Optimizing C Compiler

for

XC800 Microcontroller.

SDCC for XC800

Edition Dec 12, 2006

Trademarks and Disclaimer
SDCC is an open source product and is covered under GPL.

Attention please!
The information herein is given to describe certain components and shall not be considered as warranted characteristics. Terms of delivery and rights to technical change reserved. We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Warnings
Due to technical requirements, components may contain dangerous substances.

Optimizing C Compiler

for

XC800 Microcontroller

Revision History

User Manual for Small Device C Compiler First Release: May 30, 2005		
References	**CR**	**Subjects (Major changes since last revision)**
Version 2.4.0 C Dated: Jan 06, 2006		
Section 3.14		Startup Code
Section 3.14.1		Customizing XC866 Startup Code
Section 4.1.5		Signed-ness of bitwise operators
Version 2.4.0 B3 Dated: Feb 27, 2006		
Sections 2.3.1 and 2.3.2		Install Paths and Search Paths
Section 3.2		Command Line Options
Section 3.14		Startup code
Section 3.19		xc800 Memory Models
Section 4.1		SDCC for xc800 port
Appendix A		xc800 Specific Instruction Generation
Multiple sections		xc866 v2.4.0 C changed to xc800 v2.4.0 B3
Version 2.5.4 B4 Dated: June 15, 2006		
Section 3.8		Absolute addressing
Section 3.18		Usage of boot ROM library for SDCC
Appendix C		List of boot ROM library functions used by SDCC
Appendix D		Support for Silicon Bug : Flash XC8.004.

Version 2.5.4 B4 Dated: Dec 12, 2006

Section 4.1.2.2		Usage of large memory model for XC856

1 Introduction . . . 9
1.1 About SDCC . . . 9
1.2 Open Source . . . 9
1.3 Typographic conventions . . . 10
1.4 Other Resources . . . 10
1.5 Wishes for the future . . . 10

2 Software Installation . . . 11
2.1 System Requirements . . . 11
2.2 Components of SDCC Toolchain . . . 11
2.3 Installation . . . 13
2.3.1 Install Paths . . . 14
2.3.2 Search Paths . . . 15

3 Using SDCC . . . 17
3.1 Compiling . . . 17
3.1.1 Single Source File Projects . . . 17
3.1.2 Projects with Multiple Source Files . . . 17
3.1.3 Projects with Additional Libraries . . . 18
3.1.4 Using sdcclib to Create and Manage Libraries . . . 18
3.2 Command Line Options . . . 20
3.2.1 Processor Selection Options . . . 20
3.2.2 Preprocessor Options . . . 20
3.2.3 Linker Options . . . 21
3.2.4 XC800 Options . . . 22
3.2.5 Optimization Options . . . 23
3.2.6 Other Options . . . 24
3.2.7 Intermediate Dump Options . . . 26
3.2.8 Redirecting output on Windows Shells . . . 26
3.3 SDCC Wrapper options . . . 26
3.4 Examples on SDCC Usage . . . 27
3.5 Environment variables . . . 27
3.6 Data types . . . 28
3.7 Storage Class Language Extensions . . . 28
3.7.1 XC800 Storage Class Language Extensions . . . 28
3.8 Absolute Addressing . . . 31
3.9 Parameters & Local Variables . . . 33
3.10 Overlaying . . . 34
3.11 Interrupt Service Routines . . . 35
3.11.1 General Information . . . 35
3.11.2 XC800 Interrupt Service Routines . . . 36
3.12 Functions using private register banks (xc800) . . . 36
3.13 Critical Functions and Critical Statements . . . 37
3.14 Startup Code . . . 37
3.14.1 XC800 Startup Code . . . 38
3.15 Inline Assembly . . . 38
3.15.1 A Step by Step Introduction . . . 38
3.15.2 Naked Functions . . . 41
3.15.3 Use of Labels within Inline Assembler . . . 42

3.16 Interfacing with Assembler Code . . . 43
3.16.1 Global Registers used for Parameter Passing . . . 43
3.16.2 Assembler Routine (non-reentrant) . . . 43
3.17 int (16 bit), long (32 bit) and Floating point support . . . 44
3.18 Library Routines . . . 45
3.18.1 Compiler support routines (_gptrget, _mulint etc.) . . . 46
3.18.2 Stdclib functions (puts, printf, strcat etc.) . . . 46
3.18.3 Math functions (sin, pow, sqrt etc.) . . . 46
3.18.4 Other libraries . . . 46
3.19 XC800 Memory Models . . . 46
3.19.1 Small Model . . . 46
3.19.2 Large Model . . . 47
3.19.3 XC800 Address Spaces . . . 47
3.20 External Stack . . . 48
3.21 Pragmas . . . 49
3.22 Defines Created by the Compiler . . . 50

4 Notes on SDCC support for XC800 . . . 51
4.1 SDCC for XC800 port . . . 51
4.1.1 Limitations of SDCC for XC800 . . . 51
4.1.2 Using printf functions . . . 51
4.1.3 Using Dynamic Memory in XC800 . . . 53
4.1.4 ANSI-Compliance . . . 53
4.1.5 Signed-ness of bitwise operators . . . 55
4.1.6 Guidelines to generate efficient code . . . 55

5 SDCC Technical Data . . . 59
5.1 Optimizations . . . 59
5.1.1 Sub-expression Elimination . . . 59
5.1.2 Dead-Code Elimination . . . 59
5.1.3 Copy-Propagation . . . 60
5.1.4 Loop Optimizations . . . 60
5.1.5 Loop Reversing . . . 61
5.1.6 Algebraic Simplifications . . . 62
5.1.7 'switch' Statements . . . 62
5.1.8 Bit-shifting Operations . . . 64
5.1.9 Bit-rotation . . . 64
5.1.10 Nibble and Byte Swapping . . . 65
5.1.11 Highest Order Bit . . . 65
5.2 Cyclomatic Complexity . . . 68

6 Tools included in the distribution . . . 71

XC800 Specific Instruction Generation 73

Example XC866 code for SDCC 75

List of boot ROM library used by SDCC 77

Support for Silicon Bug : FLASH XC8.004. 79

1 Introduction

1.1 About SDCC

SDCC (Small Device C Compiler) is a freeware, retargetable, optimizing ANSI-C compiler by Sandeep Dutta designed for 8 bit Microprocessors.

The entire source code for the compiler is distributed under GPL. SDCC uses ASXXXX & ASLINK, a Freeware, retargetable assembler & linker. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware effectively.

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

- global sub expression elimination
- loop optimizations (loop invariant, strength reduction of induction variables and loop reversing)
- constant folding and 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 peephole optimizer uses a rule based substitution mechanism which is MCU independent. The compiler also allows an 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 hard coded in the assembly if needed.

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

Note: The compiler will probably always be some steps ahead of this documentation.

1.2 Open Source

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

distributed with the package. This documentation is maintained using a freeware word processor (LyX).

