SDCC Compiler User Guide

SDCC 4.6.0
$Date:: 2024-01-29 #$
$Revision: 14650 $

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 . . . 12
- 1.7 Other Resources . . . 13

2 Installing SDCC **14**
- 2.1 Configure Options . . . 14
- 2.2 Install paths . . . 17
- 2.3 Search Paths . . . 17
- 2.4 Building SDCC . . . 19
  - 2.4.1 Building SDCC on Linux . . . 19
  - 2.4.2 Building SDCC on Mac OS X . . . 19
  - 2.4.3 Cross compiling SDCC on Linux for Windows . . . 19
  - 2.4.4 Building SDCC using Cygwin and Mingw32 . . . 19
  - 2.4.5 Building SDCC Using Microsoft Visual C++ 2010 (MSVC) . . . 20
  - 2.4.6 Windows Install Using a ZIP Package . . . 21
  - 2.4.7 Windows Install Using the Setup Program . . . 21
  - 2.4.8 VPATH feature . . . 22
- 2.5 Building the Documentation . . . 22
- 2.6 Reading the Documentation . . . 22
- 2.7 Testing the SDCC Compiler . . . 23
- 2.8 Install Trouble-shooting . . . 23
  - 2.8.1 If SDCC does not build correctly . . . 23
  - 2.8.2 What the "./configure" does . . . 24
  - 2.8.3 What the "make" does . . . 24
  - 2.8.4 What the "make install" command does. . . . 24
- 2.9 Components of SDCC . . . 24
  - 2.9.1 sdcc - The Compiler . . . 25
  - 2.9.2 sdcpp - The C-Preprocessor . . . 25
  - 2.9.3 sdas, sdld - The Assemblers and Linkage Editors . . . 25
  - 2.9.4 ucsim_51, ucsim_z80, ucsim_stm8 etc. - The Simulators . . . 25
  - 2.9.5 sdcdb - Source Level Debugger . . . 25

3 Using SDCC **26**
- 3.1 Standard-Compliance . . . 26
  - 3.1.1 ISO C90 and ANSI C89 . . . 26
  - 3.1.2 ISO C94 (aka ISO C95) . . . 26
  - 3.1.3 ISO C99 . . . 27
  - 3.1.4 ISO C11 and ISO C17 . . . 27
  - 3.1.5 ISO C23 . . . 27
  - 3.1.6 ISO C2y preview . . . 28
  - 3.1.7 Embedded C . . . 28
  - 3.1.8 Implementation-defined behavior . . . 28
    - 3.1.8.1 Translation . . . 28

3.1.8.2 Environment . . . 28
3.1.8.3 Identifiers . . . 28
3.1.8.4 Characters . . . 28
3.1.8.5 Integers . . . 29
3.1.8.6 Floating point . . . 29
3.1.8.7 Arrays and Pointers . . . 29
3.1.8.8 Hints . . . 29
3.1.8.9 Structures, unions, enumerations and bit-fields . . . 29
3.1.8.10 Qualifiers . . . 29
3.1.8.11 Preprocessing directives . . . 29
3.1.8.12 Library functions . . . 29
3.1.8.13 Architecture . . . 30
3.2 Compiling . . . 30
3.2.1 Single Source File Projects . . . 30
3.2.2 Postprocessing the Intel Hex file . . . 31
3.2.3 Projects with Multiple Source Files . . . 31
3.2.4 Projects with Additional Libraries . . . 31
3.2.5 Using sdar to Create and Manage Libraries . . . 32
3.3 Command Line Options . . . 32
3.3.1 Language dialect Options . . . 32
3.3.2 Processor Selection Options . . . 33
3.3.3 Preprocessor Options . . . 34
3.3.4 Optimization Options . . . 34
3.3.5 Other Options . . . 35
3.3.6 Linker Options . . . 37
3.3.7 MCS51 Options . . . 38
3.3.8 DS390 / DS400 Options . . . 38
3.3.9 Options common to all z80-related ports (z80, z80n, z180, r2k, r3ka, r4k, r5k, r6k, sm83, tlcs90, ez80, r800) . . . 39
3.3.10 Z80 Options (apply to z80, z80n, z180, r2k, r3ka, tlcs90, ez80) . . . 39
3.3.11 SM83 Options . . . 39
3.3.12 STM8 Options . . . 39
3.3.13 MOS6502 Options (apply to mos6502, mos65c02) . . . 39
3.3.14 Intermediate Dump Options . . . 40
3.3.15 Redirecting output on Windows Shells . . . 40
3.4 Environment variables . . . 40
3.5 SDCC Language Extensions . . . 40
3.5.1 MCS51/DS390 intrinsic named address spaces . . . 41
3.5.1.1 __data / __near . . . 41
3.5.1.2 __xdata / __far . . . 41
3.5.1.3 __idata . . . 41
3.5.1.4 __pdata . . . 41
3.5.1.5 __code . . . 42
3.5.1.6 __bit . . . 42
3.5.1.7 __sfr / __sfr16 / __sfr32 / __sbit . . . 42
3.5.1.8 Pointers to MCS51/DS390 intrinsic named address spaces . . . 43
3.5.1.9 Notes on MCS51 memory layout . . . 43
3.5.2 Z80/Z180/eZ80 intrinsic named address spaces . . . 44
3.5.2.1 __sfr (in/out to 8-bit addresses) . . . 44
3.5.2.2 __banked __sfr (in/out to 16-bit addresses) . . . 44
3.5.2.3 __far (eZ80 only) . . . 44
3.5.2.4 __sfr (in0/out0 to 8 bit addresses on Z180/HD64180) . . . 44
3.5.2.5 __far (eZ80 only) . . . 44
3.5.3 SM83 intrinsic named address spaces . . . 45
3.5.3.1 __sfr . . . 45
3.5.4 Rabbit intrinsic named address spaces . . . 45
3.5.4.1 __sfr . . . 45
3.5.4.2 __far . . . 45
3.5.5 TLCS-90 intrinsic named address spaces . . . 45

3.5.6 HC08/S08 intrinsic named address spaces . . . 45
3.5.6.1 __data . . . 45
3.5.6.2 __xdata . . . 45
3.5.7 PDK14/PDK15 intrinsic named address spaces . . . 45
3.5.7.1 __sfr . . . 45
3.5.7.2 __sfr16 . . . 45
3.5.8 MOS6502 intrinsic named address spaces . . . 45
3.5.8.1 __zp/__data /__near . . . 45
3.5.8.2 __xdata /__far . . . 46
3.5.9 Non-intrinsic named address spaces . . . 46
3.5.10 Absolute Addressing . . . 46
3.5.11 __sdcc_external_startup . . . 47
3.5.12 Preserved register specification . . . 48
3.5.13 Binary constants . . . 48
3.5.14 Returning void . . . 48
3.5.15 Omitting promotion on arguments of vararg function (mcs51, ds390, mos6502, mos65c02, pic14 and pic16 only) . . . 48
3.5.16 Parameter forward declarations . . . 48
3.5.17 _Optional . . . 48
3.6 Parameters and Local Variables . . . 48
3.7 Overlaying . . . 49
3.8 Interrupt Service Routines . . . 49
3.8.1 General Information . . . 49
3.8.1.1 Common interrupt pitfall: variable not declared *volatile* . . . 50
3.8.1.2 Common interrupt pitfall: *non-atomic access* . . . 50
3.8.1.3 Common interrupt pitfall: *stack overflow* . . . 50
3.8.1.4 Common interrupt pitfall: *use of non-reentrant functions* . . . 50
3.8.2 MCS51/DS390 Interrupt Service Routines . . . 50
3.8.3 HC08 Interrupt Service Routines . . . 51
3.8.4 Z80, Z180 and eZ80 Interrupt Service Routines . . . 51
3.8.5 Rabbit 2000, 3000, 3000A, 4000, 5000, and 6000 Interrupt Service Routines . . . 51
3.8.6 SM83 and TLCS-90 Interrupt Service Routines . . . 52
3.8.7 STM8 Interrupt Service Routines . . . 52
3.9 Enabling and Disabling Interrupts . . . 52
3.9.1 Critical Functions and Critical Statements . . . 52
3.9.2 Enabling and Disabling Interrupts directly . . . 52
3.9.3 Semaphore locking (mcs51/ds390) . . . 53
3.10 Functions using private register banks (mcs51/ds390) . . . 53
3.11 Inline Assembler Code . . . 54
3.11.1 Inline Assembler Code Formats . . . 54
3.11.1.1 Old __asm ... __endasm; Format . . . 54
3.11.1.2 New __asm__ ("inline_assembler_code") Format . . . 54
3.11.2 A Step by Step Introduction . . . 54
3.11.3 Naked Functions . . . 56
3.11.4 Use of Labels within Inline Assembler . . . 57
3.12 Support routines for integer multiplicative operators . . . 58
3.13 Floating Point Support . . . 58
3.14 Library Routines . . . 59
3.14.1 Compiler support routines (_gptrget, _mulint etc.) . . . 59
3.14.2 Stdclib functions (puts, printf, strcat etc.) . . . 59
3.14.2.1 <stdio.h> . . . 59
3.14.2.2 <malloc.h> . . . 60
3.14.3 Math functions (sinf, powf, sqrtf etc.) . . . 60
3.14.3.1 <math.h> . . . 60
3.14.4 Other libraries . . . 60
3.15 Memory Models . . . 61
3.15.1 MCS51 Memory Models . . . 61
3.15.1.1 Small, Medium, Large and Huge . . . 61
3.15.1.2 External Stack . . . 61

3.15.2 DS390 Memory Model . . . 61
3.15.3 STM8 Memory Models . . . 61
3.15.4 MOS6502 Memory Models . . . 62
3.16 Pragmas . . . 62
3.17 Defines Created by the Compiler . . . 65

4 Notes on supported Processors **66**
4.1 MCS51 variants . . . 66
4.1.1 pdata access by SFR . . . 66
4.1.2 Other Features available by SFR . . . 66
4.1.3 Bankswitching . . . 66
4.1.3.1 Hardware . . . 66
4.1.3.2 Software . . . 67
4.1.4 MCS51/DS390 Startup Code . . . 67
4.1.5 Interfacing with Assembler Code . . . 69
4.1.5.1 Global Registers used for Parameter Passing . . . 69
4.1.5.2 Register usage . . . 69
4.1.5.3 Assembler Routine (non-reentrant) . . . 70
4.1.5.4 Assembler Routine (reentrant) . . . 70
4.2 DS400 port . . . 71
4.3 The Z80, Z80N, Z180, Rabbit 2000, Rabbit 2000A, Rabbit 3000A, Rabbit 4000, Rabbit 5000, Rabbit 6000, SM83 (GameBoy), TLCS-90, eZ80, and R800 ports . . . 71
4.3.1 Startup Code . . . 71
4.3.2 Rabbit ports . . . 71
4.3.2.1 Port choice . . . 72
4.3.2.2 Rabbit hardware bugs . . . 72
4.3.2.3 Rabbit interrupt handlers . . . 73
4.3.2.4 Rabbit memory map and memory models . . . 73
4.3.3 Z80, Z180, Z80N and R800 calling conventions . . . 74
4.3.3.1 Z80 SDCC calling convention, version 1 . . . 74
4.3.3.2 Z80 SDCC calling convention, version 0 . . . 75
4.3.4 Rabbit 2000, Rabbit 2000A, Rabbit 3000A, Rabbit 4000, Rabbit 5000, Rabbit 6000, eZ80 and TLCS-90 calling conventions . . . 76
4.3.4.1 Rabbit SDCC calling convention, version 1 . . . 76
4.3.5 SM83 calling conventions . . . 77
4.3.5.1 SM83 SDCC calling convention, version 1 . . . 77
4.3.5.2 SM83 SDCC calling convention, version 0 . . . 78
4.3.6 Small-C calling convention . . . 78
4.3.7 Dynamic C calling convention . . . 78
4.3.8 Complex instructions . . . 78
4.3.9 Unsafe reads . . . 79
4.3.10 Z80 banked calls . . . 79
4.4 The HC08 and S08 ports . . . 79
4.4.1 Startup Code . . . 79
4.5 The STM8 port . . . 79
4.5.1 Calling conventions . . . 79
4.5.1.1 SDCC calling convention, version 1 . . . 79
4.5.1.2 SDCC calling convention, version 0 . . . 80
4.5.1.3 Raisonance calling convention . . . 80
4.5.1.4 IAR calling convention . . . 80
4.5.1.5 Cosmic calling convention . . . 80
4.6 The f8 port . . . 80
4.6.1 Calling convention . . . 80
4.7 The MOS6502 and WDC65C02 ports . . . 80
4.7.1 Startup Code . . . 80
4.7.2 Hooking up interrupts . . . 80
4.7.3 Reentrancy . . . 81
4.7.4 Code and Data placement . . . 81
4.8 The PIC14 port . . . 81
4.8.1 PIC Code Pages and Memory Banks . . . 82

4.8.2 Adding New Devices to the Port . . . 82
4.8.3 Interrupt Code . . . 83
4.8.4 Configuration Bits . . . 83
4.8.5 Linking and Assembling . . . 83
4.8.6 Command-Line Options . . . 84
4.8.7 Environment Variables . . . 84
4.8.8 The Library . . . 84
4.8.8.1 Enhanced cores . . . 84
4.8.8.2 Accessing bits of special function registers . . . 84
4.8.8.3 Naming of special function registers . . . 85
4.8.8.4 error: missing definition for symbol "__gptrget1" . . . 85
4.8.8.5 Processor mismatch in file "XXX". . . . 85
4.8.9 Known Bugs . . . 85
4.8.9.1 Function arguments . . . 85
4.8.9.2 Regression tests fail . . . 85
4.9 The PIC16 port . . . 85
4.9.1 Global Options . . . 87
4.9.2 Port Specific Options . . . 87
4.9.2.1 Code Generation Options . . . 87
4.9.2.2 Optimization Options . . . 87
4.9.2.3 Assembling Options . . . 88
4.9.2.4 Linking Options . . . 88
4.9.2.5 Debugging Options . . . 88
4.9.3 Environment Variables . . . 88
4.9.4 Preprocessor Macros . . . 88
4.9.5 Directories . . . 89
4.9.6 Pragmas . . . 89
4.9.7 Header Files and Libraries . . . 91
4.9.8 Header Files . . . 91
4.9.9 Libraries . . . 91
4.9.10 Adding New Devices to the Port . . . 92
4.9.11 Memory Models . . . 92
4.9.12 Stack . . . 93
4.9.13 Functions . . . 93
4.9.14 Function return values . . . 93
4.9.15 Interrupts . . . 94
4.9.16 Generic Pointers . . . 94
4.9.17 Configuration Bits . . . 95
4.9.18 PIC16 C Libraries . . . 95
4.9.18.1 Standard I/O Streams . . . 95
4.9.18.2 Printing functions . . . 96
4.9.18.3 Signals . . . 96
4.9.19 PIC16 Port – Tips . . . 97
4.9.19.1 Stack size . . . 97
4.9.20 Known Bugs . . . 97
4.9.20.1 Extended Instruction Set . . . 97
4.9.20.2 Regression Tests . . . 98

5 Debugging **99**
5.1 Debugging with SDCDB . . . 100
5.1.1 Compiling for Debugging . . . 100
5.1.2 How the Debugger Works . . . 100
5.1.3 Starting the Debugger SDCDB . . . 100
5.1.4 SDCDB Command Line Options . . . 100
5.1.5 SDCDB Debugger Commands . . . 101
5.1.6 Interfacing SDCDB with DDD . . . 102
5.1.7 Interfacing SDCDB with XEmacs . . . 102
5.2 Debugging with other debuggers (e.g. GDB): ELF / DWARF . . . 103

6 TIPS **104**
6.1 Porting code from or to other compilers . . . 105
6.2 Tools included in the distribution . . . 105
6.3 Documentation included in the distribution . . . 106
6.4 Communication online at SourceForge . . . 107
6.5 Related open source tools . . . 107
6.6 Related documentation / recommended reading . . . 107
6.7 Application notes specifically for SDCC . . . 108
6.8 Some Questions . . . 108

7 Support **109**
7.1 Reporting Bugs . . . 109
7.2 Requesting Features . . . 109
7.3 Submitting patches . . . 110
7.4 Getting Help . . . 110
7.5 ChangeLog . . . 110
7.6 Subversion Source Code Repository . . . 110
7.7 Release policy . . . 110
7.8 Quality control . . . 110
7.9 Examples . . . 111
7.10 Use of SDCC in Education . . . 111

8 SDCC Technical Data **112**
8.1 Optimizations . . . 112
8.1.1 Sub-expression Elimination . . . 112
8.1.2 Dead-Code Elimination . . . 112
8.1.3 Copy-Propagation . . . 113
8.1.4 Loop Optimizations . . . 113
8.1.5 Loop Reversing . . . 114
8.1.6 Algebraic Simplifications . . . 114
8.1.7 'switch' Statements . . . 114
8.1.8 Bit-shifting Operations. . . . 115
8.1.9 Bit-rotation . . . 116
8.1.10 Nibble and Byte Swapping . . . 116
8.1.11 Getting a Bit . . . 116
8.1.12 Higher Order Byte / Higher Order Word . . . 117
8.1.13 Placement of Bank-Selection Instructions . . . 118
8.1.14 Lifetime-Optimal Speculative Partial Redundancy Elimination . . . 118
8.1.15 Register Allocation . . . 118
8.1.16 Peephole Optimizer . . . 119
8.2 Cyclomatic Complexity . . . 120
8.3 Retargetting for other Processors . . . 121

9 Compiler internals **122**
9.1 The anatomy of the compiler . . . 122
9.2 A few words about basic block successors, predecessors and dominators . . . 127

10 Acknowledgments **128**

Chapter 1

Introduction

1.1 About SDCC

SDCC (*S*mall *D*evice *C* *C*ompiler) 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 MCS-51 based Microprocessors (8031, 8032, 8051, 8052, etc.), Dallas DS80C390 variants, NXP (formerly Freescale/Motorola) HC08 based (HC08, S08), Zilog Z80 based MCUs (Z80, Z80N, Z180, SM83, Rabbit 2000/3000/3000A, Toshiba TLCS-90, Zilog eZ80, ASCII R800), STMicroelectronics STM8 , Padauk PDK14 and PDK15. It can be retargeted for other microprocessors; support for Padauk PDK13, Rabbit 4000/5000/6000, and MOS6502 is under development, whereas Microchip PIC is currently unmaintained. 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. You might also want to have a look at the wiki https://sourceforge.net/p/sdcc/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:

type	width	default	signed range	unsigned range
_Bool / bool	8 bits, 1 byte	unsigned	-	0, 1
char	8 bits, 1 byte	unsigned	-128, +127	0, +255
short	16 bits, 2 bytes	signed	-32.768, +32.767	0, +65.535
int	16 bits, 2 bytes	signed	-32.768, +32.767	0, +65.535
long	32 bits, 4 bytes	signed	-2.147.483.648, +2.147.483.647	0, +4.294.967.295
long long[1]	64 bits, 8 bytes	signed		
_BitInt[2]	8 to 64 bits, 1 to 8 bytes			
float	4 bytes similar to IEEE 754	signed		1.175494351E-38, 3.402823466E+38
pointer	1, 2, 3 or 4 bytes	generic		
__bit[3]	1 bit	unsigned	-	0, 1

[1] Incomplete support in the pic14 and pic16 backends.

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

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

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

The latest SDCC version can be downloaded from http://sdcc.sourceforge.net/snap.php. *Please note: the compiler will probably always be some steps ahead of this documentation*[4].

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 (GPLv3 might apply depending on the libraries used when building).
    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:
    derived from GCC cpp preprocessor http://gcc.gnu.org/; GPLv3 license
  - sdas assemblers and sdld linker:
    derived from ASXXXX https://shop-pdp.net/ashtml/; 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.
    A possible exception are pic device libraries and header files which are generated 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, if Microchip has any copyright in them (which might depend on local copyright laws). 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):
    derived from GNU Binutils http://www.gnu.org/software/binutils/; GPLv3 license
  - ucsim simulators:
    GPLv2 license
  - sdcdb debugger:
    GPLv2 license
  - gcc-test regression tests:
    derived from gcc-testsuite; no license explicitely specified, but since it is a part of GCC is probably GPLv3 licensed
  - packihx:
    public domain
  - makebin:
    zlib/libpng License
  - pic libraries in device/non-free:
    Microchip Technology Inc. claims to have copyrights on this, and their term are non-free. However, a more common opinion is that Microchip Technology Inc. is just claiming a copyright on uncopyrightable facts.

