own crt0.o.

3.12 Inline Assembler Code

3.12.1 A Step by Step Introduction

Starting from a small snippet of c-code this example shows for the MCS51 how to use inline assembly, access variables, a function parameter and an array in xdata memory. The example uses an MCS51 here but is easily adapted for other architectures. This is a buffer routine which should be optimized:

[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

```
unsigned char far at 0x7f00 buf[0x100];
unsigned char head,tail;

void to_buffer( unsigned char c )
{
    if( head != tail-1 )
        buf[ head++ ] = c;    /* access to a 256 byte aligned array */
}
```

If the code snippet (assume it is saved in buffer.c) is compiled with SDCC then a corresponding buffer.asm file is generated. We define a new function `to_buffer_asm()` in file buffer.c in which we cut and paste the generated code, removing unwanted comments and some ':'. Then add "_asm" and "_endasm;" to the beginning and the end of the function body:

```
/* With a cut and paste from the .asm file, we have something to start with.
   The function is not yet OK! (registers aren't saved) */
void to_buffer_asm( unsigned char c )
{
    _asm
    mov  r2,dpl
;buffer.c if( head != tail-1 )
    mov  a,_tail
    dec  a
    mov  r3,a
    mov  a,_head
    cjne a,ar3,00106$
    ret
00106$:
;buffer.c buf[ head++ ] = c; /* access to a 256 byte aligned array */
    mov  r3,_head
    inc  _head
    mov  dpl,r3
    mov  dph,#(_buf >> 8)
    mov  a,r2
    movx @dptr,a
00103$:
    ret
    _endasm;
}
```

The new file buffer.c should compile with only one warning about the unreferenced function argument 'c'. Now we hand-optimize the assembly code and insert an #define USE_ASSEMBLY (1) and finally have:

```
unsigned char far at 0x7f00 buf[0x100];
unsigned char head,tail;
#define USE_ASSEMBLY (1)

#if !USE_ASSEMBLY

void to_buffer( unsigned char c )
{
    if( head != tail-1 )
        buf[ head++ ] = c;
}

#else

void to_buffer( unsigned char c )
```

```
{
    c; // to avoid warning:  unreferenced function argument
    _asm
        ; save used registers here.
        ; If we were still using r2,r3 we would have to push them here.
; if( head != tail-1 )
        mov  a,_tail
        dec  a
        xrl  a,_head
        ; we could do an ANL a,#0x0f here to use a smaller buffer (see below)
        jz   t_b_end$
        ;
; buf[ head++ ] = c;
        mov  a,dpl        ; dpl holds lower byte of function argument
        mov  dpl,_head    ; buf is 0x100 byte aligned so head can be used directly
        mov  dph,#(_buf>>8)
        movx @dptr,a
        inc  _head
        ; we could do an ANL _head,#0x0f here to use a smaller buffer (see above)
t_b_end$:
        ; restore used registers here
    _endasm;
}
#endif
```

The inline assembler code can contain any valid code understood by the assembler, this includes any assembler directives and comment lines[3]. The compiler does not do any validation of the code within the `_asm ... _endasm;` keyword pair. Specifically it will not know which registers are used and thus register pushing/popping has to be done manually.

It is 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. There are only a few (if any) cases where this option makes sense, it might cause some unexpected changes in the inline assembler code. Please go through the peephole optimizer rules defined in file *SDCCpeeph.def* before using this option.

3.12.2 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 ; does not change flags, no need to save psw
```

[3]The assembler does not like some characters like ':' or "' in comments. You'll find an 100+ pages assembler manual in sdcc/as/doc/asxhtm.html

```
      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 ; does not change flags, no need to save psw
    reti            ; MUST explicitly include ret or reti in _naked function
```

The related directive #pragma exclude allows a more fine grained control over 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.

3.12.3 Use of Labels within Inline Assembler

SDCC allows the use of in-line assembler with a few restrictions regarding labels. In older versions of the compiler all labels defined within inline assembler code *had 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*).

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

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 referenced 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, i.e. labels defines in inline assembly can not be accessed by C statements.

3.13 Interfacing with Assembler Code

3.13.1 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).

3.13.2 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    dph,#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 a one byte parameter, "dptr" for two bytes, "b,dptr" for three bytes and "acc,b,dptr" for a four bytes parameter. 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
```

3.13.3 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 leftmost 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
    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.

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

For signed & unsigned int (16 bit) and long (32 bit) variables, division, multiplication and modulus operations are implemented by support routines. These support routines are all developed in ANSI-C to facilitate porting to other MCUs, although some model specific assembler 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-reent* option. Notice that you don't have to call these routines directly. The compiler will use them automatically every time an integer operation is required.

3.15 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 consist 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

These support routines are developed in ANSI-C so there is room for space and speed improvement[4]. Note if all these routines are used simultaneously the data space might overflow. For serious floating point usage the large model might be needed. 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.

3.16 Library Routines

<pending: this is messy and incomplete - a little more information is in sdcc/doc/libdoc.txt >

3.16.1 Compiler support routines (_gptrget, _mulint etc.)

3.16.2 Stdclib functions (puts, printf, strcat etc.)

3.16.2.1 <stdio.h>

As usual on embedded systems you have to provide your own `getchar()` and `putchar()` routines. SDCC does not know whether the system connects to a serial line with or without handshake, LCD, keyboard or other device. You'll find examples for serial routines f.e. in sdcc/device/lib.

The default `printf()` implementation in `printf_large.c` does not support float (except on ds390). To enable this recompile it with the option *-DUSE_FLOATS=1* on the command line. Use*--model-large* for the mcs51 port, since this uses a lot of memory.

If you're short on memory you might want to use `printf_small()` *instead* of `printf()`. For the mcs51 there additionally are assembly versions `printf_tiny()` and `printf_fast()` and `printf_fast_f()` which should fit the requirements of many embedded systems (printf_fast() can be customized by unsetting #defines to *not* support long variables and field widths).