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

1.3 Typographic conventions

Throughout this manual, we will use the following convention.

Commands you have to type in are printed in "sans serif". Code samples are printed in typewriter font. Interesting items and new terms are printed in italics.

1.4 Other Resources

The SDCC home page at [http://sdcc.sourceforge.net/] is a great place to find distribution sets. The current distribution with support for XC800 is not in sourceforge CVS repository.

1.5 Wishes for the future

There are and will always be some things that could be done. Here are some of these:

- Char KernelFunction3(char p) at 0x340;
- Support for multiple data pointers for XC800.

2 Software Installation

This chapter describes how you can install the SDCC Toolchain on Windows and Linux hosts. The Version 2.5.4B4, as of June 15th, 2006, has only a Windows version.

2.1 System Requirements

Use of SDCC Toolchain requires:

- The SDCC Toolchain package
- A host-PC with minimum 64MB RAM
- Windows NT, Windows 2000/XP

2.2 Components of SDCC Toolchain

SDCC is not just a compiler, but a collection of tools by various developers. These include linkers, assemblers, and other components. Here is a summary of some of the components.

Note: The included assembler has separate documentation which you can find in the <installdir>\doc directory.

You might want to look at the files which are installed in <installdir>.
In **Windows**, the structure is as follows:

In <installdir>\bin:

- sdcc1 - The compiler.
- sdcc - The wrapper for compiling multiple source files using sdcc1.
- sdcpp - The C preprocessor.
- as-xc800 - XC800 assembler.
- link-xc800 - XC800 linkers.
- sdcclib – The sdcc librarian, to create and manage libraries.
- packihx - A tool to pack (compress) Intel hex files.

In <installdir>\include

- The include files

In <installdir>\lib\xc866

- The subdirs small and large with the precompiled relocatables for xc866.

In <installdir>\lib\xc886

- The subdirs small and large with the precompiled relocatables for xc886.

In <installdir>\doc

- The documentation

Note: <installdir> is the directory where you have unzipped the software.

In **Linux**, the structure is as follows,

In <installdir>/usr/local/bin:

- sdcc1 - The compiler.
- sdcc - The wrapper for compiling multiple source files using sdcc1.
- sdcpp - The C preprocessor.
- as-xc866 - XC866 assembler.
- link-xc866 - XC866 linkers.
- sdcclib -The sdcc librarian, to create and manage libraries.
- packihx - A tool to pack (compress) Intel hex files.

In <installdir>/usr/local/share/sdcc/include

- The include files

In <installdir>/usr/local/share/sdcc/lib/xc866

- The subdirs small and large with the precompiled relocatables.

Note: The XC886 release does not have, as of date, Linux support.

In <installdir>/usr/local/share/sdcc/doc

- The documentation

Note: <installdir> is the directory where you have unzipped the software.

sdcc1 - The Compiler

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

sdcc - The Wrapper

This wrapper can be used to compile multiple source files. It in turn uses sdcc1 compiler for compiling the source files. All the sdcc1 compiler options are valid for this wrapper.

sdcpp - The C-Preprocessor

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

as-xc800, **link-xc800** - The Assembler and Linkage Editor for XC800.
This is the retargettable assembler and linkage editor.

sdcclib – The Librarian
This is a tool to create and manage libraries.

packihx - The hex file compressor.
This is an useful tool for packing Intel Hex files and reduces the actual code size to be downloaded to the board. The compressed hex file can be downloaded to the hardware just like the original hex file.

Usage:

```
packihx hello.ihx >hello.hex
```

OR

```
packihx hello.ihx > compressed-hello.ihx
```

2.3 Installation

The SDCC Toolchain package is shipped as a self extracting archive. All the components of SDCC are in executable formats (.exe for Windows). Refer Sec 2.2 for components of SDCC.

When unzipped at a location $(PATH) for example, the extracted directory structure is as shown in the table in the next section.

Sdcc on linux can be installed by running the following commands:

```
cd / (assuming root)
gunzip sdcc_bin.tar.gz
tar –xvf sdcc_bin.tar
```

The various installation paths are listed in the following table:

2.3.1 Install Paths

Description	Win32 Builds	Linux Builds
Binary files*	\sdcc\bin	/usr/local/bin
Include files	\sdcc\include	/usr/local/share/sdcc/include
Library file**	\sdcc\lib\xc866 (For xc866 port)	/usr/local/share/sdcc/lib/xc866
	\sdcc\lib\xc886 (For xc886 port)	
Documentation	\sdcc\doc	/usr/local/share/sdcc/doc
Examples	\sdcc\examples	

*compiler, preprocessor, assembler, and linker

** under lib\xc866, there are small and large directories for the 2 supported memory models.

** under lib\xc886, there are small and large directories for the 2 supported memory models.

2.3.2 Search Paths

The paths searched when running the compiler are as follows (the first catch wins):

2.3.2.1 Binary files (preprocessor, assembler and linker)

Search Path	Win32 Builds	Linux 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

2.3.2.2 Include Files

Search Path	Win32 Builds	Linux Builds
--I dir	--I dir	--I dir (default:/usr/local/share/sdcc/include)
$SDCC_INCLUDE	$SDCC_INCLUDE	$SDCC_INCLUDE
$SDCC_HOME/ $PREFIX2DATA_DIR/ $INCLUDE_DIR_SUFFIX	$SDCC_HOME\include	$SDCC_HOME/include
path(argv[0])/ $BIN2DATADIR/ $INCLUDE_DIR_SUFFIX	path(argv[0])\..\include	path(argv[0]/./include

2.3.2.3 Library Files

<table>
<tr><th>Search Path</th><th>Win32 Builds</th><th>Linux Builds</th></tr>
<tr><td>--L dir</td><td>--L dir</td><td>--L dir
(default: /usr/loca/share/sdcc/lib/xc866)</td></tr>
<tr><td>$SDCC_LIB/
<model></td><td>$SDCC_LIB\ xc866\
<model></td><td>$SDCC_LIB/ xc866/<model>
(default: /usr/loca/share/sdcc/lib/xc866/
<model>) where model is {small,large}</td></tr>
<tr><td rowspan="2">$SDCC_HOME/
$PREFIX2DATA_DIR/
$LIB_DIR_SUFFIX/
<model></td><td>$SDCC_HOME\lib\
xc866\<model></td><td rowspan="2">$SDCC_HOME/lib/ xc866/<model></td></tr>
<tr><td>$SDCC_HOME\lib\
xc886\<model></td></tr>
<tr><td rowspan="2">path(argv[0])/
$BIN2DATADIR/
$LIB_DIR_SUFFIX/
<model></td><td>path(argv[0])\
..\lib\xc866\<model></td><td rowspan="2">path(argv[0])\/
../lib\xc866x<model></td></tr>
<tr><td>path(argv[0])\
..\lib\xc886\<model></td></tr>
</table>

