## Contents

1 Introduction ........................................... 7  
1.1 About SDCC ........................................ 7  
1.2 SDCC Suite Licenses .................................. 8  
1.3 Documentation ...................................... 9  
1.4 Typographic conventions ............................. 9  
1.5 Compatibility with previous versions ............... 9  
1.6 System Requirements ................................ 11  
1.7 Other Resources .................................... 11

2 Installing SDCC ........................................ 12  
2.1 Configure Options .................................... 12  
2.2 Install paths ......................................... 14  
2.3 Search Paths ......................................... 15  
2.4 Building SDCC ........................................ 17  
2.4.1 Building SDCC on Linux ........................... 17  
2.4.2 Building SDCC on Mac OS X ....................... 18  
2.4.3 Cross compiling SDCC on Linux for Windows .... 18  
2.4.4 Building SDCC using Cygwin and Mingw32 ....... 18  
2.4.5 Building SDCC Using Microsoft Visual C++ 2010 (MSVC) ............ 19  
2.4.6 Windows Install Using a ZIP Package ........... 20  
2.4.7 Windows Install Using the Setup Program ....... 20  
2.4.8 VPATH feature ..................................... 20  
2.5 Building the Documentation ......................... 21  
2.6 Reading the Documentation ........................... 21  
2.7 Testing the SDCC Compiler ............................ 21  
2.8 Install Trouble-shooting .............................. 22  
2.8.1 If SDCC does not build correctly ................. 22  
2.8.2 What the "/configure" does ...................... 22  
2.8.3 What the "make" does ............................ 22  
2.8.4 What the "make install" command does .......... 22  
2.9 Components of SDCC .................................. 23  
2.9.1 sdcc - The Compiler ............................... 24  
2.9.2 sdcpp - The C-Preprocessor ....................... 24  
2.9.3 sdas, sdld - The Assemblers and Linkage Editors .... 24  
2.9.4 s51, sz80, shc08, sstm8 - The Simulators ......... 24  
2.9.5 sdcdb - Source Level Debugger ................... 24

3 Using SDCC ........................................... 25  
3.1 Standard-Compliance ................................ 25  
3.1.1 ISO C90 and ANSI C89 ............................ 25  
3.1.2 ISO C95 .......................................... 26  
3.1.3 ISO C99 .......................................... 26  
3.1.4 ISO C11 and ISO C17 ............................. 26  
3.1.5 ISO C2X .......................................... 26  
3.1.6 Embedded C ...................................... 26
4 Notes on supported Processors

4.1 MCS51 variants

4.1.1 pdata access by SFR

4.1.2 Other Features available by SFR

4.1.3 Bankswitching

4.1.4 MCS51/DS390 Startup Code

4.1.5 Interfacing with Assembler Code

4.2 DS400 port

4.3 The Z80, Z180, Rabbit 2000/3000, Rabbit 3000A, GBZ80, eZ80 and TLCS-90 ports

4.3.1 Startup Code

4.3.2 Complex instructions

4.3.3 Calling conventions

4.3.4 Small-C calling convention

4.3.5 Unsafe reads

4.4 The HC08 and S08 ports
5 Debugging
5.1 Debugging with SDCDB
  5.1.1 Compiling for Debugging
  5.1.2 How the Debugger Works
  5.1.3 Starting the Debugger SDCDB
  5.1.4 SDCDB Command Line Options
  5.1.5 SDCDB Debugger Commands
  5.1.6 Interfacing SDCDB with DDD
  5.1.7 Interfacing SDCDB with XEmacs

6 TIPS
6.1 Porting code from or to other compilers
6.2 Tools included in the distribution
6.3 Documentation included in the distribution
6.4 Communication online at SourceForge
6.5 Related open source tools
6.6 Related documentation / recommended reading
6.7 Application notes specifically for SDCC
6.8 Some Questions

7 Support
7.1 Reporting Bugs
7.2 Requesting Features
7.3 Submitting patches
7.4 Getting Help
7.5 ChangeLog
7.6 Subversion Source Code Repository
7.7 Release policy
7.8 Quality control
7.9 Examples
7.10 Use of SDCC in Education

8 SDCC Technical Data
8.1 Optimizations
  8.1.1 Sub-expression Elimination
  8.1.2 Dead-Code Elimination
  8.1.3 Copy-Propagation
  8.1.4 Loop Optimizations
  8.1.5 Loop Reversing
  8.1.6 Algebraic Simplifications
  8.1.7 ‘switch’ Statements
  8.1.8 Bit-shifting Operations
  8.1.9 Bit-rotation
  8.1.10 Nibble and Byte Swapping
  8.1.11 Highest Order Bit / Any Order Bit
  8.1.12 Higher Order Byte / Higher Order Word
  8.1.13 Placement of Bank-Selection Instructions
  8.1.14 Lifetime-Optimal Speculative Partial Redundancy Elimination
  8.1.15 Register Allocation
  8.1.16 Peephole Optimizer
  8.2 Cyclomatic Complexity
  8.3 Retargetting for other Processors

9 Compiler internals
9.1 The anatomy of the compiler
9.2 A few words about basic block successors, predecessors and dominators
10 Acknowledgments 120
Chapter 1

Introduction

1.1 About SDCC

SDCC (Small Device C Compiler) is free open source, retargettable, optimizing standard (ANSI C89 / ISO C90, ISO C99, ISO C11 / ISO C17) C compiler suite originally written 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 based (hc08, s08), Zilog Z80 based MCUs (Z80, Z180, gbz80, Rabbit 2000/3000, Rabbit 3000A ), Toshiba TLCS-90, Zilog eZ80 in Z80 mode, STMicroelectronics STM8 and Padauk PDK14. It can be retargeted for other microprocessors, support for Microchip PIC and Padauk PDK15 is under development. The entire source code for the compiler is distributed under GPL. SDCC uses a modified version of ASXXXX & ASLINK, free open source retargetable assembler & linker. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware effectively.

In addition to the wiki [http://sdcc.sourceforge.net/wiki/](http://sdcc.sourceforge.net/wiki/).

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:
## 1.2 SDCC Suite Licenses

SDCC suite is a collection of several components derived from different sources with different licenses:

- **executables**:
  - sdcc compiler:
    sdcc compiler is licensed under the GPLv2.
    The code or object files generated by SDCC suite are not licensed, so they can be used in FLOSS or proprietary (closed source) applications.
  - sdcpp preprocessor:
  - sdas assemblers and sddl linker:
    derived from ASXXXX [http://shop-pdp.kent.edu/ashtml/asxxxx.htm](http://shop-pdp.kent.edu/ashtml/asxxxx.htm); GPLv3 license
  - SDCC run-time libraries:
    The great majority of SDCC run-time libraries are licensed under the GPLv2+LE which allows linking of SDCC run-time libraries with proprietary (closed source) applications. Exception are pic device libraries and header files which are derived from Microchip header (.inc) and linker script (.lkr) files. Microchip requires that "The header files should state that they are only to be used with authentic Microchip devices" which makes them incompatible with the GPL. Pic device libraries and header files are located at non-free/lib and non-free/include directories respectively. SDCC should be run with the --use-non-free command line option in order to include non-free header files and libraries.
  - sdbinutils utilities (sdar, sdranlib, sdnm, sdobjcopy):

- **ucsim simulators**:
  GPLv2 license

---

1. Incomplete support in the pic14 and pic16 backends.
2. Only supported in the mcs51, ds390, ds400 backends.
3. Obviously this has pros and cons.
1.3. DOCUMENTATION

- sdcdb debugger:
  GPLv2 license

- gcc-test regression tests:
  derived from gcc-testsuite; no license explicitly specified, but since it is a part of GCC is probably
  GPLv3 licensed

- packihx:
  public domain

- makebin:
  zlib/libpng License

• libraries:
  - dbuf library:
    zlib/libpng License
  - Boost C++ libraries:
    http://www.boost.org/; Boost Software License 1.0 (BSL-1.0)

Links to licenses:
- GPLv2 license: http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
- LGPLv2.1 license: http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html
- GPLv3 license: http://www.gnu.org/licenses/gpl.html
- zlib/libpng License: http://www.opensource.org/licenses/Zlib
- Boost Software License 1.0 (BSL-1.0): http://www.opensource.org/licenses/BSL-1.0

1.3 Documentation

This documentation is maintained using a free open source word processor (LYX) http://www.lyx.org/.

1.4 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.

1.5 Compatibility with previous versions

Newer versions have usually numerous bug fixes compared with the previous version. But we also sometimes introduce some incompatibilities with older versions. Not just for the fun of it, but to make the compiler more stable, efficient and standard compliant (see section 3.1 for Standard-Compliance). This is a list of such changes.

- short is now equivalent to int (16 bits), it used to be equivalent to char (8 bits) which is not ANSI compliant. To maintain compatibility, old programs could be compiled using the --short-is-8bits command line option (option removed after the 3.6.0 release).

- 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 and --std-c89 and --std-c99 command line option are not defined, e.g.:

  ```c
  char a=3;
  printf ("%d %c\n", a, (char)a);
  ```

  will push a as an int and as a char resp if --std-c89 and --std-c99 command line options are not defined, will push a as two ints if --std-c89 or --std-c99 command line option is defined.
• pointer type parameters to vararg functions are casted to generic pointers on Harvard architectures (e.g. mcs51, ds390) unless explicitly casted and \texttt{--std-c89} and \texttt{--std-c99} command line option are not defined.

• option \texttt{--regextend} has been removed.

• option \texttt{--noregparms} has been removed.

• option \texttt{--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.:

\begin{verbatim}
    bit b;
    b = ~b; /* equivalent to b=1 instead of toggling b */
    b = !b; /* toggles b */
\end{verbatim}

In previous versions, both forms would have toggled the bit.

• in older versions, the preprocessor was always called with \texttt{--std-c99} regardless of the \texttt{--std-xxx} setting. This is no longer true, and can cause compilation failures on code built with \texttt{--std-c89} but using c99 preprocessor features, such as one-line (//) comments

• in versions older than 2.8.4 the pic16 *printf() and printf_tiny() library functions supported undocumented and not standard compliant 'b' binary format specifier ("%b", "%hb" and "%lb"). The 'b' specifier is now disabled by default. It can be enabled by defining BINARY_SPECIFIER macro in files device/lib/pic16/libc/stdio/vfprintf.c and device/lib/pic16/libc/stdio/printf_tiny.c and recompiling the library.

• in versions older then 2.8.5 the unnamed bit-field structure members participated in initialization, which is not conforming with ISO/IEC 9899:1999 standard (see section Section 6.7.8 Initialization, clause 9)

Old behaviour, before version 2.8.5:

\begin{verbatim}
    struct {
        int a : 2;
        char : 2;
        int b : 2;
    } s = {1, 2, 3};
    //* s.a = 1, s.b = 3 */
\end{verbatim}

New behaviour:

\begin{verbatim}
    struct {
        int a : 2;
        char : 2;
        int b : 2;
    } s = {1, 2};
    //* s.a = 1, s.b = 2 */
\end{verbatim}

• in 2.9.0 libraries included in SDCC packages, are in ar format. See section 3.2.5.

• In 3.0.0 targets for xa51 and avr are disabled by default.

• In 3.0.0 sdldgb and sdldz80 don’t support generation of Game Boy binary image format. The makebin utility can be used to convert Intel Hex format to Game Boy binary image format.

• In 3.0.0 sdldgb and sdldz80 don’t support generation of rrgb (Game Boy simulator) map file and no$gmb symbol file formats. The as2gbmap utility can be used to convert sdld map format to rrgb and no$gmb file formats.

• In 3.1.0 asranlib utility was renamed to sdranlib.

• In 3.1.0 pic14 target, structured access to SFR via \texttt{<sfrname>_bits.<bitname>} is deprecated and replaced by \texttt{<sfrname>bits.<bitname>}. It will be obsoleted (removed) in one of next SDCC releases. See section 4.6.8.3.
1.6 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.

1.7 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. The latest snapshot build version of this document in pdf format 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 from Subversion on http://sourceforge.net/p/sdcc/code/8805/tree/trunk/sdcc/.
Chapter 2

Installing SDCC

For most users it is sufficient to skip to either section 2.4.1 (Unix) or section 2.4.7 (Windows). More detailed 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
--datarootdir see table below
--docdir environment variable, see table below
--include_dir_suffix environment variable, see table below
--non_free_include_dir_suffix environment variable, see table below
--lib_dir_suffix environment variable, see table below
--non_free_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-z80-port Excludes the z80 port
--disable-z180-port Excludes the z180 port
--disable-r2k-port Excludes the r2k port
--disable-r3ka-port Excludes the r3ka port
--disable-gbz80-port Excludes the Game Boy gbz80 port
--disable-avr-port Excludes the AVR port (disabled by default)
--disable-ds390-port Excludes the DS390 port
2.1. CONFIGURE OPTIONS

--disable-hc08-port Excludes the HC08 port
--disable-s08-port Excludes the S08 port
--disable-stm8-port Excludes the STM8 port
--disable-pic-port Excludes the PIC14 port
--disable-pic16-port Excludes the PIC16 port
--disable-xa51-port Excludes the XA51 port (disabled by default)
--disable-ucsim Disables configuring and building of ucsim
--disable-device-lib Disables automatically building device libraries
--disable-packihx Disables building packihx
--enable-doc Build pdf, html and txt files from the lyx sources
--enable-libgc Use the Bohem memory allocator. Lower runtime footprint.
--without-ccache Do not use ccache even if available

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 SDCC Distributed Compile Farm.

See the examples, how to pass the Win32 settings to `configure`. The other Win32 builds using VC or whatever don’t use `configure`, but a header file sdcc_vc.h.in is the same as sdcconf.h built by `configure` for Win32.

These defaults are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>default</th>
<th>Win32 builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX</td>
<td>/usr/local</td>
<td>\sdcc</td>
</tr>
<tr>
<td>EXEC_PREFIX</td>
<td>$PREFIX</td>
<td>$PREFIX</td>
</tr>
<tr>
<td>BINDIR</td>
<td>$EXEC_PREFIX/bin</td>
<td>$EXEC_PREFIX\bin</td>
</tr>
<tr>
<td>DATADIR</td>
<td>$DATAROOTDIR</td>
<td>$DATAROOTDIR</td>
</tr>
<tr>
<td>DATAROOTDIR</td>
<td>$PREFIX/share</td>
<td>$PREFIX</td>
</tr>
<tr>
<td>DOCDIR</td>
<td>$DATAROOTDIR/sdcc/doc</td>
<td>$DATAROOTDIR/doc</td>
</tr>
<tr>
<td>INCLUDE_DIR_SUFFIX</td>
<td>sdcc/include</td>
<td>include</td>
</tr>
<tr>
<td>NON_FREE_INCLUDE_DIR_SUFFIX</td>
<td>sdcc/non-free/include</td>
<td>non-free/include</td>
</tr>
<tr>
<td>LIB_DIR_SUFFIX</td>
<td>sdcc/lib</td>
<td>lib</td>
</tr>
<tr>
<td>NON_FREE_LIB_DIR_SUFFIX</td>
<td>sdcc/non-free/lib</td>
<td>non-free/lib</td>
</tr>
</tbody>
</table>
2.2 INSTALL PATHS

'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):

<table>
<thead>
<tr>
<th>Variable (computed)</th>
<th>default</th>
<th>Win32 builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN2DATA_DIR</td>
<td>../share</td>
<td>..</td>
</tr>
<tr>
<td>PREFIX2BIN_DIR</td>
<td>bin</td>
<td>bin</td>
</tr>
<tr>
<td>PREFIX2DATA_DIR</td>
<td>share/sdcc</td>
<td></td>
</tr>
</tbody>
</table>

Examples:

```
./configure
./configure --prefix="/usr/bin" --datarootdir="/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" \
--datarootdir="/sdcc" \
docdir="\${datarootdir}/doc" \
include_dir_suffix="include" \
non_free_include_dir_suffix="non-free/include" \
lib_dir_suffix="lib" \
non_free_lib_dir_suffix="non-free/lib" \
sdccconf_h_dir_separator="\\\\" \
--disable-device-lib\n--host=i586-mingw32msvc\n--build=unknown-unknown-linux-gnu
```

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

```
./configure -C \
--prefix="/sdcc" \
--datarootdir="/sdcc" \
docdir="\${datarootdir}/doc" \
include_dir_suffix="include" \
non_free_include_dir_suffix="non-free/include" \
lib_dir_suffix="lib" \
non_free_lib_dir_suffix="non-free/lib" \
sdccconf_h_dir_separator="\\\\" \
CC="gcc -mno-cygwin" \
CXX="g++ -mno-cygwin"
```

'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
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)

<table>
<thead>
<tr>
<th>Description</th>
<th>Path</th>
<th>Default</th>
<th>Win32 builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary files*</td>
<td>$EXEC_PREFIX</td>
<td>/usr/local/bin</td>
<td>\sdcc\bin</td>
</tr>
<tr>
<td>Include files</td>
<td>$DATADIR/INCLUDE_DIR_SUFFIX</td>
<td>/usr/local/share/sdcc/include</td>
<td>\sdcc\include</td>
</tr>
<tr>
<td>Non-free include files</td>
<td>$DATADIR/non-free/INCLUDE_DIR_SUFFIX</td>
<td>/usr/local/share/sdcc/non-free/include</td>
<td>\sdcc\non-free\include</td>
</tr>
<tr>
<td>Library file**</td>
<td>$DATADIR/LIB_DIR_SUFFIX</td>
<td>/usr/local/share/sdcc/lib</td>
<td>\sdcc\lib</td>
</tr>
<tr>
<td>Library file**</td>
<td>$DATADIR/non-free/LIB_DIR_SUFFIX</td>
<td>/usr/local/share/sdcc/non-free/lib</td>
<td>\sdcc\non-free\lib</td>
</tr>
<tr>
<td>Documentation</td>
<td>$DOCDIR</td>
<td>/usr/local/share/sdcc/doc</td>
<td>\sdcc\doc</td>
</tr>
</tbody>
</table>

*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. Include files

1. Binary files (preprocessor, assembler and linker)
2.3. SEARCH PATHS

<table>
<thead>
<tr>
<th>#</th>
<th>Search path</th>
<th>default</th>
<th>Win32 builds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--I dir</td>
<td>--I dir</td>
<td>--I dir</td>
</tr>
<tr>
<td>2</td>
<td>$SDCC_INCLUDE</td>
<td>$SDCC_INCLUDE</td>
<td>$SDCC_INCLUDE</td>
</tr>
<tr>
<td>3</td>
<td>$SDCC_HOME/</td>
<td>$SDCC_HOME/</td>
<td>$SDCC_HOME/</td>
</tr>
<tr>
<td></td>
<td>$PREFIX2DATA_DIR/</td>
<td>share/sdcc/include</td>
<td>share/sdcc/include</td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>path(argv[0])/</td>
<td>path(argv[0])/../</td>
<td>path(argv[0])/../</td>
</tr>
<tr>
<td></td>
<td>$BIN2DATADIR/</td>
<td>sdcc/include</td>
<td>include</td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$DATADIR/</td>
<td>/usr/local/share/</td>
<td>(not on Win32)</td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td>sdcc/include</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$SDCC_HOME/</td>
<td>$SDCC_HOME/</td>
<td>$SDCC_HOME/</td>
</tr>
<tr>
<td></td>
<td>$PREFIX2DATA_DIR/</td>
<td>share/sdcc/non-free/include</td>
<td>share/sdcc/non-free/include</td>
</tr>
<tr>
<td></td>
<td>non-free/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>path(argv[0])/</td>
<td>path(argv[0])/../</td>
<td>path(argv[0])/../</td>
</tr>
<tr>
<td></td>
<td>$BIN2DATADIR/</td>
<td>sdcc/non-free/include</td>
<td>include</td>
</tr>
<tr>
<td></td>
<td>non-free/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$DATADIR/</td>
<td>/usr/local/share/</td>
<td>(not on Win32)</td>
</tr>
<tr>
<td></td>
<td>non-free/</td>
<td>sdcc/non-free/include</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$INCLUDE_DIR_SUFFIX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The option --nostdinc disables all search paths except #1 and #2.

3. Library files

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

2.4.1 Building SDCC on Linux

1. Download the source package either from the SDCC Subversion repository or from snapshot builds, it will be named something like sdcc-src-yyyyymmd-rrrr.tar.bz2 [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 -xvjf sdcc-src-yyyyymmd-rrrr.tar.bz2", 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. When the treedec library is available, it should be found and used automatically (improving the compilation time / code quality trade-off). As of SDCC 3.7.0, the current develop branch from https://github.com/freetdi/tdlib is a suitable version of treedec.

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.

The option --nostdlib disables all search paths except #1 and #2.
2.4.2 Building SDCC on Mac OS X

Follow the instruction for Linux.