[4] The floating point routines for the mcs51 are implemented in assembler

3.16.3 Math functions (sin, pow, sqrt etc.)

3.16.4 Other libraries

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

If you have ported some library or want to share experience about some code which f.e. falls into any of these categories Busses (I^2C, CAN, Ethernet, Profibus, Modbus, USB, SPI, JTAG ...), Media (IDE, Memory cards, eeprom, flash...), En-/Decryption, Remote debugging, Realtime kernel, Keyboard, LCD, RTC, FPGA, PID then the sdcc-user mailing list http://sourceforge.net/mail/?group_id=599 would certainly like to hear about it. Programmers coding for embedded systems are not especially famous for being enthusiastic, so don't expect a big hurray but as the mailing list is searchable these references are very valuable. Let's help to create a climate where information is shared.

3.17 Memory Models

3.17.1 MCS51 Memory Models

3.17.1.1 Small and Large

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 recommended that the small model be used unless absolutely required.

3.17.1.2 External Stack

The external stack (--xstack option) is located in pdata memory (usually at the start of the external ram segment) and uses all unused space in pdata (max. 256 bytes). 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. There is a predefined target in the library makefile).

The compiler outputs the higher order address byte of the external ram segment into port P2 (see also section 4.1), therefore when using the External Stack option, this port *may not* be used by the application program.

3.17.2 DS390 Memory Model

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.

3.18 Pragmas

SDCC supports the following #pragma directives:

- save - this will save all current options to the save/restore stack. See #pragma 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 #pragma save.
- 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.
- exclude none | {acc[,b[,dpl[,dph]]] - The exclude pragma disables the generation of pairs of push/pop instructions in *I*nterrupt *S*ervice *R*outines. 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. See also the related keyword _naked.
- less_pedantic - the compiler will not warn you anymore for obvious mistakes, you'r on your own now ;-(
- disable_warning <nnnn> - the compiler will not warn you anymore about warning number <nnnn>.
- nogcse - will stop global common subexpression elimination.
- noinduction - will stop loop induction optimizations.
- noinvariant - will not do loop invariant optimizations. For more details see Loop Invariants in section8.1.4.
- 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). More elegantly this can be achieved by obmitting the optional interrupt number after the interrupt keyword, see section 3.8 about interrupts.
- nojtbound - will not generate code for boundary value checking, when switch statements are turned into jump-tables (dangerous). For more details see section 8.1.7.
- noloopreverse - Will not do loop reversal optimization
- nooverlay - the compiler will not overlay the parameters and local variables of a function.
- stackauto- See option --stack-auto and section 3.6 Parameters and Local Variables.
- opt_code_speed - The compiler will optimize code generation towards fast code, possibly at the expense of code size.
- opt_code_size - The compiler will optimize code generation towards compact code, possibly at the expense of code speed.
- opt_code_balanced - The compiler will attempt to generate code that is both compact and fast, as long as meeting one goal is not a detriment to the other (this is the default).
- std_sdcc89 - Generally follow the C89 standard, but allow SDCC features that conflict with the standard (default).

- std_c89 - Follow the C89 standard and disable SDCC features that conflict with the standard.
- std_sdcc99 - Generally follow the C99 standard, but allow SDCC features that conflict with the standard (incomplete support).
- std_c99 - Follow the C99 standard and disable SDCC features that conflict with the standard (incomplete support).

SDCPP supports the following #pragma directives:

- preproc_asm (+ | -) - switch _asm _endasm block preprocessing on / off. Default is on.

The pragma's are intended to be used to turn-on or 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.

3.19 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, __ds390, __hc08, __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

Chapter 4

Notes on supported Processors

4.1 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.

4.1.1 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. Needless to say that they didn't agree on a common name either. So that the startup code can correctly initialize xdata variables, you should define an sfr with the name _XPAGE at the appropriate location if the default, port P2, is not used for this. Some examples are:

```
sfr at 0x92 _XPAGE; /* Cypress EZ-USB family */
sfr at 0xaf _XPAGE; /* some Silicon Labs (Cygnal) chips */
sfr at 0xaa _XPAGE; /* some Silicon Labs (Cygnal) chips */
```

For more exotic implementations further customizations may be needed. See section 3.11 for other possibilities.

4.1.2 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.

4.2 DS400 port

The DS80C400 microcontroller has a rich set of peripherals. In its built-in ROM library it includes functions to access some of the features, among them is a TCP stack with IP4 and IP6 support. Library headers (currently in beta status) and other files are provided at ftp://ftp.dalsemi.com/pub/tini/ds80c400/c_libraries/sdcc/index.html.

4.3 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 the MCS51 and DS390 ports, so floating point support, support for long variables and bitfield support is fine. See mailing lists and forums about interrupt routines.

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 and IY are used as a temporary registers, and BC and DE are available for holding variables. Return values for the Z80 port are stored in L (one byte), HL (two bytes), or DEHL (four bytes). The gbz80 port use the same set of registers for the return values, but in a different order of significance: E (one byte), DE (two bytes), or HLDE (four bytes).

4.4 The HC08 port

The port to the Motorola HC08 family has been added in October 2003, and is still undergoing some basic development. The code generator is complete, but the register allocation is still quite unoptimized. Some of the SDCC's standard C library functions have embedded non-HC08 inline assembly and so are not yet usable.

4.5 The PIC14 port

The 14bit PIC port still requires a major effort from the development community. However it can work for very simple code.

4.5.1 C code and 14bit PIC code page and RAM banks

The linker organizes allocation for the code page and RAM banks. It does not have intimate knowledge of the code flow. It will put all the code section of a single asm file into a single code page. In order to make use of multiple code pages, separate asm files must be used. The compiler treats all functions of a single C file as being in the same code page unless it is non static. The compiler treats all local variables of a single C file as being in the same RAM bank unless it is an extern.