** <model> under xc866 and xc886, there are **small** and **large** directories for the 2 supported memory models.

3 Using SDCC

3.1 Compiling

3.1.1 Single Source File Projects

For single source file XC800 projects, the process is very simple. Compile your programs with the following command:

"sdcc sourcefile.c"

*Note: Please read the section **Processor Selection Options** for differentiation between processors.The default is xc866. So it does not need to be added in the command line.*

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 the linkage editor
- **sourcefile.sym** - symbol listing for the sourcefile, created by the assembler
- **sourcefile.rel** or **sourcefile.o** - Object file created by the assembler, input to the Linkage editor
- **sourcefile.map** - The memory map for the load module, created by the Linker
- **sourcefile.mem** - A file with a summary of the memory usage
- **sourcefile.ihx** - The load module in Intel hex format (you can select the Motorola S19 format with --out-fmt-s19. If you need another format, you might want to use objdump or srecord). Both formats are documented in the documentation of srecord
- **sourcefile.adb** - An intermediate file containing debug information needed to create the .cdb file (with --debug)
- **sourcefile.cdb** - An optional file (with --debug) containing debug information. The format is documented in cdbfileformat.pdf.
- **sourcefile.** - (no extension) An optional AOMF or AOMF51 file containing debug information (generated with option --debug). The (Intel) absolute object module format is commonly used by third party tools (debuggers, simulators, emulators)
- **sourcefile.dump*** - Dump file to debug the compiler itself (generated with option --Dumpall)

3.1.2 Projects with Multiple Source Files

Multiple source file can be compiled with the following option

sdcc a.c b.c –o out.ihx (Compile a.c and b.c to generate output out.ihx file)

3.1.3 Projects with Additional Libraries

Some reusable routines may be compiled into a library. See the documentation for the assembler and linkage editor (which are in <installdir>/doc) for how to create a .lib library file. Reference to these documents can be found at the end of this document. Libraries created in this manner can be included in the command line. Make sure you include the -L <library-path> option to tell the linker where to look for these files if they are not in the current directory.

Here is an example, assuming you have the source file *foomain.c* and a library *foolib.lib* in the directory mylib (if that is not the same as your current project):

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

Note 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>/lib/small.

3.1.4 Using sdcclib to Create and Manage Libraries

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

To display a list of options supported by sdcclib type:

```
sdcclib -?
```

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

For example:

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

This will create files _divsint.rel, _divuint.rel, _modsint.rel, _moduint.rel, and _mulint.rel.

The next step is to add the .rel files to the library file:

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

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

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

If the source files are compiled using --debug, the corresponding debug information file .adb will be included in the library file as well. The library files created with sdcclib are plain text files, so they can be viewed with a text editor. It is not recommended to modify a library file created with sdcclib using a text editor, as there are file index numbers located across the file

used by the linker to quickly locate the required module to link. Once a .rel file (as well as a .adb file) is added to a library using sdcclib, it can be safely deleted, since all the information required for linking is embedded in the library file itself. Library files created using sdcclib are used as described in the preceding sections.

3.2 Command Line Options

3.2.1 Processor Selection Options

-mxc866 Generate code for XC866 microcontroller. This is the default option.

-mxc886 Generate code for XC886 microcontroller.

3.2.2 Preprocessor Options

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

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

-M Tell the preprocessor to output a rule suitable for 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 for question, in case it is tested with a preprocessor condition such as `#if #question(answer)'. `-A-' disables the standard assertions that normally describe the target machine.

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

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

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

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

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

3.2.3 Linker Options

-L --lib-path <absolute path to additional libraries> This option is passed to the linkage editor's additional libraries search path. The path name must be absolute. Additional library files may be specified in the command line.

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

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

--stack-loc <Value> By default the stack is placed after the data segment. Using this option the stack can be placed anywhere in the internal memory space of the Xc800. The value entered can be in Hexadecimal or Decimal format, e.g. --stack-loc 0x20 or --stack-loc 32. Since the sp register is incremented before a push or call, the initial sp will be set to one byte prior to the provided value. The provided value should not overlap any other memory areas such as used register banks or the data segment and with enough space for the current application.

--data-loc <Value> The start location of the internal ram data segment. The value entered can be in Hexadecimal or Decimal format, e.g. --data-loc 0x20 or --data-loc 32. (By default, the start location of the internal ram data segment is set as low as possible in memory, taking into account the used register banks and the bit segment at address 0x20.

For example, if register banks 0 and 1 are used without bit variables, the data segment will be set, if --data-loc is not used to location 0x10.)

--idata-loc <Value> The start location of the indirectly addressable internal ram of the Xc800; default value is 0x80. The value entered can be in Hexadecimal or Decimal format, e.g. --idata-loc 0x88 or --idata-loc 136.

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

-Wa asmOption[,asmOption]... Pass the asmOption to the assembler. See file SDCC/docs/ SDCC_XC866_ASM_LNK_UserManual.doc for more information on assembler options.

-Wl linkOption[,linkOption]... Pass the linkOption to the linker. See file SDCC/docs/ SDCC_XC866_ASM_LNK_UserManual.doc for more information on linker options.

3.2.4 XC800 Options

--model-small Generate code for Small Model programs. This is the default model.

--model-large Generate code for Large model programs. If this option is used, all source files in the project have to be compiled with this option.

--xstack Uses a pseudo stack in the first 256 bytes in the external ram for allocating variables and passing parameters. See section 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 a minimum of <Value> bytes for stack.

--uart-init Tells the Compiler to generate code for initializing UART in the startup code.

3.2.5 Optimization Options

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

--noinvariant Will not do loop invariant optimizations; this may be turned off for reasons explained for the previous option. For more details of loop optimizations performed, see section on Loop Optimizations. It is recommended that this option should 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 on Loop Optimizations for more details. It is recommended that this option is NOT used; #pragma noinduction can be used to turn off induction optimizations for a given function only.

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

--noloopreverse Will not do loop reversal optimization.

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

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

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

--no-peep Disable peep-hole optimization.

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

3.2.6 Other Options

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

--c1mode It 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 a standard output.

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

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

--callee-saves function1[,function2][,function3].... The compiler by default uses a caller saves convention for register saving across function calls; however this can cause unnecessary register pushing 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 & 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 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 files should be compiled with the same --callee-saves option string. Also see #pragma callee_saves.

--debug When this option is used, the compiler will generate debug information. The debug information collected in a file with .cdb extension can be used with the SDCDB. Another file with no extension contains debug information in OMF51 format which is commonly used by third party tools.

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

--int-long-reent Integer (16 bit) and long (32 bit) libraries have been compiled as reentrant. Note by default that 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 the section on Cyclomatic Complexity for more details.

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

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

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

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

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

-V Shows the actual commands the compiler is executing.

--no-c-code-in-asm Hides the c-code from the asm file.

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

--less-pedantic Disable some of the more pedantic warnings. If you want more warnings, you should consider using a separate tool dedicated to syntax checking like splint [www.splint.org].

