Application Note, V1.0, Sep. 2007

AP08065

XC800

Migration from Keil to SDCC

Microcontrollers

Never stop thinking

Edition 2007-09-17

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

AP08065		
Revision History:	2007-09	V1.0
Previous Version:	none	

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents Page

1 Introduction & Scope

SDCC for XC800 family and Keil compilers are both ANSI C compliant. The main motivation for the migration from Keil to SDCC: SDCC is a free compiler where as Keil is a commercial compiler. Migrating from Keil C51 to SDCC for XC800 may need more care than just translating the source files.

The scope of this application note is to outline clearly the SDCC extensions and syntax and their Keil equivalents. The scope of this document does NOT include listing the differences between both compilers. SDCC v2.5.4 B4 R1.6, the beta release of standalone SDCC for XC800 family and Keil Compiler (C51) v8.04b are considered for discussion in this application note.

For more information, please refer to SDCC User Manual for Infineon XC800 family (sdcc_xc800_usermanual.pdf) and SDCC User Manual for Assembler and Linker for Infineon XC800 family (sdcc_xc800_asm_lnk.pdf).

2 Memory Models

Keil supports 3 memory models.

- SMALL (All variables are located by default to internal data memory),
- COMPACT (All variables are located by default to one page or 256 bytes of external data memory pdata accessed indirectly via registers R0 & R1)
- LARGE (All variables are located by default to external data memory or xdata accessed indirectly via the data page pointer DPTR)

SDCC supports only 2 memory models:

- SMALL (All variables are default to internal data memory),
- LARGE (All variables are default to external data memory or xdata)

Source programs written for COMPACT model in Keil should be adapted to LARGE model in SDCC.

3 Storage classes for variables

Both compilers support all the memory qualifiers available for XC800. The only difference is that Keil supports ***bdata*** for bit-addressable objects in internal data memory and ***bit*** for bit variables where as SDCC supports only bit variables using ***bit*** keyword. This means, using SDCC only bit variables can be defined in bit-addressable memory where as in Keil variables of type int, char etc., and arrays can also be defined and located in bit-addressable memory.

Example 1

Keil:

```
int bdata ibase;                /* Bit-addressable int */
char bdata bary [4];            /* Bit-addressable array */
sbit mybit15 = ibase ^ 15;      /* bit 15 of ibase */
sbit Ary07 = bary[0] ^ 7;       /* bit 7 of bary[0] */
bit test_bit = 0;               /* bit variable */
```

SDCC:

```
bit test_bit = 1;               /* bit variable */
```

An alternative method for simulating bit addressable objects using SDCC is suggested in Example 5.

Also, both compilers support XC800 memory specific pointers. Both compilers support bit variables as return values.

4 Absolute addressing

Keil uses **_at_** keyword for locating a variable at an absolute memory location where as SDCC uses *at* keyword.

Example 2

```
Keil: int chksum _at_ 0x23; /*chksum at 0x23 */

SDCC:  unsigned int at 0x23 chksum; /*chksum at 0x23 */
```

While locating the variables at absolute addresses, storage classes can also be specified in the variable definitions for both the compilers.

Example 3

```
Keil: int xdata chksum _at_ 0xF010; /* chksum at xdata
0xF010 */

SDCC:  xdata at 0xF010 unsigned int chksum; /* chksum at xdata 0xF010 */
```

Keil has an exception that variables defined with *bit* storage class cannot be located at an absolute address unlike in SDCC.

Example 4

```
Keil:          Not available

SDCC:  bit at 0x02 bit_var;
/* Allocate the variable at offset 0x02 in the bit addressable space */
```

The SDCC will not track variables declared at absolute addresses and may declare other variables so that they will overlap.

A use for absolute addressing is to simulate bit-addressable variables. In the following example, we will define the variable of type char ***num_byte*** at address 0x0020 which is the first byte in bit-addressable memory. Next, we will define 8 variables of type bit ***num_bit0*** to ***num_bit7*** and locate them such that they will overlap with the address of the variable ***num_byte***.

Example 5

```
SDCC: Simulating bit-addressable variables /* simulate_bdata.c */
#include <xc886.h>
data unsigned int at 0x0020 num_byte;

bit                    at 0x0 num_bit0;
/* Locate num_bit0 at offset 0 in BSEG, bit-addressable segment */
bit                    at 0x1 num_bit1;
bit                    at 0x2 num_bit2;
bit                    at 0x3 num_bit3;
bit                    at 0x4 num_bit4;
bit                    at 0x5 num_bit5;
bit                    at 0x6 num_bit6;
bit                    at 0x7 num_bit7;

void main(void)
{
        uart_init();
        num_byte = 0x00;

        printf("num_byte value before modifying is %d \r\n",num_byte);

        num_bit4   = 1;

        IEN0 = num_byte;

        printf("num_byte value after modifying is %d \r\n",num_byte);

        while ( 1 ) ; /* Program Loop */
}
Command-line option for compiling the code:
sdcc -mxc886 -Wl -bBSEG=0x20 simulate_bdata.c -o simulate_bdata.hex
/* -Wl -bBSEG=0x20 → locates segment BSEG at 0x20 */

Output:

num_byte value before modifying is 0

num_byte value after modifying is 16
```

5 SFR/SBIT definitions

The on-chip peripherals of the XC800 are accessed using special function registers or SFRs. These core registers occupy direct internal data memory locations in the range 80_H to FF_H.