[2]Incomplete support in the pic14 and pic16 backends.
[3]Only supported in the mcs51, ds390, ds400 backends.
[4]Obviously this has pros and cons

- 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 neither of the **--std-c90, --std-c99, --std-c11** or **--std-c23** command line options is used, e.g.:
  ```
  char a=3;
  printf ("%d %c\n", a, (char)a);
  ```
  will push a as an int and as a char resp if none of the above command line options are not defined,
  will push a as two ints if none of the above 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 neither of the **--std-c90, --std-c99, --std-c11** or **--std-c23** command line options is used.
- option --regextend has been removed.
- option --noregparms has been removed.
- option --stack-after-data has been removed.

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

- in older versions, the preprocessor was always called with **--std-c99** regardless of the **--std-xxx** setting. This is no longer true, and can cause compilation failures on code built with **--std-c90** 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:
```
struct {
int a :  2;
char  :  2;
int b :  2;
} s = {1, 2, 3};
/* s.a = 1, s.b = 3 */
```

  New behaviour:
```
struct {
int a :  2;
char  :  2;
int b :  2;
} s = {1, 2};
/* s.a = 1, s.b = 2 */
```

- 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 `<sfrname>_bits.<bitname>` is deprecated and replaced by `<sfrname>bits.<bitname>`. It will be obsoleted (removed) in one of next SDCC releases. See section 4.8.8.3.

- sdar archive managing utility and sdnm utilities were introduced in version 3.2.0. sdar, sdranlib and sdnm are derived from GNU Binutils package.

- In 3.2.0 the sdcclib utility is deprecated. Sdar utility should be used to create SDCC object file archives. Sdcclib utility will become obsolete in one of next SDCC releases and will be removed from SDCC packages.

- In 3.2.0 special SDCC keywords which are not preceded by a double underscore are obsoleted (removed). See section 3.1 Standard-Compliance.

- In 3.2.0 compiler macro definitions not starting with double underscore characters are deprecated if `--std-cXX` command line option is defined. They have been obsoleted (removed) after the 3.4.0 release (except for the macro SDCC, which has been removed after the 3.6.0 release (and brought back for mcs51 for the 3.7.0 release)).
- In 3.2.0 new compiler macros for processor definition were introduced for pic14 and pic16 targets: -D__SDCC_PIC16XXXX and -D__SDCC_PIC18FXXX respectively. The pic16 macro definition -D__18fXXX is deprecated. It was obsoleted (removed) after the 3.4.0 release.
- In 3.2.0 pragma config for pic16 target was introduced. See section 4.9.6
- In 3.2.0 new inline assembler format `__asm__ ('inline_assembler_code');` as an addition to `__asm ... __endasem;` format introduced. See section 3.11
- sdobjcopy utility was introduced in version 3.3.0. It is derived from GNU Binutils package.
- Before 3.4.0 release, intrinsic named address spaces were called "storage classes" in this manual.
- In 3.6.0, the default for char changed from signed to unsigned.
- In 3.7.0, the prototype for putchar() changed from void putchar(char) to int putchar(int).
- In 3.7.0 mcs51 and ds390 got a full _Bool/bool type, separate from __bit.
- In 3.7.0, the option **--nojtbound** and the corresponding pragma have been deprecated.
- In 3.7.0, the prototype for getchar() changed from char getchar(void) to int getchar(void).
- In 3.8.6, the deprecated sdcclib was removed.
- In 4.0.3, _itoa, _uitoa, _ltoa, _ultoa were renamed to __itoa, __uitoa, __ltoa, __ultoa.
- In 4.1.1, typeof.h has been removed.
- In 4.1.3, support for **--oldralloc** has been removed for the z80-related backends.
- In 4.1.10, the default calling convention switched from __sdccall(0) to __sdccall(1) for stm8 and gbz80.
- In 4.1.10, support for **--profile** has been removed for gbz80.
- In 4.1.11, the minimum Z80N Core version for the z80n port has been raised from 1.0 to 2.0.
- In 4.1.12, the default calling convention switched from __sdccall(0) to __sdccall(1) for z80, z180 and z80n.
- In 4.1.12, support for **--profile** has been removed for z80, z180, z80n.
- In 4.1.13, support for **--profile** has been removed.
- In 4.1.14, the gbz80 port was renamed to sm83.
- In 4.2.3, support for non-parenthesized arguments to __using and __interrupt was dropped.
- In 4.2.3, support for non-parenthesized arguments to __at that are not constants was dropped.
- In 4.2.4, the placement of __at in declarations was restricted.
- In 4.2.6, bool is chosen as the underlying integer type for enumerations with just two values.
- In 4.2.9, support for**--pedantic-parse-number** and #pragma pedantic_parse_numer was dropped.
- In 4.2.9, support for #pragma sdcc_hash was dropped, necessitating a wrapper macro for literal "#" characters in macro bodies.
- In 4.2.9, support for arguments to -MMD was dropped. The output file can be specified via -MF instead.
- In 4.2.10, _sdcc_external_startup was renamed to __sdcc_external_startup and support for it was added to further ports.
- In 4.2.13, for the sm83 port, __sfr addresses need to be specified using the full 16-bit address.

- In 4.2.14, byte order in output from the %p specifier of printf()-family functions was corrected for big-endian platforms (stm8, hc08, s08, mos6502) to match uintptr_t.
- In 4.3.1, __builtin_rlc, __builtin_rrc and __builtin_swap were replaced by __builtin_rot.
- In 4.3.6, support for **--parms-in-bank1** was removed from the mcs51 port.
- In 4.4.1, the address width in .rel files was increased from 24 bits to 32 bits for the z80 (and related) ports.
- In 4.5.2, the C23 standard library bit and byte utilities (stdbit.h) more closely follow the final ISO C23 standard instead of the 2022 C2X draft.
- In 4.5.3, signed, short, long on __bit or __sbit gives an error instead of being ignored silently.
- In 4.5.6, bool is no longer used as an underlying type for enums without a fixed underlying type.
- In 4.5.9, support for __builtin_rot was removed. The C2Y features from stdbit.h can be used instead.
- In 4.5.10, the semantics of realloc(ptr, 0) changed from C90 to C99.
- In 4.5.11, the ez80_z80 port has been replaced by the ez80 port, and the crt0 now needs to set up both sps and spl stack pointers, and interrupt handlers now need to save all 24 bits of registers BC, DE, HL, IY if they use them (SDCC will do this automatically for __interrupt, but hand-written interrupt handlers might need to be adapted).
- In 4.5.11, for the r2k, r2ka, r3ka ports the internal interrupt table is generated by the compiler (instead of being part of the crt0).
- In 4.5.11, for the tlcs90 port requires interrupt handlers need to save register BY if they use it (SDCC will do this automatically for __interrupt, but hand-written interrupt handlers might need to be adapted).
- In 4.5.12, void parameters may no longer be named or qualified.
- In 4.5.12, struct/union members other than anonymous struct/union or bit-fields must be named.
- In 4.5.13, stray non-ASCII characters in source files are an error instead of being silently ignored.
- In 4.5.15, the omission of integer promotions on explicitly cast variable arguments only happens for the mcs51, ds390, mos6502, mos65c02, pic14 and pic16 ports.
- In 4.5.17, the names of mixed-sign support functions for multiplicative operators have been fixed.
- In 4.5.18, plain int bit-fields are signed.
- In 4.5.20, some non-matching return types that previously raised warnings raise errors.
- In 4.5.23, `_Generic` works as clarified by C23, more attention is payed to implicit `char` signedness, string literals are only `const` if requested with `--fconst-str`, and a read-only `__code` address space only implies `const` if requested with `--fconst-code`.
- In 4.5.24, `--fconst-str` has been renamed to `--fconst-stringlit`.

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-z80n-port Excludes the z80n port

--disable-r2k-port Excludes the r2k port

--disable-r2ka-port Excludes the r2ka port

--disable-r3ka-port Excludes the r3ka port

--disable-r4k-port Excludes the r4k port

--disable-r5k-port Excludes the r5k port

--disable-r6k-port Excludes the r6k port

--disable-sm83-port Excludes the SM83 port

--disable-tlcs90-port Excludes the TLCS-90 port

--disable-ez80-port Excludes the e80 port

--disable-r800-port Excludes the r800 port

--disable-ds390-port Excludes the DS390 port

--disable-ds400-port Excludes the DS400 port

--disable-hc08-port Excludes the HC08 port

--disable-s08-port Excludes the S08 port

--disable-stm8-port Excludes the STM8 port

--disable-pdk13-port Excludes the PDK13 port

--disable-pdk14-port Excludes the PDK14 port

--disable-pdk15-port Excludes the PDK15 port

--disable-pic14-port Excludes the PIC14 port

--disable-pic16-port Excludes the PIC16 port

--disable-mos6502-port Excludes the MOS6502

--disable-mos65c02-port Excludes the MOS65C02

--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 sdccconf.h built by 'configure' for Win32.

These defaults are:

Variable	default	Win32 builds
PREFIX	/usr/local	\sdcc
EXEC_PREFIX	*$PREFIX*	*$PREFIX*
BINDIR	*$EXEC_PREFIX*/bin	*$EXEC_PREFIX*\bin
DATADIR	*$DATAROOTDIR*	*$DATAROOTDIR*
DATAROOTDIR	*$PREFIX*/share	*$PREFIX*
DOCDIR	*$DATAROOTDIR*/sdcc/doc	*$DATAROOTDIR*\doc
INCLUDE_DIR_SUFFIX	sdcc/include	include
NON_FREE_INCLUDE_DIR_SUFFIX	sdcc/non-free/include	non-free/include
LIB_DIR_SUFFIX	sdcc/lib	lib
NON_FREE_LIB_DIR_SUFFIX	sdcc/non-free/lib	non-free/lib

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

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

Examples:

```
./configure
./configure --prefix=''/usr/bin'' --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 \
--host=i586-mingw32msvc \
--build=unknown-unknown-linux-gnu
```

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

```
./configure \
--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''
```

2.2 Install paths

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

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

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

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

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

Moreover the install path can be changed by defining DESTDIR:

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

Please note that DESTDIR must have a trailing slash!

2.3 Search Paths

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

1. Binary files (preprocessor, assembler and linker)

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

2. Include files

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

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.).

#	Search path	default	Win32 builds
1	--L dir	--L dir	--L dir
2	$SDCC_LIB/<*model*>	$SDCC_LIB/<*model*>	$SDCC_LIB/<*model*>
3	$SDCC_LIB	$SDCC_LIB	$SDCC_LIB
4	$SDCC_HOME/ *$PREFIX2DATA_DIR/* *$LIB_DIR_SUFFIX/* <*model*>	$SDCC_HOME/ share/sdcc/lib/<*model*>	$SDCC_HOME\ lib\<*model*>
5	path(argv[0])/ *$BIN2DATADIR/* *$LIB_DIR_SUFFIX/* <*model*>	path(argv[0])/../sdcc/ lib/<*model*>	path(argv[0])\ ..\lib\ <*model*>
6	*$DATADIR/non-free/* *$LIB_DIR_SUFFIX/* <*model*>	/usr/local/share/sdcc/ lib/<*model*>	(not on Win32)
7	$SDCC_HOME/ *$PREFIX2DATA_DIR/* *non-free/* *$LIB_DIR_SUFFIX/* <*model*>	$SDCC_HOME/share/sdcc/ non-free/lib/<*model*>	$SDCC_HOME\ lib\non-free\<*model*>
8	path(argv[0])/ *$BIN2DATADIR/* *non-free/* *$LIB_DIR_SUFFIX/* <*model*>	path(argv[0])/../sdcc/ non-free/lib/<*model*>	path(argv[0])\..\ lib\non-free\<*model*>
9	*$DATADIR/non-free/* *$LIB_DIR_SUFFIX/* <*model*>	/usr/local/share/sdcc/ non-free/lib/ <*model*>	(not on Win32)

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

2.4 Building SDCC

SDCC can be built for various host platforms using the instructions provided below. Note that the PIC14 and PIC16 library folders in the source distribution contain Autotools-generated files. These are included for convenience and to avoid introducing Autotools as an additional dependency. They have to be regenerated in case of a version mismatch. Alternatively, the PIC backends can be disabled.

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-yyyymmdd-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-yyyymmdd-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.

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 MinGW gcc cross compiler SDCC can be cross-compiled for Win32. See section 'Configure Options'. SDCC requires boost, but the header-only parts should be sufficient: Get a current boost tarball from www.boost.org, unpack it, and choose suitable configure options so the headers are found by the C++ compiler.

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

- flex
- bison
- gcc ; version 3.x is fine, no need to use the old 2.9x
- binutils ; selected with gcc
- make
- 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)

TODO: This section is outdated. Current SDCC requires features noit present in MSVC++ 2010. MSVC++ 2019 or newer is required.

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-yyyymmdd-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-*

yyyymmdd-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-yyyymmdd-rrrr.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!
make
```

That's it! **configure** will create the directory tree will all the necessary Makefiles in ~/sdcc.build. It automagically computes the variables srcdir, top_srcdir and top_buildir 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 usefull 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[1].
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 f.e. was a reference like http://www.opencores.org together with a statement "some processors which are targeted by SDCC can be implemented in a *f*ield *p*rogrammable *g*ate *a*rray" 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.

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

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/z80n/z180/r2k/r2ka/r3ka/sm83/tlcs90/ez80/r800/ds390/pic16/pic14/TININative/ds400/hc08/
4.4.0 #14546 (Linux)
```

If it doesn't run, or gives a message about not finding sdcc program, then you need to check over your installation. Make sure that the sdcc bin directory is in your executable search path defined by the PATH environment setting (see section 2.8 Install trouble-shooting for suggestions). Make sure that the sdcc program is in the bin folder, if not perhaps something did not install correctly.

SDCC is commonly installed as described in section "Install and search paths".

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

```
char test;

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

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

The next step is to try it with the linker. Type in **"sdcc test.c**". If all goes well the compiler will link with the libraries and produce a test.ihx output file. If this step fails (no test.ihx, and the linker generates warnings), then the problem is most likely that SDCC cannot find the /usr/local/share/sdcc/lib directory (see section 2.8 Install trouble-shooting for suggestions).

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

```
#include <string.h>

char str1[10];

void main(void) {
  strcpy(str1, "testing");
}
```

Compile this by typing **"sdcc test.c"**. This should generate a test.ihx output file, and it should give no warnings such as not finding the string.h file. If it cannot find the string.h file, then the problem is that SDCC cannot find the /usr/local/share/sdcc/include directory (see the section 2.8 Install trouble-shooting section for suggestions). Use option **--print-search-dirs** to find exactly where SDCC is looking for the include and lib files.

2.8 Install Trouble-shooting

2.8.1 If SDCC does not build correctly

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

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

and build it like:

make 2>&1 | tee make.log

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

2.8.2 What the "./configure" does

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

2.8.3 What the "make" does

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

2.8.4 What the "make install" command does.

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

2.9 Components of SDCC

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

You might want to look at the files which are installed in <installdir>. At the time of this writing, we find the following programs, among others, 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.
- sdasz80**,** sdasgb - The Z80 and GameBoy Z80 assemblers.
- sdas6808 - The 6808 assembler.
- sdasstm8 - The STM8 assembler.
- sdld -The linker for 8051 and STM8 type processors.
- sdldz80**,** sdldgb - The Z80 and GameBoy Z80 linkers.
- sdld6808 - The 6808 linker.
- ucsim_51, s51 - The ucSim 8051 simulator.
- ucsim_m68hc08 - The ucSim 6808 simulator.
- ucsim_stm8 - The ucSim STM8 simulator.
- ucsim_z80 - The ucSim Z80 simulator.
- sdcdb - The source debugger.
- sdar, sdranlib, sdnm, sdobjcopy - The sdcc archive managing and indexing utilites.
- 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

2.9.1 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 retargettable 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 (f.e. macros, more targets) ASxxxx Cross Assemblers nowadays available from Alan Baldwin https://shop-pdp.net/. 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 ucsim_51, ucsim_z80, ucsim_stm8 etc. - The Simulators

ucsim_51 (or s51), ucsim_z80 ucsim_m68hc08 and ucsim_stm8 , as well as similarly named programs provided with SDCC, 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 the STM8 and various others.

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

3.1.1 ISO C90 and ANSI C89

Use `--std-c90` to compile in this mode.

Deviations from standard compliance

- initialization of structure arrays must be fully braced.

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

- `float` is substituted for (`long`) `double` and a warning is emitted.
- K&R style functions are syntactically supported, but with the semantics of ISO style functions.
- ISO C90 semantics for realloc(ptr, 0) are unclear. SDCC follows ISO C99 semantics.

Features missing in some ports

- pic14, pic16: structures and unions cannot be passed as function parameters
- hc08, s08, mos6502, pic14, pic16: they cannot be a return value from a function, e.g.:

```
struct s { ...  };
struct s foo1 (struct s parms) /* unsupported in these SDCC ports */
{
    struct s rets;
    ...
    return rets; /* unsupported in these SDCC ports */
}
```

- mcs51, ds390, hc08, s08, pdk13, pdk14, pdk15 and mos6502 ports: functions are not reentrant unless explicitly declared as such or `--stack-auto` is specified.

3.1.2 ISO C94 (aka ISO C95)

Use `--std-c94` or `--std-c95` to compile in this mode.

Implementation status

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.

Deviations from standard compliance

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

- Objects of variably-modified types.
- `ptrdiff_t` has 16 bits, while the standard requires at least 17 bits.

Features missing in some ports

- pic14: there is no support for 64 bit integer types.

3.1.4 ISO C11 and ISO C17

Use `--std-c11` to compile in this mode.

Implementation status

Except for the issues mentioned in the section above, this standard is supported. Note: Variably-modified types became optional in this version.

3.1.5 ISO C23

Use `--std-c23` to compile in this mode.

Deviations from standard compliance

- initialization of structure arrays must be fully braced.

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

- `float` is substituted for (`long`) `double` and a warning is emitted.
- Support for attributes is slightly incomplete.
- Checked integer arithmetic is not supported for (`unsigned`) `long long`.
- #embed is not supported

Features missing in some ports

- pic14, pic16: structures and unions cannot be passed as function parameters
- hc08, s08, mos6502, pic14, pic16: they cannot be a return value from a function, e.g.:

```
struct s { ...  };
struct s foo1 (struct s parms) /* unsupported in these SDCC ports */
{
    struct s rets;
    ...
    return rets; /* unsupported in these SDCC ports */
}
```

- mcs51, ds390, hc08, s08, pdk13, pdk14, pdk15 and mos6502 ports: functions are not reentrant unless explicitly declared as such or `--stack-auto` is specified.
- pic14: there is no support for 64 bit integer types.
- pic14, pic16: `_BitInt` is not supported.

3.1.6 ISO C2y preview

Use `--std-c2y` to compile in this mode.

Implementation status

In anticipation of the upcoming version of the language standard, C2y, SDCC supports the following prospective features:

- N3260: `_Generic` selection expression with a type operand
- N3353: Obsolete Octal and Provide New, Proper Escape Sequences
- N3356: `if` Declarations
- N3369: The `_Lengthof` Operator (Note: renamed to `_Countof` following N3469)
- N3370: Case range expressions
- N3398: The `containerof` Macro
- N3485: Literal Suffixes for `size_t` (as an extension)
- N3804: The Elvis operator `?:`

3.1.7 Embedded C

SDCC supports objects in named address spaces and to some degree pointers to such objects. The support for fixed-point math in SDCC is inconsistent with the standard. Other parts of the standard are not supported.

3.1.8 Implementation-defined behavior

3.1.8.1 Translation

- Diagnostics are output to stderr, in the format <filename>:<line-number>: <warning|error> <diagnostic-number>: <diagnostic-description>
- Nonempty sequences of white-space are retained in translation phase 3.

3.1.8.2 Environment

- When no custom crt0 is used (i.e. the port does not support a custom crt0, or the option is not used), the name and type of the function called at program startup is `void main(void)` or `int main(void)`. Otherwise the name and type depends on the custom crt0.
- See SDCC source (and your own code if you use a custom crt0 for a target that supports it) for any other information on the environment.

3.1.8.3 Identifiers

- See the compiler and assembler source for information on characters that may appear in identifiers and on the number of significant initial characters.

3.1.8.4 Characters

- There are 8 bits in a byte.
- Values of members of the execution character set: TODO.
- Values of members of the execution character set for escape sequences: TODO.
- Value of char with something weird stored in it: TODO.
- unsigned char has the same range, representation and behavior as "plain" char (unless --fsigned-char is used).
- See the SDCC source for further information on character sets.

3.1.8.5 Integers

- There are no extended integer types.

3.1.8.6 Floating point

- See the implementation (soft float library) for any information on floating point.

3.1.8.7 Arrays and Pointers

- For the result of converting between pointers and integers see the SDCC source code.
- For the size of the result of subtracting two pointers to elements of the same array see the SDCC source code.

3.1.8.8 Hints

- The extend to which suggestions made by register are effective depends on the target.
- SDCC will inline functions if and only if they are declared using inline and do not have variable arguments.

3.1.8.9 Structures, unions, enumerations and bit-fields

- A plain int bit-field is treated as a signed int bit-field.
- There are no allowed bit-field types other than _Bool, int, signed int, unsigned int and the bit-precise integer types.
- Atomic types are not permitted for bit-fields.
- If a bit-fields does not fit into the same byte as the previous bit-fields, it starts on the next byte.
- bit-fields are allocated in the same order as they appear in the source.
- Non-bit-field members of structures are aligned on byte boundaries (i.e. there are no padding bytes).
- structures do not need to have named members.
- For enumerations, the compatible type is the first from the following list that fits the constants: unsigned char, signed char, unsigned int, signed int, unsigned long int, signed long int, unsigned long long int, signed long long int.

3.1.8.10 Qualifiers

- SDCC shall preserve all volatile reads and writes, but does not guarantee them to be atomic (except for atomic types and volatile sig_atomic_t).
- Function types may not be qualified.

3.1.8.11 Preprocessing directives

- See the preprocessor source for information on preprocessing directives.

3.1.8.12 Library functions

- See the respective library headers for the library functions available.
- See assert.h and library source for the format of the diagnostic printed by the assert macro.
- There is no fegetexceptflag function.
- There is no feraiseexcept function.
- There is no setlocale function.
- There is no FLT_EVAL_METHOD macro.
- There is no DEC_EVAL_METHOD macro.

- There are no non-required domain errors for mathematics functions.
- See library source for the values returned by mathematical functions on domain error and pole error (and anything else on mathematical functions and floating type encodings).
- calloc, malloc and realloc return a pointer to an allocated object when the requested size is zero.
- See library headers for the null-pointer constant to which NULL expands.
- See library headers and source for anything else about the library.

3.1.8.13 Architecture

- See the respective library headers for the values or expressions for macros specified in float.h, limits.h, stdint.h.
- Multithreading is not supported.
- The number of bytes in any object is the minimum allowed (except for some padding bits on bit-fields), byte order depends on the target.
- No extended alignments are supported.
- There are no valid alignments other than those returned by _Alignof.
- sizeof always returns the smallest value allowed assuming an 8-bit char. _Alignof always returns 0.

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 *srecord* - see also section 3.2.2 on the following page). Both formats are documented in the documentation of srecord
- sourcefile.adb - An intermediate file containing debug information needed to create the .cdb file (with --debug)
- sourcefile.cdb - An optional file (with --debug) containing debug information. The format is documented in cdbfileformat.pdf
- sourcefile.omf - An optional AOMF or AOMF51 file containing debug information (generated with option --debug). The (Intel) *a*bsolute *o*bject *m*odule *f*ormat 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 it self (generated with option --dumpall) (see section 3.3.14 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 0xfffe -little-endian-checksum-negative 0xfffe 0x02 0x02  -o sourcefile.hex -intel
```

The srecord package is available at 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
```

Note that in this case, **the source file being compiled is always linked in first**, regardless of its position in the command line. This can be problematic when a custom crt0 is supplied as object file. As an alternative to the above, *foomain.c* can be compiled separately, and the resulting object file can then be linked with the other object files using a separate command:

```
sdcc -c foomain.c
sdcc custom_crt0.rel foomain.rel foo1.rel foo2.rel
```

The file containing the *main()* function must be the first file specified in the command line, since the linkage editor processes object files 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. Section 3.2.5 below contains a minimal example. Libraries created in this manner can be included in the command line. Make sure

[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).

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 Language dialect Options

--std-<arg> Determine the language standard. For enhanced compatibility with other compilers, **--std** can also be used with a single dash (i.e. **-std**) and with = or ␣ (whitespace) as delimiter. The language standard, specified via <**arg**>, can be one of the following:

- **c90** Follow the ISO/IEC 9899 First Edition standard (ANSI C89 / ISO C90). Alternative spellings: **c89**, **iso9899:1990**
- **c94** Follow the ISO/IEC 9899 First Edition standard as modified in amendment 1. Alternative spelling: **c95, iso9899:199409**
- **c99** Follow the ISO/IEC 9899 Second Edition standard (ISO C99). Alternative spelling: **iso9899:1999**
- **c11** Follow the ISO/IEC 9899 Third Edition standard (ISO C11). Alternative spelling: **iso9899:2011**
- **c17** Follow the ISO/IEC 9899 Fourth Edition standard (ISO C17). Alternative spellings: **iso9899:2017**, **c18**, **iso9899:2018**
- **c23** Follow the ISO/IEC 9899 Fifth Edition standard (ISO C23). Alternative spelling: **c2x**
- **c2y** Enable features anticipated for the Sixth Edition standard (currently abbreviated ISO C2y).
- **sdcc90** Generally follow ANSI C89 / ISO C90, but allow some SDCC behaviour that conflicts with the standard. Alternative spelling: **sdcc89**
- **sdcc99** Generally follow ISO C99, but allow some SDCC behaviour that conflicts with the standard.
- **sdcc11** Generally follow ISO C11, but allow some SDCC behaviour that conflicts with the standard (default).
- **sdcc17** Generally follow ISO C17, but allow some SDCC behaviour that conflicts with the standard. Alternative spelling: **sdcc18**
- **sdcc23** Generally follow ISO C23, but allow some SDCC behaviour that conflicts with the standard. Alternative spelling: **sdcc2x**
- **sdcc2y** Generally follow ISO C2y, but allow some SDCC behaviour that conflicts with the standard.

--fdollars-in-identifiers Permit '$' as an identifier character.

--fsigned-char By default char is `unsigned`. To set the signedness for characters to signed, use the option --fsigned-char. If this option is not set and no signedness keyword (unsigned/signed) is given, a char will be unsigned. All other types are unaffected.

--const-stringlit Make string literals const.

3.3.2 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.

-mz80n Generate code for the Z80N processor.

-mz180 Generate code for the Zilog Z180 family of processors.

-mr2k Generate code for the Rabbit 2000 processor.

-mr2ka Generate code for the Rabbit 2000A / Rabbit 3000 family of processors.

-mr3ka Generate code for the Rabbit 3000A processor.

-mr4k Generate code for the Rabbit 4000 processor.

-mr5k Generate code for the Rabbit 5000 processor.

-mr6k Generate code for the Rabbit 6000 processor.

-msm83 Generate code for the Sharp SM83 processor.

-mtlcs90 Generate code for the Toshiba TLCS-90 processor.

-mez80 Generate code for the Zilog eZ80 processor.

-mr800 Generate code for the ASCII R800 processor.

-mstm8 Generate code for the STMicroelectronics STM8 family of processors.

-mpdk13 Generate code for Padauk processors with 13 bit wide program memory.

-mpdk14 Generate code for Padauk processors with 14 bit wide program memory.

-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).

-mmos6502 Generate code for the original MOS Technology NMOS 6502 processor and compatible derivatives including the 6510 and 8502.

-mmos65c02 Generate code for the CMOS Rockwell/WDC 65C02.

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

3.3.3 Preprocessor Options

SDCC uses *sdcpp*, an adapted version of the GNU Compiler Collection preprocessor *cpp* (*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/.

-I<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.

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

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

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

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

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

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

3.3.4 Optimization Options

--**no**purity Disable optimization based on functions being pure.

--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 (*s*pill *loc*ations, sloc). A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It is recommended that this option NOT be used, #pragma nogcse can be used to turn off global subexpression elimination for a given function only.

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

--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 increased memory use during compilation) and more optimized code being generated. Setting it to lower values speeds up compilation, but does not optimize as much. The default value is 3000. This option currently does not affect the mcs51, ds390, pic14 and pic16 ports.

--**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.

3.3.5 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.

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

--syntax-only Parse and verify syntax only, no output files are produced.

-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\test3\output 1\\"* or put after the final quote, for example: *-o "F:\Projects\test3\output 1"*. The path using slashes for directory delimiters can be used too, for example: *-o "F:/Projects/test3/output 1/"*.

-x <type> The specified type overrides the file type that SDCC detected based on the file name extension. The currently supported options are "c", "c-header" and "none". The option "none" restores the default behavior.

--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 62.

--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. By default, 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. When –out-gmt-elf is used, the debug information is in DWARF format instead.

-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.

--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.

--norestartseqatomics Omit restartable sequence support routines for atomics; can reduce code size, but disables support for atomic that use these support routines; trying to use such atomics will then result in a diagnostic.

--verbose Shows the various actions the compiler is performing.

-V Shows the actual commands the compiler is executing.

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

--no-peep-comments 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 62.

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

--Werror Treat all warnings as errors.

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

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

--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.

--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

3.3.6 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> The value entered can be in Hexadecimal or Decimal format, e.g. --stack-loc 0x20 or --stack-loc 32.

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

--idata-loc <Value> The start location of the indirectly addressable internal ram of the 8051, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, eg. --idata-loc 0x88 or --idata-loc 136.

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

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

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

--out-fmt-elf The linker output (final object code) is in ELF format. (Currently only supported for the HC08, S08 and STM8 processors). When used with –debug, the debug info is in DWARF format instead of CDB.

-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.

3.3.7 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.

--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.

3.3.8 DS390 / DS400 Options

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

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

--stack-10bit Generate code for the 10 bit stack mode of the Dallas DS80C390 part. This is the one and only that the ds390 code generator supports right now and is default when using *-mds390*. In this mode, the stack is located in the lower 1K of the internal RAM, which is mapped to 0x400000. Note that the support is incomplete, since it still uses a single byte as the stack pointer. This means that only

the lower 256 bytes of the potential 1K stack space will actually be used. However, this does allow you to reclaim the precious 256 bytes of low RAM for use for the DATA and IDATA segments. The compiler will not generate any code to put the processor into 10 bit stack mode. It is important to ensure that the processor is in this mode before calling any re-entrant functions compiled with this option. In principle, this should work with the *--stack-auto* option, but that has not been tested. It is incompatible with the *--xstack* option. It also only makes sense if the processor is in 24 bit contiguous addressing mode (see the *--model-flat24 option*).

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

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

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

3.3.9 Options common to all z80-related ports (z80, z80n, z180, r2k, r3ka, r4k, r5k, r6k, sm83, tlcs90, ez80, r800)

--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.

3.3.10 Z80 Options (apply to z80, z80n, z180, r2k, r3ka, tlcs90, ez80)

--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-regs-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.

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

3.3.11 SM83 Options

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

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

3.3.12 STM8 Options

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

--model-large Generate code for Large model programs, see section Memory Models for more details. If this option is used all source files in the project have to be compiled with this option. It must also be used when invoking the linker.

3.3.13 MOS6502 Options (apply to mos6502, mos65c02)

--model-small Generate code for small model programs, see section Memory Models for more details.

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

--no-zp-spill Force the compiler to spill registers to 16-bit addressable memory (xdata) instead of Page Zero. When running out of Page Zero space, this option will allow to free many Page Zero locations at the expense of slightly larger and slower code

3.3.14 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.

3.3.15 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 won't be documented[2].

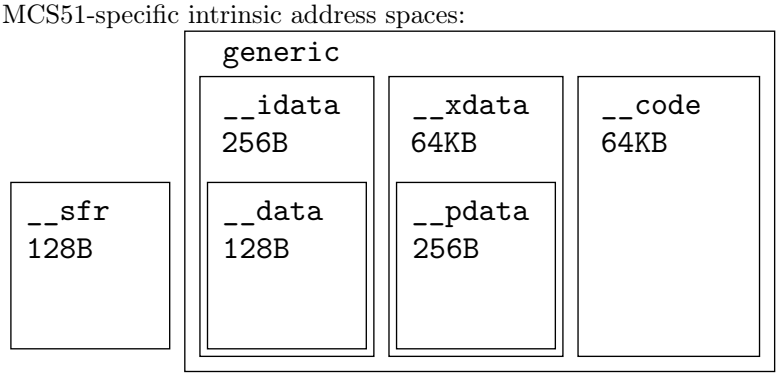
3.5 SDCC Language Extensions

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. Depending on the target architecture, support can be limited to objects in such address spaces and exclude pointer-based access to those.

[2] if you are curious search in SDCC's sources for `"getenv"`

3.5.1 MCS51/DS390 intrinsic named address spaces

SDCC supports the following MCS51-specific intrinsic address spaces:



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.:

```
__data unsigned char test_data;
```

Writing 0x01 to this variable generates the assembly code:

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

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.:

```
__xdata unsigned char test_xdata;
```

Writing 0x01 to this variable generates the assembly code:

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

3.5.1.3 __idata

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

```
__idata unsigned char test_idata;
```

Writing 0x01 to this variable generates the assembly code:

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

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

```
__pdata unsigned char test_pdata;
```

Writing 0x01 to this variable generates the assembly code:

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

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

3.5.1.5 __code

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

```
__code unsigned char test_code;
```

Read access to this variable generates the assembly code:

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

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

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

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

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

3.5.1.6 __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[3]. 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 *s*pecial *f*unction *r*egisters 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,

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

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;

/* 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 __*data* 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 MCS-51 family may have up to 128 bytes of additional, indirectly addressable, internal RAM memory (__*idata*). Furthermore, some chips may have some built in external memory (__*xdata*) which should not be confused with the internal, directly addressable RAM memory (__*data*). Sometimes this built in __*xdata* memory has to be activated before using it (you can probably find this information on the datasheet of the microcontroller your are using, see also section 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 (__*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 *i*nput/*o*utput memory. I/O memory is accessed with special instructions, e.g.:

```
__sfr __at(0x78) IoPort;  /* define a var in I/O space at 78h called IoPort */
```

Writing 0x01 to this variable generates the assembly code:

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

3.5.2.2 __banked __sfr (in/out to 16-bit addresses)

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

```
__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 __far (eZ80 only)

The ez80 port supports a 1 MB __far address space that contains the generic address space. Support for __far is still very incomplete and should be considered experimental.

3.5.2.4 __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.

3.5.2.5 __far (eZ80 only)

The ez80 port supports a 24 MB __far address space that contains the generic address space. This support is still very incomplete and should be considered experimental.

3.5.3 SM83 intrinsic named address spaces

3.5.3.1 __sfr

The keyword __*sfr* is an alternative way to access memory locations 0xff00 to 0xffff, which are typically used for memory-mapped I/O e.g.:

```
__sfr __at(0xff01) IoPort;
```

3.5.4 Rabbit intrinsic named address spaces

3.5.4.1 __sfr

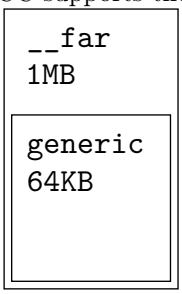
The Rabbit ports support a 64 KB __sfr address space corresponding to the "internal i/o"space of the Rabbits.

3.5.4.2 __far

The Rabbit ports supports a 1 MB __far address space that contains the generic address space. Due to limitations of the Rabbit ldp instructions, variables declared as __far can be neither volatile nor atomic.

3.5.5 TLCS-90 intrinsic named address spaces

SDCC supports the following TLCS-90-specific intrinsic address spaces:



Support for __far is still very incomplete and should be considered experimental.

3.5.6 HC08/S08 intrinsic named address spaces

3.5.6.1 __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.

3.5.6.2 __xdata

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

3.5.7 PDK14/PDK15 intrinsic named address spaces

3.5.7.1 __sfr

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

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

3.5.7.2 __sfr16

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

3.5.8 MOS6502 intrinsic named address spaces

3.5.8.1 __zp/__data /__near

Variables in the address space __zp reside in the first 256 bytes of memory (the Page Zero). The MOS6502 is most efficient at accessing variables stored here. Pointers can only be dereferenced directly if they reside in Page Zero. This is the default for the small memory model.

3.5.8.2 __xdata / __far

Variables in the address space __xdata can reside anywhere in the 64K memory space. This is the default (for the large memory model).

3.5.9 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:

```
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:

```
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.

3.5.10 Absolute Addressing

Data items can be assigned an absolute address with the __*at* <*address*> keyword (the address needs to be either a parenthesized constant expression or a constant), which can also be used in addition to a named address space, e.g.:

```
__xdata unsigned int __at (0x7ffe) 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.:

```
__code char __at (0x7ff0) 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:

```
volatile __xdata unsigned char __at (0x8000) 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.:

```
__bit __at (0x02) bvar;
```

The above example will allocate the variable at offset 0x02 in the bit-addressable space. There is no real advantage to assigning absolute addresses to variables in this manner, unless you want strict control over all the variables allocated. One possible use would be to write hardware portable code. For example, if you have a routine that uses one or more of the microcontroller I/O pins, and such pins are different for two different hardwares, you can declare the I/O pins in your routine using:

```
extern volatile __bit MOSI;    /* master out, slave in */
extern volatile __bit MISO;    /* master in, slave out */
extern volatile __bit MCLK;    /* master clock */

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

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

```
__bit __at (0x80) MOSI;    /* I/O port 0, bit 0 */
__bit __at (0x81) MISO;    /* I/O port 0, bit 1 */
__bit __at (0x82) MCLK;    /* I/O port 0, bit 2 */
```

Similarly, for the second hardware you would use

```
__bit __at (0x83) MOSI;    /* I/O port 0, bit 3 */
__bit __at (0x91) MISO;    /* I/O port 1, bit 1 */
__bit __at (0x92) MCLK;    /* I/O port 1, bit 2 */
```

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

3.5.11 __sdcc_external_startup

When a function `unsigned char __sdcc_external_startup(void)` is present, it is executed before both the initialization of static and global variables, as well as `main`. This allows to implement functionality that needs to be done early. For example: disabling a hardware watchdog that would otherwise bite during the time it takes to initialize global variables; setup or calibration of system/peripheral clocks; or, a memory check that needs to be done before any memory (other than the return address for `__sdcc_external_startup` itself) is in use.

If this routine returns a non-zero value, the static and global variable initialization will be skipped and the function `main` will be invoked. Otherwise static and global variables will be initialized before the function `main` is invoked.

For mos6502, z80, z80n, z180, r2k, r2ka, r3ka, r4k, r5k, r6k, sm83, tlcs90, ez80, r800, when using a custom crt0, support depends on the custom crt0.

3.5.12 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:

```
void f(void) __preserves_regs(b, c, iyl, iyh);
```

3.5.13 Binary constants

SDCC supports the use of C23 binary constants, such as 0b01100010 when the compiler is invoked using `--std-sdccxx`, even when the corresponding `--std-cxx`, does not. Note: `xx` is a placeholder for the desired version of the C standard.

3.5.14 Returning void

SDCC allows functions to return expressions of type void. This feature is only enabled when the compiler is invoked using `--std-sdccxx`. Note: `xx` is a placeholder for the desired version of the C standard.

3.5.15 Omitting promotion on arguments of vararg function (mcs51, ds390, mos6502, mos65c02, pic14 and pic16 only)

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 compability 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. Note: `xx` is a placeholder for the desired version of the C standard.

3.5.16 Parameter forward declarations

SDCC (when using `--std-sdccxx`) supports parameter forward declarations for s single parameter of integer type. The supported functionality is thus a subset of the GCC extension:

void f(size_t n; char a[static n], size_t n);

3.5.17 _Optional

SDCC supports the _ *Optional* qualifier.

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/MOS6502/PDK13/PDK14/PDK15 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.:

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

Since stack space on 8051 and MOS6502 is limited, and accessing the stack is slow for the Padauk, 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.:

```
unsigned char foo(__xdata int parm)
{
    __xdata unsigned char i;
    __bit bvar;
    __data unsigned char __at (0x31) 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-bitregisters 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.:

```
#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 *i*nterrupt *s*ervice *r*outines to be coded in C, with some extended keywords.

```
void timer_isr (void) __interrupt (1) __using (1)
{
    ...
}
```

The optional number following the __interrupt keyword is the interrupt number this routine will service. When present, the compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. If you have multiple source files in your project, interrupt service routines can be present in any of them, but a prototype of the isr MUST be present or included in the file that contains the function *main*. The optional (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.