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

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

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

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

3.2.7 Intermediate Dump Options

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

--dumpraw This option will cause the compiler to dump the intermediate code into a file of named <source filename>.dumpraw just after the intermediate code has been generated for a function, i.e. before any optimizations are done. The basic blocks at this stage ordered in the depth first number, so they may not be in sequence of execution.

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

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

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

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

3.2.8 Redirecting output on Windows Shells

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

Something like this should work:

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

3.3 SDCC Wrapper options

SDCC wrapper has its own options apart from all the sdcc1 compiler options.

--sdccverbose Will show the commands being executed

--sdccsavetemps Will save the temporary files. By default this option is off.

3.4 Examples on SDCC Usage

Invocation	Explanation
sdcc a.c –o a.ihx	Compile a.c and generate hex file a.ihx
sdcc a.c b.c –o out.ihx	Compile a.c and b.c to generate output out.ihx file
sdcc –nolabelopt a.c –o out.ihx	Compile a.c with the option –nolabelopt option
sdcc –noinvariant –d a.c b.asm	Compile a.c with – noinvariant and b.asm with –d option
sdcc a.c b.rel	Compile a.c to generate a.rel. Link a.rel and b.rel to generate the output file.

3.5 Environment variables

SDCC recognizes the following environment variables:

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

TMP, TEMP, TMPDIR The 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 is no need to set them. On Windows, it is recommended to set one of them.

SDCC_HOME The path is the directory where sdcc is installed.See section on Installation.

SDCC_INCLUDE The path is the path to the include directory .See section on Installation

SDCC_LIB The path is the path to the lib directory. See section on Installation.

3.6 Data types

Supported data-types are:

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

3.7 Storage Class Language Extensions

3.7.1 XC800 Storage Class Language Extensions

In addition to the ANSI storage classes, SDCC allows the following XC800 specific storage classes:

3.7.1.1 data / near

This is the default storage class for the Small Memory model (data and near can be used synonymously). Variables declared with this storage class will be allocated in the directly addressable portion of the internal RAM e.g.:

```
data unsigned char test_data;
```

Writing 0x01 to this variable generates the assembly code:

```
75*08 01   mov  _test_data,#0x01
```

3.7.1.2 xdata / far

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

```
xdata unsigned char test_xdata;
```

Writing 0x01 to this variable generates the assembly code:

```
90 00 00   mov  dptr,#_test_xdata
74 01      mov  a,#0x01
F0         movx @dptr,a
```

3.7.1.3 idata

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

```
idata unsigned char test_idata;
```

Writing 0x01 to this variable generates the assembly code:

```
78 80       mov  r0,#_test_idata
76 01       mov  @r0,#0x01
```

Please note that the first 128 bytes of idata physically accesses the same RAM as the data memory. The stack is located in idata memory.

3.7.1.4 pdata

It is typically located at the start of xdata and has a maximum size of 256 bytes. The following example writes 0x01 to the address pointed to. Please note that pdata physically accesses xdata memory. The high byte of the address is determined by port P2.

```
pdata unsigned char *test_pdata_ptr;

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

Generates the assembly code:

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

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.

3.7.1.5 code

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

```
code unsigned char test_code;
```

Read access to this variable generates the assembly code:

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

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

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

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

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

3.7.1.6 bit

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

3.7.1.7 sfr / sbit

Like the bit keyword, sfr / sbit signifies both a data-type and a storage class. They are used to describe the special function registers and special bit variables; eg:

```
sfr at 0x80 P0;  /* special function register P0 at
location 0x80 */
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.

3.7.1.8 Pointers to XC800 specific memory spaces

SDCC allows (via language extensions) pointers to explicitly point to any of the memory spaces of XC800. 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 toobject in internal ram */
data unsigned char * xdata p;

/* pointer physically in code rom pointing to data in xdata space */
xdata unsigned char * code p;
```

```
/* pointer physically in code space pointing to data in code space */
code unsigned char * code p;

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

All unqualified pointers are treated as 3-byte generic pointers.

The highest order byte of the generic pointers contains 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.

Note: Explicit type-cast is needed to convert a memory specific pointer to a generic pointer.

eg:

```
int *p1;
int xdata *p2;
int idata *p3;

p1 = p2;
p3 = (idata*)p1; /* (generic*) to (idata*) */
```

3.8 Absolute Addressing

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

```
xdata at 0x7ffe unsigned int chksum;
```

In the above example, the variable chksum will located at 0x7ffe and 0x7fff of the external RAM. The compiler does not reserve any space for variables declared in this way (they are implemented with an equate in the assembler). Thus it is left to the programmer to make sure there are no overlaps with other variables that are declared without the absolute address. The assembler listing file (.lst) and the linker output files (.rst and .map) are good places to look for such overlaps. Variables with an absolute address are not initialized.

For example, if we have declarations for two variables at 0x0022 and 0x0024, each 2 bytes in length, a workaround would be to tell the compiler to start the data location from 0x0026 by using the --data-loc 0x0026 option.

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

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

For some architectures, array accesses are more efficient if an (xdata/far) array starts at a block (256 byte) boundary.

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

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

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

For example, if you have a routine that uses one or more of the microcontroller I/O pins, and such pins are different for two different hardwares, you can declare the I/O pins in your routine using:

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

void C800_put(unsigned char value)
{
    unsigned char mask=0x80;

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

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

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

Similarly, for the second hardware you would use:

```
bit at 0x83 SDI;    /* I/O port 0, bit 3 */
```

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

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.

3.9 Parameters & Local Variables

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

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

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

Since stack space on XC800 is limited, the reentrant keyword or the --stack-auto option should be used sparingly. Note that the reentrant keyword just means that the parameters & local variables will be allocated to the stack, it does not mean that the function is register bank independent.

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

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

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

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

3.10 Overlaying

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

Note: 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.