On Mac OS X 10.2.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
```

Universal (ppc and i386) binaries can be produced on Mac OS X 10.4.x with Xcode. Run 'configure' with:

```
./configure \
LDFLAGS="-Wl,-syslibroot,/Developer/SDKs/MacOSX10.4u.sdk -arch i386 -arch ppc" \
CXXFLAGS = "-O2 -isysroot /Developer/SDKs/MacOSX10.4u.sdk -arch i386 -arch ppc" \
CFLAGS = "-O2 -isysroot /Developer/SDKs/MacOSX10.4u.sdk -arch i386 -arch ppc"
```

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 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. 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
- libboost-dev
- 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
- svn ; only if you use Subversion 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.

### 2.4.5 Building SDCC Using Microsoft Visual C++ 2010 (MSVC)

Download the source package either from the SDCC Subversion repository or from the snapshot builds http://sdcc.sourceforge.net/snap.php, it will be named something like sdcc-src-yyyyymmdd-rrrr.tar.bz2. SDCC is distributed with all the project, solution and other files you need to build it using Visual C++ 2010 (except for ucSim). The solution name is ‘sdcc.sln’. 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.

Apart from the SDCC sources you also need to have the BOOST libraries installed for MSVC. Get it here http://www.boost.org/

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

If UnxUtils didn’t work well, msys (http://www.mingw.org/wiki/msys) or msys2(https://msys2.github.io) can be an alternative.

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) Add ‘C:\user\local\wbin’ to VC++ Directories / Executable Directories.

   (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) Add ‘C:\util’ to VC++ Directories / Executable Directories. 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.sln’ 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.

2.4.6 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.

2.4.7 Windows Install Using the Setup Program

Download the setup program sdcc-x.y.z-setup.exe for an official release from http://sourceforge.net/projects/sdcc/files/ or a setup program for one of the snapshots sdcc-yyyyymmdd-xxxx-setup.exe from http://sdcc.sourceforge.net/snap.php and execute it. A windows typical installer will guide you through the installation process.

2.4.8 VPATH feature

SDCC supports the VPATH feature provided by configure and make. It allows to separate the source and build trees. Here’s an example:

```
cd - # cd $HOME
tar -xjf sdcc-src-yyyyymmdd-yyyy.tar.bz2 # extract source to directory
sdcc mkdir sdcc.build # put output in sdcc.build
cd sdcc.build
../sdcc/configure # configure is doing all the magic!
magic
make
```

That’s it! configure will create the directory tree will all the necessary Makefiles in ~/sdcc.build. It automagically computes the variables srccd, top_srccd and top_buildid for each directory. After running make the generated files will be in ~/sdcc.build, while the source files stay in ~/sdcc.

This is not only useful for building different binaries, e.g. when cross compiling. It also gives you a much better overview in the source tree when all the generated files are not scattered between the source files. And the best thing is: if you want to change a file you can leave the original file untouched in the source directory. Simply copy it to the build directory, edit it, enter ‘make clean’, ‘rm Makefile.dep’ and ‘make’. make will do the rest for you!
2.5 Building the Documentation

Add --enable-doc to the configure arguments to build the documentation together with all the other stuff. You will need several tools (LyX, LATEX, LATEX2HTML, pdflatex, dvipdf, dvips and makeindex) to get the job done. Another possibility is to change to the doc directory and to type "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 is straightforward. Prebuilt documentation is available from http://sdcc.sourceforge.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. You’ll find the pdf version at http://sdcc.sourceforge.net/doc/sdccman.pdf.

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 was a reference like http://www.opencores.org together with a statement "some processors which are targeted by SDCC can be implemented in a field programmable gate array" or http://sourceforge.net/projects/fpgac/ "have you ever heard of an open source compiler that compiles a subset of C for an FPGA?" 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 output its version like:

```
SDCC : mcs51/z80/avr/ds390/pic16/pic14/ds400/hc08 2.5.6 #4169 (May 8 2006)
(UNIX)
```

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

---

1 If you should know why please drop us a note
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 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:

```c
#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.
- `sdas8051` - The assembler for 8051 type processors.
- `sdas390` - The assembler for DS80C390 processor.
- `sdas80`, `sdas80gb` - The Z80 and GameBoy Z80 assemblers.
- `sdas6808` - The 6808 assembler.
- `sdasstm8` - The STM8 assembler.
- `sdld` - The linker for 8051 and STM8 type processors.
- `sdld80`, `sdld80gb` - The Z80 and GameBoy Z80 linkers.
- `sdld6808` - The 6808 linker.
- `s51` - The ucSim 8051 simulator.
- `sz80` - The ucSim Z80 simulator.
- `s6808` - The ucSim 6808 simulator.
- `sdcdb` - The source debugger.
- `sdar`, `sdranlib`, `sdsn`, `sdobjcopy` - The sdcc archive managing and indexing utilities.
- `packihx` - A tool to pack (compress) Intel hex files.
- `makebin` - A tool to convert Intel Hex file to a binary and GameBoy binary image file format.

In <installdir>/share/sdcc/include:

- the include files

In <installdir>/share/sdcc/non-free/include:

- the non-free include files

In <installdir>/share/sdcc/lib:

- the src and target subdirectories with the precompiled relocatables.

In <installdir>/share/sdcc/non-free/lib:

- the src and target subdirectories with the non-free precompiled relocatables.

In <installdir>/share/sdcc/doc:

- the documentation
2.9. COMPONENTS OF SDCC

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 cpp preprocessor http://gcc.gnu.org/. The C preprocessor is used to pull in #include sources, process #ifdef statements, #defines and so on.

2.9.3 sdas, sdld - The Assemblers and Linkage Editors
This is a set of retargetable assemblers and linkage editors, which 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.

SDCC used an about 1998 branch of asxxxx version 2.0 which unfortunately is not compatible with the more advanced (i.e. macros, more targets) ASxxxx Cross Assemblers nowadays available from Alan Baldwin http://shop-pdp.kent.edu/. In 2009 Alan made his ASxxxx Cross Assemblers version 5.0 available under the GPL license (GPLv3 or later), so a reunion is now a work in progress. Thanks Alan!

2.9.4 s51, sz80, shc08, sstm8 - The Simulators
s51, sz80 shc08 and sstm8 are free open source simulators developed by Daniel Drotos. The simulators are 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, the Phillips XA51 family, the Z80, 6808 and the STM8.

2.9.5 sdcdb - Source Level Debugger
SDCDB is the companion source level debugger. More about SDCDB in section 5.1. 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 Standard-Compliance

SDCC aims to be a conforming freestanding implementation of the C programming language.

3.1.1 ISO C90 and ANSI C89

Use --std-c89 to compile in this mode.

The latest publicly available version of the standard ISO/IEC 9899 - Programming languages - C should be available at: http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899.

Deviations from standard compliance:

- Structures and unions cannot be passed as function parameters and cannot be a return value from a function, e.g.:

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

- Initialization of structure arrays must be fully braced.

```c
struct s { char x } a[] = {1, 2}; /* invalid in SDCC */  
struct s { char x } a[] = {(1), (2)}; /* OK */
```

- 'double' precision floating point not supported. Instead a warning is emitted, and float is used instead. long double is not supported.

- Old K&R style function declarations are not supported.

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

Some features of this standard are not supported in some ports:

- Functions are not reentrant unless explicitly declared as such or --stack-auto is specified in the mcs51, ds390, hc08 and s08 ports.
3.2. Compiling

3.1.2 ISO C95
Use `--std-c95` to compile in this mode.

Except for the issues mentioned in the section above, this standard is supported.

3.1.3 ISO C99
Use `--std-c99` to compile in this mode.

In addition to what is mentioned in the section above, the following features of this standard are not supported by SDCC:

- Compound literals.
- Variable-length arrays.

Some features of this standard are not supported in some ports:

- There is no support for data types long long, unsigned long long, int_fast64_t, int_least64_t, int64_t, uint_fast64_t, uint_least64_t, uint64_t in the pic14 and pic16 ports.

3.1.4 ISO C11 and ISO C17
Use `--std-c11` to compile in this mode.

Except for the issues mentioned in the section above, this standard is supported.

3.1.5 ISO C2X
Use `--std-c2x` to compile in this mode.

Standard and support are still quite incomplete.

3.1.6 Embedded C
SDCC supports named address spaces. The support for fixed-point math in SDCC is inconsistent with the standard. Other parts of the standard are not supported.

3.2 Compiling

3.2.1 Single Source File Projects
For single source file 8051 projects the process is very simple. Compile your programs with the following command `sdcc sourcefile.c`. This will compile, assemble and link your source file. Output files are as follows:

- sourcefile.asm - Assembler source file created by the compiler
- sourcefile.lst - Assembler listing file created by the Assembler
- sourcefile.rst - Assembler listing file updated with linkedit information, created by linkage editor
- sourcefile.sym - symbol listing for the sourcefile, created by the assembler
- sourcefile.rel - Object file created by the assembler, input to Linkage editor
- sourcefile.map - The memory map for the load module, created by the Linker
- sourcefile.mem - A file with a summary of the memory usage
- sourcefile.ihx - The load module in Intel hex format (you can select the Motorola S19 format with `--out-fmt-s19`. If you need another format you might want to use `objdump` or `srecedit` - see also section 3.2.2 on the following page). Both formats are documented in the documentation of srecedit
- 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.omf - An optional AOMF or AOMF51 file containing debug information (generated with option --debug). The (Intel) absolute object module format is a subformat of the OMF51 format and is commonly used by third party tools (debuggers, simulators, emulators).

• sourcefile.dump* - Dump file to debug the compiler itself (generated with option --dumpall) (see section 3.3.11 and section 9.1 “Anatomy of the compiler”).

### 3.2.2 Postprocessing the Intel Hex file

In most cases this won’t be needed but the Intel Hex file which is generated by SDCC might include lines of varying length and the addresses within the file are not guaranteed to be strictly ascending. If your toolchain or a bootloader does not like this you can use the tool `packihx` which is part of the SDCC distribution:

```
packihx sourcefile.ihx >sourcefile.hex
```

The separately available `srecord` package additionally allows to set undefined locations to a predefined value, to insert checksums of various flavours (crc, add, xor) and to perform other manipulations (convert, split, crop, offset, ...).

```
srec_cat sourcefile.ihx -intel -o sourcefile.hex -intel
```

An example for a more complex command line\(^1\) could look like:

```
srec_cat sourcefile.ihx -intel -fill 0x12 0x0000 0xffffe -little-endian-checksum-negative 0xffffe 0x02 0x02 -o sourcefile.hex -intel
```

The srecord package is available at [http://sourceforge.net/projects/srecord/](http://sourceforge.net/projects/srecord/).

### 3.2.3 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

\(^1\)the command backfills unused memory with 0x12 and the overall 16 bit sum of the complete 64 kByte block is zero. If the program counter on an mcs51 runs wild the backfill pattern 0x12 will be interpreted as an `lcall` to address 0x1212 (where an emergency routine could sit).
3.3. COMMAND LINE OPTIONS

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.

3.2.4 Projects with Additional Libraries

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

    sdcc foomain.c foolib.lib -L mylib

Note here that mylib must be an absolute path name.

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

3.2.5 Using sdar to Create and Manage Libraries

Support for sdar format libraries was introduced in SDCC 2.9.0.

Both the GNU and BSD ar format variants are supported by sdld linkers.

To create a library containing sdas object files, you should use the following sequence:

    sdar -rc <library name>.lib <list of .rel files>

3.3 Command Line Options

3.3.1 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 (aka 68HC08) family of processors.
- **-ms08** Generate code for the Freescale/Motorola S08 (aka 68HCS08, HCS08, CS08) family of processors.
- **-mz80** Generate code for the Zilog Z80 family of processors.
- **-mz180** Generate code for the Zilog Z180 family of processors.
- **-mr2k** Generate code for the Rabbit 2000 / Rabbit 3000 family of processors.
- **-mr3ka** Generate code for the Rabbit 3000A family of processors.
- **-mgbz80** Generate code for the LR35902 GameBoy Z80 processor.
- **-mtlcs90** Generate code for the Toshiba TLCS-90 processor.
- **-mez80_z80** Generate code for the Zilog eZ80 processor in Z80 mode.
- **-stm8** Generate code for the STMicroelectronics STM8 family of processors.
- **-mpdk14** Generate code for Padauk processors with 14 bit wide program memory.
3.3. COMMAND LINE OPTIONS

-mpdk15 Generate code for Padauk processors with 15 bit wide program memory.

-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).

SDCC inspects the program name it was called with so the processor family can also be selected by renaming the sdcc binary (to e.g. z80-sdcc) or by calling SDCC from a suitable link. Option -m has higher priority than setting from program name.

3.3.2 Preprocessor Options

SDCC uses sdcpp, an adapted version of the GNU Compiler Collection preprocessor epp (gcc http://gcc.gnu.org/). If you need more dedicated options than those listed below please refer to the GCC CPP Manual at http://www.gnu.org/software/gcc/onlinedocs/.

-<path> The additional location where the preprocessor 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.

-A(question(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.

-pedantic-parse-number Pedantic parse numbers so that situations like 0xfe-LO_B(3) are parsed properly and the macro LO_B(3) gets expanded. See also #pragma pedantic_parse_number on page 58 in section3.16 Note: this functionality is not in conformance with C99 standard!

-Wp preprocessorOption[,preprocessorOption]... Pass the preprocessorOption to the preprocessor sdcpp.

3.3.3 Optimization Options

--nogcse Will not do global common subexpression elimination, this option may be used when the compiler creates undesirably large stack/data spaces to store compiler temporaries (spill locations, 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.
3.3. COMMAND LINE OPTIONS

--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.

--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 with built-in rules.

--peep-file <filename> This option can be used to use additional rules to be used by the peep hole optimizer. See section 8.1.16 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.

--peep-return Let the peep hole optimizer do return optimizations. This is the default without --debug.

--no-peep-return Do not let the peep hole optimizer do return optimizations. This is the default with --debug.

--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.

--fomit-frame-pointer Frame pointer will be omitted when the function uses no local variables. On the z80-related ports this option will result in the frame pointer always being omitted.

--max-allocs-per-node Setting this to a high value will result in increased compilation time and more optimized code being generated. Setting it to lower values speed up compilation, but does not optimize as much. The default value is 3000. This option currently only affects the hc08, s08, z80, z180, r2k, r3ka and gbz80 ports.

--nolospre Disable lospre. lospre is an advanced redundancy elimination technique, essentially an improved variant of global subexpression elimination.

--allow-unsafe-read Allow optimizations to generate unsafe reads. This will enable additional optimizations, but can result in spurious reads from undefined memory addresses, which can be harmful if the target system uses certain ways of doing memory-mapped I/O.

--nostdlibcall Disable the optimization of calls to the standard library.

3.3.4 Other Options

-v --version displays the sdcc version.

-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.
3.3. COMMAND LINE OPTIONS

-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 where everything will be placed or the file name used for all generated output files. If the parameter is a path, it must have a trailing slash (or backslash for the Windows binaries) to be recognized as a path. Note for Windows users: if the path contains spaces, it should be surrounded by quotes. The trailing backslash should be doubled in order to prevent escaping the final quote, for example: -o "F:\Projects\test\output 1\" or put after the final quote, for example: -o "F:\Projects\test\output 1". The path using slashes for directory delimiters can be used too, for example: -o "F:/Projects/test3/output 1/".

--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 and 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 and exit (function prologue and epilogue) for these functions to save and restore the registers used by these functions, this can SUBSTANTIALLY reduce code and 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 on page 57.

--all-callee-saves Function of --callee-saves will be applied to all functions by default.

--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 a .omf 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.

--fsigned-char By default char is unsigned. To set the signess for characters to signed, use the option --fsigned-char. If this option is set and no signedness keyword (unsigned/signed) is given, a char will be unsigned. All other types are unaffected.

--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.

31
--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  Don’t include peep-hole comments in the generated asm files even if --fverbose-asm option is specified.

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

--less-pedantic  Disable some of the more pedantic warnings. For more details, see the less_pedantic pragma on page 57.

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

--Werror  Treat all warnings as errors.

--vc  Display errors and warnings using MSVC style, so you can use SDCC with the visual studio IDE. With SDCC both offering a GCC-like (the default) and a MSVC-like output style, integration into most programming editors should be straightforward.

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

-Wa asmOption[,asmOption]... Pass the asmOption to the assembler. See file sdcc/sdas/doc/asmlnk.txt for assembler options.cd

--std-sdcc89  Generally follow the ANSI C89 / ISO C90 standard, but allow some SDCC behaviour that conflicts with the standard.

--std-c89  Follow the ANSI C89 / ISO C90 standard.

--std-sdcc99  Generally follow the ISO C99 standard, but allow some SDCC behaviour that conflicts with the standard.

--std-c99  Follow the ISO C99 standard.

--std-sdcc11  Generally follow the ISO C11 standard, but allow some SDCC behaviour that conflicts with the standard (default).

--std-c11  Follow the ISO C11 standard.

--codeseg <Name>  The name to be used for the code segment, default CSEG. This is useful if you need to tell the compiler to put the code in a special segment so you can later on tell the linker to put this segment in a special place in memory. Can be used for instance when using bank switching to put the code in a bank.

--constseg <Name>  The name to be used for the const segment, default CONST. This is useful if you need to tell the compiler to put the const data in a special segment so you can later on tell the linker to put this segment in a special place in memory. Can be used for instance when using bank switching to put the const data in a bank.

--fdollars-in-identifiers  Permit ‘$’ as an identifier character.

--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

--use-non-free  Search / include non-free licensed libraries and header files, located under the non-free directory - see section 2.3
3.3.5 Linker Options

--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.

-L <absolute path to additional libraries> Same as above.

--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 for mcs51. By using this option the stack can be placed anywhere in the internal memory space of the mcs51. 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.

For stm8, by default the stack is placed at the device-specific reset value. By using this option, the stack can be placed anywhere in the lower 16-bits of the stm8 memory space. This is particularly useful for working around the stack roll-over antifeature present in some stm8 devices.

--xstack-loc <Value> By default the external stack is placed after the __pdata segment. Using this option the xstack can be placed anywhere in the external memory space of the 8051. The value entered can be in Hexadecimal or Decimal format, e.g. --xstack-loc 0x8000 or --xstack-loc 32768. The provided value should not overlap any other memory areas such as the pdata or xdata segment and with enough space for the current application.

--data-loc <Value> The start location of the internal ram data segment. The value entered can be in Hexadecimal or Decimal format, e.g. --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, e.g. --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 and S08 processors)

-Wl linkOption[,linkOption]... Pass the linkOption to the linker. If a bootloader is used an option like “-Wl -bCSEG=0x1000” would be typical to set the start of the code segment. Either use the double quotes around this option or use no space (e.g. -Wl-bCSEG=0x1000). See also #pragma constseg and #pragma codeseg in section3.16. File sdcc/sdas/doc/asmlnk.txt has more on linker options.
3.3.6 MCS51 Options

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

--model-medium Generate code for Medium 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. It must also be used when invoking the linker.

--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. It must also be used when invoking the linker.

--model-huge Generate code for Huge 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. It must also be used when invoking the linker.

--xstack Uses a pseudo stack in the __pdata area (usually the first 256 bytes in the external ram) for allocating variables and passing parameters. See section 3.15.1.2 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 and this option will probably be removed along with the removal of --no-pack-iram.

--no-pack-iram (deprecated) Causes the linker to use old style for allocating memory areas. This option is now deprecated and will be removed in future versions.

--acall-ajmp Replaces the three byte instructions lcall/ljmp with the two byte instructions acall/ajmp. Only use this option if your code is in the same 2k block of memory. You may need to use this option for some 8051 derivatives which lack the lcall/ljmp instructions.

--no-ret-without-call Causes the code generator to insert an extra lcall or acall instruction whenever it needs to use a ret instruction in a context other than a function returning. This option is needed when using the Infineon XC800 series microcontrollers to keep its Memory Extension Stack balanced.

3.3.7 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-acceleratorenerate code for DS390 Arithmetic Accelerator.

3.3.8 Options common to all z80-related ports (z80, z180, r2k, r3ka, gbz80, tlc90, ez80_z80)

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

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

--codeseg <Value> Use <Value> for the code segment name.

--constseg <Value> Use <Value> for the const segment name.

3.3.9 Z80 Options (apply to z80, z180, r2k, r3ka, tlc90, ez80_z80)

--portmode=<Value> Determinate PORT I/O mode (<Value> is z80 or z180).

--asm=<Value> Define assembler name (<Value> is rgbds, sdasz80, isas or z80asm).

--reserve-reg-iy This option tells the compiler that it is not allowed to use register pair iy. The option can be useful for systems where iy is reserved for the OS. This option is incompatible with --fomit-frame-pointer.

--oldralloc Use old register allocator. The old register allocator is typically is faster than the current one, but the current one generates better code. This differences are the most noticeable, when a high value for --max-allocs-per-node is used.

--fno-omit-frame-pointer Never omit the frame pointer.

3.3.10 GBZ80 Options

-bo <Num> Use code bank <Num>.

-ba <Num> Use data bank <Num>.

3.3.11 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”.

--dum-ast This option will cause the compiler to dump the abstract syntax tree to the econsole.

--dump-i-code Will dump iCodes at various stages into files named <source filename>.dump<stage>.

--dump-graphs Will dump internal representations as graphviz .dot files. Depending on other options, this can include the control-flow graph at lospre, insertion of bank selection instructions or register allocation and the conflict graph and tree-decomposition at register allocation.

--fverbose-asm Include code generator and peep-hole comments in the generated asm files.

3.3.12 Redirecting output on Windows Shells

By default SDCC writes its 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)
3.4 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 mainly used for debugging purposes. They can change or disappear very quickly, and will never be documented².

3.5 SDCC Language Extensions

SDCC supports some language extensions useful for embedded systems. These include named address spaces (see also section 5.1 of the Embedded C standard). SDCC supports both intrinsic named address spaces (which ones are supported depends on the target architecture) and non-intrinsic named address spaces (defined by the user using the keyword __addressmod, they are particularly useful with custom bank-switching hardware). Unlike the Embedded C standard, SDCC allows local variables to have an intrinsic named address space even when not explicitly declared as static or extern.

3.5.1 MCS51/DS390 intrinsic named address spaces

SDCC supports the following MCS51-specific intrinsic address spaces:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>__idata</td>
<td>256B</td>
</tr>
<tr>
<td>__data</td>
<td>128B</td>
</tr>
<tr>
<td>__xdata</td>
<td>64KB</td>
</tr>
<tr>
<td>__pdata</td>
<td>256B</td>
</tr>
<tr>
<td>__code</td>
<td>64KB</td>
</tr>
</tbody>
</table>

3.5.1.1 __data / __near

This is the default (generic) address space for the Small Memory model. Variables in this address space will be allocated in the directly addressable portion of the internal RAM of a 8051, e.g.:

```c
__data unsigned char test_data;
```

Writing 0x01 to this variable generates the assembly code:

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

²If you are curious search in SDCC’s sources for “getenv”
3.5.1.2 __xdata / __far

Variables in this address space will be placed in the external RAM. This is the default (generic) address space for the Large Memory model, e.g.:

```c
__xdata unsigned char test_xdata;
```

Writing 0x01 to this variable generates the assembly code:

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

3.5.1.3 __idata

Variables in this address space will be allocated into the indirectly addressable portion of the internal ram of a 8051, e.g.:

```c
__idata unsigned char test_idata;
```

Writing 0x01 to this variable generates the assembly code:

```assembly
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 (unless --xstack is specified).

3.5.1.4 __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). This is the default (generic) address space for the Medium Memory model, e.g.:

```c
__pdata unsigned char test_pdata;
```

Writing 0x01 to this variable generates the assembly code:

```assembly
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.