To get the best follow these guide lines:

1. make local functions static, as non static functions require code page selection overhead.
2. Make local variables static as extern variables require RAM bank selection overhead.
3. For devices that have multiple code pages it is more efficient to use the same number of files as pages, i.e. for the 16F877 use 4 separate files and i.e. for the 16F874 use 2 separate files. This way the linker can put the code for each file into different code pages and the compiler can allocate reusable variables more efficiently and there's less page selection overhead. And as for any 8 bit micro (especially for PIC 14 as they have a very simple instruction set) use 'unsigned char' whereever possible instead of 'int'.

4.5.2 Creating a device include file

For generating a device include file use the support perl script inc2h.pl kept in directory support/script.

4.5.3 Interrupt code

For the interrupt function, use the keyword 'interrupt' with level number of 0 (PIC14 only has 1 interrupt so this number is only there to avoid a syntax error - it ought to be fixed). E.g.:

```
void Intr(void) interrupt 0
{
  T0IF = 0; /* Clear timer interrupt */
}
```

4.5.4 Linking and assembling

For assembling you can use either GPUTILS' gpasm.exe or MPLAB's mpasmwin.exe. GPUTILS is available from http://sourceforge.net/projects/gputils. For linking you can use either GPUTIL's gplink or MPLAB's mplink.exe. If you use MPLAB and an interrupt function then the linker script file vectors section will need to be enlarged to link with mplink.

Here is a `Makefile` using GPUTILS:

```
.c.o:
        sdcc -S -V -mpic14 -p16F877 $<
        gpasm -c $*.asm

$(PRJ).hex:  $(OBJS)
        gplink -m -s $(PRJ).lkr -o $(PRJ).hex $(OBJS)
```

Here is a `Makefile` using MPLAB:

```
.c.o:
        sdcc -S -V -mpic14 -p16F877 $<
        mpasmwin /q /o $*.asm

$(PRJ).hex:  $(OBJS)
        mplink /v $(PRJ).lkr /m $(PRJ).map /o $(PRJ).hex $(OBJS)
```

Please note that indentations within a `Makefile` have to be done with a tabulator character.

4.6 The PIC16 port

The PIC16 port is the portion of SDCC that is responsible to produce code for the Microchip(TM) microcontrollers with 16 bit core. Currently this family of microcontrollers contains the PIC18Fxxx and PIC18Fxxxx. Currently supported devices are:

18F242	18F248	18F252	18F258	18F442	18F448
18F452	18F458	18F1220	18F2220	18F2550	18F4331
18F4455	18F6520	18F6620	18F6680	18F6720	18F8520
18F8620	18F8680	18F8720			

4.6.1 Global Options

PIC16 port supports the standard command line arguments as supposed, with the exception of certain cases that will be mentioned in the following list:

--callee-saves See --all-callee-saves

--all-callee-saves All function arguments are passed on stack by default. *There is no need to specify this in the command line.*

--fommit-frame-pointer Frame pointer will be omitted when the function uses no local variables.

4.6.2 Port Specific Options

The port specific options appear after the global options in the sdcc –help output.

4.6.2.1 General Options

General options enable certain port features and optimizations.

--stack-model=[model] Used in conjuction with the command above. Defines the stack model to be used, valid stack models are :

- *small* Selects small stack model. 8 bit stack and frame pointers. Supports 256 bytes stack size.
- *large* Selects large stack model. 16 bit stack and frame pointers. Supports 65536 bytes stack size.

--preplace-udata-with=[kword] Replaces the default udata keyword for allocating unitialized data variables with [kword]. Valid keywords are: "udata_acs", "udata_shr", "udata_ovr".

--ivt-loc <nnnn> positions the Interrupt Vector Table at location <nnnn>. Useful for bootloaders.

--asm= sets the full path and name of an external assembler to call.

--link= sets the full path and name of an external linker to call.

--mplab-comp MPLAB compatibility option. Currently only suppresses special gpasm directives.

4.6.2.2 Optimization Options

--optimize-goto Try to use (conditional) BRA instead of GOTO

--optimize-cmp Try to optimize some compares.

--obanksel=nn Set optimization level for inserting BANKSELs.

0	no optimization
1	checks previous used register and if it is the same then does not emit BANKSEL, accounts only for labels.
2	tries to check the location of (even different) symbols and removes BANKSELs if they are in the same bank. *Important: There might be problems if the linker script has data sections across bank borders!*

4.6.2.3 Linking Options

--nodefaultlibs do not link default libraries when linking

--no-crt Don't link the default run-time modules

--use-crt= Use a custom run-time module instead of the defaults.

4.6.2.4 Debugging Options

Debugging options enable extra debugging information in the output files.

--debug-xtra Similar to --debug, but dumps more information.

--debug-ralloc Force register allocator to dump <source>.d file with debugging information. <source> is the name of the file compiled.

--pcode-verbose Enable pcode debugging information in translation.

--denable-peeps Force the usage of peepholes. Use with care.

--gstack Trace push/pops for stack pointer overflow

--call-tree dump call tree in .calltree file

4.6.3 Enviromental Variables

There is a number of enviromental variables that can be used when running SDCC to enable certain optimizations or force a specific program behaviour. these variables are primarily for debugging purposes so they can be enabled/disabled at will.

Currently there is only two such variables available:

OPTIMIZE_BITFIELD_POINTER_GET when this variable exists reading of structure bitfields is optimized by directly loading FSR0 with the address of the bitfield structure. Normally SDCC will cast the bitfield structure to a bitfield pointer and then load FSR0. This step saves data ram and code space for functions that perform heavy use of bitfields. (ie. 80 bytes of code space are saved when compiling malloc.c with this option).

NO_REG_OPT do not perform pCode registers optimization. This should be used for debugging purposes. In some where bugs in the pcode optimizer are found, users can benefit from temporarily disabling the optimizer until the bug is fixed.

4.6.4 Preprocessor Macros

PIC16 port defines the following preprocessor macros while translating a source.