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 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.

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000b
2	External 1	0x0013
3	Timer 1	0x001b
4	Serial	0x0023
5	Timer 2 (8052)	0x002b
...		...
n		0x0003 + 8*n

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

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

3.8.3 HC08 Interrupt Service Routines

Since the number of interrupts available is chip specific and the interrupt vector table always ends at the last byte of memory, the interrupt numbers corresponds to the interrupt vectors in reverse order of address. For example, interrupt 1 will use the interrupt vector at 0xfffc, interrupt 2 will use the interrupt vector at 0xfffa, and so on. However, interrupt 0 (the reset vector at 0xfffe) is not redefinable in this way; instead see section 4.1.4 for details on customizing startup.

3.8.4 Z80, Z180 and eZ80 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:

```
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.

Type	Syntax	Behaviour
Interruptible interrupt handler	void f(void) __interrupt	Interrupt handler can be interrupted by further interrupts
Non-interruptible interrupt handler	void f(void) __critical __interrupt(0)	Interrupt handler can be interrupted by NMI only
NMI handler	void f(void) __critical __interrupt	Interrupt handler can be interrupted by NMI only

3.8.5 Rabbit 2000, 3000, 3000A, 4000, 5000, and 6000 Interrupt Service Routines

SDCC generates an internal interrupt vector table, but does not attempt to generate an external interrupt vector table.

Type	Syntax	Behaviour
Interruptible interrupt handler	void f(void) __interrupt	Interrupt handler can be interrupted by further interrupts of same priority
Non-interruptible interrupt handler	void f(void) __critical __interrupt(0)	Interrupt handler can be interrupted by interrupts of higher priority only. amd will be added to internal interrupt table as first entry.

3.8.6 SM83 and TLCS-90 Interrupt Service Routines

SDCC does not attempt to generate an interrupt vector table.

Type	Syntax	Behaviour
Interruptible interrupt handler	void f(void) __interrupt	Interrupt handler can be interrupted by further interrupts
Non-interruptible interrupt handler	void f(void) __critical __interrupt(0)	Interrupt handler cannot be interrupted by further interrupts

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 (for architectures where there is no efficient way to do so (sm83, tlcs90, stm8), interrupts will be unconditionally enabled instead). Nesting critical functions will need one additional byte on the stack for each call.

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

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

```
__critical{ i++; }
```

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

3.9.2 Enabling and Disabling Interrupts directly

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

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

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

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

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.

3.9.3 Semaphore locking (mcs51/ds390)

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

SDCC generates this instruction if the source follows this pattern:

```
volatile bit resource_is_free;

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

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

3.10 Functions using private register banks (mcs51/ds390)

Some architectures have support for quickly changing register sets. SDCC supports this feature with the __ *using* attribute (which tells the compiler to use a register bank other than the default bank zero). It should only be applied to *interrupt* functions (see footnote below). This will in most circumstances make the generated ISR code more efficient since it will not have to save registers on the stack.

The __ *using* attribute will have no effect on the generated code for a *non-interrupt* function (but may occasionally be useful anyway[4]).

(pending: Note, nowadays the __using *attribute has an effect on the generated code for a* non-interrupt *function.)*

[4] 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

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:

```
__asm
    ; This is a comment
    label:
        nop
__endasm;
```

Note: As of SDCC 4.2.9, assembler comments occurring in this type of inline assembler block are affected by macro expansion.

3.11.1.2 New __asm__ ("inline_assembler_code") Format

The __asm__ inline assembler code format was introduced in SDCC version 3.2.0. Its main advantage is that it is compatible with all standard compliant C preprocessors.

Example:

```
__asm__ ('; This is a comment\nlabel:\n\tnop');
```

Or for better readability:

```
__asm__ (
"; This is a comment\n"
"label:\n"
"    nop"
);
```

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:

```
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;**"[5] to the beginning and the end of the function body:

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

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

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

#if !USE_ASSEMBLY

void to_buffer( unsigned char c )
{
    if( head != (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)
```

[5] 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.

```
        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.

```
#define DELAY   \
    __asm       \
        nop     \
        nop     \
    __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:

```
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:

```
Note, this is an outdated example, recent versions of SDCC generate
the same code for simpleInterrupt() and nakedInterrupt()!

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

whereas nakedInterrupt looks like:

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

The related directive #pragma exclude allows a more fine grained control over pushing & popping the registers.

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

3.11.4 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 nnnnn is a number less than 100 (which implies a limit of utmost 100 inline assembler labels *per function*).[6]

```
__asm
    mov     b,#10
00001$:
    djnz    b,00001$
__endasm ;
```

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

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

[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.

```
    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.

Function	**Description**
_mulint.c	16 bit multiplication
_divsint.c	signed 16 bit division (calls _divuint)
_divuint.c	unsigned 16 bit division
_modsint.c	signed 16 bit modulus (calls _moduint)
_moduint.c	unsigned 16 bit modulus
_mullong.c	32 bit multiplication
_divslong.c	signed 32 division (calls _divulong)
_divulong.c	unsigned 32 division
_modslong.c	signed 32 bit modulus (calls _modulong)
_modulong.c	unsigned 32 bit modulus

In the mcs51, ds390, hc08, s08, pdk13, pdk14, pdk15, pic14 and pic16 backends they are by default compiled as *non-reentrant*; when targeting on 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:

Function	Description
_fsadd.c	add floating point numbers
_fssub.c	subtract floating point numbers
_fsdiv.c	divide floating point numbers
_fsmul.c	multiply floating point numbers
_fs2uchar.c	convert floating point to unsigned char
_fs2schar.c	convert floating point to signed char
_fs2uint.c	convert floating point to unsigned int
_fs2sint.c	convert floating point to signed int
_fs2ulong.c	convert floating point to unsigned long
_fs2slong.c	convert floating point to signed long
_uchar2fs.c	convert unsigned char to floating point
_schar2fs.c	convert signed char to floating point
_uint2fs.c	convert unsigned int to floating point
_sint2fs.c	convert signed int to floating point
_ulong2fs.c	convert unsigned long to floating point
_slong2fs.c	convert signed long to floating point
_ulonglong2fs.c	convert unsigned long long to floating point
_slonglong2fs.c	convert singed long long to floating point

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*>*

3.14.1 Compiler support routines (_gptrget, _mulint etc.)

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

3.14.2.1 <stdio.h>

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 f.e. in sdcc/device/lib. For the mcs51 this minimalistic polling `putchar()` routine might be a start:

```
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 4.9.9.

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.

mcs51	printf	printf USE_FLOATS=1	printf_small	printf_fast	printf_fast_f	printf_tiny
filename	printf_large.c	printf_large.c	printfl.c	printf_fast.c	printf_fast_f.c	printf_tiny.c
"Hello World" size small / large	1.7k / 2.4k	4.3k / 5.6k	1.2k / 1.8k	1.3k / 1.3k	1.9k / 1.9k	0.44k / 0.44k
code size small / large	1.4k / 2.0k	2.8k / 3.7k	0.45k / 0.47k (+ _ltoa)	1.2k / 1.2k	1.6k / 1.6k	0.26k / 0.26k
formats	cdi*o*psux	cdfi*o*psux	cd*o*sx	cdsux	cdfsux	cdsux
long (32 bit) support	x	x	x	x	x	-
byte arguments on stack	b	b	-	-	-	-
float format	-	%f	-	-	%f[8]	-
float formats %e %g	-	-	-	-	-	-
field width	x	x	-	x	x	-

[8] Range limited to +/- 4294967040, precision limited to 8 digits past decimal

mcs51	printf	printf USE_FLOATS=1	printf_small	printf_fast	printf_fast_f	printf_tiny
filename	printf_large.c	printf_large.c	printfl.c	printf_fast.c	printf_fast_f.c	printf_tiny.c
"Hello World" size small / large	1.7k / 2.4k	4.3k / 5.6k	1.2k / 1.8k	1.3k / 1.3k	1.9k / 1.9k	0.44k / 0.44k
code size small / large	1.4k / 2.0k	2.8k / 3.7k	0.45k / 0.47k (+ _ltoa)	1.2k / 1.2k	1.6k / 1.6k	0.26k / 0.26k
string speed[9], small / large	1.52 / 2.59 ms	1.53 / 2.62 ms	0.92 / 0.93 ms	0.45 / 0.45 ms	0.46 / 0.46 ms	0.45 / 0.45 ms
int speed[10], small / large	3.01 / 3.61 ms	3.01 / 3.61 ms	3.51 / 18.13 ms	0.22 / 0.22 ms	0.23 / 0.23 ms	0.25 / 0.25 ms[11]
long speed[12], small / large	5.37 / 6.31 ms	5.37 / 6.31 ms	8.71 / 40.65 ms	0.40 / 0.40 ms	0.40 / 0.40 ms	-
float speed[13], small / large	-	7.49 / 22.47 ms	-	-	1.04 / 1.04 ms	-

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 (I^2C, CAN, Ethernet, Profibus, Modbus, USB, SPI, JTAG ...), Media (IDE, Memory cards, eeprom, flash...), En-/Decryption, Remote debugging, Realtime kernel, Keyboard, LCD, RTC, FPGA, PID then the sdcc-user mailing list http://sourceforge.net/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.

[9] Execution time of printf("%s%c%s%c%c%c", "Hello", ' ', "World", '!', '\r', '\n'); 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()
[13] Execution time of printf("%.3f", -12345.678); standard 8051 @ 22.1184 MHz, empty putchar()

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 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 *banked4.1.3* 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 4.1), 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 *tinibios* in the ds390 libraries will do that for you). If you don't use *tinibios*, the boot loader or similar code must ensure that the processor is in 24 bit contiguous addressing mode before calling the SDCC startup code.

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

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

3.15.3 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.

3.15.4 MOS6502 Memory Models

SDCC implements two memory models for the MOS6502: *small* and *large* (default). 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 small model all data objects are placed by default in Page Zero. This is normally only useful in embedded systems.

In the large memory model, data is placed by default in 16-bit addressable memory. Critical data can still be placed in Page Zero using the __ZP or __near storage modifier. Code generated for the large model is slightly bigger and slower than code generated for the small 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 32).
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 36), 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 *I*nterrupt *S*ervice *R*outines. The directive should be placed immediately before the ISR function definition and it affects ALL ISR functions following it. To enable the normal register saving for ISR functions use #pragma exclude none. See also the related keyword __*naked*.
- **less_pedantic** - the compiler will not warn you anymore for obvious mistakes, you're on your own now ;-(. See also the command line option --less-pedantic on page 36.
  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>.
- **nogcse** - will stop global common subexpression elimination.
- **noinduction** - will stop loop induction optimizations.
- **noinvariant** - will not do loop invariant optimizations. For more details see Loop Invariants in section8.1.4.
- **noiv** - Do not generate interrupt vector table entries for all ISR functions defined after the pragma. This is useful in cases where the interrupt vector table must be defined manually, or when there is a secondary, manually defined interrupt vector table (e.g. for the autovector feature of the Cypress EZ-USB FX2). More elegantly this can be achieved by 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.
- **max_allocs_per_node** <value>- Use this value as limit for some optimizations. See option --max-allocs-per-node.
- **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.
- **std_c11** - Follow the C11 standard and disable SDCC features that conflict with the standard.
- **std_sdcc11** - Generally follow the C11 standard, but allow SDCC features that conflict with the standard.
- **std_c23**- Follow the C23 standard and disable SDCC features that conflict with the standard (for backwards compatibility, **std_c2x** still exists as an alias).
- **std_sdcc23** - Generally follow the C23 standard, but allow SDCC features that conflict with the standard.
- **std_c2y**- Follow the C2y standard and disable SDCC features that conflict with the standard.
- **std_sdcc2y** - Generally follow the C2y standard, but allow 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:

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

```
#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 it's own __asm __endasm block. Below is an example for multiline assembly defines.

```
#define Nop __asm \
nop \
__endasm

#define ThreeNops Nop; \
Nop; \
Nop

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

- **sdcc_hash** (+ | -) - *Until SDCC 4.2.8:* Allow "naked" hash in macro definition, for example: `#define DIR_LO(x) #(x & 0xff)`
  Default is off. Below is an example on how to use this pragma.

```
#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.

```
#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:

#define	**Description**
__SDCC	Always defined. Version number string (e.g. SDCC_3_2_0 for sdcc 3.2.0).
SDCC	OBSOLETE. WILL BE REMOVED IN THE FUTURE. CURRENTLY Only defined for the mcs51 backend (and only if –std-cXX 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!
__SDCC_mcs51 or __SDCC_ds390 or __SDCC_z80, etc.	depending on the model used (e.g.: -mds390). Older versions used SDCC_mcs51, etc instead.
__SDCC_STACK_AUTO	when *--stack-auto* option is used
__SDCC_MODEL_SMALL	when *--model-small* is used
__SDCC_MODEL_MEDIUM	when *--model-medium* is used
__SDCC_MODEL_LARGE	when *--model-large* is used
__SDCC_MODEL_HUGE	when *--model-huge* is used
__SDCC_USE_XSTACK	when *--xstack* option is used
__SDCC_STACK_TENBIT	when *-mds390* is used
__SDCC_MODEL_FLAT24	when *-mds390* is used
__SDCC_VERSION_MAJOR	Always defined. SDCC major version number. E.g. 3 for SDCC 3.5.0
__SDCC_VERSION_MINOR	Always defined. SDCC minor version number. E.g. 5 for SDCC 3.5.0
__SDCC_VERSION_PATCH	Always defined. SDCC patchlevel version number. E.g. 0 for SDCC 3.5.0
__SDCC_REVISION	Always defined. SDCC svn revision number. Older versions of sdcc used SDCC_REVISION instead.
SDCC_PARMS_IN_BANK1	when *--parms-in-bank1* is used
__SDCC_ALL_CALLEE_SAVES	when *--all-callee-saves* is used
__SDCC_FLOAT_REENT	when *--float-reent* is used
__SDCC_INT_LONG_REENT	when *--int-long-reent* is used
__SDCC_OPTIMIZE_SPEED	when *--opt-code-speed* is used
__SDCC_OPTIMIZE_SIZE	when *--opt-code-size* is used
__SDCCCALL	Default ABI version for calling convention

Chapter 4

Notes on supported Processors

4.1 MCS51 variants

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

4.1.1 pdata access by SFR

With the upcome of devices with internal xdata and flash memory devices using port P2 as dedicated I/O port is becoming more popular. Switching the high byte for __pdata access which was formerly done by port P2 is then achieved by a Special Function Register. In well-established MCS51 tradition the address of this *sfr* is where the chip designers decided to put it. Needless to say that they didn't agree on a common name either. So that the startup code can correctly initialize xdata variables, you should define an sfr with the name _XPAGE at the appropriate location if the default, port P2, is not used for this. Some examples are:

```
__sfr __at (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 (0xaf) _XPAGE; /* some Silicon Labs (Cygnal) chips a.k.a.  EMI0CN */
__sfr __at (0xaa) _XPAGE; /* some Silicon Labs (Cygnal) 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

8000-FFFF	bank1	bank2	bank3
0000-7FFF	common		

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. Therefor 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 3.3.5) 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 3.3.6). 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 .
```

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-size <n>* 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 -plosgff*[1] *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.

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, non-struct) 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.

Further such parameters (and all struct parameters, as well as the bit parameters from the 9th onwards) are either allocated on the stack (for reentrant routines or if --stack-auto is used) or in data/xdata memory (depending on the memory model).

The first 8 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 therefor 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.

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

4.1.5.3 Assembler Routine (non-reentrant)

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

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

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

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

The corresponding assembler function is:

```
        .globl _asm_func_PARM_2
        .globl _asm_func
        .area OSEG
_asm_func_PARM_2:
        .ds 1
        .area CSEG
_asm_func:
        mov    a,dpl
        add    a,_asm_func_PARM_2
        mov    dpl,a
        mov    dph,#0x00
        ret
```

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:

```
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:

```
.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, Z80N, Z180, Rabbit 2000, Rabbit 2000A, Rabbit 3000A, Rabbit 4000, Rabbit 5000, Rabbit 6000, SM83 (GameBoy), TLCS-90, eZ80, and R800 ports

SDCC can target the Z80, Z80N, Z180, eZ80, Rabbit 2000, Rabbit 2000A, Rabbit 3000A and LR35902, the Sharp SM83 (used .e.g in the Nintendo GameBoy) sm83.

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.

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 Rabbit ports

SDCC has six Rabbit-supporting ports: r2k for the Rabbit 2000, r2ka for the Rabbit 2000A, 2000B, 2000C and 3000, r3ka for the Rabbit 3000A, r4k for the Rabbit 4000, r5k for the Rabbit 5000, and r6k for the Rabbit 6000. Some Rabbits support two different memory types (Rabbit 4000 and later), two different memory access modes (Rabbit 5000 and later), four different instruction set modes (Rabbit 4000 and later). All SDCC ports assume 8-bit memories, 8-bit memory access mode. The code generated by the r2k, r2ka, r3ka ports uses 00/default instruction set mode (the code generated is also compatible with 10 and 01 instruction set mode,

but not 11/enhanced instruction set mode). The code generated by the r4k, r5k, r6k ports uses 10 instruction set mode, and is not compatible with other instruction set modes.

4.3.2.1 Port choice

Port vs. Device	R2K	R2KA	R2KB	R2KC	R3K	R3KA	R4K	R5K	R6K
r2k	Y	y	y	y	y	y	y	N	N
r2ka	n	Y	Y	Y	Y	y	y	N	N
r3ka	N	N	N	N	N	Y	y	N	N
r4k	N	N	N	N	N	N	Y	N	N
r5k	N	N	N	N	N	N	y	Y	y
r6k	N	N	N	N	N	N	N	N	Y

Legend:
Y - The code this port generates is the best for this CPU.
y - The code this port generates will work on this CPU.
N - The code this port generates will not work on this CPU.
n - The code this port generates will typically not work on this CPU.

There are multiple wait state bugs present in some of the the Rabbits. The difference between the r2k and r2ka port is additional wait state bug workarounds only. If all memory used has zero wait states, code from the r2ka backend can be safely run on the original Rabbit 2000.

The r2k and r2ka port assume that the whole stack has the same number of wait states (code from the r2k and r2ka ports can fail is the stack spans memories with a different amount of wait states).

The Rabbit 2000 has some wait state bugs that SDCC does not work around. These bugs result in the number of wait states used being one less than configured for some instructions. The workaround has to be supplied by the user, by configuring all memories that do use wait states to use on additional wait state.

For all Rabbit ports, SDCC assumes that all data memory is at least as fast (i.e. does not need more wait states) as all code memory. Code where this is not the case (e.g. code in fast Flash writing into slow battery-backed SRAM) will have to be written in assembler by hand.

4.3.2.2 Rabbit hardware bugs

Bug	R2K	R2KA	R2KB	R2KC	R3K	R3KA	R4K	R5K	R6K
ioi / ioe prefix bug	x								
ddcb / fdcb wait state bug	U								
conditional jump wait state bug	U								
ldir / lddr wait state bug	U								
mul wait state bug	w								
ldir / lddr split bug		S							
new ldir / lddr wait state bug	w	w	w	w	w				
16-bit mode alignment bug								x	
ioi / ioe bit bug								x	x
ret cc bug								x	x
rmw wait state bug	W	W	W	W	W	W	W	W	
16-bit vs. 8-bit wait state bug							W		
dma vs stack prot. bug							X		
16-bit dma vs. ldir / lddr / etc. bug							X		
cplx bug									x
dma match bug									X
ex jkhl, bcde alt reg bug									x
ldbyte bug									x
puma bug									x

Legend:
x s w u - bug present, worked around by SDCC.
X S W U - bug present, not worked around by SDCC.
w W u U - wait state bug, only relevant when the Rabbit is configured to have wait states for some memory.
s S - instruction / data split bug, only relevant when instruction / data split is enabled, which is currently not supported by SDCC.
u U - bug can be worked around by the user by configuring memory that needs wait states to use one additional wait state.

The 16-bit mode alignment bug only affects 16-bit mode, which is not currently supported by SDCC.

The rmw wait state bug is only relevant when executing fast code writing to slow memory, such as code in fast RAM writing to slow battery-backed SRAM.

The 16-bit vs. 8-bit wait state bug is only relevant when the target system uses both 16-bit and 8-bit memories for code, and the 8-bit memory requires wait states.