3.5.1.5 __code

'Variables' in this address space will be placed in the code memory:

```c
__code unsigned char test_code;
```

Read access to this variable generates the assembly code:

```assembly
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:

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

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

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

This is a data-type and an address space. 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 intrinsic named address space most architectures support ANSI-C bit-fields. In accordance with ISO/IEC 9899 bits and bitfields without an explicit signed modifier are implemented as unsigned.

### 3.5.1.7 __sfr / __sfr16 / __sfr32 / __sbit

Like the __bit keyword, __sfr / __sfr16 / __sfr32 / __sbit signify both a data-type and named address space, they are used to describe the special function registers and special __bit variables of a 8051, eg:

```
__sfr __at (0x80) P0; /* special function register P0 at location 0x80 */

/* 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. Although SDCC usually accesses them Least Significant Byte (LSB) first, this is not guaranteed.

Please note, if you use a header file which was written for another compiler then the __sfr / __sfr16 / __sfr32 / __sbit intrinsic named address spaces will most likely be not compatible. Specifically the syntax sfr P0 = 0x80; is compiled without warning by SDCC to an assignment of 0x80 to a variable called P0. Nevertheless with the file compiler.h it is possible to write header files which can be shared among different compilers (see section 6.1).

### 3.5.1.8 Pointers to MCS51/DS390 intrinsic named address spaces

SDCC allows (via language extensions) pointers to explicitly point to any of the named address 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;
```

---

3Not really meant as examples, but nevertheless showing what bit-fields are about: device/include/mc68hc908qy.h and support/regression/tests/bitfields.c
3.5. SDCC LANGUAGE EXTENSIONS

CHAPTER 3. USING SDCC

/* 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;

/* generic pointer physically located in xdata space */
unsigned char * __xdata p;

/* generic pointer physically located in default memory space */
unsigned char * 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.

3.5.1.9 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 you are using, see also section 4.1.4 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 for example 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.

3.5.2 Z80/Z180/eZ80 intrinsic named address spaces

3.5.2.1 __sfr (in/out to 8-bit addresses)
The Z80 family has separate address spaces for memory and input/output memory. I/O memory is accessed with special instructions, e.g.:

```c
__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
```

3.5.2.2 __banked __sfr (in/out to 16-bit addresses)
The keyword __banked is used to support 16 bit addresses in I/O memory e.g.:

```c
__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
```

3.5.2.3 __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.

3.5.3 HC08/S08 intrinsic named address spaces

3.5.3.1 __data
Variables in the address space __data resides in the first 256 bytes of memory (the direct page). The HC08 is most efficient at accessing variables (especially pointers) stored here.

3.5.3.2 __xdata
Variables in the address space __xdata can reside anywhere in memory. This is the default (generic address space).

3.5.4 PDK14/PDK15 intrinsic named address spaces

3.5.4.1 __sfr
The Padauk family has separate address spaces for memory and input/output memory. I/O memory is accessed with special instructions, e.g.:

```c
__sfr __at 0x18 gpcc; /* define a var in I/O space at 18h called gpcc */
```
3.5.4.2 __sfr16

The Padauk family has a 16-bit timer accessed with special instructions.

3.5.5 Non-intrinsic named address spaces

SDCC supports user-defined non-intrinsic named address spaces. So far SDCC only supports them for bank-switching. You need to have a function that switches to the desired memory bank and declare a corresponding named address space:

```c
void setb0(void); // The function that sets the currently active memory bank to b0
void setb1(void); // The function that sets the currently active memory bank to b1
__addressmod setb0 spaceb0; // Declare a named address space called spaceb0 that uses setb0()
__addressmod setb1 spaceb1; // Declare a named address space called spaceb1 that uses setb1()
spaceb0 int x; // An int in address space spaceb0
spaceb1 int *y; // A pointer to an int in address space spaceb1
spaceb0 int *spaceb1 z; // A pointer in address space spaceb1 that points to an int in address space spaceb0
```

Non-intrinsic named address spaces for data in ROM are declared using the const keyword:

```c
void setb0(void); // The function that sets the currently active memory bank to b0
void setb1(void); // The function that sets the currently active memory bank to b1
__addressmod setb0 const spaceb0; // Declare a named address space called spaceb0 that uses setb0() and resides in ROM
__addressmod setb1 spaceb1; // Declare a named address space called spaceb1 that uses setb1() and resides in RAM
const spaceb0 int x = 42; // An int in address space spaceb0
spaceb1 int *y; // A pointer to an int in address space spaceb1
const spaceb0 int *spaceb1 z; // A pointer in address space spaceb1 that points to a constant int in address space spaceb0
```

Variables in non-intrinsic named address spaces will be placed in areas of the same name (this can be used for the placement of named address spaces in memory by the linker).

SDCC will automatically insert calls to the corresponding function before accessing the variable. SDCC inserts the minimum possible number calls to the bank selection functions. See Philipp Klaus Krause, "Optimal Placement of Bank Selection Instructions in Polynomial Time" for details on how this works.

3.5.6 Absolute Addressing

Data items can be assigned an absolute address with the __at <address> keyword, which can also be used in addition to a named address space, e.g.:

```c
__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.

If however you provide an initializer actual memory allocation will take place and overlaps will be detected by the linker. E.g.:

```c
__code __at (0x7ff0) char Id[5] = "SDCC";
```
In the above example the variable Id will be located from 0x7ff0 to 0x7ff4 in code memory.

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:

```c
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 3.11.2 has an example).

Absolute addresses can be specified for variables in all named address spaces, e.g.:

```c
__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:

```c
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 */

```c
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

```c
__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

```c
__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.

### 3.5.7 Preserved register specification

SDCC allows to specify preserved registers in function declarations, to enable further optimizations on calls to functions implemented in assembler. Example for the Z80 architecture specifying that a function will preserve register pairs bc and iy:

```c
void f(void) __preserves_regs(b, c, iy1, iyh);
```
3.6. Parameters and Local Variables

3.5.8 Binary constants

SDCC supports the use of binary constants, such as 0b01100010. This feature is only enabled when the compiler is invoked using --std-sdccxx.

3.5.9 Returning void

SDCC allows functions to return expressions of type void. This feature is only enabled when the compiler is invoked using --std-sdccxx.

3.5.10 Omitting promotion on arguments of vararg function (does not apply to pdk14, pdk15)

Arguments to vararg functions are not promoted when explicitly cast. This feature is only enabled when the compiler is invoked using --std-sdccxx. This breaks compatibility with the C standards, so linking code compiled with --std-sdccxx with code compiled using --std-cxx can result in failing programs when arguments to vararg functions are explicitly cast.

3.6 Parameters and Local Variables

Automatic (local) variables and parameters to functions are placed on the stack for most targets. For MCS51/DS390/HC08/S08 they 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 medium or 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.:

```c
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 intrinsic named address spaces and absolute addresses, e.g.:

```c
unsigned char foo(__xdata int parm)
{
    __xdata unsigned char i;
    __bit bvar;
    __data __at (0x31) unsigned char j;
    ...
}
```

In the above example the parameter parm and 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.

It is however allowed to use bit parameters in reentrant functions and also non-static local bit variables are supported. Efficient use is limited to 8 semi-bit registers in bit space. They are pushed and popped to stack as a single byte just like the normal registers.

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 intrinsic named address space is specified for a local variable, it will NOT be overlaid.

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 overlaid since these are implemented using external functions, e.g.:

```c
#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 behaviour when called from an interrupt service routine. The #pragma nooverlay ensures that the parameters and local variables for the function are NOT overlaid.

### 3.8 Interrupt Service Routines

#### 3.8.1 General Information

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

```c
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 (8051 specific) keyword `__using` can be used to tell the compiler to use the specified register bank when generating code for this function.

Interrupt service routines open the door for some very interesting bugs:

#### 3.8.1.1 Common interrupt pitfall: variable not declared `volatile`

If an interrupt service routine changes variables which are accessed by other functions these variables have to be declared `volatile`. See [http://en.wikipedia.org/wiki/Volatile_variable](http://en.wikipedia.org/wiki/Volatile_variable).

#### 3.8.1.2 Common interrupt pitfall: non-atomic access

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
3.8. INTERRUPT SERVICE ROUTINES  
CHAPTER 3. USING SDCC

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.

3.8.1.3 Common interrupt pitfall: stack overflow

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.

3.8.1.4 Common interrupt pitfall: use of non-reentrant functions

A special note here, integer multiplicative operators and floating-point operations might be implemented using external support routines, depending on the target architecture. If an interrupt service routine needs to do any of these operations on a target where functions are non-reentrant by default, 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.

Note, the type promotion required by ANSI C can cause 16 bit routines to be used without the programmer being aware of it. See f.e. the cast (unsigned char) (tail-1) within the if clause in section 3.11.2.

Calling other functions from an interrupt service routine on a target where functions are non-reentrant by default 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 non-reentrant functions should not be called from the main program while the interrupt service routine might be active. They also must not be called from low priority interrupt service routines while a high priority interrupt service routine might be active. You could use semaphores or make the function critical if all parameters are passed in registers.

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 to the maximum interrupt number specified.

<table>
<thead>
<tr>
<th>Interrupt #</th>
<th>Description</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>External 0</td>
<td>0x0003</td>
</tr>
<tr>
<td>1</td>
<td>Timer 0</td>
<td>0x000b</td>
</tr>
<tr>
<td>2</td>
<td>External 1</td>
<td>0x0013</td>
</tr>
<tr>
<td>3</td>
<td>Timer 1</td>
<td>0x001b</td>
</tr>
<tr>
<td>4</td>
<td>Serial</td>
<td>0x0023</td>
</tr>
<tr>
<td>5</td>
<td>Timer 2 (8052)</td>
<td>0x002b</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>0x0003 + 8*n</td>
</tr>
</tbody>
</table>

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 correspond 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 4.1.4 for details on customizing startup.

3.8.4 Z80 and Z180 Interrupt Service Routines

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

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

```c
void nmi_isr (void) __critical __interrupt
{
    ...
}
```

Since interrupts on the Z80 and Z180 are level-triggered (except for the NMI), interruptible interrupt handlers should only be used where hardware acknowledge is available.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interruptible interrupt handler</td>
<td>void f(void) __interrupt</td>
<td>Interrupt handler can be interrupted by further interrupts</td>
</tr>
<tr>
<td>Non-interruptible interrupt handler</td>
<td>void f(void) __critical __interrupt(0)</td>
<td>Interrupt handler can be interrupted by NMI only</td>
</tr>
<tr>
<td>NMI handler</td>
<td>void f(void) __critical __interrupt</td>
<td>Interrupt handler can be interrupted by NMI only</td>
</tr>
</tbody>
</table>

3.8.5 Rabbit 2000, 3000, 3000A and 4000 Interrupt Service Routines

SDCC does not attempt to generate an interrupt vector table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interruptible interrupt handler</td>
<td>void f(void) __interrupt</td>
<td>Interrupt handler can be interrupted by further interrupts</td>
</tr>
<tr>
<td>Non-interruptible interrupt handler</td>
<td>void f(void) __critical __interrupt(0)</td>
<td>Interrupt handler can be interrupted by interrupts of higher priority only</td>
</tr>
</tbody>
</table>

3.8.6 GBZ80 and TLCS-90 Interrupt Service Routines

SDCC does not attempt to generate an interrupt vector table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interruptible interrupt handler</td>
<td>void f(void) __interrupt</td>
<td>Interrupt handler can be interrupted by further interrupts</td>
</tr>
<tr>
<td>Non-interruptible interrupt handler</td>
<td>void f(void) __critical __interrupt(0)</td>
<td>Interrupt handler cannot be interrupted by further interrupts</td>
</tr>
</tbody>
</table>

3.8.7 STM8 Interrupt Service Routines

The STM8 interrupt table contains 31 entries: Reset (used by SDCC for program startup), trap and user interrupts 0 to 29. Where the keyword __interrupt is used for normal user interrupts, the __trap keyword is used for the trap handler:
void handler (void) __trap
{
  ...
}

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.

```c
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:

```c
__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):

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

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

```c
#define CLI __asm cli __endasm;
#define SEI __asm sei __endasm;

or for SDCC version 3.2.0 or newer:
#define CLI asm ("cli");
#define SEI asm ("sei");
```

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 re-enable 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 re-enabling interrupts within the interrupt routine. Check there is sufficient space on the stack and don’t add complexity unless you have to.
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.

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 Inline Assembler Code

3.11.1 Inline Assembler Code Formats

SDCC supports two formats for inline assembler code definition:

3.11.1.1 Old __asm ... __endasm; Format

Most of inline assembler code examples in this manual use the old inline assembler code format, but the new format could be used equivalently.

Example:

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.
3.11. INLINE ASSEMBLER CODE

3.11.1.2 New __asm__ ("inline_assembler_code") Format

The __asm__ inline assembler code format was introduced in SDCC version 3.2.0.

Example:

```c
__asm__ ("; This is a comment\nlabel:\nnop\n__endasm;"

3.11.2 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:

```c
unsigned char __far __at(0x7f00) buf[0x100];
unsigned char head, tail; /* if interrupts are involved see
                             section 3.8.1.1 about volatile */

void to_buffer( unsigned char c )
{
    if( head != (unsigned char)(tail-1) ) /* cast needed to avoid promotion to integer */
        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:

```asm
/* 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 != (unsigned char)(tail-1) ) /* cast needed to avoid promotion to integer */
        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 dp1,r3
        mov dph,#{_buf >> 8}
        mov a,r2
```

Note, that the single underscore form (__asm and __endasm) are not C99 compatible, and for C99 compatibility, the double-underscore form (__asm and __endasm) has to be used. The latter is also used in the library functions.
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:

```c
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 != (unsigned char)(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 != (unsigned char)(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. The assembler does not like some characters like ':' or '"' in comments. You'll find an 100+ pages assembler manual in sdcc/sdas/doc/asmlnk.txt or online at http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/sdas/doc/asmlnk.txt.

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 required that each assembly instruction be placed on a separate line. This is also recommended for labels (as the example shows). This is especially important to note when the inline assembler is placed in a C preprocessor macro as the preprocessor will normally put all replacing code on a single line. Only when the macro has each
assembly instruction on a single line that ends with a line continuation character will it be placed as separate lines in the resulting .asm file.

```c
#define DELAY \
__asm \n    nop \n    nop \n__endasm
```

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 peeph.def before using this option.

### 3.11.3 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:

```c
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
        reti ; MUST explicitly include ret or reti in _naked function.
    __endasm;
}
```

For an 8051 target, the generated simpleInterrupt looks like:

```asm
_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:

```
```
3.12. SUPPORT ROUTINES FOR INTEGER MULTIPLICATIVE OPERATORS  CHAPTER 3. USING SDCC

```c
__asm
    mov   b,#10
00001$:  ; some assembler code
    djnz  b,00001$
__endasm;
```

3.11.4 Use of Labels within Inline Assembler

SDCC allows the use of in-line assembler with a few restrictions regarding labels. All labels defined within inline assembler code have 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).\(^6\)

```
foo() {  /* some c code */
    __asm
        ; some assembler code
        ljmp 0003$
    __endasm;
    /* some more c code */
    clabel: /* inline assembler cannot reference this label */ \(^7\)
        __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.

3.12 Support routines for integer multiplicative operators

Depending on the target architecture, some integer multiplicative operators might be implemented by support routines. These support routines exist in portable C versions to facilitate porting to other MCUs, although depending on the target, assembler routines might be used instead. The following files contain some of the described routines, all of them can be found in `<installdir>/share/sdcc/lib`.

---

\(^6\)This is a slightly more stringent rule than absolutely necessary, but stays always on the safe side. Labels in the form of `nnnnn$` are local labels in the assembler, locality of which is confined within two labels of the standard form. The compiler uses the same form for labels within a function (but starting from `nnnnn=00100`); and places always a standard label at the beginning of a function, thus limiting the locality of labels within the scope of the function. So, if the inline assembler part would be embedded into C-code, an improperly placed non-local label in the assembler would break up the reference space for labels created by the compiler for the C-code, leading to an assembling error.

The numeric part of local labels does not need to have 5 digits (although this is the form of labels output by the compiler), any valid integer will do. Please refer to the assemblers documentation for further details.

\(^7\)Here, the C-label `clabel` is translated by the compiler into a local label, so the locality of labels within the function is not broken.
In the mcs51, ds390, hc08, s08, pic14 and pic16 backends they are by default compiled as non-reentrant; when targeting any of these architectures, 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.

### 3.13 Floating Point Support

SDCC supports (single precision 4 bytes) floating point numbers; the format is somewhat similar to IEEE, but it is not IEEE; in particular, denormalized floating point numbers are not supported. The floating point support routines are derived from gcc’s floatlib.c and consist of the following routines:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_fsadd.c</td>
<td>add floating point numbers</td>
</tr>
<tr>
<td>_fssub.c</td>
<td>subtract floating point numbers</td>
</tr>
<tr>
<td>_fsdiv.c</td>
<td>divide floating point numbers</td>
</tr>
<tr>
<td>_fsmul.c</td>
<td>multiply floating point numbers</td>
</tr>
<tr>
<td>_fs2uchar.c</td>
<td>convert floating point to unsigned char</td>
</tr>
<tr>
<td>_fs2char.c</td>
<td>convert floating point to signed char</td>
</tr>
<tr>
<td>_fs2uint.c</td>
<td>convert floating point to unsigned int</td>
</tr>
<tr>
<td>_fs2int.c</td>
<td>convert floating point to signed int</td>
</tr>
<tr>
<td>_fs2ulong.c</td>
<td>convert floating point to unsigned long</td>
</tr>
<tr>
<td>_fs2long.c</td>
<td>convert floating point to signed long</td>
</tr>
<tr>
<td>_uchar2fs.c</td>
<td>convert unsigned char to floating point</td>
</tr>
<tr>
<td>_char2fs.c</td>
<td>convert char to floating point number</td>
</tr>
<tr>
<td>_uint2fs.c</td>
<td>convert unsigned int to floating point</td>
</tr>
<tr>
<td>_int2fs.c</td>
<td>convert int to floating point numbers</td>
</tr>
<tr>
<td>_ulong2fs.c</td>
<td>convert unsigned long to floating point number</td>
</tr>
<tr>
<td>_long2fs.c</td>
<td>convert long to floating point number</td>
</tr>
</tbody>
</table>

### 3.14 Library Routines

<pending: this is messy and incomplete - a little more information is at [http://sdcc.sourceforge.net/wiki/index.php/List_of_the_SDCC_library](http://sdcc.sourceforge.net/wiki/index.php/List_of_the_SDCC_library)>

#### 3.14.1 Compiler support routines (_gptrget, _mulint etc.)

#### 3.14.2 Stdclib functions (puts, printf, strcat etc.)

**getchar(), putchar()**  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. And whether a lf to crlf conversion within putchar() is intended. You’ll find examples for serial routines for instance sdcc/device/lib. For the mcs51 this minimalistic polling putchar() routine might be a start:

```c
int putchar (int c) {
    while (!TI) /* assumes UART is initialized */
    ;
    TI = 0;
    SBUF = c;
    return c;
}
```

**printf()** The default printf() implementation in printf_large.c does not support float (except on ds390), only <NO FLOAT> will be printed instead of the value. To enable floating point output, 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. To enable float support for the pic16 targets, see 4.7.9.

If you’re short on code memory you might want to use printf_small() instead of printf(). For the mcs51 there additionally are assembly versions printf_tiny() (subset of printf using less than 270 bytes) and printf_fast() and printf_fast_f() (floating-point aware version of printf_fast) 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). Be sure to use only one of these printf options within a project.

Feature matrix of different printf options on mcs51.

<table>
<thead>
<tr>
<th>mcs51</th>
<th>printf</th>
<th>printf</th>
<th>printf_small</th>
<th>printf_fast</th>
<th>printf_fast_f</th>
<th>printf_tiny</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>USE_FLOATS=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>filename</td>
<td>printf_large.c</td>
<td>printf_large.c</td>
<td>printf.c</td>
<td>printf_fast.c</td>
<td>printf_fast_f.c</td>
<td>printf_tiny.c</td>
</tr>
<tr>
<td>&quot;Hello World&quot;</td>
<td>1.7k / 2.4k</td>
<td>4.3k / 5.6k</td>
<td>1.2k / 1.8k</td>
<td>1.3k / 1.3k</td>
<td>1.9k / 1.9k</td>
<td>0.44k / 0.44k</td>
</tr>
<tr>
<td>size</td>
<td>1.4k / 2.0k</td>
<td>2.8k / 3.7k</td>
<td>0.45k / 0.47k (+_itoa)</td>
<td>1.2k / 1.2k</td>
<td>1.6k / 1.6k</td>
<td>0.26k / 0.26k</td>
</tr>
<tr>
<td>code size</td>
<td>cdipsux</td>
<td>cdipsux</td>
<td>cdosx</td>
<td>cdxux</td>
<td>cdfsux</td>
<td>cdxux</td>
</tr>
<tr>
<td>formats</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>long (32 bit) support</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>byte arguments on stack</td>
<td>b</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float format</td>
<td>-</td>
<td>%f</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>%f</td>
</tr>
<tr>
<td>float formats %e %g</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>field width</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>string speed(^8), small / large</td>
<td>1.52 / 2.59 ms</td>
<td>1.53 / 2.62 ms</td>
<td>0.92 / 0.93 ms</td>
<td>0.45 / 0.45 ms</td>
<td>0.46 / 0.46 ms</td>
<td>0.45 / 0.45 ms</td>
</tr>
<tr>
<td>int speed(^9), small / large</td>
<td>3.01 / 3.61 ms</td>
<td>3.01 / 3.61 ms</td>
<td>3.51 / 18.13 ms</td>
<td>0.22 / 0.22 ms</td>
<td>0.23 / 0.23 ms</td>
<td>0.25 / 0.25 ms(^11)</td>
</tr>
<tr>
<td>long speed(^12), small / large</td>
<td>5.37 / 6.31 ms</td>
<td>5.37 / 6.31 ms</td>
<td>8.71 / 40.65 ms</td>
<td>0.40 / 0.40 ms</td>
<td>0.40 / 0.40 ms</td>
<td>-</td>
</tr>
</tbody>
</table>

---

\(^8\)Range limited to +/- 4294967040, precision limited to 8 digits past decimal

\(^9\)Execution time of printf("%s%c%sc%c", "Hello", ' ', "World", '!', 'r', 'u'); standard 8051 @ 22.1184 MHz, empty putchar()

\(^10\)Execution time of printf("%d", -12345); standard 8051 @ 22.1184 MHz, empty putchar()

\(^11\)printf_tiny integer speed is data dependent, worst case is 0.33 ms

\(^12\)Execution time of printf("%ld", -123456789); standard 8051 @ 22.1184 MHz, empty putchar()
### 3.14.2.2 <malloc.h>

As of SDCC 2.6.2 you no longer need to call an initialization routine before using dynamic memory allocation and a default heap space of 1024 bytes is provided for malloc to allocate memory from. If you need a different heap size you need to recompile _heap.c with the required size defined in HEAP_SIZE. It is recommended to make a copy of this file into your project directory and compile it there with:

```
sdcc -c _heap.c -D HEAP_SIZE=2048
```

And then link it with:

```
sdcc main.rel _heap.rel
```

### 3.14.3 Math functions (sinf, powf, sqrtf etc.)

#### 3.14.3.1 <math.h>

See definitions in file <math.h>.