Macro	Description
SDCC_pic16	Port identification
__pic16	Port identification (same as above)
pic18fxxxx	MCU Identification. *xxxx* is the microcontrol identification number, i.e. 452, 6620, etc
__18Fxxxx	MCU Identification (same as above)
STACK_MODEL_nnn	nnn = SMALL or LARGE respectively according to the stack model used

In addition the following macros are defined when calling assembler:

Macro	Description
__18Fxxxx	MCU Identification. *xxxx* is the microcontrol identification number, i.e. 452, 6620, etc
SDCC_MODEL_nnn	nnn = SMALL or LARGE respectively according to the memory model used for SDCC
STACK_MODEL_nnn	nnn = SMALL or LARGE respectively according to the stack model used

4.6.5 Directories

PIC16 port uses the following directories for searching header files and libraries.

Directory	Description	Target	Command prefix
PREFIX/sdcc/include/pic16	PIC16 specific headers	Compiler	-I
PREFIX/sdcc/lib/pic16	PIC16 specific libraries	Linker	-L

4.6.6 Pragmas

PIC16 port currently supports the following pragmas:

stack pragma stack forces the code generator to initialize the stack & frame pointers at a specific address. This is an adhoc solution for cases where no STACK directive is available in the linker script or gplink is not instructed to create a stack section.
The stack pragma should be used only once in a project. Multiple pragmas may result in indeterminate behaviour of the program.[1]
The format is as follows:

```
#pragma stack bottom_address [stack_size]
```

bottom_address is the lower bound of the stack section. The stack pointer initially will point at address (bottom_address+stack_size-1).

```
Example:
/* initializes stack of 100 bytes at RAM address 0x200 */
#pragma stack 0x200 100
```

If the stack_size field is omitted then a stack is created with the default size of 64. This size might be enough for most programs, but its not enough for operations with deep function nesting or excessive stack usage.

wparam *This pragma is deprecated. Its use will cause a warning message to be issued.*

[1] The old format (ie. #pragma stack 0x5ff) is deprecated and will cause the stack pointer to cross page boundaries (or even exceed the available data RAM) and crash the program. Make sure that stack does not cross page boundaries when using the SMALL stack model.

code place a function symbol at static FLASH address

```
Example:
/* place function test_func at 0x4000 */
#pragma code test_func 0x4000
```

library instructs the linker to use a library module.
Usage:

```
#pragma library module_name
```

module_name can be any library or object file (including its path). Note that there are four reserved keywords which have special meaning. These are:

Keyword	Description	Module to link
ignore	ignore all library pragmas	*(none)*
c	link the C library	*libc18f*.lib
math	link the Math libarary	*libm18f*.lib
io	link the I/O library	*libio18f**.lib
debug	link the debug library	*libdebug*.lib

* is the device number, i.e. 452 for PIC18F452 MCU.

This feature allows for linking with specific libraries withoug having to explicit name them in the command line. Note that the IGNORE keyword will reject all modules specified by the library pragma.

udata pragma udata instructs the compiler to emit code so that linker will place a variable at a specific memory bank

```
Example:
/* places variable foo at bank2 */
#pragma udata bank2 foo
char foo;
```

In order for this pragma to work extra SECTION directives should be added in the .lkr script. In the following example a sample .lkr file is shown:

```
// Sample linker script for the PIC18F452 processor
LIBPATH .
CODEPAGE   NAME=vectors    START=0x0           END=0x29          PROTECTED
CODEPAGE   NAME=page       START=0x2A          END=0x7FFF
CODEPAGE   NAME=idlocs     START=0x200000      END=0x200007      PROTECTED
CODEPAGE   NAME=config     START=0x300000      END=0x30000D      PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE      END=0x3FFFFF      PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000      END=0xF000FF      PROTECTED
ACCESSBANK NAME=accessram  START=0x0           END=0x7F
DATABANK   NAME=gpr0       START=0x80          END=0xFF
DATABANK   NAME=gpr1       START=0x100         END=0x1FF
DATABANK   NAME=gpr2       START=0x200         END=0x2FF
DATABANK   NAME=gpr3       START=0x300         END=0x3FF
DATABANK   NAME=gpr4       START=0x400         END=0x4FF
DATABANK   NAME=gpr5       START=0x500         END=0x5FF
ACCESSBANK NAME=accesssfr  START=0xF80         END=0xFFF         PROTECTED
SECTION    NAME=CONFIG     ROM=config
SECTION    NAME=bank0      RAM=gpr0       # these SECTION directives
SECTION    NAME=bank1      RAM=gpr1       # should be added to link
SECTION    NAME=bank2      RAM=gpr2       # section name 'bank?' with
SECTION    NAME=bank3      RAM=gpr3       # a specific DATABANK name
SECTION    NAME=bank4      RAM=gpr4
SECTION    NAME=bank5      RAM=gpr5
```

The linker will recognise the section name set in the pragma statement and will position the variable at the memory bank set with the RAM field at the SECTION line in the linker script file.

4.6.7 Header Files

There is one main header file that can be included to the source files using the pic16 port. That file is the **pic18fregs.h**. This header file contains the definitions for the processor special registers, so it is necessary if the source accesses them. It can be included by adding the following line in the beginning of the file:

```
#include <pic18fregs.h>
```

The specific microcontroller is selected within the pic18fregs.h automatically, so the same source can be used with a variety of devices.

4.6.8 Libraries

The libraries that PIC16 port depends on are the microcontroller device libraries which contain the symbol definitions for the microcontroller special function registers. These libraries have the format pic18fxxxx.lib, where *xxxx* is the microcontroller identification number. The specific library is selected automatically by the compiler at link stage according to the selected device.

Libraries are created with gplib which is part of the gputils package http://sourceforge.net/projects/gputils.

Building the libraries

Before using SDCC/pic16 there are some libraries that need to be compiled. This process is not done automatically by SDCC since not all users use SDCC for pic16 projects. So each user should compile the libraries separately.