Both the compilers have processor specific header files that contain the definitions of SFRs.

Example 6

```
Keil:   sfr P0 = 0x80;
SDCC:   sfr at 0x80 P0; /* special function register P0 at location
                           0x80 */
```

Keil provides ***sfr16*** keyword to define 16-bit SFRs. SDCC does not have any equivalent keyword. It can be converted to ***volatile unsigned int*** as below:

Example 7

```
Keil:   sfr16 T2 = 0xCC;      /* Timer 2: T2L 0CCh, T2H 0CDh */

SDCC:   volatile unsigned int at 0xCC T2;
```

Special function registers which are located on an address divisible by 8 are bit-addressable. A variable of type sbit addresses a specific bit within these sfrs.

Keil supports 3 variants for specifying the address of a sbit register/variable:

- sbit sbit_name = sfr-address ^ bit-position;
- sbit sbit_name = sbit-address;
- sbit sbit_name = sfr-name ^ bit-position;

SDCC supports 2 variants for defining a sbit:

- sbit at sfr-address ^ bit-position sbit_name;
- sbit at sbit-address sbit_name;

Example 8

```
Keil:  sbit CY = 0xD0 ^ 7; /* sbit CY at bit-position 7 of the byte at
                              address 0x00D0 */

SDCC:  sbit at 0xD0^7 CY;
```

Example 9

```
Keil:  sbit CY = 0xD7;     /* sbit CY at bit address 0xD7 */

SDCC:  sbit at 0xD7 CY;
```

Example 10

```
Keil:  sfr PSW = 0xD0; /* Base address, 0xD0 is evenly divisibly by 8 */
       sbit CY = PSW ^ 7; /* Previously declared sfr PSW is the base
address for the sbit CY. sbit CY is declared at bit address 0xD7 */

SDCC:  sbit at 0xD7 CY; /*Bit address is directly specified */
```

6 Double Precision Floating Point Support

Keil has support for double precision floating point where as SDCC does not have support for the same. If any variable is declared as double, SDCC issues a warning and assumes it as a float.

Note: SDCC supports the float data type in large memory model only where as Keil supports float in small memory model also.

7 Interrupt Functions

Both Keil and SDCC have the same syntax for defining interrupt routines. The syntax is as below:

```
void interrupt_id (void) interrupt interrupt_num using bank_num
{
….
}
```

Where,

interrupt_id is the interrupt function name,

interrupt_num is the interrupt's position within the interrupt vector table,

bank_num indicates which register bank has to be used for storing local variables. This parameter is optional.

SDCC has the following exception:

If a source program containing main() uses any ISR which is NOT present in the same source file, then a prototype of the ISR should be included in the source file containing main().

8 Relocating Functions

Functions can be relocated to a specific memory address in both the compilers. Keil provides the following macro as a linker directive to locate code segments.

CODE (« *range*, ... » « *segment* « (*address*) » « , ... » »)

Where

range specifies the address range(s) to use for CODE segments.

segment is the name of a segment.

address is a physical address at which the segment is to be located.

Example 11

```
Keil:  CODE(?PR?FUNC1?A (0x2800))
/* Locates the "?PR?FUNC1?A" segment at 2800h  */
```

Note: In Keil, each segment has a prefix, e.g. "?PR?" which denotes that the segments corresponds to Executable Program Code. Segment names include a module_name which is the name of the source file in which the object is declared. In this example, "FUNC1" is the function name and "A" is the source file name.

In SDCC, relocation of segments can be done using the following linker option:

-barea=expression

This option sets the area base address. Where, "area" is the segment name and the expression may contain constants and/or defined symbols from the linked files.

If multiple segments have to be located to different addresses, this option should be put as one definition per line in the linker script or should be issued multiple times, one for each segment, to the linker using "-Wl" option of SDCC.

The directive "**.area**" can be used to define multiple programming sections. "CSEG" is the default code segment.

.area TEST (REL, CON)

Where, "TEST" is the section name and REL, CON denotes that this section is relocatable and concatenated with other sections of this program area.

Example 12

```
SDCC: Example on usage of ".area" directive /* sample.s */
;--------------------------------------------------------
; overlayable register banks
        .area REG_BANK_0      (REL,OVR,DATA)
        .ds 8
;--------------------------------------------------------
; internal ram data
        .area DSEG    (DATA)
_num1   =       0x0021
;--------------------------------------------------------
; Stack segment in internal ram
        .area   SSEG    (DATA)
__start__stack:
        .ds     1
;--------------------------------------------------------
; external ram data
        .area XSEG    (XDATA)
_counter =    0x10
;--------------------------------------------------------
; Home
        .area HOME    (CODE)
        .area CSEG    (CODE)
__sdcc_program_startup:
        lcall   _main
;       return from main will lock up
        sjmp .
;--------------------------------------------------------
; code
        .area CSEG    (CODE)
_main:
       ...
```

The following can also be used to define a program section:

```
Sdcc test.c -S --codeseg TESTCODE
```

This defines a code segment with name “TESTCODE”.