### 3.14.4 Other libraries

Libraries included in SDCC should have a license at least as liberal as the GPLv2+LE. Exception are pic device libraries and header files which are derived from Microchip header (.inc) and linker script (.lkr) files. Microchip requires that "The header files should state that they are only to be used with authentic Microchip devices" which makes them incompatible with GPL.

If you have ported some library or want to share experience about some code which f.e. falls into any of these categories Busses (I²C, 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/p/sdcc/mailman/sdcc-user/](http://sourceforge.net/p/sdcc/mailman/sdcc-user/) 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.

### 3.15 Memory Models

#### 3.15.1 MCS51 Memory Models

##### 3.15.1.1 Small, Medium, Large and Huge

SDCC allows four memory models for MCS51 code, small, medium, large and huge. 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 for all models (however, the libraries for --stack-auto are compiled

---

<table>
<thead>
<tr>
<th>mcs51</th>
<th>printf</th>
<th>printf</th>
<th>printf_small</th>
<th>printf_fast</th>
<th>printf_fast_f</th>
<th>printf_tiny</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>printf_large.c</td>
<td>printf_large.c</td>
<td>printf.c</td>
<td>printf_fast.c</td>
<td>printf_fast_f.c</td>
<td>printf_tiny.c</td>
</tr>
<tr>
<td>&quot;Hello World&quot; size</td>
<td>1.7k / 2.4k</td>
<td>4.3k / 5.6k</td>
<td>1.2k / 1.8k</td>
<td>1.3k / 1.3k</td>
<td>1.9k / 1.9k</td>
<td>0.44k / 0.44k</td>
</tr>
<tr>
<td>code size</td>
<td>1.4k / 2.0k</td>
<td>2.8k / 3.7k</td>
<td>0.45k / 0.47k (+ ltoa)</td>
<td>1.2k / 1.2k</td>
<td>1.6k / 1.6k</td>
<td>0.26k / 0.26k</td>
</tr>
<tr>
<td>float speed</td>
<td>-</td>
<td>7.49 / 22.47 ms</td>
<td>-</td>
<td>-</td>
<td>1.04 / 1.04 ms</td>
<td>-</td>
</tr>
</tbody>
</table>

---

13 Execution time of printf("%.3f", -12345.678); standard 8051 @ 22.1184 MHz, empty putchar()
3.15. MEMORY MODELS

for the small and large models only). The compiled library modules are contained in separate directories as small, medium, large and huge so that you can link to the appropriate set.

When the medium, large or huge model is used all variables declared without specifying an intrinsic named address space will be allocated into the external ram, this includes all parameters and local variables (for non-reentrant functions). Medium model uses pdata and large and huge models use xdata. When the small model is used variables without an explicitly specified intrinsic named address space are allocated in the internal ram.

The huge model compiles all functions as banked\textsuperscript{4.1.3} and is otherwise equal to large for now. All other models compile the functions without bankswitching by default.

Judicious usage of the processor specific intrinsic named address spaces 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.

3.15.1.2 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 \ref{sec:port-p2}), therefore when using the External Stack option, this port may not be used by the application program.

3.15.2 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 bit mode (although \texttt{tinibios} in the ds390 libraries will do that for you). If you don’t use \texttt{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.

3.15.3 STM8 Memory Models

SDCC implements two memory models for the STM8: medium (default) and large. Modules compiled with different memory models should never be combined together. The library routines supplied with the compiler are compiled for all models.

In the medium model the address space is 16 bits for both objects and functions, allowing for a memory space of 64 KB. Since the STM8 typically has Flash starting at 0x8000, this means that only up to 32 KB of Flash can be used (most STM8 devices don’t have more than 32 KB of Flash).

In the large memory model, the address space is 16 bits for objects and 24 bits for functions. Since the STM8 typically has flash starting at 0x8000, this means that up to 32 KB of flash can be used for constant data, while the whole Flash can be used for functions. Code generated for the large model is slightly bigger and slower and needs slightly more stack space than code generated for the medium model.
3.16 Pragmas

Pragmas are used to turn on and/or off certain compiler options. Some of them are closely related to corresponding command-line options (see section 3.3 on page 28). Pragmas should be placed before and/or after a function, placing pragmas inside a function body could have unpredictable results.

SDCC supports the following #pragma directives:

- **save** - this will save most 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 and 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 and exit for these functions to save and restore the registers used by these functions, this can SUBSTANTIALLY reduce code and 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 (see page on page 31), 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[,bits]]]]} - The exclude pragma disables the generation of pairs of push/pop instructions in Interrupt Service Routines. 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’re on your own now ;-(. See also the command line option --less-pedantic on page 32.

More specifically, the following warnings will be disabled: `comparison is always [true/false] due to limited range of data type` (94); `overflow in implicit constant conversion` (158); `[the (in)famous] conditional flow changed by optimizer: so said EVELYN the modified DOG` (110); `function 'function name' must return value` (59).

Furthermore, warnings of less importance (of PEDANTIC and INFO warning level) are disabled, too, namely: `constant value ']' , out of range` (81); `[left/right] shifting more than size of object changed to zero` (116); `unreachable code` (126); `integer overflow in expression` (165); `unmatched #pragma save and #pragma restore` (170); `comparison of 'signed char' with 'unsigned char' requires promotion to int` (185); `ISO C90 does not support flexible array members` (187); `extended stack by [number] bytes for compiler temp(s) :in function 'function name'`: `[ ]` (114); `function 'function name'`: `# edges [number] , # nodes [number] , cyclomatic complexity [number]` (121).

- **disable_warning <nnnn>** - the compiler will not warn you anymore about warning number <nnnn>.

- **nogese** - 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 section 8.1.4.

- **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 omitting the optional interrupt number after the __interrupt keyword, see section 3.8 about interrupts.

- **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.

• **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.

• **std_c99** - Follow the C99 standard and disable SDCC features that conflict with the standard.

• **codeseg <name>** - Use this name (max. 8 characters) for the code segment. See option --codeseg.

• **constseg <name>** - Use this name (max. 8 characters) for the const segment. See option --constseg.

The preprocessor SDCPP supports the following #pragma directives:

• **pedantic_parse_number (+ | -)** - Pedantic parse numbers so that situations like 0xfe-LO_B(3) are parsed properly and the macro LO_B(3) gets expanded. Default is off. See also the --pedantic-parse-number command line option on page 29.

  Below is an example on how to use this pragma. *Note: this functionality is not in conformance with standard!*

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

• **preproc_asm (+ | -)** - switch the __asm __endasm block preprocessing on / off. Default is on. Below is an example on how to use this pragma.

  ```c
  #pragma preproc_asm -
  /* this is a c code nop */
  #define NOP ;
  void foo (void)
  {
    ...
    while (--i)
      NOP
    ...
    __asm
    ; this is an assembler nop instruction
    ; it is not preprocessed to ‘;’ since the asm preprocessing is disabled
    NOP
    __endasm;
  ```
The pragma preproc_asm should not be used to define multilines of assembly code (even if it supports it), since this behavior is only a side effect of sdcpp __asm __endasm implementation in combination with pragma preproc_asm and is not in conformance with the C standard. This behavior might be changed in the future sdcpp versions. To define multilines of assembly code you have to include each assembly line into its own __asm __endasm block. Below is an example for multiline assembly defines.

```c
#define Nop __asm \
  nop \ 
__endasm

#define ThreeNops Nop; \
Nop; \ 
Nop

void foo (void)
{
  ... 
  ThreeNops; 
  ...
}
```

- `sdcc_hash (+1 -)` - Allow "naked" hash in macro definition, for example:

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

Default is off. Below is an example on how to use this pragma.

```c
#pragma preproc_asm +
#pragma sdcc_hash +

#define ROMCALL(x) \
  mov R6_B3, #(x & 0xff) \
  mov R7_B3, #((x >> 8) & 0xff) \
  lcall __romcall

  ... 
__asm
  ROMCALL(72) 
__endasm;
  ...
```

Some of the pragmas are intended to be used to turn-on or off certain optimizations which might cause the compiler to generate extra stack and/or 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 and optimizations for a given function.

```c
#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 pragmas be used when changing options for a function.
### 3.17 Defines Created by the Compiler

Besides defines from the C standards, the compiler creates the following `#defines`:

<table>
<thead>
<tr>
<th><code>#define</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__SDCC</code></td>
<td>Always defined. Version number string (e.g. <code>SDCC_3_2_0</code> for sdcc 3.2.0).</td>
</tr>
<tr>
<td><code>SDCC</code></td>
<td>OBSOLETE. WILL BE REMOVED IN THE FUTURE. CURRENTLY Only defined for the mcs51 backend (and only if <code>--std-cXX</code> is not used). This macro has been available since SDCC 2.5.6 and is the version number as an int (ex. 256). PLEASE USE OTHER VERSION MACROS INSTEAD!</td>
</tr>
<tr>
<td><code>__SDCC_mcs51</code> or <code>__SDCC_ds390</code> or <code>__SDCC_z80</code>, etc.</td>
<td>depending on the model used (e.g.: <code>-mds390</code>). Older versions used <code>SDCC_mcs51</code>, etc instead.</td>
</tr>
<tr>
<td><code>__SDCC_STACK_AUTO</code></td>
<td>when <code>--stack-auto</code> option is used</td>
</tr>
<tr>
<td><code>__SDCC_MODEL_SMALL</code></td>
<td>when <code>--model-small</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_MODEL_MEDIUM</code></td>
<td>when <code>--model-medium</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_MODEL_LARGE</code></td>
<td>when <code>--model-large</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_MODEL_HUGE</code></td>
<td>when <code>--model-huge</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_USE_XSTACK</code></td>
<td>when <code>--xstack</code> option is used</td>
</tr>
<tr>
<td><code>__SDCC_STACK_TENBIT</code></td>
<td>when <code>-mds390</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_MODEL_FLAT24</code></td>
<td>when <code>-mds390</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_VERSION_MAJOR</code></td>
<td>Always defined. SDCC major version number. E.g. 3 for SDCC 3.5.0</td>
</tr>
<tr>
<td><code>__SDCC_VERSION_MINOR</code></td>
<td>Always defined. SDCC minor version number. E.g. 5 for SDCC 3.5.0</td>
</tr>
<tr>
<td><code>__SDCC_VERSION_PATCH</code></td>
<td>Always defined. SDCC patchlevel version number. E.g. 0 for SDCC 3.5.0</td>
</tr>
<tr>
<td><code>__SDCC_REVISION</code></td>
<td>Always defined. SDCC svn revision number. Older versions of sdcc used <code>SDCC_REVISION</code> instead.</td>
</tr>
<tr>
<td><code>SDCC_PARMS_IN_BANK1</code></td>
<td>when <code>--parms-in-bank1</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_ALL_CALLEE_SAVES</code></td>
<td>when <code>--all-callee-saves</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_FLOAT_REENT</code></td>
<td>when <code>--float-reent</code> is used</td>
</tr>
<tr>
<td><code>__SDCC_INT_LONG_REENT</code></td>
<td>when <code>--int-long-reent</code> is used</td>
</tr>
</tbody>
</table>
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:

```c
__sfr __at (0x85) __XPAGE;  /* Ramtron VRS51 family a.k.a. MPAGE */
__sfr __at (0x92) __XPAGE;  /* Cypress EZ-USB family, Texas Instruments (Chipcon) a.k.a. MPAGE */
__sfr __at (0x91) __XPAGE;  /* Infineon (Siemens) C500 family a.k.a. XPAGE */
__sfr __at (0xa1) __XPAGE;  /* some Silicon Labs (Cynthal) chips a.k.a. EMI0CN */
__sfr __at (0x9a) __XPAGE;  /* some Silicon Labs (Cynthal) chips a.k.a. EMI0CN */
```

There are also devices without anything resembling __XPAGE, but luckily they usually have dual data-pointers. For these devices a different method can be used to correctly initialize xdata variables. A default implementation is already in crtxinit.asm but it needs to be assembled manually with DUAL_DPTR set to 1.

For more exotic implementations further customizations may be needed. See section 4.1.4 for other possibilities.

4.1.2 Other Features available by SFR

Some MCS51 variants offer features like Dual 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.

4.1.3 Bankswitching

Bankswitching (a.k.a. code banking) is a technique to increase the code space above the 64k limit of the 8051.
4.1.3.1 Hardware

<table>
<thead>
<tr>
<th>8000-FFFF</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-7FFF</td>
<td>common</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SiLabs C8051F120 example

Usually the hardware uses some sfr (an output port or an internal sfr) to select a bank and put it in the banked area of the memory map. The selected bank usually becomes active immediately upon assignment to this sfr and when running inside a bank it will switch out this code it is currently running. Therefore you cannot jump or call directly from one bank to another and need to use a so-called trampoline in the common area. For SDCC an example trampoline is in crtbank.asm and you may need to change it to your 8051 derivative or schematic. The presented code is written for the C8051F120.

When calling a banked function SDCC will put the LSB of the functions address in register R0, the MSB in R1 and the bank in R2 and then call this trampoline __sdcc_banked_call. The current selected bank is saved on the stack, the new bank is selected and an indirect jump is made. When the banked function returns it jumps to __sdcc_banked_ret which restores the previous bank and returns to the caller.

4.1.3.2 Software

When writing banked software using SDCC you need to use some special keywords and options. You also need to take over a bit of work from the linker.

To create a function that can be called from another bank it requires the keyword __banked. The caller must see this in the prototype of the callee and the callee needs it for a proper return. Called functions within the same bank as the caller do not need the __banked keyword nor do functions in the common area. Beware: SDCC does not know or check if functions are in the same bank. This is your responsibility!

Normally all functions you write end up in the segment CSEG. If you want a function explicitly to reside in the common area put it in segment HOME. This applies for instance to interrupt service routines as they should not be banked.

Functions that need to be in a switched bank must be put in a named segment. The name can be mostly anything up to eight characters (e.g. BANK1). To do this you either use --codeseg BANK1 (See 3.3.4) on the command line when compiling or #pragma codeseg BANK1 (See 3.16) at the top of the C source file. The segment name always applies to the whole source file and generated object so functions for different banks need to be defined in different source files.

When linking your objects you need to tell the linker where to put your segments. To do this you use the following command line option to SDCC: -Wl-b BANK1=0x18000 (See 3.3.5). This sets the virtual start address of this segment. It sets the banknumber to 0x01 and maps the bank to 0x8000 and up. The linker will not check for overflows, again this is your responsibility.

4.1.4 MCS51/DS390 Startup Code

The compiler triggers the linker to link certain initialization modules from the runtime library called crt<something>. Only the necessary ones are linked, for instance crtxstack.asm (GSINIT1, GSINIT5) is not linked unless the --xstack option is used. These modules are highly entangled by the use of special segments/areas, but a common layout is shown below:

(main.asm)

.area HOME {CODE}
__interrupt_vect:
    ljmp __sdcc_gsinit_startup

(crtstart.asm)

.area GSINIT0 {CODE}
__sdcc_gsinit_startup:
    mov sp,__start__stack - 1

(crtxstack.asm)
.area GSINIT1 (CODE)
__sdcc_init_xstack::
; Need to initialize in GSINIT1 in case the user’s __sdcc_external_startup uses the xstack.
    mov __XPAGE,#((__start__xstack >> 8))
    mov __spx,#__start__xstack
(crtstart.asm)

.area GSINIT2 (CODE)
lcall __sdcc_external_startup
mov a,dpl
jz __sdcc_init_data
ljmp __sdcc_program_startup
__sdcc_init_data:
(crtxinit.asm)

.area GSINIT3 (CODE)
__mcs51_genXINIT::
    mov r1,#l_XINIT
    mov a,r1
    orl a,#(l_XINIT >> 8)
    jz 00003$
    mov r2,#((l_XINIT+255) >> 8)
    mov dptr,#s_XINIT
    mov r0,#s_XISEG
    mov __XPAGE,#(s_XISEG >> 8)
00001$: clr a
    movc a,@a+dptr
    movx @r0,a
    inc dptr
    inc r0
    cjne r0,#0,00002$
    inc __XPAGE
00002$: djnz r1,00001$
    djnz r2,00001$
    mov __XPAGE,#0xFF
00003$:
(crtclear.asm)

.area GSINIT4 (CODE)
__mcs51_genRAMCLEAR::
    clr a
    mov r0,#(l_IRAM-1)
00004$: mov @r0,a
    djnz r0,00004$
    ; __mcs51_genRAMCLEAR() end
(crtxclear.asm)

.area GSINIT4 (CODE)
__mcs51_genXRAMCLEAR::
    mov r0,#l_PSEG
    mov a,r0
    orl a,#(l_PSEG >> 8)
    jz 00006$
    mov r1,#s_PSEG
    mov __XPAGE,#(s_PSEG >> 8)
    clr a
00005$: movx @r1,a
    inc r1
    djnz r0,00005$
00006$:  
    mov r0,#l_XSEG  
    mov a,r0  
    orl a,#((l_XSEG >> 8))  
    jz 00008$  
    mov r1,((l_XSEG + 255) >> 8)  
    mov dptr,#s_XSEG  
    clr a  
00007$:  
    movx @dptr,a  
    inc dptr  
    djnz r0,00007$  
    djnz r1,00007$  
00008$:  

(crtxstack.asm)  

.area GSINIT5 (CODE)  
; Need to initialize in GSINIT5 because __mcs51_genXINIT modifies __XPAGE  
; and __mcs51_genRAMCLEAR modifies _spx.  
    mov __XPAGE,#(__start__xstack >> 8)  
    mov _spx,#__start__xstack  

(application modules)  

.area GSINIT (CODE)  

(main.asm)  

.area GSFINAL (CODE)  
    ljmp __sdcc_program_startup  
    ;--------------------------------------------------------  
    ; Home  
    ;--------------------------------------------------------  
    .area HOME (CODE)  
    .area CSEG (CODE)  
    __sdcc_program_startup:  
    lcall _main  
    ; return from main will lock up  
    sjmp .  

One of these modules (crtstart.asm) contains a call to the C routine _sdcc_external_startup() at the start of the CODE area. This routine is also in the runtime library and returns 0 by default. 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 an _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.  

While these initialization modules are meant as generic startup code there might be the need for customization. Let’s assume the return value of _sdcc_external_startup() in crtstart.asm should not be checked (or _sdcc_external_startup() should not be called at all). The recommended way would be to copy crtstart.asm (f.e. from http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/device/lib/mcs51/crtstart.asm) into the source directory, adapt it there, then assemble it with sdas8051 -plosgff1 crtstart.asm and when linking your project explicitly specify crtstart.rel. As a bonus a listing of the relocated object file crtstart.rst is generated.  

1"-plosgff" are the assembler options used in http://sdcc.svn.sourceforge.net/viewvc/sdcc/trunk/sdcc/device/lib/mcs51/Makefile.in?view=markup
4.1.5 Interfacing with Assembler Code

4.1.5.1 Global Registers used for Parameter Passing

The compiler always uses the global registers $DPL$, $DPH$, $B$ and $ACC$ to pass the first (non-bit) parameter to a function, and also to pass the return value of function; according to the following scheme: one byte return value in $DPL$, two byte value in $DPL$ (LSB) and $DPH$ (MSB). three byte values (generic pointers) in $DPH$, $DPL$ and $B$, and four byte values in $DPH$, $DPL$, $B$ and $ACC$. Generic pointers contain type of accessed memory in $B$: $0x00$ – xdata/far, $0x40$ – idata/near – , $0x60$ – pdata, $0x80$ – code.

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).

Bit parameters are passed in a virtual register called ‘bits’ in bit-addressable space for reentrant functions or allocated directly in bit memory otherwise.

Functions (with two or more parameters or bit parameters) that are called through function pointers must therefore be reentrant so the compiler knows how to pass the parameters.

4.1.5.2 Register usage

Unless the called function is declared as _naked, or the --callee-saves/--all-callee-saves command line option or the corresponding callee_saves pragma are used, the caller will save the registers ($R0-R7$) around the call, so the called function can destroy they content freely.

If the called function is not declared as _naked, the caller will swap register banks around the call, if caller and callee use different register banks (having them defined by the _using modifier).

The called function can also use $DPL$, $DPH$, $B$ and $ACC$ observing that they are used for parameter/return value passing.

4.1.5.3 Assembler Routine (non-reentrant)

In the following example the function $c\_func$ calls an assembler routine $asm\_func$, which takes two parameters.

```c
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:

```assembly
.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
```

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 $DPH$, $DPL$, $B$ and $ACC$ according to the
description above. The variable name for the second parameter will be _<function_name>_PARM_2.