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

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

```
#pragma save
#pragma nooverlay
void set_error(unsigned char errcd)
{
    P3 = errcd;
}
#pragma restore

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

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

3.11 Interrupt Service Routines

3.11.1 General Information

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

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

The number following the interrupt keyword is the interrupt number this routine will service. The compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. The range of interrupt numbers allowed are from 0 to 14.

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 keyword can be used to tell the compiler to use the specified register bank (XC800 specific) when generating code for this function. The range of register banks allowed are from 0 to 3.

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

- If the interrupt service routines change variables which are accessed by other functions, these variables should be declared volatile.
- If the access to these variables is not atomic (i.e. the processor needs more than one instruction for the access and could be interrupted while accessing the variable), the interrupt must 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.

Note: ***int (16 bit) and long (32 bit) integer division, multiplication & modulus and floating-point operations are implemented using external support routines developed in ANSI-C. If an interrupt service routine needs to do any of these operations, then the support routines (as mentioned in a following section) will have to be recompiled using the --stack-auto option and the source file needs to be compiled using the --int-long-reent compiler option.***

Calling other functions from an interrupt service routine is not recommended and 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. Further, non-reentrant functions should not be called from the main program while the interrupt service routine is active.

Also see the section on Overlaying and section on Functions, using private register banks.

3.11.2 XC800 Interrupt Service Routines

For a detailed description of XC800, please read the Architecture manual. SDCC will automatically adjust the interrupt vector table to the maximum interrupt number specified.

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 and 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 outines.

3.12 Functions using private register banks (xc800)

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. 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 any way.

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, this means that if a high-priority ISR using a particular bank occurs while processing a low-priority ISR using the same bank, terrible 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. Use the default bank zero and take a small performance hit.

It is most efficient if your ISR calls no other function. If your ISR must call other functions, it is most efficient if those functions use the same bank as the ISR; 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.

3.13 Critical Functions and Critical Statements

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

Example:

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

The critical attribute can be 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.

3.14 Startup Code

The startupxc866.s file contains the startup code for the XC800 target program. This source file is located in the lib/xc866 directory for xc866 and in lib/xc886 for xc886. The compiler inserts a call to the C routine *_sdcc_external_startup()* at the start of the CODE area. This routine is present in the startupxc866.s file for xc866 and the startupxc886.s file for xc886, which is used to setup the hardware or perform some other operation prior to static and global variable initialization. Include a copy of this file in each xc800 project that needs custom startup code.

3.14.1 XC800 Startup Code

The user can edit the Startup file provided with SDCC to put in his own code.
The following can be done through the startup file:

- Clearing/initializing the external data memory
- Initializing the *uart_init* function
- Any other initializations which the user wants to be executed on target reset

Assembler Constant	Description
UART	If this is set to a value greater than 0, the user can do *uart* initialization in the startup instead of using the *–uart-init* option in the command line. *Note: –uart-init option and this initialization should not be used simultaneously.*
CLEAR_XDATA_ADDR	This is the start address of the xdata memory which the user wants to clear/initialize
CLEAR_XDATA_LEN	This is the length of the *xdata* which the user wants to clear/initialize starting from CLEAR_XDATA_ADDR

The *startupxc866.s* file for xc866 and *startupxc886.s* file for xc886 provide the user with the above assembly constants that the user may change to control the actions taken at startup.

The SDCC compiler does not check whether the assembler constants provided, are correctly specified. The user must ensure that these parameters and any other parameters defined in the startup file, contain suitable values.

3.15 Inline Assembly

3.15.1 A Step by Step Introduction

Starting from a small snippet of c-code, this example shows how to use inline assembly, access variables, a function parameter and an array in xdata memory. This is a buffer routine which should be optimized:

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

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

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

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

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

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

#if !USE_ASSEMBLY

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

#else
```

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

The inline assembler code can contain any valid code understood by the assembler and this includes any assembler directives and comment lines . 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 the register pushing/popping has to be done manually.

It is recommended that each assembly instruction (including labels) be placed in a separate line (as the example shows). When the --peep-asm command line option is used, the inline assembler code will be passed through the peephole optimizer. There are only a few (if any) cases where this option makes sense. It might cause some unexpected changes in the inline assembler code. Please go through the peephole optimizer rules defined in file SDCCpeeph.def before using this option.

3.15.2 Naked Functions

A special keyword may be associated with a function declaring it as _naked. The _naked function modifier attribute prevents the compiler from generating prologue and epilogue code for that function. This means that the user is entirely responsible for such things as saving any registers that may need to be preserved, selecting the proper register bank, generating the return instruction at the end, etc. Practically, this means that the contents of the function must be written in the 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 XC800 target, the generated simpleInterrupt looks like:

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

whereas *nakedInterrupt* looks like:

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

The related directive #pragma exclude allows a more fine grained control over pushing and 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 the inline assembler.

3.15.3 Use of Labels within Inline Assembler

SDCC allows the use of in-line assembler with a few restrictions regarding labels:

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

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

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

In other words, inline assembly code can access labels defined in inline assembly within the scope of the function. The same goes the other way, i.e. labels defined in inline assembly can not be accessed by C statements.

3.16 Interfacing with Assembler Code

3.16.1 Global Registers used for Parameter Passing

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

3.16.2 Assembler Routine (non-reentrant)

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

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

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

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

The corresponding assembler function is:

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

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

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

Assemble the assembler routine with the following command:

```
as-xc800 -losg asmfunc.asm
```

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

```
sdcc cfunc.c asmfunc.rel
```

3.17 int (16 bit), long (32 bit) and Floating point support

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

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

Since they are compiled as non-reentrant, interrupt service routines should not do any of the above operations. If this is unavoidable, then the above routines will need to be compiled with the --stack-auto option, after which the source program will have to be compiled with --int-long-reent option. Notice that you don't have to call these routines directly. The compiler will use them automatically every time an integer operation is required.

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

Function	Description
__fsadd.c	add floating point numbers
__fssub.c	subtract floating point numbers
__fsdiv.c	divide floating point numbers
__fsmul.c	multiply floating point numbers
__fs2uchar.c	convert floating point to unsigned char
__fs2char.c	convert floating point to signed char
__fs2uint.c	convert floating point to unsigned int
__fs2int.c	convert floating point to signed int
__fs2ulong.c	convert floating point to unsigned long
__fs2long.c	convert floating point to signed long
__uchar2fs.c	convert unsigned char to floating point
__char2fs.c	convert char to floating point number
__uint2fs.c	convert unsigned int to floating point
__int2fs.c	convert int to floating point numbers
__ulong2fs.c	convert unsigned long to floating point number
__long2fs.c	convert long to floating point number

Note: Float operations consume more memory. So, code using float data types should be compiled in model large. Float data types should be used cautiously.

3.18 Library Routines

XC886/XC888 boot ROM contains arithmetic and floating point library. To make use of it in SDCC include the file xc88x_bromlib.s in your project.The source file(a.c) which uses boot ROM library has to be compiled with the following command line option

sdcc -mxc886 --uart-init --model-large --xram-loc 0xf000 --xram-size 1536 a.c xc88x_bromlib.s

This will automatically direct the call to the boot ROM library. xc88x_bromlib.s file has to be used only in model large.

Note: List of the SDCC library is in sdcc\doc\libdoc.txt

Note: List of boot ROM library used by SDCC is in Appendix C. xc88x_bromlib.s *file is in sdcc\lib\xc886*xc88x_bromlib.s

3.18.1 Compiler support routines (_gptrget, _mulint etc.)

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

SDCC supports the following printf formats:

```
%d              int
%f              float
%u              unsigned int
%ld             long
%lu             unsigned long
%o              octal
%lo             long octal
%x              hex
%lx             long hex
%c              character
```

3.18.2.1 <stdio.h>

As usual, on embedded systems you have to provide your own *getchar()* and *putchar()* routines. If you're short on memory, you might want to use *printf_small()* instead of *printf()*.There is an assembly version *printf_fast()* which should fit the requirements of many embedded systems (by unsetting #defines, it can be customized to not support long variables and field widths).