The steps to compile the pic16 libraries under Linux are:

```
cd device/lib/pic16
./configure
make
cd ..
make model-pic16
su -c 'make install'     # install the libraries, you need the root password
```

If you need to install the headers too, do:

```
cd device/include
su -c 'make install'     # install the headers, you need the root password
```

There exist a special target to build the I/O libraries. This target is not automatically build because it will build the I/O library for *every* supported device. This way building will take quite a lot of time. Users are advised to edit the **device/lib/pic16/pics.build** file and then execute:

```
make lib-io
```

4.6.9 Memory Models

The following memory models are supported by the PIC16 port:

- small model
- large model

Memory model affects the default size of pointers within the source. The sizes are shown in the next table:

Pointer sizes according to memory model	small model	large model
code pointers	16-bits	24-bits
data pointers	16-bits	16-bits

It is advisable that all sources within a project are compiled with the same memory model. If one wants to override the default memory model, this can be done by declaring a pointer as **far** or **near**. Far selects large memory model's pointers, while near selects small memory model's pointers.

The standard device libraries (see 4.6.7) contain no reference to pointers, so they can be used with both memory models.

4.6.10 Stack

The stack implementation for the PIC16 port uses two indirect registers, FSR1 and FSR2.

FSR1 is assigned as stack pointer

FSR2 is assigned as frame pointer

The following stack models are supported by the PIC16 port

- SMALL model
- LARGE model

SMALL model means that only the FSRxL byte is used to access stack and frame, while LARGE uses both FSRxL and FSRxH registers. The following table shows the stack/frame pointers sizes according to stack model and the maximum space they can address:

Stack & Frame pointer sizes according to stack model	small	large
Stack pointer FSR1	8-bits	16-bits
Frame pointer FSR2	8-bits	16-bits

LARGE stack model is currently not working properly throughout the code generator. So its use is not advised. Also there are some other points that need special care:

1. Do not create stack sections with size more than one physical bank (that is 256 bytes)
2. Stack sections should no cross physical bank limits (i.e. #pragma stack 0x50 0x100)

These limitations are caused by the fact that only FSRxL is modified when using SMALL stack model, so no more than 256 bytes of stack can be used. This problem will disappear after LARGE model is fully implemented.

4.6.11 Functions

In addition to the standard SDCC function keywords, PIC16 port makes available two more:

wparam Use the WREG to pass one byte of the first function argument. This improves speed but you may not use this for functions with arguments that are called via function pointers, otherwise the first byte of the first parameter will get lost. Usage:

```
void func_wparam(int a) wparam
{
    /* WREG hold the lower part of a */
    /* the high part of a is stored in FSR2+2 (or +3 for large stack model) */
...
}
```

This keyword replaces the deprecated wparam pragma.

shadowregs When entering/exiting an ISR, it is possible to take advantage of the PIC18F hardware shadow registers which hold the values of WREG, STATUS and BSR registers. This can be done by adding the keyword *shadowregs* before the *interrupt* keyword in the function's header.

```
void isr_shadow(void) shadowregs interrupt 1
{
...
}
```

shadowregs instructs the code generator not to store/restore WREG, STATUS, BSR when entering/exiting the ISR.

4.6.12 Function return values

Return values from functions are placed to the appropriate registers following a modified Microchip policy optimized for SDCC. The following table shows these registers:

size	destination register
8 bits	WREG
16 bits	PRODL:WREG
24 bits	PRODH:PRODL:WREG
32 bits	FSR0L:PRODH:PRODL:WREG
>32 bits	on stack, FSR0 points to the beginning

4.6.13 Interrupts

An interrupt servive routine (ISR) is declared using the *interrupt* keyword.

```
void isr(void) interrupt n
{
...
}
```

n is the interrupt number, which for PIC18F devices can be:

n	Interrupt Vector	Interrupt Vector Address
0	RESET vector	0x000000
1	HIGH priority interrupts	0x000008
2	LOW priority interrupts	0x000018

When generating assembly code for ISR the code generator places a GOTO instruction at the *Interrupt Vector Address* which points at the genetated ISR. This single GOTO instruction is part of an automatically generated *interrupt entry point* function. The actuall ISR code is placed as normally would in the code space. Upon interrupt request, the GOTO instruction is executed which jumps to the ISR code. When declaring interrupt functions as _naked this GOTO instruction is **not** generated. The whole interrupt functions is therefore placed at the Interrupt Vector Address of the specific interrupt. This is not a problem for the LOW priority interrupts, but it is a problem for the RESET and the HIGH priority interrupts because code may be written at the next interruptt's vector address and cause undeterminate program behaviour if that interrupt is raised.[2]

n is possible to be omitted. This way a function is generated similar to an ISR, but it is not assigned to any interrupt.

When entering an interrupt, currently the PIC16 port automatically saves the following registers:

- WREG
- STATUS
- BSR
- PROD (PRODL and PRODH)
- FSR0 (FSR0L and FSR0H)

These registers are restored upon return from the interrupt routine.[3]

[2] This is not a problem when

1. this is a HIGH interrupt ISR and LOW interrupts are *disabled* or not used.
2. when the ISR is small enough not to reach the next interruptt's vector address.

[3] NOTE that when the _naked attribute is specified for an interrupt routine, then NO registers are stored or restored.

4.6.14 Generic Pointers

Generic pointers are implemented in PIC16 port as 3-byte (24-bit) types. There are 3 types of generic pointers currently implemented data, code and eeprom pointers. They are differentiated by the value of the 7th and 6th bits of the upper byte:

pointer type	7th bit	6th bit	rest of the pointer	descrption
data	1	0	uuuuuu uuuuxxxx xxxxxxxx	a 12-bit data pointer in data RAM memory
code	0	0	uxxxxx xxxxxxxx xxxxxxxx	a 21-bit code pointer in FLASH memory
eeprom	0	1	uuuuuu uuuuuuxx xxxxxxxx	a 10-bit eeprom pointer in EEPROM memory
(unimplemented)	1	1	xxxxxx xxxxxxxx xxxxxxxx	unimplemented pointer type

Generic pointer are read and written with a set of library functions which read/write 1, 2, 3, 4 bytes.