Assemble the assembler routine with the following command:

`sdas8051 -losg asmfunc.asm`

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

`sdcc cfunc.c asmfunc.rel`

### 4.1.5.4 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. before the call the second leftmost parameter will be on the top of the stack (the leftmost parameter is passed in registers). Here is an example:

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

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

int main()
{
    return c_func(10,9,8);
}
```

The corresponding (unoptimized) assembler routine is:

```asm
.globl _asm_func
_asm_func:
push _bp
mov _bp,sp ; stack contains: _bp, return address, second parameter, third parameter
mov r2,dpl
mov a,__bp
add a,#0xfd ; calculate pointer to the second parameter
mov r0,a
mov a,__bp
add a,#0xfc ; calculate pointer to the rightmost parameter
mov r1,a
mov a,@r0
add a,@r1
add a,r2 ; calculate the result (= sum of all three parameters)
mov dpl,a ; return value goes into dptr (cast into int)
mov dph,#0x00
mov sp,_bp
pop _bp
ret
```

The compiling and linking procedure remains the same, however note the extra entry & exit linkage required for the assembler code, _bp is the stack frame pointer and is used to compute the offset into the stack for parameters and local variables.
4.2 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, Z180, Rabbit 2000/3000, Rabbit 3000A, GBZ80, eZ80 and TLCS-90 ports

SDCC can target the Z80, Z180, Rabbit 2000/3000, Rabbit 3000A and LR35902, the Nintendo GameBoy’s Z80-like gbz80.

When a frame pointer is used, it resides in IX. Register A, B, C, D, E, H, L and IY are used as a temporary registers 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).

When enabling optimizations using --opt-code size and a sufficiently high value for --max-allocs-per-node sdcc typically generates much better code for these architectures than many other compilers. A comparison of compilers for these architecture can be found at http://sdcc.sourceforge.net/wiki/index.php/Z80_code_size.

4.3.1 Startup Code

On the Z80 the startup code is inserted by linking with crt0.rel 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.rel. When using a custom crt0.rel it needs to be listed first when linking.

4.3.2 Complex instructions

The Z80 and some derivatives support complex instructions, such as ldir, cpir, ... . SDCC only emits these instructions for functions in the standard library. Thus, e.g. copying one array into another is more efficient when using memcpy() than by using a a user-written loop.

Depending on the target, code generation options and the parameters to the call, SDCC emits ldir for memcpy(), ldir or lsdir for memset(), ldi for strcpy(), ldi for strncpy(). Other library functions use the complex instructions as well, but for those, function calls are generated.

4.3.3 Calling conventions

By default, all parameters are passed on the stack, right-to-left. 8-bit return values are passed in l, 16-bit values in hl, 32-bit values in dehl. Except for the GBZ80, where 8-bit values are passed in e, 16-bit values in de, 32-bit values in hlde. Larger return values are passed in memory in a location specified by the caller through a hidden pointer argument.

There are three other calling conventions supported, which can be specified using the keywords __smallc, __z88dk_fastcall (not on GBZ80) and __z88dk_callee. They are primarily intended for compatibility with libraries written for other compilers. For __z88dk_fastcall, there may be only one parameter of at most 32 bits, which is passed the same way as the return value. For __z88dk_callee, parameters are passed on the stack, but the stack is not adjusted for the parameters after the call (thus the callee has to do this instead). __z88dk_callee is currently supported on the caller side only. __z88dk_callee can be combined with __smallc.

4.3.4 Small-C calling convention

Functions declared as __smallc are called using the Small-C calling convention (passing arguments on-stack left-to-right, 1 byte arguments are passed as 2 bytes, with the value in the lower byte). This way assembler routines originally written for Small-C or code generated by Small-C can be called from SDCC. Currently variable arguments are not yet supported (neither on the caller nor on the callee side).
4.4. THE HC08 AND S08 PORTS

4.3.5 Unsafe reads

Usually, Z80-based systems (except for the Gameboy) have separate I/O and memory spaces, and any memory location can be read without side-effects. For such systems, the option –allow-unsafe-reads can be used to enable some extra optimizations.

4.4 The HC08 and S08 ports

The port to the Freescale/Motorola HC08 and S08 does not yet generate code as compact as that generated by some non-free compilers. A comparison of compilers for these architecture can be found at http://sdcc.sourceforge.net/wiki/index.php/Hc08_code_size.

4.4.1 Startup Code

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

4.5 The STM8 port

4.5.1 Calling convention

Arguments are passed on the stack right-to-left. Return values are in a (8 bit), x (16 bit), xyl (24 bit), xy (32 bit) or use a hidden extra pointer parameter pointing to the location (anything wider than 32 bit).

4.6 The PIC14 port

The PIC14 port adds support for Microchip™ PIC™ MCUs with 14 bit wide instructions. This port is not yet mature and still lacks many features. However, it can work for simple code.

Currently supported devices include:

10F320, 10F322, 10LF320, 10LF322
12F609, 12F615, 12F617, 12F629, 12F635, 12F675, 12F683
12F752
12HV752
16C62, 16C63A, 16C65B
16C71, 16C72, 16C73B, 16C74B
16C432, 16C433
16C554, 16C557, 16C558
16C620, 16C620A, 16C621, 16C621A, 16C622, 16C622A
16C710, 16C711, 16C715, 16C717, 16C745, 16C765, 16C770, 16C771, 16C773, 16C774, 16C781, 16C782
16C925, 16C926
16CR73, 16CR74, 16CR76, 16CR77
16CR620A
16F72, 16F73, 16F74, 16F76, 16F77
16F84, 16F84A, 16F87, 16F88
16F610, 16F616, 16F627, 16F627A, 16F628, 16F628A, 16F630, 16F631, 16F636, 16F639, 16F648A
16F676, 16F677, 16F684, 16F685, 16F687, 16F688, 16F689, 16F690
16F707, 16F716, 16F720, 16F721, 16F722, 16F722A, 16F723, 16F723A, 16F724, 16F726, 16F727
16F737, 16F747, 16F753, 16F767, 16F777, 16F785
16F818, 16F819, 16F870, 16F871, 16F872, 16F873, 16F873A, 16F874, 16F874A, 16F876, 16F876A
16F877, 16F877A, 16F882, 16F883, 16F884, 16F886, 16F887
16F913, 16F914, 16F916, 16F917, 16F946
16LF74, 16LF76, 16LF77
16LF84, 16LF84A, 16LF87, 16LF88
16LF627, 16LF627A, 16LF628, 16LF628A, 16LF648A
16LF707, 16LF720, 16LF721, 16LF722, 16LF722A, 16LF723, 16LF723A, 16LF724, 16LF726, 16LF727
16LF747, 16LF767, 16LF777
4.6. THE PIC14 PORT

CHAPTER 4. NOTES ON SUPPORTED PROCESSORS

16LF818, 16LF819, 16LF870, 16LF871, 16LF872, 16LF873, 16LF873A, 16LF874, 16LF874A
16LF876, 16LF876A, 16LF877, 16LF877A
16HV610, 16HV616, 16HV753, 16HV785

Supported devices with enhanced cores:

12F1501, 12F1571, 12F1572, 12F1612, 12F1822, 12F1840
12LF1501, 12LF1552, 12LF1571, 12LF1572, 12LF1612, 12LF1822, 12LF1840, 12LF1840T39A,
12LF1840T48A
16F1454, 16F1455, 16F1458, 16F1459
16F1503, 16F1507, 16F1508, 16F1509, 16F1512, 16F1513, 16F1516, 16F1517, 16F1518, 16F1519
16F1526, 16F1527, 16F1574, 16F1575, 16F1578, 16F1579
16F1613, 16F1614, 16F1615, 16F1618, 16F1619
16F1703, 16F1704, 16F1705, 16F1707, 16F1708, 16F1709, 16F1713, 16F1716, 16F1717, 16F1718, 16F1719
16F1764, 16F1765, 16F1768, 16F1769, 16F1773, 16F1776, 16F1777, 16F1778, 16F1779
16F1782, 16F1783, 16F1784, 16F1786, 16F1787, 16F1788, 16F1789
16F1823, 16F1824, 16F1825, 16F1826, 16F1827, 16F1828, 16F1829, 16F1829LIN, 16F1847
16F1933, 16F1934, 16F1936, 16F1937, 16F1938, 16F1939, 16F1946, 16F1947
16F18313, 16F18323, 16F18324, 16F18325, 16F18344, 16F18345,
16F18855, 16F18875
16LF1454, 16LF1455, 16LF1458, 16LF1459
16LF1503, 16LF1507, 16LF1508, 16LF1509, 16LF1512, 16LF1513, 16LF1516, 16LF1517, 16LF1518,
16LF1519,
16LF1526, 16LF1527
16LF1554, 16LF1559, 16LF1566, 16LF1567, 16LF1574, 16LF1575, 16LF1578, 16LF1579
16LF1613, 16LF1614, 16LF1615, 16LF1618, 16LF1619
16LF1703, 16LF1704, 16LF1705, 16LF1707, 16LF1708, 16LF1709, 16LF1713, 16LF1716, 16LF1717,
16LF1718, 16LF1719
16LF1764, 16LF1765, 16LF1768, 16LF1769, 16LF1773, 16LF1776, 16LF1777, 16LF1778, 16LF1779
16LF1782, 16LF1783, 16LF1784, 16LF1786, 16LF1787, 16LF1788, 16LF1789,
16LF1823, 16LF1824, 16LF1825, 16LF1827, 16LF1828, 16LF1829, 16LF1847
16LF1902, 16LF1903, 16LF1904, 16LF1906, 16LF1907
16LF1933, 16LF1934, 16LF1936, 16LF1937, 16LF1938, 16LF1939, 16LF1946, 16LF1947
16LF18313, 16LF18323, 16LF18324, 16LF18325, 16LF18344, 16LF18345,
16LF18855, 16LF18875

An up-to-date list of currently supported devices can be obtained via `sdcc -mpic14 -phelp foo.c` (foo.c
must exist...).

4.6.1 PIC Code Pages and Memory 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 assigns all static functions of a single .c file into the
same code page.

To get the best results, follow these guidelines:

1. Make local functions static, as non static functions require code page selection overhead.
   Due to the way sdcc handles functions, place called functions prior to calling functions in the file wherever
   possible: Otherwise sdcc will insert unnecessary pagesel directives around the call, believing that the called
   function is externally defined.

2. For devices that have multiple code pages it is more efficient to use the same number of files as pages: Use
   up to 4 separate .c files for the 16F877, but only 2 files for the 16F874. This way the linker can put the code
   for each file into different code pages and there will be less page selection overhead.

3. And as for any 8 bit micro (especially for PIC14 as they have a very simple instruction set), use ‘unsigned
   char’ wherever possible instead of ‘int’.

69
4.6.2 Adding New Devices to the Port

Adding support for a new 14 bit PIC MCU requires the following steps:

1. Create a new device description.
   Each device is described in two files: pic16f*.h and pic16f*.c. These files primarily define SFRs, structs to access their bits, and symbolic configuration options. Both files can be generated from gputils’ .inc files using the perl script support/scripts/inc2h.pl. This file also contains further instructions on how to proceed.

2. Copy the .h file into SDCC’s include path and either add the .c file to your project or copy it to device/lib/pic/libdev. Afterwards, rebuild and install the libraries.

3. Edit pic14devices.txt in SDCC’s include path (device/include/pic/ in the source tree or /usr/local/share/sdcc/include/pic after installation).
   You need to add a device specification here to make the memory layout (code banks, RAM, aliased memory regions, ...) known to the compiler. Probably you can copy and modify an existing entry. The file format is documented at the top of the file.

4.6.3 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.:

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

4.6.4 Configuration Bits

Configuration bits (also known as fuses) can be configured using ‘__code’ and ‘__at’ modifiers. Possible options should be ANDed and can be found in your processor header file. Example for PIC16F88:

```c
#include <pic16f88.h> //Contains config addresses and options
#include <stdint.h> //Needed for uint16_t

static __code uint16_t __at (_CONFIG1) configword1 = _INTRC_IO & _CP_ALL & _WDT_OFF & [...] ;
static __code uint16_t __at (_CONFIG2) configword2 = [...] ;
```

Although data type is ignored if the address (__at()) refers to a config word location, using a type large enough for the configuration word (uint16_t in this case) is recommended to prevent changes in the compiler (implicit, early range check and enforcement) from breaking the definition.

If your processor header file doesn’t contain config addresses you can declare it manually or use a literal address:

```c
static __code uint16_t __at (0x2007) configword1 = _INTRC_IO & _CP_ALL & _WDT_OFF & [...] ;
```

4.6.5 Linking and Assembling

For assembling you can use either GPUTILS’ gpasm.exe or MPLAB’s mpasmwin.exe. GPUTILS are available from http://sourceforge.net/projects/gputils. For linking you can use either GPUTILS’ 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.

Pic device specific header and c source files are automatically generated from MPLAB include files, which
are published by Microchip with a special requirement that they are only to be used with authentic Microchip devices. This requirement prevents to publish generated header and c source files under the GPL compatible license, so they are located in non-free directory (see section 2.3). In order to include them in include and library search paths, the \texttt{--use-non-free} command line option should be defined.

NOTE: the compiled code, which use non-free pic device specific libraries, is not GPL compatible!

Here is a Makefile using GPUTILS:

\begin{verbatim}
.c.o:
    sdcc -V --use-non-free -mpic14 -p16f877 -c $<

$(PRJ).hex: $(OBJS)
    gplink -m -s $(PRJ).lkr -o $(PRJ).hex $(OBJS) libsdcc.lib
\end{verbatim}

Here is a Makefile using MPLAB:

\begin{verbatim}
.c.o:
    sdcc -S -V --use-non-free -mpic14 -p16f877 $<
    mpasmwin /q /o $*.asm

$(PRJ).hex: $(OBJS)
    mplink /v $(PRJ).lkr /m $(PRJ).map /o $(PRJ).hex $(OBJS)
    libsdcc.lib
\end{verbatim}

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

\section*{4.6.6 Command-Line Options}

Besides the switches common to all SDCC backends, the PIC14 port accepts the following options (for an updated list see sdcc --help):

\begin{itemize}
\item \texttt{--debug-xtra} emit debug info in assembly output
\item \texttt{--no-pcode-opt} disable (slightly faulty) optimization on pCode
\item \texttt{--stack-loc} sets the lowest address of the argument passing stack (defaults to a suitably large shared databank to reduce BANKSEL overhead)
\item \texttt{--stack-size} sets the size if the argument passing stack (default: 16, minimum: 4)
\item \texttt{--use-non-free} make non-free device headers and libraries available in the compiler’s search paths (implicit \texttt{-I} and \texttt{-L} options)
\item \texttt{--no-extended-instructions} forbid use of the extended instruction set (e.g., ADDFSR)
\end{itemize}

\section*{4.6.7 Environment Variables}

The PIC14 port recognizes the following environment variables:

\texttt{SDCC\_PIC14\_SPLIT\_LOCALS} If set and not empty, sdcc will allocate each temporary register (the ones called r0xNNNN) in a section of its own. By default (if this variable is unset), sdcc tries to cluster registers in sections in order to reduce the BANKSEL overhead when accessing them.

\section*{4.6.8 The Library}

The PIC14 library currently only contains support routines required by the compiler to implement multiplication, division, and floating point support. No libc-like replacement is available at the moment, though many of the common sdcc library sources (in \texttt{device/lib}) should also compile with the PIC14 port.
4.6. THE PIC14 PORT  

4.6.8.1 Enhanced cores

SDCC/PIC14 has experimental support for devices with the enhanced 14-bit cores (such as pic12f1822). Due to differences in required code, the libraries provided with SDCC (libm.lib and libsdcc.lib) are now provided in two variants: libm.lib and libsdcc.lib are compiled for the regular, non-enhanced devices. libm.e.lib and libsdcc.e.lib (note the trailing ‘e’) are compiled for enhanced devices. When linking manually, make sure to select the proper variant!

When SDCC is used to invoke the linker, SDCC will automatically select the libsdcc.lib-variant suitable for the target device. However, no such magic has been conjured up for libm.lib!

4.6.8.2 Accessing bits of special function registers

Individual bits within SFRs can be accessed either using <sfrname>bits.<bitname> or using a shorthand <bitname>, which is defined in the respective device header for all <bitname>s. In order to avoid polluting the global namespace with the names of all the bits, you can #define NO_BIT_DEFINES before inclusion of the device header file.

4.6.8.3 Naming of special function registers

If NO_BIT_DEFINES is used, individual bits of the SFRs can be accessed as <sfrname>bits.<bitname>. With the 3.1.0 release, the previously used <sfrname>_bits.<bitname> (note the underscore) is deprecated. This was done to align the naming conventions with the PIC16 port and competing compiler vendors. To avoid polluting the global namespace with the legacy names, you can prevent their definition using #define NO_LEGACY_NAMES prior to the inclusion of the device header.

You must also #define NO_BIT_DEFINES in order to access SFRs as <sfrname>bits.<bitname>, otherwise <bitname> will expand to <sfrname>bits.<bitname>, yielding the undefined expression <sfrname>bits.<sfrname>bits.<bitname>.

4.6.8.4 Error: missing definition for symbol “__gptrget1”

The PIC14 port uses library routines to provide more complex operations like multiplication, division/modulus and (generic) pointer dereferencing. In order to add these routines to your project, you must link with PIC14’s libsdcc.lib. For single source file projects this is done automatically, more complex projects must add libsdcc.lib to the linker’s arguments. Make sure you also add an include path for the library (using the -I switch to the linker)!

4.6.8.5 Processor mismatch in file “XXX”.

This warning can usually be ignored due to the very good compatibility amongst 14 bit PIC devices.

You might also consider recompiling the library for your specific device by changing the ARCH=p16f877 (default target) entry in device/lib/pic/Makefile.in and device/lib/pic/Makefile to reflect your device. This might even improve performance for smaller devices as unnecessary BANKSELS might be removed.

4.6.9 Known Bugs

4.6.9.1 Function arguments

Functions with variable argument lists (like printf) are not yet supported. Similarly, taking the address of the first argument passed into a function does not work: It is currently passed in WREG and has no address...

4.6.9.2 Regression tests fail

Though the small subset of regression tests in src/regression passes, SDCC regression test suite does not, indicating that there are still major bugs in the port. However, many smaller projects have successfully used SDCC in the past...
4.7 The PIC16 port

The PIC16 port adds support for Microchip™ PIC™ MCUs with 16 bit wide instructions. This port is not yet mature and still lacks many features. However, it can work for simple code. Currently this family of microcontrollers contains the PIC18Fxxx and PIC18Fxxxx; devices supported by the port include:

18F13K22 18F13K50
18F14K22 18F14K50
18F23K20 18F23K22
18F24J10 18F24J11 18F24J50 18F24K20 18F24K22 18F24K50
18F26J11 18F26J13 18F26J50 18F26J53 18F26K20 18F26K22 18F26K80
18F27J13 18F27J53
18F43K20 18F43K22
18F44J10 18F44J11 18F44J50 18F44K20 18F44K22
18F45J10 18F45J11 18F45J50 18F45K20 18F45K22 18F45K50 18F45K80
18F46J11 18F46J13 18F46J50 18F46J53 18F46K20 18F46K22 18F46K80
18F47J13 18F47J53
18F63J11 18F63J90
18F64J11 18F64J90
18F65J10 18F65J11 18F65J15 18F65J50 18F65J90 18F65J94 18F65K22 18F65K80 18F65K90
18F66J10 18F66J11 18F66J15 18F66J50 18F66J55 18F66J60 18F66J65
18F66J90 18F66J93 18F66J94 18F66K22 18F66K80 18F66K90
18F67J10 18F67J11 18F67J50 18F67J60 18F67J90 18F67J93 18F67J94 18F67K22 18F67K90
18F83J11 18F83J90
18F84J11 18F84J90
18F85J10 18F85J11 18F85J15 18F85J50 18F85J90 18F85J94 18F85K22 18F85K90
18F86J10 18F86J11 18F86J15 18F86J50 18F86J55 18F86J60 18F86J65
18F86J70 18F86J90 18F86J94 18F86J99 18F86K22 18F86K90
18F87J10 18F87J11 18F87J50 18F87J60 18F87J90 18F87J93 18F87J94 18F87K22 18F87K90
18F95J94 18F95J99
18F97J60 18F97J94
18F24J0 18F24J24 18F24J25 18F24J28
18F44J0 18F44J24 18F44J25 18F44J28
18F1320 18F1330
18F1220 18F1230
18F1220 18F1230
18F2320 18F2321 18F2331
18F2410 18F2420 18F2423 18F2431 18F2439 18F2450 18F2455 18F2458 18F2480
18F2510 18F2515 18F2520 18F2523 18F2525 18F2539 18F2550 18F2553 18F2580 18F2585
18F2610 18F2620 18F2680 18F2682 18F2685
18F4220 18F4221
18F4320 18F4321 18F4322
18F4410 18F4420 18F4423 18F4431 18F4439 18F4450 18F4455 18F4458 18F4480
18F4510 18F4515 18F4520 18F4523 18F4525 18F4539 18F4550 18F4553 18F4580 18F4585
18F4610 18F4620 18F4680 18F4682 18F4685
18F6310 18F6390 18F6393
18F6410 18F6490 18F6493
18F6520 18F6525 18F6527 18F6585
18F6620 18F6621 18F6622 18F6627 18F6628 18F6680
18F6720 18F6722 18F6723
18F8310 18F8390 18F8393
18F8410 18F8490 18F8493
18F8520 18F8525 18F8527 18F8585
18F8620 18F8621 18F8622 18F8627 18F8628 18F8680
18F8720 18F8722 18F8723
18LF13K22 18LF13K50
4.7. THE PIC16 PORT

4.7.1 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

--use-non-free Make non-free device headers and libraries available in the compiler’s search paths (implicit -I and -L options).

4.7.2 Port Specific Options

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

An up-to-date list of supported devices is also available via ‘sdcc -mpic16 -plist’.
4.7.2.1 Code Generation Options

These options influence the generated assembler code.

--pstack-model=[model] Used in conjunction 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.

--pno-banksel Do not generate BANKSEL assembler directives.

--extended Enable extended instruction set/literal offset addressing mode. Use with care!

4.7.2.2 Optimization Options

--obanksel=n 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!

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

--no-optimize-goto Do not use (conditional) BRA instead of GOTO.

--optimize-cmp Try to optimize some compares.

--optimize-df Analyze the dataflow of the generated code and improve it.

4.7.2.3 Assembling Options

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

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

4.7.2.4 Linking Options

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

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

--ivt-loc=n Place the interrupt vector table at address n. Useful for bootloaders.

--nodefaultlibs Do not link default libraries when linking.

--use-crt= Use a custom run-time module instead of the default (crt0i).

--no-crt Don’t link the default run-time modules
4.7.2.5 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 being compiled.

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

--calltree Dump call tree in .calltree file.

--gstack Trace push/pops for stack pointer overflow.

4.7.3 Environment Variables

There is a number of environmental 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 bit-fields is optimized by directly loading FSR0 with the address of the bit-field structure. Normally SDCC will cast the bit-field structure to a bit-field pointer and then load FSR0. This step saves data ram and code space for functions that make heavy use of bit-fields. (i.e., 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. If bugs in the pcode optimizer are found, users can benefit from temporarily disabling the optimizer until the bug is fixed.

4.7.4 Preprocessor Macros

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__SDCC_pic16</td>
<td>Port identification</td>
</tr>
<tr>
<td>_pic18fxxxx</td>
<td>MCU Identification. xxxx is the microcontrol identification number, i.e. 452, 6620, etc</td>
</tr>
<tr>
<td>__18Fxxxx</td>
<td>MCU Identification (same as above)</td>
</tr>
<tr>
<td>STACK_MODEL_nnn</td>
<td>nnn = SMALL or LARGE respectively according to the stack model used</td>
</tr>
</tbody>
</table>

In addition the following macros are defined when calling assembler:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__18Fxxxx</td>
<td>MCU Identification. xxxx is the microcontrol identification number, i.e. 452, 6620, etc</td>
</tr>
<tr>
<td>__SDCC_MODEL_nnn</td>
<td>nnn = SMALL or LARGE respectively according to the memory model used for SDCC</td>
</tr>
<tr>
<td>STACK_MODEL_nnn</td>
<td>nnn = SMALL or LARGE respectively according to the stack model used</td>
</tr>
</tbody>
</table>

4.7.5 Directories

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

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Target</th>
<th>Command prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX/sdcc/include/pic16</td>
<td>PIC16 specific headers</td>
<td>Compiler</td>
<td>-I</td>
</tr>
<tr>
<td>PREFIX/sdcc/lib/pic16</td>
<td>PIC16 specific libraries</td>
<td>Linker</td>
<td>-L</td>
</tr>
</tbody>
</table>
4.7. THE PIC16 PORT

If the `--use-non-free` command line option is specified, non-free directories are searched:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Target</th>
<th>Command prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX/sdcc/non-free/include/pic16</td>
<td>PIC16 specific non-free headers</td>
<td>Compiler</td>
<td>-I</td>
</tr>
<tr>
<td>PREFIX/sdcc/non-free/lib/pic16</td>
<td>PIC16 specific non-free libraries</td>
<td>Linker</td>
<td>-L</td>
</tr>
</tbody>
</table>

4.7.6 Pragmas

The PIC16 port currently supports the following pragmas:

**stack** This forces the code generator to initialize the stack & frame pointers at a specific address. This is an ad hoc 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.\(^2\)

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.

**code** Force a function to a 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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Module to link</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignore</td>
<td>ignore all library pragmas</td>
<td>(none)</td>
</tr>
<tr>
<td>c</td>
<td>link the C library</td>
<td>libc18f.lib</td>
</tr>
<tr>
<td>math</td>
<td>link the Math library</td>
<td>libm18f.lib</td>
</tr>
<tr>
<td>io</td>
<td>link the I/O library</td>
<td>libio18f*.lib</td>
</tr>
<tr>
<td>debug</td>
<td>link the debug library</td>
<td>libdebug.lib</td>
</tr>
</tbody>
</table>

\(^*\) is the device number, i.e. 452 for PIC18F452 MCU.

This feature allows for linking with specific libraries without having to explicit name them in the command line. Note that the IGNORE keyword will reject all modules specified by the library pragma.

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

\(^2\)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.
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.