The 16-bit dma vs. ldir / lddr / etc. bug is only relevant when using DMA and the target system uses 16-bit memory for code. Since SDCC uses the block move instructions like ldir, lddr, etc. without applying a workaround, this means that code execution and DMA may then not happen at the same time. A possible user workaround is to not use DMA when using 16-bit memory for code. Among all Rabbit 4000-based RCM and BL modules, only the RCM4000 and the RCM4010 use 16-bit memories.

The dma match bug is only relevant when a DMA termination mask register is set to a nonzero value.

4.3.2.3 Rabbit interrupt handlers

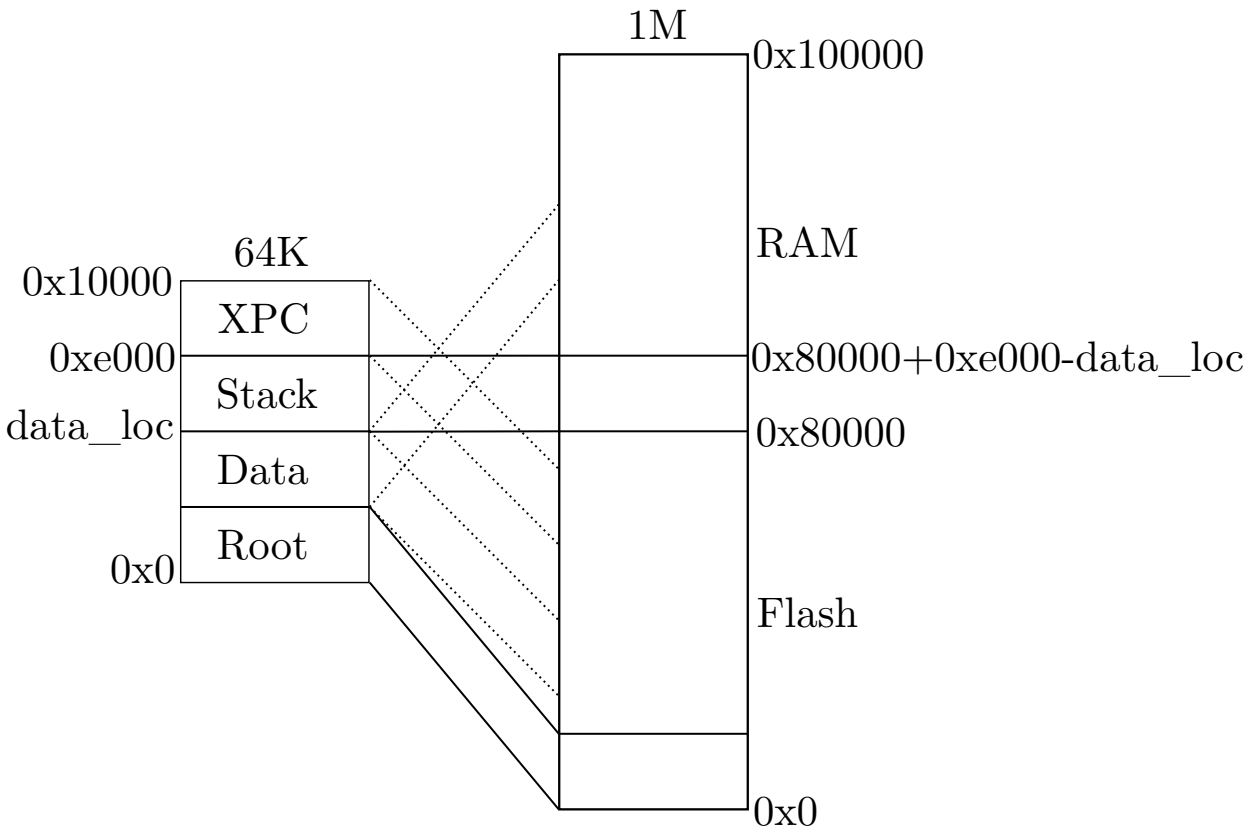
External interrupts need to be handled via a custom crt0. For internal interrupts, there is compiler support that creates the internal interrupt table from the declared interrupt handlers in the file containing the main function.

```
void handler(void) __interrupt(0) __critical
{
... // A hardware interrupt handler that runs at the priority of the interrupt (i.e. it can only be interrupted by interrupts of higher priority)
... // Automatically inserted as internal interrupt 0 (periodic interrupt) in internal interrupt vector table if declared in the file containing the main function.
}
void handler(void) __interrupt(2)
{
... // A software interrupt handler (interrupt number 2 is rst 0x10)
... // Automatically inserted as internal interrupt 2 (rst 0x10) in internal interrupt vector table if declared in the file containing the main function.
}
void handler(void) __interrupt
{
... // A hardware interrupt handler suitable for use as handler for an external interrupt
... // Can be used jumped to in a custom crt0 containing the external interrupt vector table
}
```

The internal interrupts 2 to 7 are software interrupts (for the rst and syscall instructions). Hardware interrupt handlers written in C either run at the priority of the interrupt (with __critical) or at the priority of the code executing before the interrupt (without __critical). More complex prioritites (changing priority while the handler is executing) require interrupt handlers written in assembler.

4.3.2.4 Rabbit memory map and memory models

Most Rabbit instructions operate on a 64K logical address space. The Rabbits support a 1M (up to Rabbit 3000A) or 512K to 16M (from Rabbit 4000, configured via MECR register, default is 1M) physical address space, which the logical address space is mapped into via a memory management unit (MMU).



SDCC assumes a 1M physical address space, and that memory for the program and constant data (typically flash) is located from 0x0 in the physical address space, up to at most 0x7ffff. It assumes that memory for non-constant data (typically RAM) is mapped from 0x80000. This is the typical hardware configuration for Rabbit-based systems, including the Rabbit Control Modules (RCM).

The logical address space is divided into four segments, which the MMU maps into physical address space. SDCC uses the Rabbit "Root" segment for the SDCC code, gsinit, etc segments (startup code, const data not in __far, program code in small memory model). It uses the "Stack" segment for the SDCC data segment, stack and heap (non-const data no in __far). The SDCC xdata segment (non-const data in __far) is placed in physical memory above 0x80000 (exact location can be given by --xram-loc), the SDCC xconst segment (const data in __far) is placed in physical memory above 0x0 (directly after code, gsinit, etc). The Rabbit "Data" segment is not used directly by SDCC. Its intended use is for user-defined named address spaces. The Rabbit "XPC" segment is used for the SDCC xcode segment (program code in medium memory model).

The size of the Rabbit "Root", "Data" and "Stack" segments can be adjusted according to user needs by setting up the SEGSIZE register in __sdcc_external_startup and the --data-loc command line option (the value for the option shall be 0xS000, where S matches the upper nibble of the SEGSIZE register).

Pointers to objects that are not __far-qualified objects are represented by their logical address as 16-bit value. Pointers to objects that are __far-qualified are represented by their physical address as 24-bit value.

The default crt0 that ships with SDCC uses a 32KB "Root" segment followed by a 8KB "Data" segment, a 16KB "Stack" segment and the 8KB "XPC" segment.

4.3.3 Z80, Z180, Z80N and R800 calling conventions

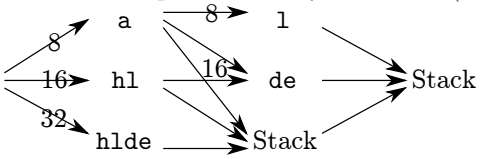
The current default is the SDCC calling convention, version 1. Using the command-line option –sdcccall 0, the default can be changed to version 0. There are four other calling conventions supported, which can be specified using the keywords __smallc, __z88dk_fastcall, __z88dk_callee and __dynamicc. They 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 of __sdcccall(0). For __z88dk_callee, the stack is not adjusted for stack parameters the parameters after the call (thus the callee has to do this instead). __z88dk_callee can be combined with __smallc, __sdcccall(0) or __sdcccall(1).

4.3.3.1 Z80 SDCC calling convention, version 1

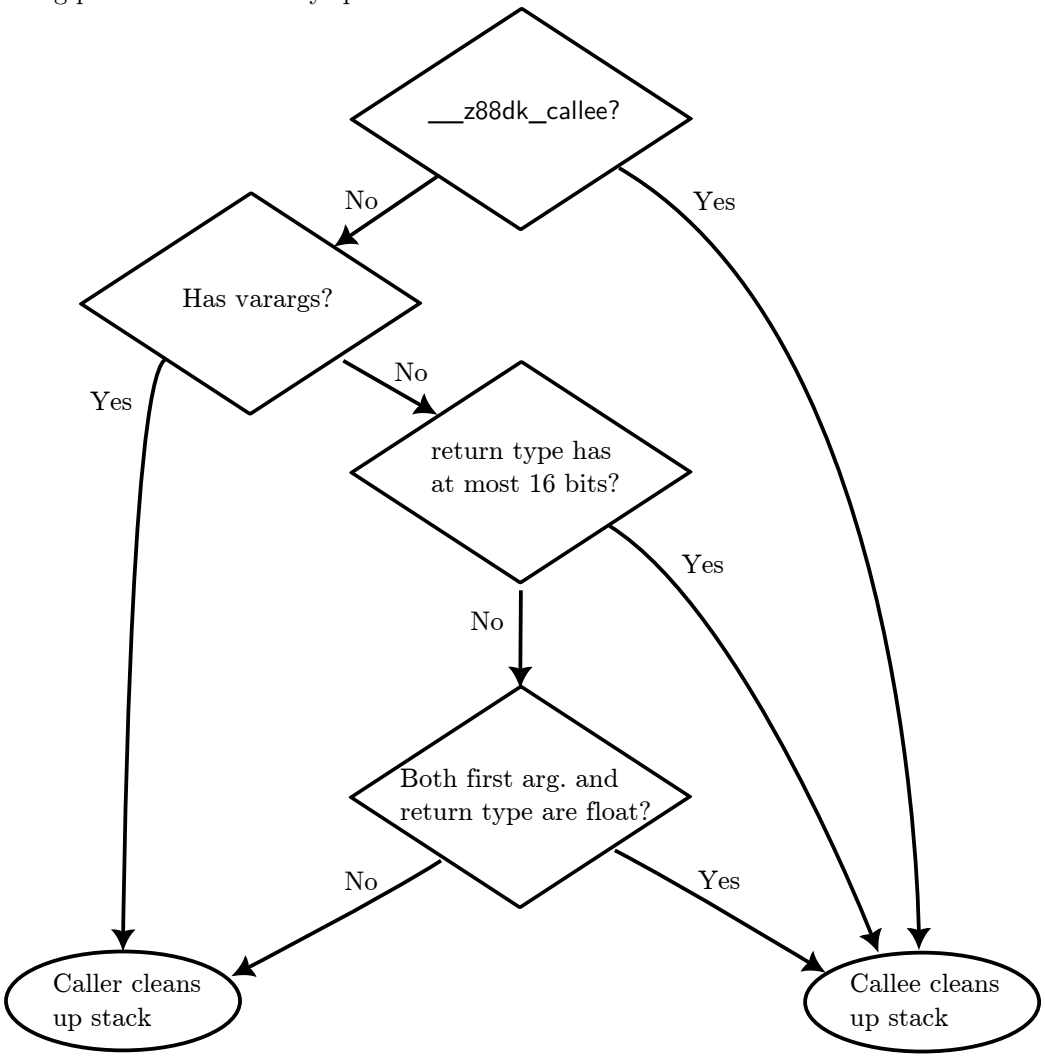
This calling convention can be chosen per function via __sdcccall(1). 8-bit return values are passed in a, 16-bit values in de, 24-bit values in lde, 32-bit values in hlde. Larger return values (as well as struct and union

independent of their size) are passed in memory in a location specified by the caller through a hidden pointer argument.

For functions that have variable arguments: All parameters are passed on the stack. The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead).



For Functions that do not have variable arguments: the first parameter is passed in a if it has 8 bits. If it has 16 bits it is passed in hl. If it has 32 bits, it is passed in hlde. If the first parameter is in a, and the second has 8 bits, it is passed in l; if the first is passed in a or hl, and the second has 16 bits, it is passed in de; all other parameters are passed on the stack, right-to-left. Independent of their size, struct / union parameters and all following parameters are always passed on the stack.



If __z88dk_callee is not used, after the call, the stack parameters are cleaned up by the caller, with the following exceptions: functions that do not have variable arguments and return void or a type of at most 16 bits, or have both a first parameter of type float and a return value of type float.

4.3.3.2 Z80 SDCC calling convention, version 0

This calling convention can be chosen per function via __sdcccall(0). All parameters are passed on the stack, right-to-left. 8-bit return values are passed in l, 16-bit values in hl, 24-bit values in ehl, 32-bit values in dehl. Except for the SM83, where 8-bit values are passed in e, 16-bit values in de, 32-bit values in hlde. Larger return values (as well as struct and union independent of their size) are passed in a memory in a location specified by the caller through a hidden pointer argument. Unless __z88dk_callee is used, all stack parameters are cleaned up by the caller.

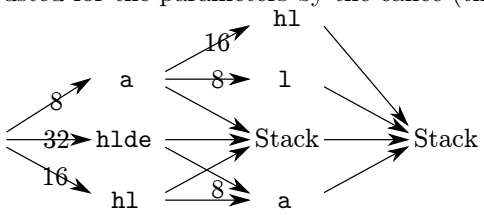
4.3.4 Rabbit 2000, Rabbit 2000A, Rabbit 3000A, Rabbit 4000, Rabbit 5000, Rabbit 6000, eZ80 and TLCS-90 calling conventions

The current default is the Rabbit calling convention described here, version 1. Using the command-line option –sdcccall 8, the default can be changed to version 0 of the Z80 calling convention, described above. There are four other calling conventions supported, which can be specified using the keywords __smallc, __z88dk_fastcall, __z88dk_callee and __dynamicc. 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, the stack is not adjusted for stack parameters the parameters after the call (thus the callee has to do this instead). __z88dk_callee can be combined with __smallc, __sdcccall(0) or __sdcccall(1).

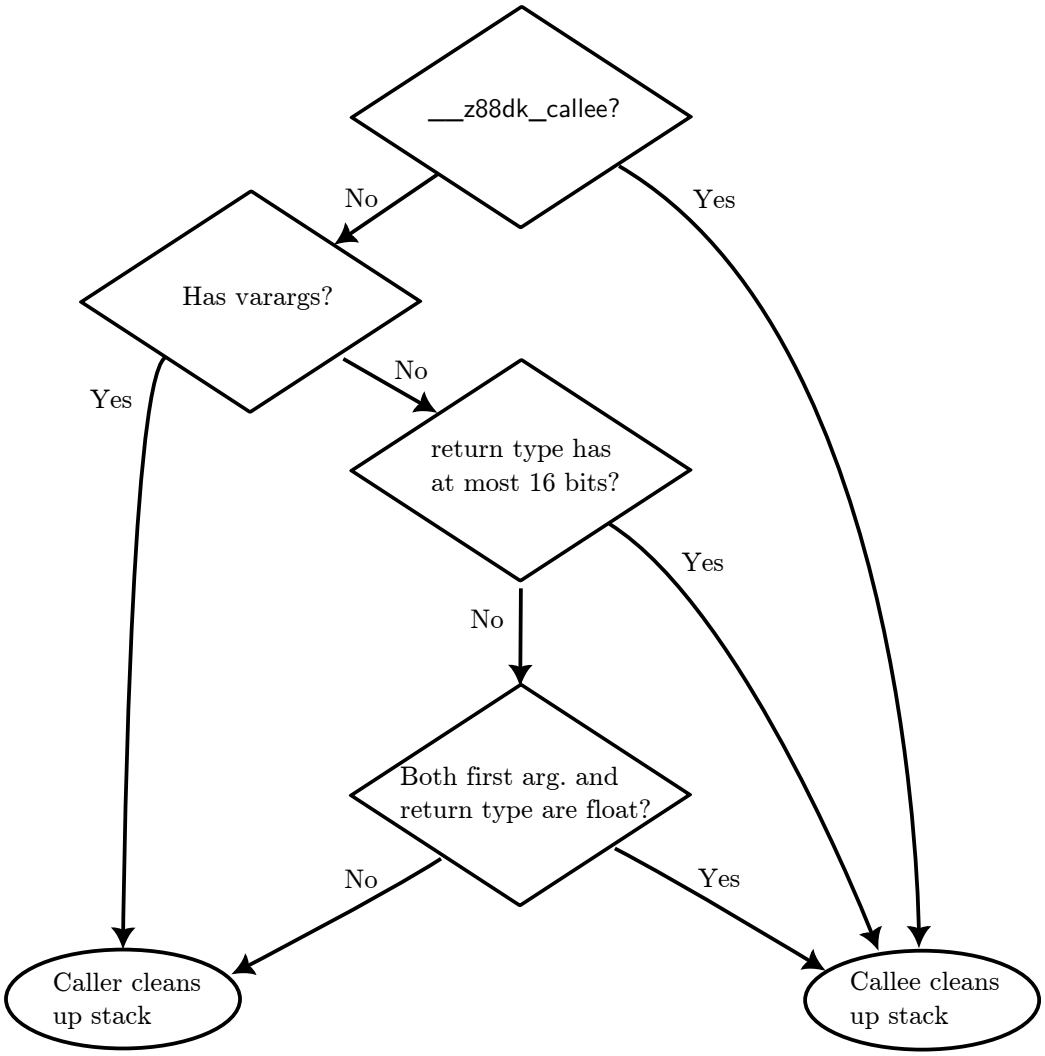
4.3.4.1 Rabbit SDCC calling convention, version 1

This calling convention can be chosen per function via __sdcccall(1). 8-bit return values are passed in a, 16-bit values in hl, 24-bit values in lde, 32-bit values in hlde. Larger return values (as well as struct and union independent of their size) are passed in memory in a location specified by the caller through a hidden pointer argument.

For functions that have variable arguments: All parameters are passed on the stack. The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead).



For Functions that do not have variable arguments: the first parameter is passed in a if it has 8 bits. If it has 16 bits it is passed in hl. If it has 32 bits, it is passed in hlde. If the first parameter is in a, and the second has 8 bits, it is passed in l; if the first is in hl or hlde, and the second has 8 bits, it is passed in a; if the first is in a, and the second has 16 bits, it is passed in hl; all other parameters are passed on the stack, right-to-left. Independent of their size, struct / union parameters and all following parameters are always passed on the stack.



If __z88dk_callee is not used, after the call, the stack parameters are cleaned up by the caller, with the following exceptions: functions that do not have variable arguments and return void or a type of at most 16 bits, or have both a first parameter of type float and a return value of type float.

4.3.5 SM83 calling conventions

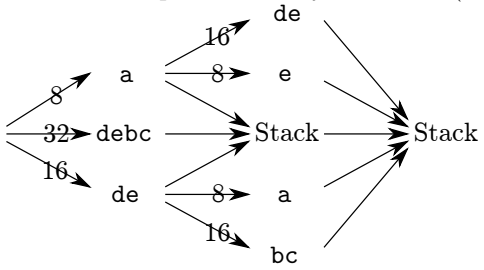
The current default is the SDCC calling convention, version 1. Using the command-line option –sdcccall 0, the default can be changed to version 0. There is another calling conventions supported, which can be specified using the keyword __dynamicc.

4.3.5.1 SM83 SDCC calling convention, version 1

This calling convention can be chosen per function via __sdcccall(1).

8-bit return values are passed in a, 16-bit values in bc, 32-bit values in debc. Larger return values (as well as struct and union independent of their size) are passed in memory in a location specified by the caller through a hidden pointer argument.

For functions that have variable arguments: All parameters are passed on the stack. The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead).



For Functions that do not have variable arguments: the first parameter is passed in a if it has 8 bits. If it has 16 bits it is passed in de. If it has 32 bits, it is passed in debc. If the first parameter is in a, and the second has 8 bits, it is passed in e; if the first is in bc or debc, and the second has 8 bits, it is passed in a; if the first is passed in a, and the second has 16 bits, it is passed in bc; if the first is passed in de, and the second has 16 bits, it is passed in bc; all other parameters are passed on the stack, right-to-left. Independent of their size, struct / union parameters and all following parameters are always passed on the stack. The stack is adjusted by the callee (thus explicitly specifying __z88dk_callee does not make a difference), unless the functionhas variable arguments.

4.3.5.2 SM83 SDCC calling convention, version 0

This calling convention can be chosen per function via __sdcccall(0). 8-bit return values are passed in e, 16-bit values in de, 32-bit values in hlde. Larger return values (as well as struct and union independent of their size) are passed in memory in a location specified by the caller through a hidden pointer argument. All parameters are passed on the stack. The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead), unless __z88dk_callee is specified. __sdcccall(0) can be combined with __z88dk_callee.