4.6.15 PIC16 C Libraries

4.6.15.1 Standard I/O Streams

In the *stdio.h* the type FILE is defined as:

```
typedef char * FILE;
```

This type is the stream type implemented I/O in the PIC18F devices. Also the standard input and output streams are declared in stdio.h:

```
extern FILE * stdin;
extern FILE * stdout;
```

The FILE type is actually a generic pointer which defines one more type of generic pointers, the *stream* pointer. This new type has the format:

pointer type	<7:6>	<5>	<4>	<3:0>	rest of the pointer	descrption
stream	00	1	0	nnnn	uuuuuuuu uuuuuuuu	upper byte high nubble is 0x2n, the rest are zeroes

Currently implemented there are 3 types of streams defined:

stream type	value	module	description
STREAM_USART	0x200000UL	USART	Writes/Reads characters via the USART peripheral
STREAM_MSSP	0x210000UL	MSSP	Writes/Reads characters via the MSSP peripheral
STREAM_USER	0x2f0000UL	(none)	Writes/Reads characters via used defined functions

The stream identifiers are declared as macros in the stdio.h header.
In the libc library there exist the functions that are used to write to each of the above streams. These are

__stream_usart_putchar writes a character at the USART stream

__stream_mssp_putchar writes a character at the MSSP stream

putchar dummy function. This writes a character to a user specified manner.

In order to increase performance *putchar* is declared in stdio.h as having its parameter in WREG (it has the wparam keyword). In stdio.h exists the macro PUTCHAR(arg) that defines the putchar function in a user-friendly way. *arg* is the name of the variable that holds the character to print. An example follows:

```
#include <pic18fregs.h>
#include <stdio.h>

PUTCHAR( c )
{
```

```
    PORTA = c;    /* dump character c to PORTA */
}

void main(void)
{
    stdout = STREAM_USER;    /* this is not necessery, since stdout points
                              * by default to STREAM_USER */
    printf (ĺThis is a printf test\nĺ);
}
```

4.6.15.2 Printing functions

PIC16 contains an implementation of the printf-family of functions. There exist the following functions:

```
extern unsigned int sprintf(char *buf, char *fmt, ...);
extern unsigned int vsprintf(char *buf, char *fmt, va_list ap);
extern unsigned int printf(char *fmt, ...);
extern unsigned int vprintf(char *fmt, va_lista ap);
extern unsigned int fprintf(FILE *fp, char *fmt, ...);
extern unsigned int vfprintf(FILE *fp, char *fmt, va_list ap);
```

For sprintf and vsprintf *buf* should normally be a data pointer where the resulting string will be placed. No range checking is done so the user should allocate the necessery buffer. For fprintf and vfprintf *fp* should be a stream pointer (i.e. stdout, STREAM_MSSP, etc...).

4.6.15.3 Signals

The PIC18F family of microcontrollers supports a number of interrupt sources. A list of these interrupts is shown in the following table:

signal name	description	signal name	descritpion
SIG_RB	PORTB change interrupt	SIG_EE	EEPROM/FLASH write complete interrupt
SIG_INT0	INT0 external interrupt	SIG_BCOL	Bus collision interrupt
SIG_INT1	INT1 external interrupt	SIG_LVD	Low voltage detect interrupt
SIG_INT2	INT2 external interrupt	SIG_PSP	Parallel slave port interrupt
SIG_CCP1	CCP1 module interrupt	SIG_AD	AD convertion complete interrupt
SIG_CCP2	CCP2 module interrupt	SIG_RC	USART receive interrupt
SIG_TMR0	TMR0 overflow interrupt	SIG_TX	USART transmit interrupt
SIG_TMR1	TMR1 overflow interrupt	SIG_MSSP	SSP receive/transmit interrupt
SIG_TMR2	TMR2 matches PR2 interrupt		
SIG_TMR3	TMR3 overflow interrupt		

The prototypes for these names are defined in the header file *signal.h* .
In order to simplify signal handling, a number of macros is provided:

DEF_INTHIGH(name) begin the definition of the interrupt dispatch table for high priority interrupts. *name* is the function name to use.

DEF_INTLOW(name) begin the definition of the interrupt dispatch table fo low priority interrupt. *name* is the function name to use.

DEF_HANDLER(sig,handler) define a handler for signal *sig.*

END_DEF end the declaration of the dispatch table.

Additionally there are two more macros to simplify the declaration of the signal handler:

SIGHANDLER(handler) this declares the function prototype for the *handler* function.

SIGHANDLERNAKED(handler) same as SIGHANDLER() but declares a naked function.

An example of using the macros above is shown below:

```
#include <pic18fregs.h>
#include <signal.h>

DEF_INTHIGH(high_int)
DEF_HANDLER(SIG_TMR0, _tmr0_handler)
DEF_HANDLER(SIG_BCOL, _bcol_handler)
END_DEF

SIGHANDLER(_tmr0_handler)
{
  /* action to be taken when timer 0 overflows */
}

SIGHANDLERNAKED(_bcol_handler)
{
  _asm
    /* action to be taken when bus collision occurs */
    retfie
 _endasm;
}
```

NOTES: Special care should be taken when using the above scheme:

- do not place a colon (;) at the end of the DEF_* and END_DEF macros.
- when declaring SIGHANDLERNAKED handler never forget to use *retfie* for proper returning.

4.6.16 PIC16 Port – Tips

Here you can find some general tips for compiling programs with SDCC/pic16.

4.6.16.1 Stack size

The default stack size (that is 64 bytes) probably is enough for many programs. One must take care that when there are many levels of function nesting, or there is excessive usage of stack, its size should be extended. An example of such a case is the printf/sprintf family of functions. If you encounter problems like not being able to print integers, then you need to set the stack size around the maximum (256 for small stack model). The following diagram shows what happens when calling printf to print an integer:

```
printf () --> ltoa () --> ultoa () --> divschar ()
```

It is should be understood that stack is easily consumed when calling complicated functions. Using command line arguments like --fommit-frame-pointer might reduce stack usage by not creating unnecessery stack frames. Other ways to reduce stack usage may exist.