If you want to print floats with more than 6 digit precision, use *printf_highfloats()* instead of *printf()*, since *printf()* is limited to 6 digit precision.

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

3.18.4 Other libraries

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

3.19 XC800 Memory Models

3.19.1 Small Model

In this model, all variables by default, reside in the internal data memory of the xc866 system. This is the same as if they were declared explicitly using the data memory type specifier. In

this memory model, variable access is very efficient. This is the default model in which the compiler works. To operate the compiler in this model, the **-- model-small** option should be used.

3.19.2 Large Model

In the large model, all variables by default, reside in external data memory (up to 64Kbytes). This is the same as if they were explicitly declared using the **xdata** memory type specifier. The data pointer (DPTR) is used for addressing.

To operate the compiler in this model the **--model-large** option should be used. Libraries compiled with --model-large should be used with this model. Libraries compiled with –model-large are provided as part of the Toolchain package. These libraries can be found in the directory:

`sdcc\lib\xc866\large.`

3.19.3 XC800 Address Spaces

The XC800 CPU manipulates operands in the following five address spaces:

- 16 KByte of Flash program memory for xc866 and 32KBytes for xc886
- 8 KByte of Boot ROM program memory
- 256 bytes of internal RAM data memory
- 512 bytes of XRAM memory for xc866 and 1536 bytes for xc886
  (XRAM can be read/written as program memory or external data memory)
- A128-byte special function register area

3.19.3.1 General Purpose Registers

The lower 32 locations of the internal RAM are assigned to four banks with eight general purpose registers (GPRs) each. Only one of these banks may be enabled at a time. Two bits in the program status word, RS0 and RS1, select the active register bank. This allows fast context switching, which is useful when entering subroutines or interrupt service routines.

The 8 general purpose registers of the selected register bank may be accessed by register addressing. With register addressing, the instruction opcode indicates which register is to be used. For indirect addressing, R0 and R1 are used as pointers or index registers to address internal or external memory (e.g. MOV @R0).

Reset initializes the stack pointer to location 07H and increments it once to start from location 08H which is also the first register (R0) of register bank 1. Thus, if one is going to use more than one register bank, the SP should be initialized to a different location of the RAM which is not used for data storage.

3.19.3.2 Program and Data Memory Organization

The XC800 can operate in four different operating modes (chipmodes) with different program and data memory organizations:

- Normal Mode

  This is the normal operating mode for the compiler.

eg: sdcc <filename.c>

- Normal XRAM Mode

  This mode is activated only in the Large Memory Model.

eg: sdcc –model-large <filename.c>

- Bootstrap Mode

- Bootstrap XRAM Mode

  The sdcc compiler does not operate in these two modes.

3.20 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, therefore when using the External Stack option, this port may not be used by the application program.

3.21 Pragmas

SDCC supports the following #pragma directives:

- **save** - This will save all current options to the save/restore stack. See restore.
- **restore** - This will restore saved options from the last save. Saves and restores can be nested. SDCC uses a save/restore stack: save pushes current options to the stack, and restore pulls current options from the stack. See save.
- **nogcse** - This will stop global common sub-expression elimination.
- **noinduction** - This will stop loop induction optimizations.
- **nojtbound** - This will not generate the code for the boundary value checking when switch statements are turned into jump-tables (dangerous).
- **nooverlay** - The compiler will not overlay the parameters and local variables of a function.
- **less_pedantic** - The compiler will not warn you anymore for obvious mistakes
- **noloopreverse** - This will not do loop reversal optimization
- **exclude none | {acc[,b[,dpl[,dph]]]** - The exclude pragma disables generation of the pair of push/pop instructions in ISR function (using interrupt keyword). The directive should be placed immediately before the ISR function definition and it affects ALL ISR functions following it. To enable the normal register saving for ISR functions, use #pragma exclude none.
- **noiv** - Do not generate interrupt vector table entries for all ISR functions defined after the pragma. This is useful in cases where the interrupt vector table must be defined manually, or when there is a secondary, manually defined interrupt vector table (e.g. for the autovector feature of the Cypress EZ-USB FX2). More elegantly, this can be achieved by omitting the optional interrupt number after the interrupt keyword.
- **callee_saves function1[,function2[,function3...]]** - The compiler by default uses a caller saves convention for the 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 and extra code needs 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 the code and improve run time performance of the generated code. In future, the compiler (with inter procedural analysis) may be able to determine the appropriate scheme to use for each function call. If --callee-saves command line option is used, the function names specified in #pragma callee_saves is appended to the list of functions specified in the command line.

SDCPP supports the following #pragma directives:

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

The pragmas are intended to be used to turn-on or turn-off certain optimizations which might cause the compiler to generate extra stack / data space to store compiler generated temporary

variables. This usually happens in large functions. Pragma directives should be used as shown in the following example; they are used to control options and optimizations for a given function; pragmas should be placed before and/or after a function; placing pragmas inside a function body could have unpredictable results.

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

The compiler will generate a warning message when extra space is allocated. It is strongly recommended that the save and restore pragmas be used when changing options for a function.

3.22 Defines Created by the Compiler

The compiler creates the following #defines:

#define	Description
SDCC	this Symbol is always defined
SDCC_xc866, etc	depending on the model used (e.g. -mxc866)
SDCC_STACK_AUTO	when --stack-auto option is used
SDCC_MODEL_SMALL	when --model-small is used
SDCC_MODEL_LARGE	when --model-large is used
SDCC_USE_XSTACK	when --xstack option is used

4 Notes on SDCC support for XC800

4.1 SDCC for XC800 port

Assembler supports the extra instructions supporting the new variant of MOVC instruction and the TRAP instruction. Macros for these instructions are provided by the compiler. Currently the compiler generates code with just one DPTR.

4.1.1 Limitations of SDCC for XC800

- The compiler still does not support generation of code using two data pointers (DPSEL register). The assembler identifies DPSEL register and generates the code accordingly.
- Compiler does not generate code for the new variant of MOVC instruction. This is supported by the macro defined in the xc866.h file.
- TRAP instruction is emitted only by the macro defined in the xc866.h file.
- Extern variables cannot be assigned before its extern declaration.
- Programs requiring dynamic memory allocation using “malloc()” should be compiled in model large.
- No support for Signed Bit fields.

4.1.2 Using printf functions

All variants of printf(), like printf(), vprintf() etc. are implemented as library. Before any variant of printf() is used, UART has to be initialized. To initialize UART, uart_init() function is provided in the SDCC package. This init function needs to be called only once at the beginning. Also when using printf(), header files stdio.h and xc866.h for xc866 (xc886.h for xc886 and xc856.h for XC856) have to be included.

Note: Command line option ‘--uart-init’ can be used instead of explicit call to uart_init().