4.3.6 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). 8-bit return values are passed in a, 16-bit values in de, 32-bit values in hlde. Larger return values (as well as struct and union independent of their size) are passed in memory in a location specified by the caller through a hidden pointer argument. 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 by SDCC (neither on the caller nor on the callee side).

4.3.7 Dynamic C calling convention

Functions declared __dynamicc are called using the Dynamic C calling convention. This way assembler routines originally written for Dynamic C or code generated by Dynamic C can be called from SDCC. Currently variable arguments are not yet supported (neither on the caller nor on the callee side).

All arguments are passed on the stack. The first parameter is also passed in hl if it has 8 bits (sign-extended, but callees in Dynamic C don't rely on that) or 16 bits, in bcde if it has arithmetic type and 32 bits. struct and union, and pointers to __far, are passed on the stack only. Arguments of other types are not supported by the convention. Currently, variable arguments are not yet supported by SDCC (neither on the caller nor on the callee side).

8-bit (sign-extended, but callers in Dynamic C don't rely on that) and 16-bit return values are passed in hl, 32-bit return values are passed in bcde (arithmetic types) or px (pointers to __far). struct and union, independent of their size, are returned via the stack. Return values of other types are not supported by the convention, nor is returning struct/union from functions with variable arguments.

After the call, all stack parameters and the space for struct/union return values are cleaned up by the caller. All registers are to be saved/restored by the caller as needed.

Pointers to __far are a feature introduced in Dynamic C 10, which supports Rabbit 4000/5000/6000 only; no Dynamic C version targeting Z180 or Rabbit 2000/3000 supports them.

Support for the Dynamic C calling convention is slightly incomplete in SDCC vs. Dynamic C 10: pointers to __far are not supported as return values. Also, 8-bit parameters are not sign-extended by the caller (i.e. the upper byte, which is not used by Dynamic C callees, has an unspecified value). When passing or returning struct/union or pointers to those to/from a function actually written for Dynamic C, be aware these should only have members of types supported by both SDCC and Dynamic C, and for which both SDCC and Dynamic C use the same layout. This restricts the members to the following types: char, signed char, unsigned char, (signed) int, unsigned int, (signed) long, unsigned long, float, pointers (excluding pointers to __far), struct/union types to which this restriction applies recursively.

4.3.8 Complex instructions

The Z80 and some derivatives support complex instructions, such as ldir, cpir, ... SDCC uses these instructions for some functions in the standard library, and sometimes directly generates code for such functions. ldir and ldi are also used by code generation. Thus, e.g. copying one array into another is more efficient when using

memcpy() than by using a a user-written loop assigning the elements (but not more efficient than assigning a struct containing such an array).

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

4.3.9 Unsafe reads

Usually, Z80-based systems (except for the SM83 and TLCS-90) have separate I/O and memory spaces, and any normal memory location can be read without side-effects. For such systems, the option –allow-unsafe-reads can be used to enable some extra optimizations that rely on this.

4.3.10 Z80 banked calls

Banked calls are supported via __banked. Banked calls are done via a trampoline (__sdcc_bcall if --legacy-banking is specified, __sdcc_bcall_abc for z88dk_fastcall, __sdcc_bcall_ehl for other calls). Default trampolines are provided in the library. The default trampolines calls user supplied helper functions set_bank and get_bank that set the current bank to the value in register a, or return the current bank in register a.

For banked functions, the calling convention is slightly different: the stack is always cleared up by the caller. Unless __z88dk_fastcall is used, all parameters are passed on the stack.

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.

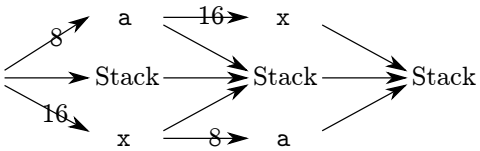
4.5 The STM8 port

4.5.1 Calling conventions

By default, the SDCC calling convention, version 1 is used. Using the option –sdcccall 0, the default can be changed to version 0.

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, and all struct / union).

4.5.1.1 SDCC calling convention, version 1



For functions that have variable arguments, all parameters are passed on the stack. For other functions, if the first parameter has 8 or 16 bits, it is passed in a or x. If the first parameter is passed in a, and the second has 16 bits, the second is passed in x. If the first parameter is passed in x, and the second has 8 bits, the second is passed in a. All other parameters are passed on the stack. Independent of their size, struct / union parameters and all following parameters are always passed on the stack. If __z88dk_callee is specified, the stack is always adjusted by the callee. Otherwise, for the large memory model, the stack is always adjusted by the caller. For the medium memory model the stack is adjusted by the caller, with the following exceptions: functions that do not have variable arguments and return void or a type of at most 16 bits, or have both a first parameter of type float and a return value of type float.

4.5.1.2 SDCC calling convention, version 0

This calling convention can be chosen per function via __sdcccall(0) (e.g. for compatibility with functions written in assembler for use with older versions of SDCC). All parameters are passed on the stack. The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead), unless __z88dk_callee is specified.

4.5.1.3 Raisonance calling convention

For compatibility with the Raisonance STM8 compiler, the __raisonance calling convention is supported. If the first parameter is 8 or 16 bits, it is passed in a or x. If the first parameter is 8 bits, and the second 16 bits, the second is passed in x. If the first parameter is 16 bits, and the second is 8 bits, the second is passed in a. All other parameters are passed on the stack. If the return value is 8 bits, it is passed in a. If it is 16 bits, it is passed in x. Raisonance passes larger return values in pseudoregisters, which is not supported by SDCC.

4.5.1.4 IAR calling convention

For compatibility with the IAR STM8 compiler, the __iar calling convention is supported. The first 8-bit parameter is passed in a, the first 16-bit parameter in x, the second 16-bit parameter in y. Further parameters of up to 32 bits are passed in pseudoregisters, which is not supported by SDCC. All other parameters are passed on the stack. If the return value is 8 bits, it is passed in a. If it is 16 bits, it is passed in x. IAR passes larger return values in pseudoregisters, which is not supported by SDCC.

4.5.1.5 Cosmic calling convention

For compatibility with the Cosmic STM8 compiler, the __cosmic calling convention is supported. If the first parameter is 8 or 16 bits, it is passed in a or x. If the return value is 8 bits, it is passed in a. If it is 16 bits, it is passed in x. Cosmic passes larger return values in pseudoregisters, which is not supported by SDCC. Even for the medium memory model, __cosmic functions use a 24-bit return address in their stack frame, and are called using callf.

4.6 The f8 port

4.6.1 Calling convention

If the the function does not have variable arguments, and the first argument is not a struct or union and has 8 or 16 bits, it is passed in xl (8 bits) or y (16 bits). All other arguments are passed on the stack right-to-left. Return values are in xl (8 bit), x (16 bit), xxl (24 bit), yx (32 bit) or use a hidden extra pointer parameter pointing to the location (anything wider than 32 bit, and all struct / union). The stack is not adjusted for the parameters by the callee (thus the caller has to do this instead).

4.7 The MOS6502 and WDC65C02 ports

The mos6502 port can target the original MOS Technology NMOS 6502 , and the CMOS Rockwell/WDC 65C02 with enhanched instruction set.

4.7.1 Startup Code

On the MOS6502 the startup code is inserted by linking with crt0.rel which is generated from sdcc/device/lib/mos6502/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.

4.7.2 Hooking up interrupts

ISR can be written in C using SDCC. However, to hook interrupts is necessary to customize crt0.asm and have the interrupt vectors point to the C functions.

4.7.3 Reentrancy

Due to the very limited stack space, the MOS6502 port by default generates non-reentrant code. Re-entrant functions should be declared using the __reentrant keyword. Alternatively the entire program can be compiled using –stack-auto. On the MOS6502 re-entrant code is, in general, much less efficient.

4.7.4 Code and Data placement

The compiler can pass segment location to the linker. On the MOS6502 the following switches are supported:
–code-loc: start address of the code segment
–data-loc: start address of the zero page
–xram-loc: start address of the data segment

4.8 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
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, 16LF1824T39A
16LF1825, 16LF1826, 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...).

4.8.1 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'.

4.8.2 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.

4.8.3 Interrupt Code

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

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

4.8.4 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:

```
#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:

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

4.8.5 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 reqirement 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 **--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:

```
.c.o:
        sdcc -V --use-non-free -mpic14 -p16f877 -c $<

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

Here is a `Makefile` using MPLAB:

```
.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
```

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

4.8.6 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):

--debug-xtra emit debug info in assembly output

--no-pcode-opt disable (slightly faulty) optimization on pCode

--stack-loc sets the lowest address of the argument passing stack (defaults to a suitably large shared databank to reduce BANKSEL overhead)

--stack-size sets the size if the argument passing stack (default: 16, minimum: 4)

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

--no-extended-instructions forbid use of the extended instruction set (e.g., ADDFSR)

4.8.7 Environment Variables

The PIC14 port recognizes the following environment variables:

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.

4.8.8 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 `device/lib`) should also compile with the PIC14 port.

4.8.8.1 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. `libme.lib` and `libsdcce.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`!

4.8.8.2 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.

4.8.8.3 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>`.

4.8.8.4 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)!

4.8.8.5 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.

4.8.9 Known Bugs

4.8.9.1 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...

4.8.9.2 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...

4.9 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
18F25J10 18F25J11 18F25J50 18F25K20 18F25K22 18F25K50 18F25K80
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 18F66J16 18F66J50 18F66J55 18F66J60 18F66J65

18F66J90 18F66J93 18F66J94 18F66J99 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 18F86J16 18F86J50 18F86J55 18F86J60 18F86J65
18F86J72 18F86J90 18F86J93 18F86J94 18F86J99 18F86K22 18F86K90
18F87J10 18F87J11 18F87J50 18F87J60 18F87J72 18F87J90 18F87J93 18F87J94 18F87K22 18F87K90
18F95J94 18F96J60 18F96J65 18F96J94 18F96J99
18F97J60 18F97J94
18F242 18F248 18F252 18F258
18F442 18F448 18F452 18F458
18F1220 18F1230
18F1320 18F1330
18F2220 18F2221
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 18F4331
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
18LF14K22 18LF14K50
18LF23K22 18LF24J10 18LF24J11 18LF24J50 18LF24K22 18LF24K50
18LF25J10 18LF25J11 18LF25J50 18LF25K22 18LF25K50 18LF25K80
18LF26J11 18LF26J13 18LF26J50 18LF26J53 18LF26K22 18LF26K80
18LF27J13 18LF27J53
18LF43K22
18LF44J10 18LF44J11 18LF44J50 18LF44K22
18LF45J10 18LF45J11 18LF45J50 18LF45K22 18LF45K50 18LF45K80
18LF46J11 18LF46J13 18LF46J50 18LF46J53 18LF46K22 18LF46K80
18LF47J13 18LF47J53
18LF65K80
18LF66K80
18LF242 18LF248 18LF252 18LF258
18LF442 18LF448 18LF452 18LF458
18LF1220 18LF1230
18LF1320 18LF1330
18LF2220 18LF2221
18LF2320 18LF2321 18LF2331
18LF2410 18LF2420 18LF2423 18LF2431 18LF2439 18LF2450 18LF2455 18LF2458 18LF2480
18LF2510 18LF2515 18LF2520 18LF2523 18LF2525 18LF2539 18LF2550 18LF2553 18LF2580 18LF2585
18LF2610 18LF2620 18LF2680 18LF2682 18LF2685
18LF4220 18LF4221
18LF4320 18LF4321 18LF4331
18LF4410 18LF4420 18LF4423 18LF4431 18LF4439 18LF4450 18LF4455 18LF4458 18LF4480

18LF4510 18LF4515 18LF4520 18LF4523 18LF4525 18LF4539 18LF4550 18LF4553 18LF4580 18LF4585
18LF4610 18LF4620 18LF4680 18LF4682 18LF4685
18LF6310 18LF6390 18LF6393
18LF6410 18LF6490 18LF6493
18LF6520 18LF6525 18LF6527 18LF6585
18LF6620 18LF6621 18LF6622 18LF6627 18LF6628 18LF6680
18LF6720 18LF6722 18LF6723
18LF8310 18LF8390 18LF8393
18LF8410 18LF8490 18LF8493
18LF8520 18LF8525 18LF8527 18LF8585
18LF8620 18LF8621 18LF8622 18LF8627 18LF8628 18LF8680
18LF8720 18LF8722 18LF8723

An up-to-date list of supported devices is also available via '`sdcc -mpic16 -plist`'.

4.9.1 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).

4.9.2 Port Specific Options

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

4.9.2.1 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!

4.9.2.2 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.

4.9.2.3 Assembling Options

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

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

4.9.2.4 Linking Options

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

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

--ivt-loc=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

4.9.2.5 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.

4.9.3 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.

4.9.4 Preprocessor Macros

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

Macro	Description
__SDCC_pic16	Port identification
pic18fxxxx	MCU Identification. *xxxx* is the microcontrol identification number, i.e. 452, 6620, etc
__18Fxxxx	MCU Identification (same as above)
STACK_MODEL_nnn	nnn = SMALL or LARGE respectively according to the stack model used

In addition the following macros are defined when calling assembler:

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

4.9.5 Directories

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

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

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

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

4.9.6 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:

[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.

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

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

This feature allows for linking with specific libraries 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.

```
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
```

4.9.7 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!

4.9.8 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.

4.9.9 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.

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, but call `./configure.gnu --enable-floats` instead of just `./configure.gnu`. Also make sure that at least `libc/stdio/vfprintf.c` is actually recompiled, e.g. by `touch`ing 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.

4.9.10 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/*/*.`
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 4.9.9 or just configure and build sdcc again from scratch (recommended).

4.9.11 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:

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

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

The standard device libraries (see 4.9.8) contain no reference to pointers, so they can be used with both memory models.

[3] In fact, the `.ignore` files are only used when auto-generating `Makefile.am` using the `.../libio/mkmk.sh` script.

4.9.12 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

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

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

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

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

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

4.9.13 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:

```
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.

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

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

4.9.14 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:

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

4.9.15 Interrupts

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

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

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

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

When generating assembly code for ISR the code generator places a GOTO instruction at the *Interrupt Vector Address* which points at the 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]

4.9.16 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:

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

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

[4] 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.

4.9.17 Configuration Bits

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

- using #pragma config (see section 4.9.6), which is a preferred method for the new code. Example:

```
#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:

```
#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".

4.9.18 PIC16 C Libraries

4.9.18.1 Standard I/O Streams

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

```
typedef char * FILE;
```

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

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

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

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

Currently implemented there are 3 types of streams defined:

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

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

__stream_usart_putchar writes a character at the USART stream

__stream_mssp_putchar writes a character at the MSSP stream

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

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

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

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

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

4.9.18.2 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_lista ap);
extern unsigned int fprintf(FILE *fp, char *fmt, ...);
extern unsigned int vfprintf(FILE *fp, char *fmt, va_list ap);
```

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

4.9.18.3 Signals

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

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

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

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

DEF_INTLOW(name) begin the definition of the interrupt dispatch table 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:

```
#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.

4.9.19 PIC16 Port – Tips

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

4.9.19.1 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.

4.9.20 Known Bugs

4.9.20.1 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.

4.9.20.2 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 3.3.5) 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 3.3.5) 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 (*i*n *c*ircuit *e*mulator). 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 used to specify the directory search list. The debugger will look into the directory list specified for source, cdb & ihx files. The items in the directory list must be separated by ':', e.g. if the source files can be in the directories /home/src1 and /home/src2, the --directory option should be --directory=/home/src1:/home/src2. Note there can be no spaces in the option.
- -cd <directory> - change to the <directory>.

- -fullname - used by GUI front ends.
- -cpu <cpu-type> - this argument is passed to the simulator please see the simulator docs for details.
- -X <Clock frequency > this options is passed to the simulator please see the simulator docs for details.
- -s <serial port file> passed to simulator see the simulator docs for details.
- -S <serial in,out> passed to simulator see the simulator docs for details.
- -k <port number> passed to simulator see the simulator docs for details.

5.1.5 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, xxgdb or xemacs) existing for the GNU debugger. If you use a graphical user interface for the debugger you can skip this section.

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

Set breakpoint at specified line or function:

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

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

Clear breakpoint at specified line or function:

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

continue

Continue program being debugged, after breakpoint.

finish

Execute till the end of the current function.

delete [n]

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

info [break | stack | frame | registers]

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

step

Step program until it reaches a different source line. Note: pressing <return> repeats the last command.

next

Step program, proceeding through subroutine calls.

run

Start debugged program.

ptype variable

Print type information of the variable.

print variable

print value of variable.

file filename

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

frame

print information about current frame.

set srcmode

Toggle between C source & assembly source.

! simulator command

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

quit

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

5.1.6 Interfacing SDCDB with DDD

The *p*ortable *n*etwork *g*raphics 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 ptypecommand for data at
;;                                             buffer point
;; x               sdcdbsrc-delete            SDCDB Delete all breakpoints if no arg
;;                                             given or delete arg (C-u arg x)
;; m               sdcdbsrc-frame             SDCDB Display current frame if no arg,
;;                                             given or display frame arg
;;                                             buffer point
;; !               sdcdbsrc-goto-sdcdb        Goto the SDCDB output buffer
;; p               sdcdb-print-c-sexp         SDCDB print command for data at
;;                                             buffer point
;; g               sdcdbsrc-goto-sdcdb        Goto the SDCDB output buffer
;; t               sdcdbsrc-mode              Toggles Sdcdbsrc mode (turns it off)
;;
;; C-c C-f         sdcdb-finish-from-src      SDCDB finish command
;;
;; C-x SPC         sdcdb-break                Set break for line with point
;; ESC t           sdcdbsrc-mode              Toggle Sdcdbsrc mode
;; ESC m           sdcdbsrc-srcmode           Toggle list mode
;;
```

5.2 Debugging with other debuggers (e.g. GDB): ELF / DWARF

For some ports, SDCC can create ELF binaries with DWARF debug information. This can e.g. be used for on-target debugging on STM8 using OpenOCD and GDB. To do so, compile with –debug –out-fmt-elf. Note that –out-fmt-elf needs to be specified both when compiling (to generate the debug info in DWARF rather than CDB format) and linking (to get an ELF binary instead of Intel Hex).