Chapter 5

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.

5.1 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.

5.2 How the Debugger Works

When the --debug option is specified the compiler generates extra symbol information some of which are put into 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.

5.3 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.

5.4 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.
- -k <port number> passed to simulator see the simulator docs for details.

5.5 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 this section.

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. Note: pressing <return> repeats the last command.

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"

5.6 Interfacing with DDD

The .eps File http://cvs.sourceforge.net/viewcvs.py/*checkout*/sdcc/sdcc/doc/figures/ddd_example.eps shows a screenshot of a debugging session with DDD (Unix only) on a simulated 8032. The debugging session might not run as smoothly as the screenshot suggests. The debugger allows setting of breakpoints, displaying and changing variables, single stepping through C and assembler code.
The source was compiled with

```
sdcc --debug ddd_example.c
```

and DDD was invoked with

```
ddd -debugger 'sdcdb -cpu 8032 ddd_example'
```

5.7 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
;;
```

Chapter 6

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'. Please note, that ANSI C requires both signed and unsigned chars to be promoted to 'signed int' before doing any operation. This promotion can be omitted, if the result is the same. The effect of the promotion rules together with the sign-extension is often surprising:

  ```
  unsigned char uc = 0xfe;
  if (uc * uc < 0) /* this is true!  */
  {
      ....
  }
  ```

  `uc * uc` is evaluated as `(int) uc * (int) uc = (int) 0xfe * (int) 0xfe = (int) 0xfc04 = -1024`.
  Another one:

  ```
  (unsigned char) -12 / (signed char) -3 = ...
  ```

  No, the result is not 4:

  ```
  (int) (unsigned char) -12 / (int) (signed char) -3 =
  (int) (unsigned char) 0xf4 / (int) (signed char) 0xfd =
  (int) 0x00f4 / (int) 0xfffd =
  (int) 0x00f4 / (int) 0xfffd =
  (int) 244 / (int) -3 =
  (int) -81 = (int) 0xffaf;
  ```

  Don't complain, that gcc gives you a different result. gcc uses 32 bit ints, while SDCC uses 16 bit ints. Therefore the results are different.
  From "comp.lang.c FAQ":

  > If well-defined overflow characteristics are important and negative values are not, or if you want to steer clear of sign-extension problems when manipulating bits or bytes, use one of the corresponding unsigned types. (Beware when mixing signed and unsigned values in expressions, though.) Although character types (especially unsigned char) can be used as "tiny" integers, doing so is sometimes more trouble than it's worth, due to unpredictable sign extension and increased code size.

- 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, bit-shifting or are using an array index.

- 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.

6.1 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

6.2 Documentation included in the distribution

Subject / Title	Where to get / filename
SDCC Compiler User Guide	You’re reading it right now
Changelog of SDCC	sdcc/Changelog
ASXXXX Assemblers and ASLINK Relocating Linker	sdcc/as/doc/asxhtm.html
SDCC regression test	sdcc/doc/test_suite_spec.pdf
Various notes	sdcc/doc/*
Notes on debugging with sdcdb	sdcc/debugger/README
Software simulator for microcontrollers	sdcc/sim/ucsim/doc/index.html
Temporary notes on the pic16 port	sdcc/src/pic16/NOTES
SDCC internal documentation (debugging file format)	sdcc/doc/cdbfileformat.pdf

6.3 Related open source tools

Name	Purpose	Where to get
gpsim	PIC simulator	http://www.dattalo.com/gnupic/gpsim.html
gputils	GNU PIC utilities	http://sourceforge.net/projects/gputils
flP5	PIC programmer	http://freshmeat.net/projects/flp5/
indent	Formats C source - Master of the white spaces	http://directory.fsf.org/GNU/indent.html
srecord	Object file conversion, checksumming, ...	http://sourceforge.net/projects/srecord
objdump	Object file conversion, ...	Part of binutils (should be there anyway)
doxygen	Source code documentation system	http://www.doxygen.org
kdevelop	IDE (has anyone tried integrating SDCC & sdcdb? Unix only)	http://www.kdevelop.org
splint	Statically checks c sources (see 3.2.8)	http://www.splint.org
ddd	Debugger, serves nicely as GUI to sdcdb (Unix only)	http://www.gnu.org/software/ddd/

6.4 Related documentation / recommended reading

Name	Subject / Title	Where to get
c-refcard.pdf	C Reference Card, 2 pages	http://refcards.com/refcards/c/index.html
c-faq	C-FAQ-list	http://www.eskimo.com/~scs/C-faq/top.html
	Latest datasheet of the target CPU	vendor
	Revision history of datasheet	vendor
S. S. Muchnick	Advanced Compiler Design and Implementation	bookstore (very dedicated, probably read other books first)

6.5 Some Questions

Some questions answered, some pointers given - it might be time to in turn ask *you* some questions:

- can you solve your project with the selected microcontroller? Would you find out early or rather late that your target is too small/slow/whatever? Can you switch to a slightly better device if it doesn't fit?
- should you solve the problem with an 8 bit CPU? Or would a 16/32 bit CPU and/or another programming language be more adequate? Would an operating system on the target device help?
- if you solved the problem, will the marketing department be happy?
- if the marketing department is happy, will customers be happy?
- if you're the project manager, marketing department and maybe even the customer in one person, have you tried to see the project from the outside?
- is the project done if you think it is done? Or is just that other interface/protocol/feature/configuration/option missing? How about website, manual(s), internationali(z|s)ation, packaging, labels, 2nd source for components, electromagnetic compatability/interference, documentation for production, production test software, update mechanism, patent issues?
- is your project adequately positioned in that magic triangle: fame, fortune, fun?

Maybe not all answers to these questions are known and some answers may even be *no*, nevertheless knowing these questions may help you to avoid burnout[1]. Chances are you didn't want to hear some of them...

[1] burnout is bad for electronic devices, programmers and motorcycle tyres

Chapter 7

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.

7.1 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.

7.2 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.

7.3 Submitting patches

Like bug reports contributed patches are forwarded to the developer mailing list. This is the link for submitting patches: http://sourceforge.net/tracker/?group_id=599&atid=300599.

You need to specify some parameters to the `diff` command for the patches to be useful. If you modified more than one file a patch created f.e. with **"diff -Naur unmodified_directory modified_directory >my_changes.patch"** will be fine, otherwise **"diff -u sourcefile.c.orig sourcefile.c >my_changes.patch"** will do.

7.4 Getting Help

These links should take you directly to the Mailing lists http://sourceforge.net/mail/?group_id=599[1] and the Forums http://sourceforge.net/forum/?group_id=599, lists and forums are archived and searchable so if you are lucky someone already had a similar problem. While mails to the lists themselves are delivered promptly their web front end on sourceforge sometimes shows a severe time lag (up to several weeks), if you're seriously using SDCC please consider subscribing to the lists.

7.5 ChangeLog

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

7.6 Release policy

Historically there often were long delays between official releases and the sourceforge download area tends to get not updated at all. Excuses in the past might have referred to problems with live range analysis, but as this was fixed a while ago, the current problem is that another excuse has 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 build 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.

7.7 Examples

You'll find some small examples in the directory *sdcc/device/examples/*. More examples and libraries are available at *The SDCC Open Knowledge Resource* ***http://sdccokr.dl9sec.de/*** web site or at http://www.pjrc.com/tech/8051/.

7.8 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*.

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

Chapter 8

SDCC Technical Data

8.1 Optimizations

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

8.1.1 Sub-expression Elimination

The compiler does local and *g*lobal *c*ommon *s*ubexpression *e*limination, 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.

8.1.2 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;
}
```