```
config
```

The pragma config instructs the compiler to emit config directive.

The format is as follows:

```
#pragma config setting=value [, setting=value]
```

Multiple settings may be defined on a single line, separated by commas. Settings for a single configuration byte may also be defined on separate lines.

Example:
```
#pragma config CP0=OFF,OSCS=ON,OSC=LP,BOR=ON,BORV=25,WDT=ON,WDTPS=128,CCP2MUX=ON
#pragma config STVR=ON
```

### 4.7.7 Header Files and Libraries

Pic device specific header and c source files are automatically generated from MPLAB include files, which are published by Microchip with a special requirement that they are only to be used with authentic Microchip devices. This requirement prevents to publish generated header and c source files under the GPL compatible license, so they are located in the non-free directory (see section 2.3). In order to include them in include and library search paths, the `--use-non-free` command line option should be defined.

NOTE: the compiled code, which use non-free pic device specific libraries, is not GPL compatible!
4.7.8 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.

4.7.9 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](http://sourceforge.net/projects/gputils).

Building the libraries

Before using SDCC/pic16 there are some libraries that need to be compiled. This process is done automatically if gputils are found at SDCC’s compile time. Should you require to rebuild the pic16 libraries manually (e.g. in order to enable output of float values via `printf()`, see below), these are the steps required to do so under Linux or Mac OS X (cygwin might work as well, but is untested):

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

If you need to install the headers too, do:

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

Output of float values via `printf()`

The library is normally built without support for displaying float values, only `<NO FLOAT>` will appear instead of the value. To change this, rebuild the library as stated above, and call `./configure.gnu --enable-floats` instead of just `./configure.gnu`. Also make sure that at least `libc/stdio/vfprintf.c` is actually re-compiled, e.g. by touching it after the configure run or deleting its `.o` file.

The more common approach of compiling `vfprintf.c` manually with `-DUSE_FLOATS=1` should also work, but is untested.

4.7.10 Adding New Devices to the Port

Adding support for a new 16 bit PIC MCU requires the following steps:

1. Create `picDEVICE.c` and `picDEVICE.h` from `pDEVICE.inc` using
   ```
   perl /path/to/sdcc/support/scripts/inc2h-pic16.pl /path/to/gputils/header/pDEVICE.inc
   ```
2. `mv picDEVICE.h /path/to/sdcc/device/non-free/include/pic16`
3. `mv picDEVICE.c /path/to/sdcc/device/non-free/lib/pic16/libdev`
4. Either

(a) add the new device to /path/to/sdcc/device/lib/pic16/libio/*.ignore to suppress building any of the I/O libraries for the new device\(^3\), or

(b) add the device (family) to /path/to/sdcc/support/scripts/pic18fam-h-gen.pl to assign I/O styles, run the pic18fam-h-gen.pl script to generate pic18fam.h.gen, replace your existing pic18fam.h with the generated file, and (if required) implement new I/O styles in /path/to/sdcc/device/include/pic16/{adc, i2c, usart}.h and /path/to/sdcc/device/lib/pic16/libio/\*\/*.\n
5. Edit /path/to/sdcc/device/include/pic16/pic18fregs.h
The file format is self-explanatory, just add

```
# elif defined(picDEVICE)
# include <picDEVICE.h>
```
at the right place (keep the file sorted, please).

6. Edit /path/to/sdcc/device/include/pic16devices.txt
Copy and modify an existing entry or create a new one and insert it at the correct place (keep the file sorted, please).

7. ( cd /path/to/sdcc/device/non-free/lib/pic16 && sh update.sh )

8. Recompile the pic16 libraries as described in 4.7.9 or just configure and build sdcc again from scratch (recommended).

### 4.7.11 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:

<table>
<thead>
<tr>
<th>Pointer sizes according to memory model</th>
<th>small model</th>
<th>large model</th>
</tr>
</thead>
<tbody>
<tr>
<td>code pointers</td>
<td>16-bits</td>
<td>24-bits</td>
</tr>
<tr>
<td>data pointers</td>
<td>16-bits</td>
<td>16-bits</td>
</tr>
</tbody>
</table>

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 4.7.8) contain no reference to pointers, so they can be used with both memory models.

### 4.7.12 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

\(^3\)In fact, the .ignore files are only used when auto-generating Makefile.am using the .../libio/mkmk.sh script.
4.7. THE PIC16 PORT  CHAPTER 4. NOTES ON SUPPORTED PROCESSES

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:

<table>
<thead>
<tr>
<th>Stack &amp; Frame pointer sizes according to stack model</th>
<th>small</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack pointer FSR1</td>
<td>8-bits</td>
<td>16-bits</td>
</tr>
<tr>
<td>Frame pointer FSR2</td>
<td>8-bits</td>
<td>16-bits</td>
</tr>
</tbody>
</table>

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.

4.7.13 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:

```c
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) */
    ...
}
```

__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.

```c
void isr_shadow(void) __shadowregs __interrupt 1
{
    ...
}
```

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

4.7.14 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:

<table>
<thead>
<tr>
<th>size</th>
<th>destination register</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>WREG</td>
</tr>
<tr>
<td>16 bits</td>
<td>PRODL:WREG</td>
</tr>
<tr>
<td>24 bits</td>
<td>PRODH:PRODL:WREG</td>
</tr>
<tr>
<td>32 bits</td>
<td>FSR0L:PRODH:PRODL:WREG</td>
</tr>
<tr>
<td>&gt;32 bits</td>
<td>on stack, FSR0 points to the beginning</td>
</tr>
</tbody>
</table>
4.7.15 Interrupts

An interrupt service routine (ISR) is declared using the __interrupt keyword.

```c
void isr(void) __interrupt n {
    ...
}
```

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>Interrupt Vector</th>
<th>Interrupt Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESET vector</td>
<td>0x000000</td>
</tr>
<tr>
<td>1</td>
<td>HIGH priority interrupts</td>
<td>0x000008</td>
</tr>
<tr>
<td>2</td>
<td>LOW priority interrupts</td>
<td>0x000018</td>
</tr>
</tbody>
</table>

When generating assembly code for ISR the code generator places a GOTO instruction at the Interrupt Vector Address which points at the generated ISR. This single GOTO instruction is part of an automatically generated interrupt entry point function. The actual 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 interrupt’s vector address and cause indeterminate program behaviour if that interrupt is raised.\(^4\)

\( n \) may 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.\(^5\)

4.7.16 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:

<table>
<thead>
<tr>
<th>pointer type</th>
<th>7th bit</th>
<th>6th bit</th>
<th>rest of the pointer</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>1</td>
<td>0</td>
<td>㎜㎜㎜㎜ ㎜㎜㎜㎜㎜ ㎜㎜㎜㎜㎜㎜</td>
<td>a 12-bit data pointer in data RAM memory</td>
</tr>
<tr>
<td>code</td>
<td>0</td>
<td>0</td>
<td>㎜㎜㎜㎜ ㎜㎜㎜㎜㎜ ㎜㎜㎜㎜㎜㎜</td>
<td>a 21-bit code pointer in FLASH memory</td>
</tr>
<tr>
<td>eeprom</td>
<td>0</td>
<td>1</td>
<td>㎜㎜㎜㎜ ㎜㎜㎜㎜㎜ ㎜㎜㎜㎜㎜㎜</td>
<td>a 10-bit eeprom pointer in EEPROM memory</td>
</tr>
<tr>
<td>(unimplemented)</td>
<td>1</td>
<td>1</td>
<td>㎜㎜㎜㎜ ㎜㎜㎜㎜㎜ ㎜㎜㎜㎜㎜㎜</td>
<td>unimplemented pointer type</td>
</tr>
</tbody>
</table>

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

\(^4\)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 interrupt’s vector address.

\(^5\)NOTE that when the _naked attribute is specified for an interrupt routine, then NO registers are stored or restored.
### 4.7.17 Configuration Bits

Configuration bits (also known as fuses) can be configured using one of two methods:

- using `#pragma config` (see section 4.7.6), which is a preferred method for the new code. Example:

  ```c
  #pragma config CP0=OFF,OSCS=ON,OSC=LP,BOR=ON,BORV=25,WDT=ON,WDTPS=128,CCP2MUX=ON
  #pragma config STVR=ON
  ```

- using `__code` and `__at` modifiers. This method is deprecated. Possible options should be ANDed and can be found in your processor header file. Example for PIC18F2550:

  ```c
  #include <pic18fregs.h> //Contains config addresses and options
  
  static __code char __at(__CONFIG1L) configword1l =
      __USBPLL_CLOCK_SRC_FROM_96MHZ_PLL_2_1L &
      __PLLDIV_NO_DIVIDE__4MHZ_INPUT__1L & [...];
  static __code char __at(__CONFIG1H) configword1h = [...];
  static __code char __at(__CONFIG2L) configword2l = [...];
  //More configuration words
  ```

Mixing both methods is not allowed and throws an error message "mixing __CONFIG and CONFIG directives".

### 4.7.18 PIC16 C Libraries

#### 4.7.18.1 Standard I/O Streams

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

```c
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`:

```c
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:

<table>
<thead>
<tr>
<th>pointer type</th>
<th>&lt;7:6&gt;</th>
<th>&lt;5&gt;</th>
<th>&lt;4&gt;</th>
<th>&lt;3:0&gt;</th>
<th>rest of the pointer</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>nnnn</td>
<td>uuuuuuuu uuuuuuuu</td>
<td>upper byte high nubble is 0x2n, the rest are zeroes</td>
</tr>
</tbody>
</table>

Currently implemented there are 3 types of streams defined:

<table>
<thead>
<tr>
<th>stream type</th>
<th>value</th>
<th>module</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM_USART</td>
<td>0x2000000UL</td>
<td>USART</td>
<td>Writes/Reads characters via the USART peripheral</td>
</tr>
<tr>
<td>STREAM_MSSP</td>
<td>0x2100000UL</td>
<td>MSSP</td>
<td>Writes/Reads characters via the MSSP peripheral</td>
</tr>
<tr>
<td>STREAM_USER</td>
<td>0x2F00000UL</td>
<td>(none)</td>
<td>Writes/Reads characters via used defined functions</td>
</tr>
</tbody>
</table>

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:

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

PUTCHAR( c )
{
    PORTA = c; /* dump character c to PORTA */
}

void main(void)
{
    stdout = STREAM_USER; /* this is not necessary, since stdout points */
    * by default to STREAM_USER */
    printf(“This is a printf test\n”);
}
```

### 4.7.18.2 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_list 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 necessary buffer. For `fprintf` and `vfprintf`, `fp` should be a stream pointer (i.e. `stdout`, `STREAM_MSSP`, etc...).

### 4.7.18.3 Signals

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

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

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 for 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:

```c
#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.

### 4.7.19 PIC16 Port – Tips

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

#### 4.7.19.1 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 --fomit-frame-pointer might reduce stack usage by not creating unnecessary stack frames. Other ways to reduce stack usage may exist.
4.7.20 Known Bugs

4.7.20.1 Extended Instruction Set

The PIC16 port emits code which is incompatible with the extended instruction set available with many newer devices. Make sure to always explicitly disable it, usually using:

• #pragma config XINST=OFF

or deprecated:

• static __code char __at(__CONFIG4L) conf4l = /* more flags & */ _XINST_OFF_4L;

Some devices (namely 18f2455, 18f2550, 18f4455, and 18f4550) use _ENHCPU_OFF_4L instead of _XINST_OFF_4L.

4.7.20.2 Regression Tests

The PIC16 port currently passes most but not all of the tests in SDCC’s regression test suite (see section 7.8), thus no automatic regression tests are currently performed for the PIC16 target.
Chapter 5

Debugging

There are several approaches to debugging your code. This chapter is meant to show your options and to give detail on some of them:

When writing your code:

- write your code with debugging in mind (avoid duplicating code, put conceptually similar variables into structs, use structured code, have strategic points within your code where all variables are consistent, ...)
- run a syntax-checking tool like splint (see --more-pedantic 3.3.4) over the code.
- for the high level code use a C-compiler (like f.e. GCC) to compile run and debug the code on your host. See (see --more-pedantic 3.3.4) on how to handle syntax extensions like _xdata, _at(), ...
- use another C-compiler to compile code for your target. Always an option but not recommended:) And not very likely to help you. If you seriously consider walking this path you should at least occasionally check portability of your code. Most commercial compiler vendors will offer an evaluation version so you can test compile your code or snippets of your code.

Debugging on a simulator:

- there is a separate section about SDCDB (section 5.1) below.
- or (8051 specific) use a free open source/commercial simulator which interfaces to the AOMF file (see 3.2.1) optionally generated by SDCC.

Debugging On-target:

- use a MCU port pin to serially output debug data to the RS232 port of your host. You’ll probably want some level shifting device typically involving a MAX232 or similar IC. If the hardware serial port of the MCU is not available search for ‘Software UART’ in your favourite search machine.
- use an on-target monitor. In this context a monitor is a small program which usually accepts commands via a serial line and allows to set program counter, to single step through a program and read/write memory locations. For the 8051 good examples of monitors are paulmon and cmon51 (see section 6.5).
- toggle MCU port pins at strategic points within your code and use an oscilloscope. A digital oscilloscope with deep trace memory is really helpful especially if you have to debug a realtime application. If you need to monitor more pins than your oscilloscope provides you can sometimes get away with a small R-2R network. On a single channel oscilloscope you could for example monitor 2 push-pull driven pins by connecting one via a 10 kΩ resistor and the other one by a 5 kΩ resistor to the oscilloscope probe (check output drive capability of the pins you want to monitor). If you need to monitor many more pins a logic analyzer will be handy.
- use an ICE (in circuit emulator). Usually very expensive. And very nice to have too. And usually locks you (for years...) to the devices the ICE can emulate.
• use a remote debugger. In most 8-bit systems the symbol information is not available on the target, and a complete debugger is too bulky for the target system. Therefore usually a debugger on the host system connects to an on-target debugging stub which accepts only primitive commands. Terms to enter into your favourite search engine could be 'remote debugging', 'gdb stub' or 'inferior debugger’. (is there one?)

• use an on target hardware debugger. Some of the more modern MCUs include hardware support for setting break points and monitoring/changing variables by using dedicated hardware pins. This facility doesn’t require additional code to run on the target and usually doesn’t affect runtime behaviour until a breakpoint is hit. For the mcs51 most hardware debuggers use the AOMF file (see 3.2.1) as input file.

Last not least:

• if you are not familiar with any of the following terms you’re likely to run into problems rather sooner than later: volatile, atomic, memory map, overlay. As an embedded programmer you have to know them so why not look them up before you have problems?)

• tell someone else about your problem (actually this is a surprisingly effective means to hunt down the bug even if the listener is not familiar with your environment). As ‘failure to communicate’ is probably one of the job-induced deformations of an embedded programmer this is highly encouraged.

5.1 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.

5.1.1 Compiling for Debugging

The --debug option must be specified for all files for which debug information is to be generated. The compiler 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.

5.1.2 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. (Currently SDCDM only connects to the simulator but newcdb at http://ec2drv.sourceforge.net/ is an effort to connect directly to the hardware.)

5.1.3 Starting the Debugger SDCDB

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.
5.1.4 SDCDB Command Line Options

- `--directory=<source file directory>` this option can be 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 option 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.

5.1.5 SDCDB 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, xxdgdb 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"
5.1.6 Interfacing SDCDB with DDD

The portable network graphics file http://sdcc.sourceforge.net/wiki_images/ddd_example.png 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"
```