Chapter 6

TIPS

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

- Use the smallest data type to represent your data-value. If it is known in advance that the value is going to be less than 256 then use an 'unsigned char' instead of a 'short' or 'int'. Please note, that ANSI C requires both signed and unsigned chars to be promoted to 'signed int' before doing any operation. This promotion can be omitted, if the result is the same. The effect of the promotion rules together with the sign-extension is often surprising: **!**

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

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

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

  No, the result is not 4:

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

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

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

- Use unsigned when it is known in advance that the value is not going to be negative. This helps especially if you are doing division or multiplication, bit-shifting or are using an array index.
- NEVER jump into a LOOP.
- Declare the variables to be local whenever possible, especially loop control variables (induction).
- 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 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).
- 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

Name	Purpose	Directory
as2gbmap.py	sdas map to rrgb map and no$gmb sym file format converter	sdcc/support/scripts
cinc2h.pl	gpasm inc to c header file converter	sdcc/support/scripts
gen_known_bugs.pl	generate knownbugs.html	sdcc/support/scripts
keil2sdcc.pl	Keil header to SDCC header file converter	sdcc/support/scripts
makebin	Intel Hex to binary and GameBoy binay format converter	sdcc/bin
mcs51-disasm.pl	disassembler to the mcs51 instructions contained hex files	sdcc/support/scripts
mh2h.c	header file conversion	sdcc/support/scripts
optimize_pic16devices.pl	optimizes or unoptimizes the pic16devices.txt file	sdcc/support/scripts
packihx	Intel Hex packer	sdcc/bin
pic14-header-parser.pl	helper to realization of peripheral-handling (PIC14)	sdcc/support/scripts
pic16-header-parser.pl	helper to realization of peripheral-handling (PIC16)	sdcc/support/scripts
pic16fam-h-gen.pl	helper to realization of peripheral-handling (PIC14)	sdcc/support/scripts
pic18fam-h-gen.pl	helper to realization of peripheral-handling (PIC16)	sdcc/support/scripts
repack_release.sh	repack sdcc to release package	sdcc/support/scripts
sdas390	assembler	sdcc/bin
sdas6808	assembler	sdcc/bin
sdas6500	assembler	sdcc/bin
sdas8051	assembler	sdcc/bin
sdasgb	assembler	sdcc/bin
sdasz80	assembler	sdcc/bin
sdcc_cygwin_mingw32	cross compile the sdcc with mingw32 under Cygwin	sdcc/support/scripts
sdcc_mingw32	cross compile the sdcc with mingw32	sdcc/support/scripts
SDCDB	simulator	sdcc/bin
sdld	linker	sdcc/bin
sdld6808	linker	sdcc/bin
sdldgb	linker	sdcc/bin
sdldz80	linker	sdcc/bin
uCsim	simulator for various architectures	sdcc/sim/ucsim

6.3 Documentation included in the distribution

Subject / Title	Filename / Where to get
SDCC Compiler User Guide	You're reading it right now *online at:* http://sdcc.sourceforge.net/doc/sdccman.pdf
Changelog of SDCC	sdcc/Changelog *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/ChangeLog
ASXXXX Assemblers and ASLINK Relocating Linker	sdcc/sdas/doc/asmlnk.txt *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/sdas/doc/asmlnk.txt
SDCC regression test	test_suite_spec *online at:* http://sdcc.sourceforge.net/wiki/index.php/Proposed_Test_Suite_Design
Various notes	sdcc/doc/* *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/doc/
Notes on debugging with SDCDB	sdcc/debugger/README *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/debugger/README
uCsim Software simulator for microcontrollers	sdcc/sim/ucsim/doc/index.html *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/sim/ucsim/doc/index.html
Temporary notes on the pic16 port	sdcc/src/pic16/NOTES *online at:* http://svn.code.sf.net/p/sdcc/code/trunk/sdcc/src/pic16/NOTES
SDCC internal documentation (debugging file format)	sdcc/doc/cdbfileformat.pdf *online at:* http://sdcc.sourceforge.net/wiki/index.php/CDB_File_Format

6.4 Communication online at SourceForge

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

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

Name	Purpose	Where to get
gpsim	PIC simulator	http://www.dattalo.com/gnupic/gpsim.html
gputils	GNU PIC utilities	http://sourceforge.net/projects/gputils
flP5	PIC programmer	http://freshmeat.net/projects/flp5/
ec2drv/newcdb	Tools for Silicon Laboratories JTAG debug adapter, partly based on SDCDB (Unix only)	http://sourceforge.net/projects/ec2drv
indent	Formats C source - Master of the white spaces	http://directory.fsf.org/GNU/indent.html
srecord	Object file conversion, checksumming, ...	http://sourceforge.net/projects/srecord
objdump	Object file conversion, ...	Part of binutils (should be there anyway)
cmon51	8051 monitor (hex up-/download, single step, disassemble)	http://sourceforge.net/projects/cmon51
doxygen	Source code documentation system	http://www.doxygen.org
kdevelop	IDE (has anyone tried integrating SDCC & SDCDB? Unix only)	http://www.kdevelop.org
paulmon	8051 monitor (hex up-/download, single step, disassemble)	http://www.pjrc.com/tech/8051/paulmon2.html
splint	Statically checks c sources (see 3.3.5)	http://www.splint.org
ddd	Debugger, serves nicely as GUI to SDCDB (Unix only)	http://www.gnu.org/software/ddd/
d52	Disassembler, can count instruction cycles, use with options -pnd	http://www.8052.com/users/disasm/
cmake	Cross platform build system, generates Makefiles and project workspaces	http://www.cmake.org and a dedicated wiki entry: http://www.cmake.org/Wiki/CmakeSdcc

6.6 Related documentation / recommended reading

Name	Subject / Title	Where to get
c-refcard.pdf	C Reference Card, 2 pages	http://refcards.com/refcards/c/index.html
c-faq	C-FAQ	http://www.c-faq.com
ISO/IEC 9899:TC2	"C-Standard"	http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899
ISO/IEC DTR 18037	"Extensions for Embedded C"	http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1021.pdf
	Latest datasheet of target CPU	vendor
	Revision history of datasheet	vendor

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.

Vendor	Subject / Title	Where to get
Maxim / Dallas	Using the SDCC Compiler for the DS80C400	http://pdfserv.maxim-ic.com/en/an/AN3346.pdf
Maxim / Dallas	Using the Free SDCC C Compiler to Develop Firmware for the DS89C420/430/440/450 Family of Microcontrollers	http://pdfserv.maxim-ic.com/en/an/AN3477.pdf
Silicon Laboratories / Cygnal	Integrating SDCC 8051 Tools Into The Silicon Labs IDE	http://www.silabs.com/public/documents/tpub_doc/anote/Microcontrollers/en/an198.pdf
Ramtron / Goal Semiconductor	Interfacing SDCC to Syn and Textpad	http://www.ramtron.com/doc/Products/Microcontroller/Support_Tools.asp
Ramtron / Goal Semiconductor	Installing and Configuring SDCC and Crimson Editor	http://www.ramtron.com/doc/Products/Microcontroller/Support_Tools.asp
Texas Instruments	MSC12xx Programming with SDCC	http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=sbaa109&fileType=pdf

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), internationali(z|s)ation, packaging, labels, 2nd source for components, electromagnetic compatability/interference, documentation for production, production test software, update mechanism, patent issues?
- is your project adequately positioned in that magic triangle: fame, fortune, fun?

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

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

Chapter 7

Support

SDCC has grown to be a large project. The compiler alone (without the preprocessor, assembler and linker) is well over 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 day[1] 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.*

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/.

[1] 220 daily downloads on average Jan-Sept 2006 and about 150 daily downloads between 2002 and 2005. This does not include other methods of distribution.

7.3 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).

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

The PIC14 port uses a different set of regression tests , you'll find them in the directory *sdcc/src/regression*.

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

7.9 Examples

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

7.10 Use of SDCC in Education

In short: *highly* encouraged[3]. 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/applying patches, using a type-setting (as opposed to word-processing) engine LyX/LaTeX, using SourceForge http://sourceforge.net/, following some 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/.
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)
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 *g*lobal *c*ommon *s*ubexpression *e*limination, e.g.:

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

will be translated to

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

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

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

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

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

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

8.1.2 Dead-Code Elimination

```
int global;

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

will be changed to

```
int global;

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

8.1.3 Copy-Propagation

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

will be changed to

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

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

8.1.4 Loop Optimizations

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

Loop Invariant:

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

changed to

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

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

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

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

changed to

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

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

8.1.5 Loop Reversing

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

- The 'for' loop is of the form

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

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

8.1.6 Algebraic Simplifications

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

```
i = j + 0;      /* changed to:  */     i = j;
i /= 2;         /* 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.

  ```
  switch(i) {                    switch (i) {
     case 4:  ...                    case 0:  ...
     case 5:  ...                    case 1:  ...
     case 3:  ...
     case 6:  ...                    case 3:  ...
     case 7:  ...                    case 4:  ...
     case 8:  ...                    case 5:  ...
     case 9:  ...                    case 6:  ...
     case 10:  ...                   case 7:  ...
     case 11:  ...                   case 8:  ...
  }                              }
  ```

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

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

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

```
switch (i) {
  case 1:  ...
  case 2:  ...
  case 3:  ...
  case 4:  ...
  case 5:  ...
  case 6:  ...
  case 7:  ...
  case 101:  ...
  case 102:  ...
  case 103:  ...
  case 104:  ...
  case 105:  ...
  case 106:  ...
  case 107:  ...
}
```

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

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

and

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

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

8.1.8 Bit-shifting Operations.

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

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

generates the following code:

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

Typically, SDCC will not setup a loop if the shift count is known. Another example:

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

will generate:

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

8.1.9 Bit-rotation

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

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

will generate the following code:

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

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

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

8.1.10 Nibble and Byte Swapping

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

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

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

Note that SDCC stores numbers in little-endian[1] format (i.e. lowest order first) for most backends. However, the hc08, s08 and stm8 backends are big-endian.

8.1.11 Getting a 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.:

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

```
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:

```
                        61 ;  hob.c 7
000A E5*01              62          mov   a,(_gint + 1)
000C 23                 63          rl    a
000D 54 01              64          anl   a,#0x01
000F F5*02              65          mov   _foo_hob1_1_1,a
                        66 ;  hob.c 8
0011 E5*01              67          mov   a,(_gint + 1)
0013 33                 68          rlc   a
0014 92*00              69          mov   _foo_hob2_1_1,c
                        66 ;  hob.c 9
0016 E5*01              67          mov   a,(_gint + 1)
0018 33                 68          rlc   a
0019 92*01              69          mov   _foo_hob3_1_1,c
                        70 ;  hob.c 10
001B E5*01              71          mov   a,(_gint + 1)
001D 03                 72          rr    a
001E 54 01              73          anl   a,#0x01
0020 F5*03              74          mov   _foo_aob1_1_1,a
                        75 ;  hob.c 11
0022 E5*01              76          mov   a,(_gint + 1)
0024 13                 77          rrc   a
0025 92*02              78          mov   _foo_aob2_1_1,c
                        79 ;  hob.c 12
0027 E5*01              80          mov   a,(_gint + 1)
0029 A2 E3              81          mov   c,acc[3]
002B 92*03              82          mov   _foo_aob3_1_1,c
```

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 = gint + ((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). For mcs51, SDCC recognizes the following expressions to yield the higher order byte or word and generates optimized code for it, e.g.:

```
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)
                          93 ;  hob.c 16
003A 85*05*07             94         mov   _foo_hob2_1_1,(_glong + 3)
                          95 ;  hob.c 17
003D 85*04*08             96         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 = gint + ((gint >> 8) & 0xFF);
```

will still be recognized.

8.1.13 Placement of Bank-Selection Instructions

For non-intrinsic named address spaces, SDCC will place the bank selection instructions optimally. For details see Philipp Klaus Krause, "Optimal Placement of Bank Selection Instructions in Polynomial Time", Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems, M-SCOPES '13, pp 23–30. Association for Computing Machinery, 2013.

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 all ports except for mcs51, ds390, pic14 and pic16. With the exception of hc08 and s08, it is also the only available register allocator for these 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.

For more details on how this register allocator works, see: Philipp Klaus Krause, "Optimal Register Allocation in Polynomial Time", Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013. Proceedings, Lecture Notes in Computer Science, volume 7791, pp. 1-20. Springer, 2013. Also: Philipp Klaus Krause, "Bytewise Register Allocation", Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15, pp 22–27. Association for Computing Machinery, 2015.

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 }
```

(NOTE: from version 2.7.3 on, you can use option --acall-ajmp, which also takes care of aligning the interrupt vectors properly.)

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

The syntax for a rule is as follows:

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

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

The optimizer will apply to the rules one by one from the top in the sequence of their appearance, it will terminate when all rules are exhausted. If the 'restart' option is specified, then the optimizer will start matching the rules again from the top, this option for a rule is expensive (performance), it is intended to be used in situations where a transformation will trigger the same rule again. An example of this (not a good one, it has side effects) is the following rule:

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

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

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

would result in:

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

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

```
; nop
; nop
```

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

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

The optimizer does a look-up of a function name table defined in function *callFuncByName* in the source file SDCCpeeph.c, with the name *labelInRange*. If it finds a corresponding entry the function is called. Note there can be no parameters specified for 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:

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

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

8.3 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 iTemp variables defined by the compiler that still survive after all the optimizations. Live range analysis is essential for register allocation, since these computation determines which of these iTemps will be assigned to registers, and for how long.
- Phase five is register allocation. 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.

More information is available on SDCC Wiki (preliminary link https://sourceforge.net/p/sdcc/wiki/Adding%20a%20port/) and in the thread http://sourceforge.net/mailarchive/message.php?msg_id=13954144 .

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 spacess, 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

iCode	Operands	Description	C Equivalent
'!'	IC_LEFT() IC_RESULT()	NOT operation	IC_RESULT = ! IC_LEFT;
'~'	IC_LEFT() IC_RESULT()	Bitwise complement of	IC_RESULT = ~IC_LEFT;
ROT	IC_LEFT() IC_RIGHT() IC_RESULT()	Rotate	IC_RESULT = (IC_LEFT << IC_RIGHT) | (IC_LEFT >> (sizeof(IC_LEFT)*8-IC_RIGHT));
UNARYMINUS	IC_LEFT() IC_RESULT()	Unary minus	IC_RESULT = - IC_LEFT;
IPUSH	IC_LEFT()	Push the operand onto stack	NONE
IPOP	IC_LEFT()	Pop the operand from the stack	NONE
CALL	IC_LEFT() IC_RESULT()	Call the function represented by IC_LEFT	IC_RESULT = IC_LEFT();
PCALL	IC_LEFT() IC_RESULT()	Call via function pointer	IC_RESULT = (*IC_LEFT)();
RETURN	IC_LEFT()	Return the value in operand IC_LEFT	return IC_LEFT;
LABEL	IC_LABEL()	Label	IC_LABEL:
GOTO	IC_LABEL()	Goto label	goto IC_LABEL();
'+'	IC_LEFT() IC_RIGHT() IC_RESULT()	Addition	IC_RESULT = IC_LEFT + IC_RIGHT
'-'	IC_LEFT() IC_RIGHT() IC_RESULT()	Subtraction	IC_RESULT = IC_LEFT - IC_RIGHT
'*'	IC_LEFT() IC_RIGHT() IC_RESULT()	Multiplication	IC_RESULT = IC_LEFT * IC_RIGHT;
'/'	IC_LEFT() IC_RIGHT() IC_RESULT()	Division	IC_RESULT = IC_LEFT / IC_RIGHT;
'%'	IC_LEFT() IC_RIGHT() IC_RESULT()	Modulus	IC_RESULT = IC_LEFT % IC_RIGHT;
'<'	IC_LEFT() IC_RIGHT() IC_RESULT()	Less than	IC_RESULT = IC_LEFT < IC_RIGHT;
'>'	IC_LEFT() IC_RIGHT() IC_RESULT()	Greater than	IC_RESULT = IC_LEFT > IC_RIGHT;
EQ_OP	IC_LEFT() IC_RIGHT() IC_RESULT()	Equal to	IC_RESULT = IC_LEFT == IC_RIGHT;
AND_OP	IC_LEFT() IC_RIGHT() IC_RESULT()	Logical and operation	IC_RESULT = IC_LEFT && IC_RIGHT;
OR_OP	IC_LEFT() IC_RIGHT() IC_RESULT()	Logical or operation	IC_RESULT = IC_LEFT || IC_RIGHT;
'^'	IC_LEFT() IC_RIGHT() IC_RESULT()	Exclusive OR	IC_RESULT = IC_LEFT ^ IC_RIGHT;
'|'	IC_LEFT() IC_RIGHT() IC_RESULT()	Bitwise OR	IC_RESULT = IC_LEFT | IC_RIGHT;
BITWISEAND	IC_LEFT() IC_RIGHT() IC_RESULT()	Bitwise AND	IC_RESULT = IC_LEFT & IC_RIGHT;
LEFT_OP	IC_LEFT() IC_RIGHT() IC_RESULT()	Left shift	IC_RESULT = IC_LEFT << IC_RIGHT

iCode	Operands	Description	C Equivalent
RIGHT_OP	IC_LEFT() IC_RIGHT() IC_RESULT()	Right shift	IC_RESULT = IC_LEFT >> IC_RIGHT
GET_VALUE_AT_ADDRESS	IC_LEFT() IC_RESULT()	Indirect fetch	IC_RESULT = (*IC_LEFT);
POINTER_SET	IC_RIGHT() IC_RESULT()	Indirect set	(*IC_RESULT) = IC_RIGHT;
'='	IC_RIGHT() IC_RESULT()	Assignment	IC_RESULT = IC_RIGHT;

IFX	IC_COND
	IC_TRUE
	IC_LABEL
		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	if (IC_COND) goto IC_TRUE;
			Or
			If (!IC_COND) goto IC_FALSE;

ADDRESS_OF	IC_LEFT() IC_RESULT()	Address of	IC_RESULT = &IC_LEFT();
JUMPTABLE	IC_JTCOND IC_JTLABELS	Jump to list of labels depending on the value of JTCOND	Switch statement
CAST	IC_RIGHT() IC_LEFT() IC_RESULT()	Cast types	IC_RESULT = (typeof IC_LEFT) IC_RIGHT;
SEND	IC_LEFT()	This is used for passing parameters in registers; move IC_LEFT to the next available parameter register.	None
RECV	IC_RESULT()	This is used for receiving parameters passed in registers; Move the values in the next parameter register to IC_RESULT	None
(some more have been added)			*see f.e.* `gen51Code()` *in* `src/mcs51/gen.c`

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

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

In addition to the operands each iCode contains information about the filename and line it corresponds to in the source file. The first field in the listing should be interpreted as follows:
Filename(linenumber: iCode Execution sequence number : ICode hash table key : loop depth of the iCode).
Then follows the human readable form of the ICode operation. Each operand of this triplet form can be of three basic types a) compiler generated temporary b) user defined variable c) a constant value. Note that local variables and parameters are replaced by compiler generated temporaries. Live ranges are computed only for temporaries (i.e. live ranges are not computed for global variables). Registers are allocated for temporaries

only. Operands are formatted in the following manner:
Operand Name [lr live-from : live-to] { type information } [registers allocated].
As mentioned earlier the live ranges are computed in terms of the execution sequence number of the iCodes, for example
the iTemp0 is live from (i.e. first defined in iCode with execution sequence number 3, and is last used in the iCode with sequence number 5). For induction variables such as iTemp21 the live range computation extends the lifetime from the start to the end of the loop.
The register allocator used the live range information to allocate registers, the same registers may be used for different temporaries if their live ranges do not overlap, for example r0 is allocated to both iTemp6 and to iTemp17 since their live ranges do not overlap. In addition the allocator also takes into consideration the type and usage of a temporary, for example itemp6 is a pointer to near space and is used as to fetch data from (i.e. used in GET_VALUE_AT_ADDRESS) so it is allocated a pointer register (r0). Some short lived temporaries are allocated to special registers which have meaning to the code generator e.g. iTemp13 is allocated to a pseudo register CC which tells the back end that the temporary is used only for a conditional jump the code generation makes use of this information to optimize a compare and jump ICode.
There are several loop optimizations performed by the compiler. It can detect induction variables iTemp21(i) and iTemp23(j). Also note the compiler does selective strength reduction, i.e. the multiplication of an induction variable in line 18 (gint = j * 3) is changed to addition, a new temporary iTemp17 is allocated and assigned a initial value, a constant 3 is then added for each iteration of the loop. The compiler does not change the multiplication in line 17 however since the processor does support an 8 * 8 bit multiplication.
Note the dead code elimination optimization eliminated the dead assignments in line 7 & 8 to I and sum respectively.

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

Finally the code generated for this function:

```
.area DSEG (DATA)
_p::
  .ds 2
_gint::
  .ds 2
```

```
; sample.c 5
; ------------------------------
; function function
; ------------------------------
_function:
; iTemp0 [lr3:5]{_near * int}[r2] = recv
  mov r2,dpl
; iTemp6 [lr5:16]{_near * int}[r0] := iTemp0 [lr3:5]{_near * int}[r2]
  mov ar0,r2
;_whilecontinue_0($1) :
00101$:
; iTemp4 [lr7:8]{int}[r2 r3] = @[iTemp6 [lr5:16]{_near * int}[r0]]
; if iTemp4 [lr7:8]{int}[r2 r3] == 0 goto _whilebreak_0($3)
  mov ar2,@r0
  inc r0
  mov ar3,@r0
  dec r0
  mov a,r2
  orl a,r3
  jz 00103$
00114$:
; iTemp7 [lr9:13]{_far * int}[DPTR] := _p [lr0:0]{_far * int}
  mov dpl,_p
  mov dph,(_p + 1)
; _p [lr0:0]{_far * int} = _p [lr0:0]{_far * int} + 0x2 {short}
  mov a,#0x02
  add a,_p
  mov _p,a
  clr a
  addc a,(_p + 1)
  mov (_p + 1),a
; iTemp10 [lr13:14]{int}[r2 r3] = @[iTemp7 [lr9:13]{_far * int}[DPTR]]
  movx a,@dptr
  mov r2,a
  inc dptr
  movx a,@dptr
  mov r3,a
; *(iTemp6 [lr5:16]{_near * int}[r0]) := iTemp10 [lr13:14]{int}[r2 r3]
  mov @r0,ar2
  inc r0
  mov @r0,ar3
; iTemp6 [lr5:16]{_near * int}[r0] =
; iTemp6 [lr5:16]{_near * int}[r0] +
; 0x2 {short}
  inc r0
; goto _whilecontinue_0($1)
  sjmp 00101$
; _whilebreak_0($3) :
00103$:
; iTemp2 [lr18:40]{short}[r2] := 0x0 {short}
  mov r2,#0x00
; iTemp11 [lr19:40]{short}[r3] := 0x0 {short}
  mov r3,#0x00
; iTemp21 [lr21:38]{short}[r4] := 0x0 {short}
  mov r4,#0x00
; iTemp23 [lr22:38]{int}[r5 r6] := 0xa {int}
  mov r5,#0x0A
  mov r6,#0x00
; iTemp17 [lr23:38]{int}[r7 r0] := 0x1e {int}
  mov r7,#0x1E
  mov r0,#0x00
; _forcond_0($4) :
00104$:
; iTemp13 [lr25:26]{char}[CC] = iTemp21 [lr21:38]{short}[r4] < 0xa {short}
; if iTemp13 [lr25:26]{char}[CC] == 0 goto _forbreak_0($7)
  clr c
  mov a,r4
  xrl a,#0x80
  subb a,#0x8a
  jnc 00107$
00115$:
; iTemp2 [lr18:40]{short}[r2] = iTemp2 [lr18:40]{short}[r2] +
; iTemp21 [lr21:38]{short}[r4]
  mov a,r4
  add a,r2
```