8.1.3 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.

8.1.4 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.

8.1.5 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.

8.1.6 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.

8.1.7 ’switch’ Statements

SDCC can optimize switch statements to jump tables. It makes the decision based on an estimate of the generated code size. SDCC is quite liberal in the requirements for jump table generation:

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

```
switch(i) {                        switch (i) {
   case 4:  ...                        case 0:  ...
   case 5:  ...                        case 1:  ...
   case 3:  ...
   case 6:  ...                        case 3:  ...
   case 7:  ...                        case 4:  ...
```

```
        case 8:  ...
        case 9:  ...
        case 10:  ...
        case 11:  ...
    }
```

```
    case 5:  ...
    case 6:  ...
    case 7:  ...
    case 8:  ...
}
```

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 not larger than supported by the target architecture.
- If the case labels are not in numerical sequence ('gaps' between cases) SDCC checks whether a jump table with additionally inserted dummy cases is still attractive.
- If the starting number is not zero and a check for the lower boundary of the jump-table can thus be eliminated SDCC might insert dummy cases 0, ...

Switch statements which have large gaps in the numeric sequence or those that have too many 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 5:  ...
  case 6:  ...
  case 7:  ...
  case 101:  ...
  case 102:  ...
  case 103:  ...
  case 104:  ...
  case 105:  ...
  case 106:  ...
  case 107:  ...
}
```

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

```
switch (i) {
  case 1:  ...
  case 2:  ...
  case 3:  ...
  case 4:  ...
  case 5:  ...
  case 6:  ...
  case 7:  ...
}
```

and

```
switch (i) {
  case 101:  ...
  case 102:  ...
  case 103:  ...
  case 104:  ...
  case 105:  ...
  case 106:  ...
  case 107:  ...
}
```

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

The pragma nojtbound can be used to turn off checking the *j*ump *t*able *bound*aries. It has no effect if a default label is supplied. Use of this pragma is dangerous: if the switch argument is not matched by a case statement the processor will happily jump into Nirvana.

8.1.8 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
```

8.1.9 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;           /* unsigned is needed for rotation */
...
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 */
```

8.1.10 Nibble and Byte Swapping

Other special cases of the bit-shift operations are nibble or byte swapping, SDCC recognizes the following expressions:

```
unsigned char i;
unsigned int j;
...
i = ((i << 4) | (i >> 4));
j = ((j << 8) | (j >> 8));
```

and generates a swap instruction for the nibble swapping or move instructions for the byte swapping. The "j" example can be used to convert from little to big-endian or vice versa. If you want to change the endianness of a *signed* integer you have to cast to `(unsigned int)` first.

Note that SDCC stores numbers in little-endian[1] format (i.e. lowest order first).

8.1.11 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.

8.1.12 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
}
```

[1] Usually 8-bit processors don't care much about endianness. This is not the case for the standard 8051 which only has an instruction to increment its *dptr*-datapointer so little-endian is the more efficient byte order.

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 innermost '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.

8.2 ANSI-Compliance

Deviations from the compliance:

- functions are not reentrant unless explicitly declared as such or the **--stack-auto** command line option is specified.
- structures and unions 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 */
{
    ...
}
```

- Certain words that are valid identifiers in the standard may be reserved words in SDCC unless the **--std-c89 or --std-c99** command line options are used. These may include (depending on the selected processor): 'at', 'banked', 'bit', 'code', 'critical', 'data', 'eeprom', 'far', 'flash', 'idata', 'interrupt', 'near', 'nonbanked', 'pdata', 'reentrant', 'sbit', 'sfr', 'shadowregs', 'sram', 'using', 'wparam', 'xdata', '_overlay', '_asm', '_endasm', and '_naked'. Compliant equivalents of these keywords are always available in a form that begin with two underscores, f.e. '__data' instead of 'data'.

8.3 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.

8.4 Retargetting for other Processors

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.

Chapter 9

Compiler internals

9.1 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 a 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 register (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
```

9.2 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.

Chapter 10

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.

Alphabetical index

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

Index