4.1.2.1 A Sample program using printf()

```
/*
 * A Sample program demonstrating the usage of printf() for XC800
 */
#include <stdio.h>
#include <xc866.h> /*Use <xc886.h> here, for xc886*/

void main()
{
volatile char a = 2, b = 4, c;//Variable declaration
/*UART Initialization done only once for printf()s - optional if using
–uart-init for compilation.*/
uart_init();
c = a + b;
if (c != 6) {
printf("Char Add FAILED \r\n");
```

```
return;
}
printf("Char Add PASS \r\n"); return;
}
```

4.1.2.2 A Sample program using Large Memory model for XC856

```
/*
 * A Sample program demonstrating the usage of  large memory model for XC856.
 */

/* HelloWorld.c */
#include <stdio.h>
#include <xc856.h>  /* Header file to be included for XC856 */

void main()
{
        printf("Hello World \r\n");
}

/* startupxc886.s */

; STARTUP FILE for USER CUSTOMIZATION

.area startup_code (code)

.globl __sdcc_external_startup
__sdcc_external_startup:

; The following two lines should be uncommented for XC856 Large Memory Model
MEX3 = 0x96             ;SFR for XC856
MOV MEX3, #0x1F         ; To access On-Chip XRAM of XC856

ret                     ;End of __sdcc_external_startup
```

The command to build the hex file for the above example in large model for XC856 is :

```
Sdcc -mxc886 --uart-init HelloWorld.c startupxc886.s --xram-loc 0xf000 --xram-size 512
```

NOTE: Please note that the compiler target switch -mxc886 has to be used for XC856 too. However, xc856.h header file should be included in the test case if it has to be built for XC856.

4.1.3 Using Dynamic Memory in XC800

For using malloc in any program,

```
init_dynamic_memory(MEMHEADER xdata *  , unsigned int );
(mem area)                    (size)
```

has to be called.

Another point for using malloc, is to always use “xdata” as shown below:

4.1.3.1 A Sample program using dynamic memory allocation

```
/*
 * A Sample program demonstrating the usage of malloc for XC800 SDCC
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <xc866.h> /*Use <xc886.h> here, for xc886*/
#include<stdlib.h>

#define DYNAMIC_MEMORY_SIZE 128
unsigned char xdata dynamic_memory_pool[DYNAMIC_MEMORY_SIZE];
unsigned char  * xdata src_str;
unsigned char  * xdata dst_str;

                 void main(void)
                 {
         init_dynamic_memory((MEMHEADER    xdata    *)dynamic_memory_pool,
DYNAMIC_MEMORY_SIZE);
        uart_init();   //optional if using –uart-init for compilation
                     src_str = malloc(10);
                     src_str  = “Hello”
        dst_str = malloc(10);
                     strcpy(dst_str, src_str);
        printf("The copied string is = %s,\r\n", dst_str);

}
```

4.1.4 ANSI-Compliance

Deviations from the compliance:

- Functions are not always reentrant.
- All Recursive functions MUST be declared as reentrant.
- Structures cannot be assigned values directly, cannot be passed as function parameters or assigned to each other and cannot be a return value from a function,

e.g.:

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

As a workaround:

- Members of a structure can be assigned values directly
- Pointer to a structure can be passed as a function parameter
- Pointer to a structure can be a return value from a function

e.g.:

```
#include <xc866.h>
#include <stdio.h>

typedef struct {
  int no_of_lines;
  char *info;
} X;

X * max(X *x, X *y)
{
  if (x->no_of_lines >= y->no_of_lines)
    return x;

  return y;
}

void main (void)
{
  X x1, x2, *m;

  x1.no_of_lines = 100;
  x2.no_of_lines = 200;

  m =  max(&x1, &x2);
```

```
  uart_init();

  printf("No of lines of code in x1 is %d \r\n", x1.no_of_lines);
  printf("No of lines of code in x2 is %d \r\n", x2.no_of_lines);

  printf("Max No of lines of code is %d \r\n", m->no_of_lines);

}
```

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

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

- functions declared as pointers must be dereferenced during the call

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

4.1.5 Signed-ness of bitwise operators

According to section 6.5 of C9X standard (References chapter), the bitwise operators (~, <<, >>, &, ^ and |) are required to have operands that have integer type. These operators return values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

In SDCC for XC800, the implementation for the result of the bitwise operators has unsigned types.

4.1.6 Guidelines to generate efficient code

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

- 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'. 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). Although character types (especially unsigned char) can be used as "tiny" integers, doing so is sometimes more trouble than it is worth, due to unpredictable sign extensions and increased code size.
- Use unsigned when it is known in advance that the value is not going to be negative. This helps especially if you are doing division or multiplication, bit-shifting or are using an array index.
- NEVER jump into a LOOP.
- Declare the variables to be local whenever possible, especially loop control variables (induction).
- Since the compiler does not always do implicit integral promotion, the programmer should do an explicit cast when integral promotion is required.
- Reducing the size of division, multiplication and modulus operations can reduce the code size substantially.

Take the following code for example.

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

For the modulus operation, the variable ch will be promoted to unsigned int first; then the modulus operation will be performed (this will lead to a call to support routine _moduint()), and the result will be casted to a char.

If it is changed to:

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

Then it would substantially reduce the code generated (future versions of the compiler will be smart enough to detect such optimization opportunities).
Have a look at the assembly listing, to get a "feel" of the code generation.

5 SDCC Technical Data

5.1 Optimizations

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

5.1.1 Sub-expression Elimination

The compiler does local and global common sub-expression elimination e.g.:

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

will be translated to:

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

Some subexpressions are not as obvious as the above example:

```
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 the registers.

5.1.2 Dead-Code Elimination

```
int global;

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

```
  global = 3; /* unreachable */
}
```

will be changed to:

```
int global;

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

5.1.3 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 the dead-code elimination.

5.1.4 Loop Optimizations

Two types of loop optimizations are done by the 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 the registers for the duration of the loop. Because of this preference of the register allocator, loop induction optimization causes an increase in the 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 the 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 and 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.

5.1.5 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.

5.1.6 Algebraic Simplifications

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

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

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

5.1.7 'switch' Statements

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

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

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

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

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

The number of case labels is at least three, since it takes two conditional statements to handle the boundary conditions.

The number of case labels is less than 84, since each label takes 3 bytes and a jump-table can be utmost 256 bytes long.

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

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

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

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

and

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

then both the switch statements will be implemented using jump-tables, whereas the unmodified switch statement will not be implemented. You might also consider inserting dummy cases 0 and 5 to 8 in this example. The pragma nojtbound can be used to turn off checking the jump table boundaries. It has no effect if a default label is supplied. Use of this pragma is dangerous; if the switch argument is not matched by a case statement, the processor will happily jump into Nirvana.

5.1.8 Bit-shifting Operations

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

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

generates the following code:

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

In general, SDCC will never setup a loop if the shift count is known.

Another example:

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

will generate:

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

5.1.9 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 */
```

5.1.10 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: SDCC stores numbers in little-endian. Usually, 8-bit processors do not care much about endianness.