5.1.7 Interfacing SDCDB 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 ptype command for data at
;; x sdcdbsrc-delete SDCDB Delete all breakpoints if no arg
; arg x)
;; m sdcdbsrc-frame SDCDB Display current frame
```

91
if no arg,
;;
;; ! sdcdbsrc-goto-sdcdb Goto the SDCDB output buffer
;; p sdcdb-print-c-sexp SDCDB print command for data
at
;;
;; 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) 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.
6.1 Porting code from or to other compilers

- NEVER jump into a LOOP.
- Declare the variables to be local whenever possible, especially loop control variables (induction).
- Have a look at the assembly listing to get a "feeling" for the code generation.

### 6.1 Porting code from or to other compilers

- check whether endianness of the compilers differs and adapt where needed.
- check the device specific header files for compiler specific syntax. Eventually include the file `<compiler.h>` [http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/device/include/mcs51/compiler.h](http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/device/include/mcs51/compiler.h) to allow using common header files. (see f.e. cc2510fx.h [http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/device/include/mcs51/cc2510fx.h](http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/device/include/mcs51/cc2510fx.h)).
- check whether the startup code contains the correct initialization (watchdog, peripherals).
- check whether the sizes of short, int, long match.
- check if some 16 or 32 bit hardware registers require a specific addressing order (least significant or most significant byte first) and adapt if needed (first and last relate to time and not to lower/upper memory location here, so this is not the same as endianness).
- check whether the keyword `volatile` is used where needed. The compilers might differ in their optimization characteristics (as different versions of the same compiler might also use more clever optimizations this is good idea anyway). See section 3.8.1.1.
- check that the compilers are not told to suppress warnings.
- check and convert compiler specific extensions (interrupts, memory areas, pragmas etc.).
- check for differences in type promotion. Especially check for math operations on `char` or `unsigned char` variables. For the sake of C99 compatibility SDCC will probably promote these to `int` more often than other compilers. Eventually insert explicit casts to `(char)` or `(unsigned char)`. Also check that the `~` operator is not used on `bit` variables, use the `!` operator instead. See sections 6 and 1.5.
- check the assembly code generated for interrupt routines (f.e. for calls to possibly non-reentrant library functions).
- check whether timing loops result in proper timing (or preferably consider a rewrite of the code with timer based delays instead).
- check for differences in printf parameters (some compilers push `(va_arg)` char variables as `int` others push them as `char`. See section 1.5). Provide a putchar() function if needed.
- check the resulting memory map. Usage of different memory spaces: code, stack, data (for mcs51/ds390 additionally idata, pdata, xdata). Eventually check if unexpected library functions are included.

### 6.2 Tools included in the distribution
<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>as2gbmap.py</td>
<td>sdas map to rrgb map and no$gmb sym file format converter</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>cinc2h.pl</td>
<td>gpasm inc to c header file converter</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>gen_known_bugs.pl</td>
<td>generate knownbugs.html</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>keil2sdcc.pl</td>
<td>Keil header to SDCC header file converter</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>makebin</td>
<td>Intel Hex to binary and GameBoy binary format converter</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>mcs51-disasm.pl</td>
<td>disassembler to the mcs51 instructions contained hex files</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>mh2h.c</td>
<td>header file conversion</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>optimize_pic16devices.pl</td>
<td>optimizes or unoptimizes the pic16devices.txt file</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>packihx</td>
<td>Intel Hex packer</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>pic14-header-parser.pl</td>
<td>helper to realization of peripheral-handling (PIC14)</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>pic16-header-parser.pl</td>
<td>helper to realization of peripheral-handling (PIC16)</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>pic16fam-h-gen.pl</td>
<td>helper to realization of peripheral-handling (PIC14)</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>pic18fam-h-gen.pl</td>
<td>helper to realization of peripheral-handling (PIC16)</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>repack_release.sh</td>
<td>repack sdcc to release package</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>sdas390</td>
<td>assembler</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sdas6808</td>
<td>assembler</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sdas8051</td>
<td>assembler</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sdas80</td>
<td>assembler</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sdc32</td>
<td>assembler</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sdc32_mingw32</td>
<td>cross compile the sdcc with mingw32 under Cygwin</td>
<td>sdcc/support/scripts</td>
</tr>
<tr>
<td>SDCDB</td>
<td>simulator</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sddl</td>
<td>linker</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sddl6808</td>
<td>linker</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sddlgb</td>
<td>linker</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>sddldz80</td>
<td>linker</td>
<td>sdcc/bin</td>
</tr>
<tr>
<td>uCsim</td>
<td>simulator for various architectures</td>
<td>sdcc/sim/ucsim</td>
</tr>
</tbody>
</table>

### 6.3 Documentation included in the distribution
### 6.4 Communication online at SourceForge

<table>
<thead>
<tr>
<th>Subject / Title</th>
<th>Note</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>wiki</td>
<td></td>
<td><a href="http://sdcc.sourceforge.net/wiki/">http://sdcc.sourceforge.net/wiki/</a></td>
</tr>
<tr>
<td>sdcc-user mailing list</td>
<td>around 650 subscribers mid 2009</td>
<td><a href="https://lists.sourceforge.net/mailman/listinfo/sdcc-user">https://lists.sourceforge.net/mailman/listinfo/sdcc-user</a></td>
</tr>
<tr>
<td>sdcc-devel mailing list</td>
<td></td>
<td><a href="https://lists.sourceforge.net/mailman/listinfo/sdcc-devel">https://lists.sourceforge.net/mailman/listinfo/sdcc-devel</a></td>
</tr>
<tr>
<td>help forum</td>
<td>similar scope as sdcc-user mailing list</td>
<td><a href="http://sourceforge.net/p/sdcc/discussion/1865">http://sourceforge.net/p/sdcc/discussion/1865</a></td>
</tr>
<tr>
<td>open discussion forum</td>
<td></td>
<td><a href="http://sourceforge.net/p/sdcc/discussion/1864">http://sourceforge.net/p/sdcc/discussion/1864</a></td>
</tr>
<tr>
<td>trackers (bug tracker, feature requests, patches, support requests, webdocs)</td>
<td></td>
<td><a href="http://sourceforge.net/p/sdcc/_list/tickets">http://sourceforge.net/p/sdcc/_list/tickets</a></td>
</tr>
<tr>
<td>rss feed</td>
<td>stay tuned with most (not all) sdcc activities</td>
<td><a href="http://sourceforge.net/export/rss2_keepsake.php?group_id=599">http://sourceforge.net/export/rss2_keepsake.php?group_id=599</a></td>
</tr>
</tbody>
</table>

With a sourceforge account you can "monitor" forums and trackers, so that you automatically receive mail on changes. You can digg out earlier communication by using the search function http://sourceforge.net/search/?group_id=599.

### 6.5 Related open source tools
### 6.6  RELATED DOCUMENTATION / RECOMMENDED READING

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Where to get</th>
</tr>
</thead>
<tbody>
<tr>
<td>gputils</td>
<td>GNU PIC utilities</td>
<td><a href="http://sourceforge.net/projects/gputils">http://sourceforge.net/projects/gputils</a></td>
</tr>
<tr>
<td>flP5</td>
<td>PIC programmer</td>
<td><a href="http://freshmeat.net/projects/flp5/">http://freshmeat.net/projects/flp5/</a></td>
</tr>
<tr>
<td>ec2drv/newcdb</td>
<td>Tools for Silicon Laboratories JTAG debug adapter, partly based on SDCDB (Unix only)</td>
<td><a href="http://sourceforge.net/projects/ec2drv">http://sourceforge.net/projects/ec2drv</a></td>
</tr>
<tr>
<td>indent</td>
<td>Formats C source - Master of the white spaces</td>
<td><a href="http://directory.fsf.org/GNU/indent.html">http://directory.fsf.org/GNU/indent.html</a></td>
</tr>
<tr>
<td>srecord</td>
<td>Object file conversion, checksumming, ...</td>
<td><a href="http://sourceforge.net/projects/srecord">http://sourceforge.net/projects/srecord</a></td>
</tr>
<tr>
<td>objdump</td>
<td>Object file conversion, ...</td>
<td>Part of binutils (should be there anyway)</td>
</tr>
<tr>
<td>cmon51</td>
<td>8051 monitor (hex up-/download, single step, disassemble)</td>
<td><a href="http://sourceforge.net/projects/cmon51">http://sourceforge.net/projects/cmon51</a></td>
</tr>
<tr>
<td>doxygen</td>
<td>Source code documentation system</td>
<td><a href="http://www.doxygen.org">http://www.doxygen.org</a></td>
</tr>
<tr>
<td>kdevelop</td>
<td>IDE (has anyone tried integrating SDCC &amp; SDCDB? Unix only)</td>
<td><a href="http://www.kdevelop.org">http://www.kdevelop.org</a></td>
</tr>
<tr>
<td>paulmon</td>
<td>8051 monitor (hex up-/download, single step, disassemble)</td>
<td><a href="http://www.pjrc.com/tech/8051/paulmon2.html">http://www.pjrc.com/tech/8051/paulmon2.html</a></td>
</tr>
<tr>
<td>splint</td>
<td>Statically checks c sources (see 3.3.4)</td>
<td><a href="http://www.splint.org">http://www.splint.org</a></td>
</tr>
<tr>
<td>ddd</td>
<td>Debugger, serves nicely as GUI to SDCDB (Unix only)</td>
<td><a href="http://www.gnu.org/software/ddd/">http://www.gnu.org/software/ddd/</a></td>
</tr>
<tr>
<td>d52</td>
<td>Disassembler, can count instruction cycles, use with options -pnd</td>
<td><a href="http://www.8052.com/users/disasm/">http://www.8052.com/users/disasm/</a></td>
</tr>
<tr>
<td>cmake</td>
<td>Cross platform build system, generates Makefiles and project workspaces</td>
<td><a href="http://www.cmake.org">http://www.cmake.org</a> and a dedicated wiki entry: <a href="http://www.cmake.org/Wiki/CmakeSdcc">http://www.cmake.org/Wiki/CmakeSdcc</a></td>
</tr>
</tbody>
</table>

### 6.7  Application notes specifically for SDCC

SDCC makes no claims about the completeness of this list and about up-to-dateness or correctness of the application notes.

---

97
6.8 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), internationalization, packaging, labels, 2nd source for components, electromagnetic compatibility/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

<table>
<thead>
<tr>
<th>Vendor / Dallas</th>
<th>Subject / Title</th>
<th>Where to get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texas Instruments</td>
<td>MSC12xx Programming with SDCC</td>
<td><a href="http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=sbaa1094&amp;fileType=pdf">http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=sbaa1094&amp;fileType=pdf</a></td>
</tr>
</tbody>
</table>
Chapter 7

Support

SDCC has grown to be a large project. The compiler alone (without the preprocessor, assembler and linker) is well over 150,000 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, wiki, rss-feed, download area and Subversion 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 make 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/p/sdcc/bugs/. With SDCC on average having more than 200 downloads on SourceForge per day1 there must be some users. So it’s not exactly easy to find a new bug. If you find one we need it: reporting bugs is good.

---

1220 daily downloads on average Jan-Sept 2006 and about 150 daily downloads between 2002 and 2005. This does not include other methods of distribution.
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/p/sdcc/feature-requests/.

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/p/sdcc/patches/.

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/p/sdcc/mailman/sdcc-user/² and the Forums http://sourceforge.net/p/sdcc/discussion/, 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 Subversion version of SDCC by watching the Changelog in the Subversion repository http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/ChangeLog.

7.6 Subversion Source Code Repository

The output of sdcc --version or the filenames of the snapshot versions of SDCC include date and its Subversion number. Subversion allows to download the source of recent or previous versions http://sourceforge.net/p/sdcc/code/8805/tree/ (by number or by date).

7.7 Release policy

Starting with version 2.4.0 SDCC in 2004 uses a time-based release schedule with one official release usually during the first half of the year.

The last digit of an official release is zero. Additionally there are daily snapshots available at http://sdcc.sourceforge.net/snap.php, and you can always build the very last version from the source code available at Sourceforge http://sdcc.sourceforge.net/snap.php#Source. The SDCC Wiki at http://sdcc.sourceforge.net/wiki/ also holds some information about past and future releases.

7.8 Quality control

The compiler is passed through daily snapshot build compile and build checks. The so called regression tests check that SDCC itself compiles flawlessly on several host platforms (i386, Opteron, 64 bit Alpha, ppc64, Mac OS X on ppc and i386, Solaris on Sparc) and checks the quality of the code generated by SDCC by running the code for several target platforms through simulators. The regression test suite comprises about 1000 files which expand to more than 1500 test cases which include about 7000 tests. A large number of tests from the GCC test suite is included in this. The results of these tests are published daily on SDCC’s snapshot page (click on the red or green symbols on the right side of http://sdcc.sourceforge.net/snap.php).

²Traffic on sdcc-devel and sdcc-user is about 100 mails/month each not counting automated messages (mid 2003)
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 i.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 PIC14 port uses a different set of regression tests, you’ll find them in the directory `sdcc/src/regression`.

### 7.9 Examples


### 7.10 Use of SDCC in Education

In short: **highly encouraged**. If your rationales are to:

1. give students a chance to understand the complete steps of code generation
2. have a curriculum that can be extended for years. Then you could use an FPGA board as target and your curriculum will seamlessly extend from logic synthesis (http://www.opencores.org/opencores.org, Oregano http://www.oregano.at/ip/ip01.htm), over assembly programming, to C to FPGA compilers (FPGAC http://sourceforge.net/projects/fpgac/) and to C.
3. be able to insert excursions about skills like using a revision control system, submitting/merging patches, using a type-setting (as opposed to word-processing) engine LaTeX/LATEX, using SourceForge http://sourceforge.net/, following some netiquette [http://en.wikipedia.org/wiki/Netiquette](http://en.wikipedia.org/wiki/Netiquette), understanding BSD/LGPL/GPL/Proprietary licensing, growth models of Open Source Software, CPU simulation, compiler regression tests.
   And if there should be a shortage of ideas then you can always point students to the ever-growing feature request list [http://sourceforge.net/p/sdcc/feature-requests/](http://sourceforge.net/p/sdcc/feature-requests/).
4. not tie students to a specific host platform and instead allow them to use a host platform of their choice (among them Alpha, i386, i386_64, Mac OS X, Mips, Sparc, Windows and eventually OLPC [http://www.laptop.org](http://www.laptop.org))
5. not encourage students to use illegal copies of educational software
6. be immune to licensing/availability/price changes of the chosen tool chain
7. be able to change to a new target platform without having to adopt a new tool chain
8. have complete control over and insight into the tool chain
9. make your students aware about the pros and cons of open source software development
10. give back to the public as you are probably at least partially publicly funded
11. give students a chance to publicly prove their skills and to possibly see a world wide impact

then SDCC is probably among the first choices. Well, probably SDCC might be the only choice.

---

3 the phrase "use in education" might evoke the association "only fit for use in education". This connotation is not intended but nevertheless risked as the licensing of SDCC makes it difficult to offer educational discounts.
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 global common subexpression elimination, e.g.:

\[
\begin{align*}
    i &= x + y + 1; \\
    j &= x + y;
\end{align*}
\]

will be translated to

\[
\begin{align*}
    &iTemp = x + y; \\
    &i = iTemp + 1; \\
    &j = iTemp;
\end{align*}
\]

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

\[
\begin{align*}
    a->b[i].c &= 10; \\
    a->b[i].d &= 11;
\end{align*}
\]

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

\[
\begin{align*}
    &iTemp = a->b[i]; \\
    &iTemp.c = 10; \\
    &iTemp.d = 11;
\end{align*}
\]

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
### 8.1.3 Copy-Propagation

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

will be changed to

```c
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:**

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

changed to

```c
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:

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

changed to

```c
itemp1 = 0;
itemp2 = 0;
```
8.1. Optimization

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: */ \( i = j; \)
- \( i /= 2; \) /* for unsigned i 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.
8.1. OPTIMIZATIONS

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.:

```c
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

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

and

```c
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.
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.:

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

generates the following code:

```assembly
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:

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

will generate:

```assembly
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:

```c
unsigned char i; /* unsigned is needed for rotation */
...
i = ((i << 1) | (i >> 7));
...
```

will generate the following code:

```assembly
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.:

```c
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:

```c
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 format (i.e. lowest order first) for most backends. However, the hc08, s08 and stm8 backends are big-endian.

### 8.1.11 Highest Order Bit / Any Order Bit

It is frequently required to obtain the highest order bit of an integral type (long, int, short or char types). Also obtaining any other order bit is not uncommon. SDCC recognizes the following expressions to yield the highest order bit and generates optimized code for it, e.g.:

```c
unsigned int gint;

foo () {
    unsigned char hob1, aob1;
    bit hob2, hob3, aob2, aob3;
    ...
    hob1 = (gint >> 15) & 1;
    hob2 = (gint >> 15) & 1;
    hob3 = gint & 0x8000;
    aob1 = (gint >> 9) & 1;
    aob2 = (gint >> 8) & 1;
    aob3 = gint & 0x0800;
    ...
}
```

will generate the following code:

```
 000A E5 01  mov a,(_gint + 1)
 000C 23  rl a
 000D F5 01  anl a,#0x01
 0011 E5 01  mov a,(__gint + 1)
 0013 33  rlc a
 0014 F5 01  mov __foo_hob1_1_1,a
 0016 E5 01  mov a,(__gint + 1)
 0018 33  rlc a
 0019 92 00  mov __foo_hob2_1_1,c
 001B E5 01  mov a,(__gint + 1)
 001D 03  rr a
 001E F5 01  mov __foo_aob1_1_1,a
 0022 E5 01  mov a,(__gint + 1)
 0024 13  rrc a
 0025 92 02  mov __foo_aob2_1_1,c
 0027 E5 01  mov a,(__gint + 1)
 0029 A2 E3  mov c,acc[3]
 002B 92 03  mov __foo_aob3_1_1,c
```

1Usually 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.
Other variations of these cases however will not be recognized. They are standard C expressions, so I heartily recommend these 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.:

\[
xyz = \text{gint} + ((\text{gint} >> 15) \& 1);
\]

will still be recognized.

### 8.1.12 Higher Order Byte / Higher Order Word

It is also frequently required to obtain a higher order byte or word of a larger integral type (long, int or short types). SDCC recognizes the following expressions to yield the higher order byte or word and generates optimized code for it, e.g.:

```c
unsigned int gint;
unsigned long int glong;

foo () {
    unsigned char hob1, hob2;
    unsigned int how1, how2;
    ...
    hob1 = (gint >> 8) & 0xFF;
    hob2 = glong >> 24;
    how1 = (glong >> 16) & 0xFFFF;
    how2 = glong >> 8;
    ...
}
```

will generate the following code:

```
91; hob.c 15
0037 85*01*06 92 mov _foo_hob1_1_1,(_gint + 1)
003A 85*05*07 94 mov _foo_hob2_1_1,(_glong + 3)
003D 85*04*08 95 mov _foo_how1_1_1,(_glong + 2)
0040 85*05*09 97 mov (_foo_how1_1_1 + 1),(_glong + 3)
0043 85*03*0A 98 mov _foo_how2_1_1,(_glong + 1)
0046 85*04*0B 99 mov (_foo_how2_1_1 + 1),(_glong + 2)
```

Again, variations of these cases may not be recognized. They are standard C expressions, so I heartily recommend these be the only way to get the higher order byte/word, (it is portable). Of course it will be recognized even if it is embedded in other expressions, e.g.:

\[
xyz = \text{gint} + ((\text{gint} >> 8) \& 0xFF);
\]

will still be recognized.

### 8.1.13 Placement of Bank-Selection Instructions

8.1.14 Lifetime-Optimal Speculative Partial Redundancy Elimination

SDCC has an implementation of lifetime-optimal speculative partial redundancy elimination based on tree-decompositions.

8.1.15 Register Allocation

SDCC currently has two register allocators. One of them is optimal when optimizing for code size. This register allocator is used by default on the hc08, s08, z80, z180, r2k, r3ka, gbz80 and stm8 ports. Even though it runs in polynomial time, it can be quite slow; therefore the --max-allocs-per-node command line option can be used for a trade-off between compilation speed and quality of the generated code: Lower values result in faster compilation, higher values can result in better code being generated.

It first creates a tree-decomposition of the control-flow graph, and then uses dynamic programming bottom-up along the tree-decomposition. Optimality is achieved through the use of a cost function, which gives cost for instructions under register assignments. The cost function is target-specific and has to be implemented for each port; in all current SDCC ports the cost function is integrated into code generation.


8.1.16 Peephole Optimizer

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

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

The above rule will change the following assembly sequence:

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

to

```
mov r1,a
```

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

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

will remain unmodified.

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

```
replace { lcall %1 } by { acall %1 }
replace { ljmp %1 } by { ajmp %1 }
```
The inline-assembler code is also passed through the peep hole optimizer, thus the peephole optimizer can also be used as an assembly level macro expander. The rules themselves are MCU dependent whereas the rule language infra-structure is MCU independent. Peephole optimization rules for other MCU can be easily programmed using the rule language.

The syntax for a rule is as follows:

\[
\text{rule := replace [ restart ] \{ \langle \text{assembly sequence} \rangle \} by \{ \langle \text{assembly sequence} \rangle \}}
\]

\[
\langle \text{assembly sequence} \rangle := \text{assembly instruction} \quad (\text{each instruction including labels must be on a separate line}).
\]

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:

\[
\text{replace restart \{ \}
\text{pop } %1
\text{push } %1 \} by \{
\text{; nop}
\}
\]

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.:

\[
\text{pop ar1}
\text{pop ar2}
\text{push ar2}
\text{push ar1}
\]

would result in:

\[
\text{pop ar1}
\text{; nop}
\text{push ar1}
\]

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:

\[
\text{; nop}
\text{; nop}
\]

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

\[
\text{replace \{ \}
\text{ljmp } %5
\%
\text{2:}
\} by \{
\text{sjmp } %5
\%
\text{2:}
\}
\]

\[
\text{if labelInRange}
\]
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 some of 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, labelRefCountChange, labelIsReturnOnly, xramMovcOption, portIsDS390, 24bitMode, notVolatile, notUsed, notSame, operandsNotRelated, labelJTInRange, canAssign, optimizeReturn, notUsedFrom, labelIsReturnOnly, operandsLiteral, labelIsUncondJump, deadMove, useAcallAjmp` and `okToRemoveSLOC`.

This whole thing is a little kludgy, but maybe some day SDCC will have some better means. If you are looking at the `peeph*.def` files, 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 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:

\[
\text{complexity} = (\text{number of edges in control flow graph}) - (\text{number of nodes in control flow graph}) + 2;
\]

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.

### 8.3 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 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 iTemps 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. For new ports register allocator described above in 8.1.15 should be used in most cases, since it can result in substantially better code. In the old register allocator, there are two parts to register allocation.

  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 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 intrinsic named address spaces are MCU specific: While there may be an __xdata intrinsic named address space for 8051 there none for z80. SDCC has MCU specific intrinsic named address spaces, i.e. __xdata will be treated as a named address space when parsing 8051 C code but will be treated as a C identifier when parsing z80 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.