```
  mov r2,a
; iTemp15 [lr29:30]{short}[r1] = iTemp21 [lr21:38]{short}[r4] * 0x3 {short}
  mov b,#0x03
  mov a,r4
  mul ab
  mov r1,a
; iTemp11 [lr19:40]{short}[r3] = iTemp11 [lr19:40]{short}[r3] +
; iTemp15 [lr29:30]{short}[r1]
  add a,r3
  mov r3,a
; iTemp17 [lr23:38]{int}[r7 r0]= iTemp17 [lr23:38]{int}[r7 r0]- 0x3 {short}
  mov a,r7
  add a,#0xfd
  mov r7,a
  mov a,r0
  addc a,#0xff
  mov r0,a
; _gint [lr0:0]{int} = _gint [lr0:0]{int} + iTemp17 [lr23:38]{int}[r7 r0]
  mov a,r7
  add a,_gint
  mov _gint,a
  mov a,r0
  addc a,(_gint + 1)
  mov (_gint + 1),a
; iTemp21 [lr21:38]{short}[r4] = iTemp21 [lr21:38]{short}[r4] + 0x1 {short}
  inc r4
; iTemp23 [lr22:38]{int}[r5 r6]= iTemp23 [lr22:38]{int}[r5 r6]- 0x1 {short}
  dec r5
  cjne r5,#0xff,00104$
  dec r6
; goto _forcond_0($4)
  sjmp 00104$
; _forbreak_0($7) :
00107$:
; ret iTemp24 [lr40:41]{short}
  mov a,r3
  add a,r2
  mov dpl,a
; _return($8) :
00108$:
  ret
```

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

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

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

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

Chapter 10

Acknowledgments

http://sdcc.sourceforge.net/#Who

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

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, 31
--acall-ajmp, 33, 114
--all-callee-saves, 31
--allow-unsafe-read, 30
--c1mode, 30
--callee-saves, 31, 64
--code-loc <Value>, 32, 39
--code-size <Value>, 33, 39
--codeseg <Value>, 32
--compile-only, 30
--const-stringlit, 28
--constseg <Value>, 32
--cyclomatic, 31
--data-loc, 69
--data-loc <Value>, 32, 39
--debug, 25, 30, 31, 83, 95
--disable-warning, 31
--dump-ast, 35
--dump-graphs, 35
--dump-i-code, 35
--dumpall, 104
--fdollars-in-identifiers, 27
--float-reent, 31
--fomit-frame-pointer, 30
--fsigned-char, 23, 28
--i-code-in-asm, 31
--idata-loc <Value>, 33
--int-long-reent, 31, 45, 53
--iram-size <Value>, 33, 39, 64
--less-pedantic, 31
--lib-path <path>, 32
--max-allocs-per-node, 30
--model-huge, 33
--model-large, 33, 34, 54
--model-medium, 33, 34
--model-small, 33, 34
--more-pedantic, 32
--no-c-code-in-asm, 31
--no-gen-comments, 35
--no-peep, 30
--no-peep-comments, 31
--no-peep-return, 30
--no-ret-without-call, 33
--no-std-crt0, 23, 66, 75
--no-xinit-opt, 29, 64
--no-zp-spill, 34
--nogcse, 29
--noinduction, 29
--noinvariant, 29
--nolabelopt, 29
--nooloopreverse, 29
--nolospre, 29
--nooverlay, 30
--norestartseqatomics, 31
--nostdinc, 31
--nostdlib, 31
--nostdlibcall, 30
--opt-code-size, 30
--opt-code-speed, 30
--out-fmt-ihx, 33
--out-fmt-s19, 25, 33
--peep-asm, 30, 51
--peep-file, 30, 114
--peep-return, 30
--print-search-dirs, 18, 31
--purity, 29
--stack-auto, 30, 34, 43, 45, 53, 56, 58
--stack-loc <Value>, 32, 39
--stack-size <Value>, 33
--std-c11, 4, 22
--std-c23, 4, 22
--std-c2y, 23
--std-c89, 27
--std-c90, 4, 5, 21
--std-c94, 21
--std-c95, 21
--std-c99, 4, 5, 22
--std-sdcc11, 27
--std-sdcc17, 27
--std-sdcc23, 27
--std-sdcc2y, 27
--std-sdcc89, 27
--std-sdcc99, 27
--syntax-only, 30
--use-non-free, 3, 32, 78, 84, 86
--use-stdout, 32, 35
--vc, 32, 35
--verbose, 31
--version, 30
--xdata-loc<Value>, 39
--xram-loc, 69
--xram-loc <Value>, 32
--xram-size <Value>, 33, 39
--xstack, 33, 34, 37, 56
--xstack-loc <Value>, 32
-Aquestion(answer), 29
-C, 29
-D<macro[=value]>, 29
-E, 29, 30
-I<path>, 29

-L <path>, 32
-M, 29
-MM, 29
-S, 31
-Umacro, 29
-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
-mez80, 28
-mhc08, 28
-mmcs51, 28
-mmos6502, 28
-mmos65c02, 28
-mpdk13, 28
-mpdk14, 28
-mpdk15, 28
-mpic14, 28
-mpic16, 28
-mr2k, 28
-mr2ka, 28
-mr3ka, 28
-mr800, 28
-ms08, 28
-msm83, 28
-mstm8, 28
-mtlcs90, 28
-mz180, 28
-mz80, 28
-mz80n, 28
-o <path/file>, 30
-v, 30
-x <type>, 30
<NO FLOAT>, 54, 86
<file>.adb, 25, 95
<file>.asm, 25
<file>.cdb, 25, 95
<file>.dump*, 25
<file>.ihx, 25, 26
<file>.lib, 26
<file>.lnk, 26
<file>.lst, 25, 41
<file>.map, 25, 39, 41
<file>.mem, 25, 39
<file>.omf, 25
<file>.rel, 25–27
<file>.rst, 25, 41
<file>.sym, 25
<stdio.h>, 54
#defines, 60
#pragma callee_saves, 31, 57
#pragma codeseg, 58
#pragma constseg, 58
#pragma disable_warning, 58
#pragma exclude, 52, 57
#pragma less_pedantic, 57
#pragma max_allocs_per_node, 58
#pragma nogcse, 29, 58
#pragma noinduction, 29, 58, 108
#pragma noinvariant, 29, 58
#pragma noiv, 58
#pragma noloopreverse, 58
#pragma nooverlay, 44, 45, 58
#pragma opt_code_balanced, 58
#pragma opt_code_size, 58
#pragma opt_code_speed, 58
#pragma preproc_asm, 58
#pragma restore, 57
#pragma save, 57
#pragma sdcc_hash, 59
#pragma stackauto, 43, 58
#pragma std_c11, 58
#pragma std_c23, 58
#pragma std_c2y, 58
#pragma std_c89, 58
#pragma std_c99, 58
#pragma std_sdcc11, 58
#pragma std_sdcc23, 58
#pragma std_sdcc2y, 58
#pragma std_sdcc89, 58
#pragma std_sdcc99, 58
_Optional, 43
_XPAGE (mcs51), 61
__addressmod, 41
__asm, 48, 50–52
__at, 37, 39–42, 49
__bit, 5, 37, 42
__code, 37
__critical, 47
__data (hc08 named address space), 40
__data (mcs51, ds390 named address space), 32, 36, 38
__data (mos6502 named address space), 40
__endasm, 48, 50–53
__far (named address space), 36, 49
__idata (mcs51, ds390 named address space), 33, 36, 38
__interrupt, 38, 45, 51
__naked, 51, 57
__near (named address space), 36
__pdata (mcs51, ds390 named address space), 32, 33, 36, 56, 61
__sbit, 5
__sdcc_external_startup, 42
__sfr, 37, 39, 40
__sfr16, 37, 40
__sfr32, 37
__using (mcs51, ds390 register bank), 38, 45, 46, 48
__xdata (hc08 named address space), 40
__xdata (mcs51, ds390 named address space, 64
__xdata (mcs51, ds390 named address space), 32, 36, 38, 41

__xdata (mos6502 named address space), 41
_far, 39, 40
_naked, 51, 57
8031, 8032, 8051, 8052, mcs51 CPU, 2

~ Operator, 5
~ Operator, 100

Absolute addressing, 41, 43
ACC (mcs51, ds390 register), 64
Aligned array, 42, 49, 50
Annotated syntax tree, 117
Any Order Bit, 111
AOMF, AOMF51, 25, 31, 94, 95
Application notes, 103
Assembler documentation, 51, 101
Assembler listing, 25
Assembler options, 32
Assembler routines, 48, 49, 64, 114
Assembler routines (non-reentrant), 65
Assembler routines (reentrant), 65
Assembler source, 25
at, 44
atomic, 45, 48

B (mcs51, ds390 register), 64
backfill unused memory, 26
banked, 62
Bankswitching, 61
Basic blocks, 122
Binary constants, 43
bit, 33, 38, 100
Bit rotation, 111
Bit shifting, 110
Bit toggling, 5
bit-fields, 37
block boundary, 42
Boost Software License 1.0 (BSL-1.0), 4
Bug reporting, 104
Building SDCC, 14
Byte swapping, 111

C FAQ, 103
C Reference card, 103
Carry flag, 37
Changelog, 105
checksum, 26
cmake, 102
code, 32
code banking, 61
code page (pic14), 77
Command Line Options, 27
Communication
  Bug report, 104
  Feature request, 104
  Forums, 102
  Mailing lists, 102, 105
  Monitor, 102
  Patch submission, 105
  RSS feed, 102
  Trackers, 102
  wiki, 102
Compatibility with previous versions, 4
Compiler internals, 117
compiler.h (include file), 38, 100
const, 32
Copy propagation, 108
cpp, *see* sdcpp, *see* sdcpp
critical, 47
Cyclomatic complexity, 31, 115

d52, 102
d52 (disassembler), 102
DDD (debugger), 97, 102
Dead-code elimination, 107, 120
Debugger, 25, 95
Defines created by the compiler, 60
DESTDIR, 12
Division, 44
Documentation, 17, 101
double (not supported), 21, 22
download, 104
doxygen (source documentation tool), 102
DPTR, 61, 64, 111
DPTR, DPH, DPL, 64, 65
DS390, 33
  Options
    --model-flat24, 33
    --protect-sp-update, 33
    --stack-10bit, 34
    --stack-probe, 34
    --tini-libid, 34
    --use-accelerator, 34
DS390 memory model, 56
DS400, 66
DS80C390, 28
DS80C400, 28, 66, 103
DS89C4x0, 103
dynamic memory allocation (malloc), 55

ELF format, 33
Emacs, 97
Endianness, 100, 111
Environment variables, 35
Examples, 105, 106
External stack (mcs51), 56

Feature request, 104
Flags, 37
Flat 24 (DS390 memory model), 56
Floating point support, 21, 22, 45, 53, 54
FPGA (field programmable gate array), 17
FpgaC ((subset of) C to FPGA compiler), 17
function epilogue, 31, 51
function parameter, 43, 44, 65
function pointer, 38
function pointers, 64
function prologue, 31, 51, 57

GBZ80

Options
-ba <Num>, 34
-bo <Num>, 34
gcc (GNU Compiler Collection), 29
gdb, 95
generic pointer, 64
getchar(), 54
GPLv2 license, 4
GPLv2+LE, 3, 55
GPLv3 license, 4
gpsim (pic simulator), 102
gputils (pic tools), 78, 102

HC08, 28, 33, 40, 46, 74
interrupt, 46, 48
Options
--out-fmt-elf, 33
HD64180 (see Z180), 39
Header files, 38, 100, 101
heap (malloc), 55
Higher Order Byte, 112
Higher Order Word, 112

I/O memory (Z80, Z180), 39
ICE (in circuit emulator), 94
iCode, 35, 117–119
IDE, 32, 103
Include files, 38, 100, 101
indent (source formatting tool), 102
Infineon, 33
Install paths, 12
Install trouble-shooting, 18
Installation, 9
instruction cycles (count), 102
Intel hex format, 25, 26, 33, 95
Intermediate dump options, 35
interrupt, 38, 44, 45, 47–49, 51, 53, 57, 58
interrupt jitter, 48
interrupt latency, 48
interrupt mask, 48
interrupt priority, 48, 49
interrupt vector table, 32, 45, 58
interrupts, 49
intrinsic named address space, 44, 56

jump tables, 109

K&R style, 21

Labels, 52
LGPLv2.1 license, 4
Libraries, 26, 31, 32, 38, 54, 55
Linker, 26
Linker documentation, 101
Linker options, 32
lint (syntax checking tool), 32, 94
little-endian, 111
Live range analysis, 116, 117, 119
local variables, 43, 44, 56, 99
lock, 48
Loop optimization, 108, 120
Loop reversing, 29, 109

mailing list, 102
Mailing list(s), 104, 105
Makefile, 102
malloc.h, 55
MCS51, 28
MCS51 memory, 38
MCS51 memory model, 56
MCS51 options, 33
MCS51 variants, 61, 114
Memory bank (pic14), 77
Memory map, 25, 100
Memory model, 38, 44, 56, 57
Microchip, 76, 80
MOS6502, 40, 75
MOS6502 memory models, 57
MOS6502 options, 34
Motorola S19 format, 25, 33
MSVC output style, 32
msys, 16
msys2, 16
Multiplication, 44, 108, 120

naked, 64
Naked functions, 51
Nibble swapping, 111
Non-intrinsic named address spaces, 41

objdump (tool), 25, 102
Object file, 25
Optimization options, 29
Optimizations, 107, 117
Options assembler, 32
Options DS390, 33
Options GBZ80, 34
Options intermediate dump, 35
Options linker, 32
Options MCS51, 33
Options MOS6502, 34
Options optimization, 29
Options other, 30
Options PIC16, 82
Options preprocessor, 29
Options processor selection, 28
Options SDCC configuration, 9
Options STM8, 34
Options Z80, 34
Oscilloscope, 94
Overlaying, 44

P2 (mcs51 sfr), 36, 56, 61
packihx (tool), 26, 101
Parameter passing, 64
Parameters, 43
Parsing, 117
Patch submission, 104, 105
PDF version of this document, 17
pedantic, 31, 32, 57
Peephole optimizer, 30, 51, 114

PIC, 80
PIC14, 28, 76, 80
  Environment variables
    SDCC_PIC14_SPLIT_LOCALS, 79
  interrupt, 78
  Options
    --debug-extra, 79
    --no-pcode-opt, 79
    --stack-loc, 79
    --stack-size, 79
    --use-non-free, 79
PIC16, 28, 80, 83, 84, 86, 88, 89, 101
  Defines
    __pic18fxxxx, 83
    pic18fxxxx, 83
    STACK_MODEL_nnn, 83
  Environment variables
    NO_REG_OPT, 83
    OPTIMIZE_BITFIELD_POINTER_GET, 83
  Header files, 86
  interrupt, 89
  Libraries, 86
  MPLAB, 83
  Options
    --callee-saves, 82
    --use-non-free, 82
  shadowregs, 88
  stack, 88, 92
  wparam, 88
Pointer, 38
Pragmas, 57
Preprocessor, 20, 30, 58
  Options, 29
  PIC16 Macros, 83
printf(), 54, 55
  floating point support, 54
  parameters, 100
  PIC16, 91
  PIC16 Floating point support, 86
  PIC16 floating point support, 86
  printf_fast() (mcs51), 54
  printf_fast_f() (mcs51), 54
  printf_small(), 54
  printf_tiny() (mcs51), 54
  putchar(), 54, 100
Processor selection options, 28
project workspace, 102
promotion to signed int, 49, 50, 99
push/pop, 51, 52, 57
putchar(), 54

Quality control, 105

reentrant, 30, 31, 43, 44, 53, 56, 65
Register allocation, 108, 117, 119
register bank (mcs51, ds390), 38, 43, 48
Register-Allocation, 113
Regression test, 101, 105, 106
Regression test (PIC14), 105
Regression test (PIC16), 93
Related tools, 102
Release policy, 105
Reporting bugs, 104
Requesting features, 104
return value, 21, 22, 64
rotating bits, 111
RSS feed, 102
Runtime library, 62

S08, 28
s51 (simulator), 20
sbit, 37
sdar, 27
sdas (sdasgb, sdas6808, sdas8051, sdasz80), 2, 51, 101
SDCC
  Defines
    __SDCC (version macro), 60
    __SDCC_ds390, 60
    __SDCC_mcs51, 60
    __SDCC_pic16, 83
    __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_OPTIMIZE_SIZE, 60
    SDCC_OPTIMIZE_SPEED, 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
    SDCCCALL, 60
  Environment variables
    NO_REG_OPT, 83
    OPTIMIZE_BITFIELD_POINTER_GET (PIC16), 83
    SDCC_HOME, 35
    SDCC_INCLUDE, 35
    SDCC_LEAVE_SIGNALS, 35
    SDCC_LIB, 35
    SDCC_PIC14_SPLIT_LOCALS, 79
    TMP, TEMP, TMPDIR, 35
    undocumented, 35
SDCC Wiki, 105
SDCDB (debugger), 20, 95, 101, 102
sdcpp (preprocessor), 20, 29, 58
sdld, 2, 101
Search path, 12
semaphore, 48
sfr, 61
shc08 (simulator), 20
signal handler, 35
sloc (spill location), 29
SM83, 34
sm83 (GameBoy Z80), 28, 66

splint (syntax checking tool), 32, 94, 102
srecord (bin, hex, ... tool), 25, 26, 33, 102
sstm8 (simulator), 20
stack, 30, 36, 39, 43–48, 56, 108
stack overflow, 45
Standard-compliance, 4, 21
static, 43
Status of documentation, 3, 17
STM8, 2
STM8 memory models, 56
STM8 options, 34
Strength reduction, 108, 120
struct, 21, 22
Subexpression, 109
Subexpression elimination, 29, 107
Subversion code repository, 104, 105
Support, 104
swapping nibbles/bytes, 111
switch statement, 109
Symbol listing, 25
sz80 (simulator), 20

tabulator spacing (8 columns), 15
Tinibios (DS390), 56
Tools, 100
Trademarks, 123
type conversion, 5
type promotion, 5, 45, 49, 50, 99
Typographic conventions, 4

uCsim, 101
union, 21, 22
UnxUtils, 16
USE_FLOATS, 54
using (mcs51, ds390 register bank), 46, 48

vararg, va_arg, 4, 100
Variable initialization, 29, 41
version, 18, 105
version macro, 60
volatile, 41, 42, 45, 48, 51, 100
VPATH, 17

Warnings, 31
watchdog, 64, 100
wiki, 102, 105, 116

XEmacs, 97
xstack, 32

Z180, 28, 39
  I/O memory, 39
  Options
    --portmode, 39
  Pragmas
    #pragma portmode, 39
Z80, 28, 34, 39, 46, 66
  I/O memory, 39
  interrupt, 46
  Options
    --asm=<Value>, 34
    --callee-saves-bc, 34
    --codeseg <Value>, 34
    --constseg <Value>, 34
    --fno-omit-frame-pointer, 34
    --no-std-crt0, 34
    --portmode=<Value>, 34
    --reserve-regs-iy, 34
Z80, Z180, SM83, Rabbit 2000/3000, Rabbit 3000A CPU, 2
Z80N, 28
zlib/libpng License, 4