5.1.11 Highest Order Bit

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

e.g.:

```
unsigned int gint;

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

```
  ..
}
```

will generate the following code:

```
                          61 ;  hob.c 7
000A E5*01                62          mov   a,(_gint + 1)
000C 23                   63          rl    a
000D 54 01                64          anl   a,#0x01
000F F5*02                65          mov   _foo_hob_1_1,a
```

Variations of this case however will not be recognized. It is a standard C expression, so it is recommended that this be the only way to get the highest order bit, (it is portable). Of course it will be recognized even if it is embedded in other expressions,
e.g.:

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

will still be recognized.

5.1.11.1 Peephole Optimizer

The compiler uses a rule based, pattern matching and re-writing mechanism for peep-hole optimization. A default set of rules are compiled into the compiler and 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.

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 MCUs 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 and it will terminate when all the 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) and it is intended to be used in such 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: 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 as illustrated below:

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

The optimizer does a look-up of a function name table defined in function callFuncByName in the source file SDCCpeeph.c, with the name labelInRange. If it finds a corresponding entry, the function is called. Note there can be no parameters specified for these functions; in this case the use of %5 is crucial, since the function labelInRange expects to find the label in that particular variable (the hash table containing the variable bindings is passed as a parameter). If you want to code more such functions, take a close look at the function labelInRange and the calling mechanism in the source file SDCCpeeph.c.

5.2 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 of 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 simplifying the function logic.

Note: 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 may be unavoidable to have a complexity level of less than 10.

For example, if you have switch statements 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.

6 Tools included in the distribution

Name	Purpose	Directory in Windows	Directory in Linux
sdcc	Wrapper to compile multiple source files using the compiler sdcc1	sdcc/bin	/usr/local/bin
sdcc1	The actual sdcc compiler executable supporting XC800	sdcc/bin	/usr/local/bin
as-xc800	Assembler of XC800	sdcc/bin	/usr/local/bin
link-xc800	Linker of XC800	sdcc/bin	/usr/local/bin
sdcclib	SDCC librarian	sdcc/bin	/usr/local/bin
packihx	ihx packe	sdcc/bin	/usr/local/bin

A XC800 Specific Instruction Generation

The XC800 instruction set has the following two specific instructions:

- MOVC @(DPTR++),A
- TRAP

Both these instructions have the machine code of A5 and the selection depends on the TRAP_EN bit in E0 register.

```
TRAP_EN = 0: Select MOVC @(DPTR++),A
TRAP_EN = 1: Select software TRAP instruction
```

The ‘MOVC @(DPTR++),A’ instruction is used to download software into the program memory where this is implemented as RAM. This instruction can also be used subsequently to modify the contents of the program memory. The ‘MOVX’ instructions are used to access the XRAM as external data memory.

The SDCC compiler cannot generate these instructions from the intermediate code, but it supports these with inline assembly macros. The assembler accordingly generates the code for these instructions.

Note: It is the responsibility of the programmer to set the TRAP_EN bit accordingly.

The corresponding macros are as follows:

- MOVC @(DPTR++),A MOVC_WRITE
- TRAP TRAP

To use these macros, include the xc866.h and the xc886.h files respectively.

B Example XC866 code for SDCC

```
#include<xc866.h>

void blinkled(void)
{
int j=10, c;
char i = 0;


P3_DIR = 0xFF;                      // P3_DIR is defined in xc866.h

while (1)
{
for (c=0;c<50100;c++)
{
j++;
}
P3_DATA = i;                                // P3_DATA is defined in xc866.h
i = ~i;

}
}
```

C List of boot ROM library used by SDCC

__mulint: int multiply
__divuint: unsigned int divide
__moduint: unsigned int Modulus
__mullong:signed/unsigned long multiply
__divulong:unsigned long divide
__modulong:unsigned long modulo
___fsadd: float addition
___fssub: float subtract
___fsmul: float multipy
___fsdiv:float divide
___fseq:float compare
___schar2fs:type cast: signed char to float
___uchar2fs:type cast: unsigned char to float
___sint2fs:type cast: signed int to float
___uint2fs:type cast: unsigned int to float
___slong2fs:type cast: signed long to float
___ulong2fs:type cast: unsigned long to float
___fs2uint:type cast: float to unsigned int
___fs2sint:type cast: float to signed int
___fs2uchar:type cast: float to unsigned char
___fs2schar:type cast: float to signed char
___fs2slong:type cast: float to signed long
___fs2ulong:type cast: float to unsigned long

D Support for Silicon Bug : FLASH XC8.004.

For hardware cache miss bug, FLASH_XC8.004, a workaround has been provided in this version of sdcc. Use the option “--hwc-wa” while compiling the .c file. The workaround in a nutshell is the following: Whenever a movc operation is generated, additionally two instructions, which are nop instructions are generated before and after the movc. This option is not listed in the “usage” display while --help is used. Also, this option is valid only for xc886, i.e this option comes into effect only if -mxc886 option is used

A

Absolute Addressing 31
ANSI-C compiler 9
Assembler Routine (non-reentrant) 43

C

Command Line Options 20
Copy-Propagation 60
Critical Functions and Critical Statements 37
Cyclomatic Complexity 68

D

Data types 28
Dead-Code Elimination 59
Dynamic Memory in XC800 53

E

Environment variables 27
Example XC866 code for SDCC 75
External Stack 48

F

Functions using private register banks (xc866) 36

G

General Purpose Registers 47

I

Inline Assembly 38
Interfacing with Assembler Code 43
Intermediate Dump Options 26
Interrupt Service Routines 35

L

lib/xc866 37
Limitations of SDCC for XC800 51
Linker Options 21
Loop Optimizations 60
Loop Reversing 61

N

Notes on SDCC support for XC800 51

O

Open Source 9
Optimization Options 23
Optimizations 59

P

Parameters & Local Variables 33
pdata 29
Peephole Optimizer 66
peephole optimizer 9
Pointers to XC800 specific memory spaces 30
Pragmas 49
Preprocessor Options 20
Processor Selection Options 20
Program and Data Memory Organization 48

S

Sample program using printf() 51, 52
SDCC 9, 11
SDCC Technical Data 59
SDCC Toolchain 11
SDCC Wrapper options 26
Single Source File Projects 17
Small Device C Compiler 9
Startup Code 37
Storage Class Language Extensions 28

T

Tools included in the distribution 71

U

Use of Labels within Inline Assembler 42
Using sdcclib to Create and Manage Libraries 18

X

XC800 Address Spaces 47
XC800 Interrupt Service Routines 36
XC800 Memory Models 46
XC800 Options 22
XC800 Specific Instruction Generation 73, 77, 79
XC800 Startup Code 38
XC800 Storage Class Language Extensions 28
xc886 16

References

- The parent document of SDCC which covers all the publicly supported processors, can be found in: http://sdcc.sourceforge.net/doc/sdccman.html/
- The assembler and linker user manual (part of the compiler zip): SDCC_XC800_ASM_LNK.pdf
- ISO Standard for C (C9X) - Final committee draft