![Figure II](image-url)

<table>
<thead>
<tr>
<th>iCode</th>
<th>Operands</th>
<th>Description</th>
<th>C Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>`!'</td>
<td>IC_LEFT()</td>
<td>NOT operation</td>
<td>IC_RESULT = ! IC_LEFT;</td>
</tr>
<tr>
<td><code>~</code></td>
<td>IC_LEFT()</td>
<td>Bitwise complement of</td>
<td>IC_RESULT = ~IC_LEFT;</td>
</tr>
<tr>
<td><code>RRC</code></td>
<td>IC_LEFT()</td>
<td>Rotate right with carry</td>
<td>IC_RESULT = (IC_LEFT &lt;&lt; 1)</td>
</tr>
<tr>
<td><code>RLC</code></td>
<td>IC_LEFT()</td>
<td>Rotate left with carry</td>
<td>IC_RESULT = (IC_LEFT &lt;&lt; (sizeof(IC_LEFT)*8-1) )</td>
</tr>
<tr>
<td><code>GETHBIT</code></td>
<td>IC_LEFT()</td>
<td>Get the highest order bit of IC_LEFT</td>
<td>IC_RESULT = (IC_LEFT &gt;&gt; (sizeof(IC_LEFT)*8 -1));</td>
</tr>
<tr>
<td><code>UNARYMINUS</code></td>
<td>IC_LEFT()</td>
<td>Unary minus</td>
<td>IC_RESULT = - IC_LEFT;</td>
</tr>
<tr>
<td><code>IPUSH</code></td>
<td>IC_LEFT()</td>
<td>Push the operand into stack</td>
<td>NONE</td>
</tr>
<tr>
<td><code>IPOP</code></td>
<td>IC_LEFT()</td>
<td>Pop the operand from the stack</td>
<td>NONE</td>
</tr>
<tr>
<td><code>CALL</code></td>
<td>IC_LEFT()</td>
<td>Call the function represented by IC_LEFT</td>
<td>IC_RESULT = IC_LEFT();</td>
</tr>
<tr>
<td><code>PCALL</code></td>
<td>IC_LEFT()</td>
<td>Call via function pointer</td>
<td>IC_RESULT = (*IC_LEFT());</td>
</tr>
<tr>
<td><code>RETURN</code></td>
<td>IC_LEFT()</td>
<td>Return the value in operand</td>
<td>return IC_LEFT;</td>
</tr>
<tr>
<td><code>LABEL</code></td>
<td>IC_LABEL()</td>
<td>Label</td>
<td>IC_LABEL:</td>
</tr>
<tr>
<td><code>GOTO</code></td>
<td>IC_LABEL()</td>
<td>Goto label</td>
<td>goto IC_LABEL();</td>
</tr>
<tr>
<td>`+'</td>
<td>IC_LEFT()</td>
<td>Addition</td>
<td>IC_RESULT = IC_LEFT + IC_RIGHT</td>
</tr>
<tr>
<td><code>-</code></td>
<td>IC_LEFT()</td>
<td>Subtraction</td>
<td>IC_RESULT = IC_LEFT - IC_RIGHT</td>
</tr>
<tr>
<td><code>*</code></td>
<td>IC_LEFT()</td>
<td>Multiplication</td>
<td>IC_RESULT = IC_LEFT * IC_RIGHT;</td>
</tr>
<tr>
<td><code>/</code></td>
<td>IC_LEFT()</td>
<td>Division</td>
<td>IC_RESULT = IC_LEFT / IC_RIGHT;</td>
</tr>
<tr>
<td>`%'</td>
<td>IC_LEFT()</td>
<td>Modulus</td>
<td>IC_RESULT = IC_LEFT % IC_RIGHT;</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>IC_LEFT()</td>
<td>Less than</td>
<td>IC_RESULT = IC_LEFT &lt; IC_RIGHT;</td>
</tr>
</tbody>
</table>
### 9.1. THE ANATOMY OF THE COMPILER

#### CHAPTER 9. COMPILER INTERNALS

<table>
<thead>
<tr>
<th>iCode</th>
<th>Operands</th>
<th>Description</th>
<th>C Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;</code></td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Greater than</td>
<td>IC_RESULT = IC_LEFT &gt; IC_RIGHT;</td>
</tr>
<tr>
<td>EQ_OP</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Equal to</td>
<td>IC_RESULT = IC_LEFT == IC_RIGHT;</td>
</tr>
<tr>
<td>AND_OP</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Logical and operation</td>
<td>IC_RESULT = IC_LEFT &amp;&amp; IC_RIGHT;</td>
</tr>
<tr>
<td>OR_OP</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Logical or operation</td>
<td>IC_RESULT = IC_LEFT</td>
</tr>
<tr>
<td><code>^</code></td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Exclusive OR</td>
<td>IC_RESULT = IC_LEFT ^ IC_RIGHT;</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>BITWISEAND</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Bitwise AND</td>
<td>IC_RESULT = IC_LEFT &amp; IC_RIGHT;</td>
</tr>
<tr>
<td>LEFT_OP</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Left shift</td>
<td>IC_RESULT = IC_LEFT &lt;&lt; IC_RIGHT</td>
</tr>
<tr>
<td>RIGHT_OP</td>
<td>IC_LEFT() IC_RIGHT() IC_RESULT()</td>
<td>Right shift</td>
<td>IC_RESULT = IC_LEFT &gt;&gt; IC_RIGHT</td>
</tr>
<tr>
<td>GET_VALUE_AT_ADDRESS</td>
<td>IC_LEFT() IC_RESULT()</td>
<td>Indirect fetch</td>
<td>IC_RESULT = (*IC_LEFT);</td>
</tr>
<tr>
<td>POINTER_SET</td>
<td>IC_RIGHT() IC_RESULT()</td>
<td>Indirect set</td>
<td>(*IC_RESULT) = IC_RIGHT;</td>
</tr>
<tr>
<td><code>=</code></td>
<td>IC_RIGHT() IC_RESULT()</td>
<td>Assignment</td>
<td>IC_RESULT = IC_RIGHT;</td>
</tr>
<tr>
<td>IFX</td>
<td>IC_COND IC_TRUE IC_LABEL</td>
<td>Conditional jump. If true label is present then jump to true label if condition is true else jump to false label if condition is false</td>
<td>if (IC_COND) goto IC_TRUE; Or If (!IC_COND) goto IC_FALSE;</td>
</tr>
<tr>
<td>ADDRESS_OF</td>
<td>IC_LEFT() IC_RESULT()</td>
<td>Address of</td>
<td>IC_RESULT = &amp;IC_LEFT();</td>
</tr>
<tr>
<td>JUMPTABLE</td>
<td>IC_JTCOND IC_JTLABELS</td>
<td>Jump to list of labels depending on the value of JTCOND</td>
<td>Switch statement</td>
</tr>
<tr>
<td>CAST</td>
<td>IC_RIGHT() IC_LEFT() IC_RESULT()</td>
<td>Cast types</td>
<td>IC_RESULT = (typeof IC_LEFT) IC_RIGHT;</td>
</tr>
<tr>
<td>SEND</td>
<td>IC_LEFT()</td>
<td>This is used for passing parameters in registers; move IC_LEFT to the next available parameter register.</td>
<td>None</td>
</tr>
<tr>
<td>RECV</td>
<td>IC_RESULT()</td>
<td>This is used for receiving parameters passed in registers; Move the values in the next parameter register to IC_RESULT</td>
<td>None</td>
</tr>
</tbody>
</table>
### 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.

```c
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. {
17. sum += i;
18. mul += i * 3; /* this multiplication remains */
19. gint += j * 3; /* this multiplication changed to addition */
20. }
21. return sum+mul;
22. }
```

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 uses 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 temp6 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) _entry($9) :
Sample.c(11:1:0:0) proc _function [lr0][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(11:11:13:8:1) iTemp10 [lr13:14][int][r2 r3] := @iTemp7 [lr9:13][_far * int][DPTR]
Sample.c(11:11:14:19:1) iTemp6 [lr5:16][_near * int][r0] := iTemp10 [lr13:14][int][r2 r3]
Sample.c(11:11:15:12:1) iTemp6 [lr5:16][_near * int][r0] := iTemp6 [lr5:16][_near * int][r0] + 0x2 {short}
Sample.c(11:11:16:20:1) goto _whilecontinue_0($1)
Sample.c(11:17:17:21:0)_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(15:26:28:1) if iTemp13 [lr25:26][char][CC] == 0 goto _forbreak_0($7)
Sample.c(15:27:29:1) iTemp15 [lr27:30][short][r1] = iTemp21 [lr21:38][short][r4] + 0x3 {short}
Sample.c(15:28:32:1) iTemp11 [lr28:40][short][r3] := iTemp15 [lr29:30][short][r1] + 0x5 {short}
Sample.c(15:32:36:1) _gint [lr0:0][int] := _gint [lr0:0][int] + 0x1e {int}
Sample.c(19:38:47:1) goto _forcond_0($4)
Sample.c(19:39:48:0)_forbreak_0($7) :
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][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) :
 001015:
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 ar, r2
 ori ar, r3
 jz 001035
 001145:
iTemp7 [lr9:13][_far * int][DPTR] := _p [lr0:0][_far * int]
 mov dpl, _p
```

117
9.1. THE ANATOMY OF THE COMPILER

CHAPTER 9. COMPILER INTERNALS

```assembly
mov dph,(_p + 1)
;
add a, _p
mov _p, a
cld
adc a, (_p + 1)
;
;iTemp10 [lr13:14] [int] [r2 r3] = @[iTemp7 [lr9:13] [far * int] [DPTR]]
movx a, @dptr
movx 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($7)
cld
cmov a, r4
xrl a, #0x80
sub a, #0x8a
jnc 00107$
00115$:
;iTemp2 [lr18:40] [short] [r2] = iTemp2 [lr18:40] [short] [r2] +
;iTemp21 [lr21:38] [short] [r4]
mov a, r2
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 r7, a
mov r0, a
add a, #0xff
mov r0, a
:_gint [lr0:0] [int] = _gint [lr0:0] [int] + iTemp17 [lr23:38] [int] [r7 r0]
mov a, r7
```

118
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.

[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 :-)!

Thanks to Sourceforge http://sourceforge.net/ which has hosted the project since 1999 and donates significant download bandwidth.

Also thanks to all SDCC Distributed Compile Farm members for donating CPU cycles and bandwidth for snapshot builds.

This document was initially written by Sandeep Dutta and updated by SDCC developers. 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

--Werror, 32
--acall-ajmp, 34, 110
--all-callee-saves, 31
--allow-unsafe-read, 30
--c1mode, 30
--callee-saves, 31, 65
--code-loc <Value>, 33, 39
--code-size <Value>, 34, 39
--codeseg <Value>, 32
--compile-only, 30
--constseg <Value>, 32
--cyclomatic, 31
--data-loc <Value>, 33, 39
--debug, 26, 30, 31, 76, 88
--disable-warning, 32
--dump-ast, 35
--dump-graphs, 35
--dump-i-code, 35
--dumpall, 99
--fdollars-in-identifiers, 32
--float-reent, 31
--fomit-frame-pointer, 30
--fsigned-char, 32
--i-code-in-asm, 32
--idata-loc <Value>, 33
--int-long-reent, 31, 45, 53
--iram-size <Value>, 34, 49, 64
--less-pedantic, 32
--lib-path <path>, 33
--max-allocs-per-node, 30
--model-huge, 34
--model-large, 34, 54
--model-medium, 34
--model-small, 34
--more-pedantic, 32
--no-c-code-in-asm, 32
--no-gen-comments, 35
--no-pack-iram, 33, 34
--no-peep, 30
--no-peep-comments, 32
--no-peep-return, 30
--no-ret-without-call, 34
--no-std-crt0, 67
--no-xinit-opt, 30, 64
--nogcse, 29
--noinduction, 30
--noinvariant, 30

--nolabelopt, 30
--noloopreverse, 30
--nolospre, 30
--nooverflow, 30
--nostdinc, 31
--nositlib, 31
--nositlibcall, 30
--opt-code-size, 30
--opt-code-speed, 30
--out-fmt-ilx, 33
--out-fmt-s19, 26, 33
--pack-iram, 33, 34
--peep-asm, 30, 51
--peep-file, 30, 109
--peep-return, 30
--print-search-dirs, 22, 32
--stack-auto, 31, 34, 43, 45, 53, 56, 58
--stack-loc <Value>, 33, 39
--stack-size <Value>, 34
--std-c11, 26
--std-c2x, 26
--std-c89, 9, 10, 25, 32
--std-c95, 26
--std-c99, 9, 10, 26
--std-sdcc11, 32
--std-sdcc89, 32
--std-sdcc99, 32
--use-non-free, 8, 32, 71, 77, 78
--use-stdout, 32, 35
--vc, 32, 35
--verbose, 31
--version, 30
--xdata-loc <Value>, 39
--xram-loc <Value>, 33
--xram-size <Value>, 34, 39
--xstack, 34, 37, 56
--xstack-loc <Value>, 33
-Aquestion(answer), 29
-C, 29
-D<macro[=value]>, 29
-E, 29, 31
-I<path>, 29
-L <path>, 33
-M, 29
-MM, 29
-S, 31
-Umacro, 29
INDEX

-V, 31
-Wa asmOption[,asmOption], 32
-Wl linkOption[,linkOption], 33
-Wp preprocessorOption[,preprocessorOption], 29
-c, 30
-dD, 29
-dM, 29
-dN, 29
-mds390, 28
-mds400, 28
-mezz80_z80, 28
-mgbz80, 28
-mhc08, 28
-mmcs51, 28
-mpdk14, 28
-mpdk15, 29
-mpic14, 29
-mpic16, 29
-mr2k, 28
-mr3ka, 28
-ms08, 28
-mstm8, 28
-mltcs90, 28
-mz180, 28
-mz80, 28
-o <path/file>, 31
-pedantic-parse-number, 29
-v, 30

<NO FLOAT>, 54, 79
<file>.adb, 26, 88
<file>.asm, 26
<file>.cdb, 27, 88
<file>.dump*, 27
<file>.ihx, 26, 27
<file>.lib, 28
<file>.incl, 28
<file>.lst, 26, 41
<file>.map, 26, 40, 41
<file>.mem, 26, 40
<file>.omf, 27
<file>.rel, 26–28
<file>.rst, 26, 41
<file>.sym, 26
<stdio.h>, 53
~ Operator, 10
~ Operator, 94

8031, 8032, 8051, 8052, mcs51 CPU, 7

Absolute addressing, 41, 43
ACC (mcs51, ds390 register), 65
__addressmod, 41
Aligned array, 42, 49
Annotated syntax tree, 113
Any Order Bit, 107
AOMF, AOMF51, 27, 31, 87, 88
Application notes, 97
__asm, 47, 49–52
Assembler documentation, 50, 96
Assembler listing, 26
Assembler options, 32
Assembler routines, 47, 48, 65, 109
Assembler routines (non-reentrant), 65
Assembler routines (reentrant), 66
Assembler source, 26
at, 43
__at, 38, 40–42, 49
atomic, 44, 48
B (mcs51, ds390 register), 65
backfill unused memory, 27
banked, 62
Bankswitching, 61
Basic blocks, 119
Binary constants, 43
bit, 10, 33, 39, 94
__bit, 38, 42
Bit rotation, 106
Bit shifting, 106
Bit toggling, 10
bit-fields, 38
block boundary, 42
Boost Software License 1.0 (BSL-1.0), 9
Bug reporting, 99
Building SDCC, 17
Byte swapping, 107

C FAQ, 97
C Reference card, 97
Carry flag, 38
Changelog, 100
checksum, 27
cmake, 97
code, 32, 33
__code, 37
code banking, 61
code page (pic14), 69
Command Line Options, 28
Communication
Bug report, 99
Feature request, 100
Forums, 96
Mailing lists, 96, 100
Monitor, 96
Patch submission, 100
RSS feed, 96
Trackers, 96
wiki, 96

Compatibility with previous versions, 9
Compiler internals, 113
compiler.h (include file), 38, 94
const, 32
Copy propagation, 103
cpp, see sdcpp, see sdcpp
critical, 47
__critical, 47
Cyclomatic complexity, 31, 111
d52, 97
d52 (disassembler), 97
__data (hc08 named address space), 40
__data (mcs51, ds390 named address space), 33, 36, 39
DDD (debugger), 91, 97
Dead-code elimination, 102, 117
Debugger, 27, 88
#defines, 60
Defines created by the compiler, 60
DESTDIR, 15
Division, 44
Documentation, 21, 95
download, 99
doxygen (source documentation tool), 97
DPTR, 61, 65, 107
DPTR, DPH, DPL, 65
DS390, 34
Options
--model-flat24, 34
--protect-sp-update, 34
--stack-10bit, 34
--stack-probe, 34
--tini-libid, 35
--use-accelerator, 35
DS390 memory model, 56
DS400, 67
DS80C390, 28
DS80C400, 28, 67, 98
DS89C4x0, 98
dynamic memory allocation (malloc), 55
ELF format, 33
Emacs, 91
__endasm, 47, 50–52
Endianness, 94, 107
Environment variables, 36
Examples, 101
External stack (mcs51), 56
__far (named address space), 37, 49
Feature request, 100
Flags, 38
Flat 24 (DS390 memory model), 56
Floating point support, 25, 45, 53, 54
FPGA (field programmable gate array), 21
FpgaC ((subset of) C to FPGA compiler), 21
function epilogue, 31, 51
function parameter, 43, 65, 66
function pointer, 39
function pointers, 65
function prologue, 31, 51, 57
GBZ80, 35
Options
-<ba <Num>>, 35
-<bo <Num>>, 35
gbz80 (GameBoy Z80), 28, 67
gcc (GNU Compiler Collection), 29
gdb, 88
generic pointer, 65
getchar(), 53
GPLv2 license, 9
GPLv2+LE, 8, 55
GPLv3 license, 9
gpsim (pic simulator), 97
gutils (pic tools), 70, 97
HC08, 28, 33, 40, 46, 68
interrupt, 46, 47
Options
--out-fmt-elf, 33
HD64180 (see Z180), 40
Header files, 38, 94, 95
heap (malloc), 55
Higher Order Byte, 108
Higher Order Word, 108
Highest Order Bit, 107
I/O memory (Z80, Z180), 40
ICE (in circuit emulator), 87
iCode, 35, 113–116
__idata (mcs51, ds390 named address space), 33, 37, 39
IDE, 32, 98
Include files, 38, 94, 95
indent (source formatting tool), 97
Infineon, 34
Install paths, 14
Install trouble-shooting, 22
Installation, 12
interrupt cycles (count), 97
Intel hex format, 26, 27, 33, 88
Intermediate dump options, 35
interrupt, 39, 44–49, 51, 53, 57
__interrupt, 39, 44, 51
interrupt jitter, 47
interrupt latency, 47
interrupt mask, 47
interrupt priority, 47, 48
interrupt vector table, 33, 44, 45, 57
interrupts, 48
intrinsic named address space, 44, 56
jump tables, 104
K&R style, 25
Labels, 52
LGPLv2.1 license, 9
Libraries, 28, 31, 33, 39, 53, 55
Linker, 27
Linker documentation, 96
#pragma noloopreverse, 57
#pragma nooverlay, 44, 45, 58
#pragma opt_code_balanced, 58
#pragma opt_code_size, 58
#pragma opt_code_speed, 58
#pragma pedantic_parse_number, 58
#pragma preproc_asm, 58
#pragma restore, 57, 59
#pragma save, 57, 59
#pragma sdcc_hash, 59
#pragma stackauto, 43, 58
#pragma std_c89, 58
#pragma std_c99, 58
#pragma std_sdcc89, 58
#pragma std_sdcc99, 58
Pragmas, 57
Preprocessor, 24, 31, 58
Options, 29
PIC16 Macros, 76
printf(), 54, 55
   floating point support, 54
   parameters, 94
PIC16, 84
PIC16 Floating point support, 79
PIC16 floating point support, 79
printf_fast() (mcs51), 54
printf_fast_f() (mcs51), 54
printf_small(), 54
printf_tiny() (mcs51), 54
putchar(), 53, 94
Processor selection options, 28
project workspace, 97
promotion to signed int, 49, 93
push/pop, 50, 52, 57
putchar(), 53
Quality control, 100
reentrant, 31, 43, 53, 56, 65, 66
Register allocation, 103, 113, 116
register bank (mcs51, ds390), 39, 43, 48
Register-Allocation, 109
Regression test, 96, 100, 101
Regression test (PIC14), 101
Regression test (PIC16), 86
Related tools, 96
Release policy, 100
Reporting bugs, 99
Requesting features, 100
return value, 25, 65
rotating bits, 106
RSS feed, 96
Runtime library, 62, 64
S08, 28
s51 (simulator), 24
sbit, 10, 38
__sbit, 10
sdar, 28
sdas (sdasgb, sdas6808, sdas8051, sdasz80), 7, 50, 96
SDCC
Defines
   __SDCC (version macro), 60
   __SDCC_ds390, 60
   __SDCC_mcs51, 60
   __SDCC_pic16, 76
   __SDCC_z80, 60
   SDCC_ALL_CALLEE_SAVES, 60
   SDCC_FLOAT_REENT, 60
   SDCC_INT_LONG_REENT, 60
   SDCC_MODEL_FLAT24 (ds390), 60
   SDCC_MODEL_LARGE, 60
   SDCC_MODEL_MEDIUM, 60
   SDCC_MODEL_SMALL, 60
   SDCC_PARMS_IN_BANK1, 60
   SDCC_REVISION (svn revision number), 60
   SDCC_STACK_AUTO, 60
   SDCC_STACK_TENBIT (ds390), 60
   SDCC_USE_XSTACK, 60
Environment variables
   NO_REG_OPT, 76
   OPTIMIZE_BITFIELD_POINTER_GET
      (PIC16), 76
   SDCC_HOME, 36
   SDCC_INCLUDE, 36
   SDCC_LEAVE_SIGNALS, 36
   SDCC_LIB, 36
   SDCC_PIC14_SPLIT_LOCALS, 71
   TMP, TEMP, TMPDIR, 36
   undocumented, 36
SDCC Wiki, 100
__sdcc_external_startup(), 64
SDCDB (debugger), 24, 88, 96, 97
sdcpp (preprocessor), 24, 29, 58
sdd, 7, 96
Search path, 15
semaphore, 48
sfr, 61
   __sfr, 38, 40
   __sfr16, 38, 41
   __sfr32, 38
shc08 (simulator), 24
signal handler, 36
sloc (spill location), 29
splint (syntax checking tool), 32, 87, 97
srecord (bin, hex, ... tool), 26, 27, 33, 97
sstm8 (simulator), 24
stack, 31, 33, 37, 39, 43, 45, 47, 56, 103
stack overflow, 45
Standard-compliance, 9, 25
static, 43
Status of documentation, 8, 21
STM8, 7
STM8 memory models, 56
Strength reduction, 103, 116
struct, 25
Subexpression, 104
Subexpression elimination, 29, 102
Subversion code repository, 99, 100
Support, 99
swapping nibbles/bytes, 106
switch statement, 104
Symbol listing, 26
sz80 (simulator), 24

tabulator spacing (8 columns), 19
Tinibios (DS390), 56
Tools, 94
Trademarks, 120
type conversion, 10
type promotion, 10, 45, 49, 93
Typographic conventions, 9

uCsim, 95, 96
union, 25
UnxUtils, 19
USE_FLOATS, 54
using (mcs51, ds390 register bank), 45, 48
__using (mcs51, ds390 register bank), 39, 44, 45, 48

vararg, va_arg, 9, 10, 94
Variable initialization, 30, 41, 64
version, 21, 100
version macro, 60
volatile, 42, 44, 48, 51, 94
VPATH, 20

Warnings, 32
watchdog, 64, 94
wiki, 96, 100, 112

__xdata (hc08 named address space), 40
__xdata (mcs51, ds390 named address space, 64
__xdata (mcs51, ds390 named address space), 33, 37,
     39, 42
XEmacs, 91
__XPAGE (mcs51), 61
xstack, 33

Z180, 28, 40
I/O memory, 40
Options
--portmode, 40
Pragmas
#pragma portmode, 40
Z80, 28, 35, 40, 46, 67
I/O memory, 40
interrupt, 46
Options
--asm=<Value>, 35
--callee-saves-bc, 35
return value, 67
Z80, Z180, GBZ80, Rabbit 2000/3000, Rabbit 3000A
   CPU, 7
zlib/libpng License, 9

126