Example 13

```
SDCC: Example for relocating two segments at different addresses

Sdcc Func1.c -S  --codeseg  FUNC1CODE
Sdcc Func2.c -S  --codeseg  FUNC2CODE

Sdcc main.c Func1.s Func2.s -Wl -bFUNC1CODE=0x2800,-bFUNC2CODE=0xA000
--code-loc 0x0000 --stack-loc 0xE0
```

Note: Code segment location and Stack pointer initial values might also have to be changed correspondingly to avoid overlaps.

9 Inline Assembly

In SDCC, assembly code can be enclosed between the keywords "**_asm**" and "**_endasm**". Inline assembly code can also access 'C' variables by prefixing the variable name with an underscore character. However, labels declared in 'C' code cannot be accessed in inline assembly and viceversa.

Example 14

```
SDCC: Usage of Inline Assembly in a 'C' program
#include <xc886.h>
unsigned char a;
void main(void)
{
        while (1)      // program loop...
        {
                a = P0_DATA;
                _asm
                nop
                nop
                nop
                inc _a          ;Accessing variable 'a' declared in 'C'
                _endasm;
                P1_DATA = a;
        }
}
```

In Keil, inline assembly should be embedded with in the directives “**asm**” and “**endasm**”, which can be used as part of **#pragma**. Variables declared in ‘C’ cannot be accessed in inline assembly.

Example 15

```
Keil: Usage of Inline Assembly in a ‘C’ program
#include “MAIN.H”
void main(void)
{
        #pragma asm
                nop
                inc P0_DATA
        #pragma endasm
}
```

10 Dynamic memory allocation

SDCC does not create heap space automatically for a user program. It is the user’s responsibility to provide heap space for malloc() to allocate memory from.

In **SDCC**, it is supported only in large memory model and can be done like the following:

```
#define DYNAMIC_MEMORY_SIZE 128

unsigned char xdata dynamic_memory_pool[DYNAMIC_MEMORY_SIZE];

void main(void)
{
        init_dynamic_memory ((MEMHEADER xdata *) dynamic_memory_pool,
        DYNAMIC_MEMORY_SIZE); /* Allocate heap space */
}
```

In **Keil**, it is done like the following:

```
unsigned char xdata malloc_mempool [0x100];
void main()
{
        init_mempool (&malloc_mempool, sizeof(malloc_mempool));
}
```

Example 16

SDCC: Example on dynamic memory allocation

```
/* malloc_test_sdcc.c */

#include <xc886.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define DYNAMIC_MEMORY_SIZE 128

unsigned char xdata dynamic_mem_pool[DYNAMIC_MEMORY_SIZE];

unsigned char * xdata src_str, *xdata dst_str;

void main (void)
{
        init_dynamic_memory((MEMHEADER xdata *)dynamic_mem_pool,
        DYNAMIC_MEMORY_SIZE);

        uart_init();

        src_str = malloc(10); src_str = "Hello";

        printf("The source string is = %s,\r\n", src_str);

        dst_str = malloc(10);

        strcpy(dst_str, src_str);

        printf("The copied string is = %s,\r\n", dst_str);
}
```

Command line option to compile the code:

```
sdcc -mxc886 malloc_test.c -model-large -xram-loc 0xF000 --xram-size 512
-o malloc_test_output.hex
```

Example 17

Keil: Example on dynamic memory allocation

```
/* malloc_test_keil.c */

#include <MAIN.H> /* Header file for XC886 */
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

unsigned char xdata malloc_mempool [0x100];

unsigned char * xdata src_str;

unsigned char * xdata dst_str;

void main (void)
{
       init_mempool (&malloc_mempool, sizeof(malloc_mempool));

       /* Include UART initialization here */

       src_str = malloc(10);

       src_str = "Hello";

       printf("The source string is = %s,\r\n", src_str);

       dst_str = malloc(10);

       strcpy(dst_str, src_str);

       printf("The copied string is = %s,\r\n", dst_str);

}
```

Note: Error checking code and UART initialization code is omitted in the examples for clarity.

11 Support for structures

Both compilers support “struct” keyword. Keil has no limitations on the usage of structures where as SDCC have the following limitations on the usage of structures:

- Structures cannot be assigned values directly
- Structures cannot be passed as function parameters
- Structures cannot be assigned to each other
- Structure cannot be a return value from a function

Example 18

```
SDCC: Invalid usage of structures

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 */
}
```

The following can be used as workaround for the above limitations:

- 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

Example 19

```
SDCC: Example on workarounds for limitations on structures usage
typedef struct {
int no_of_lines;
char *info;
} X;

X * max(X *x, X *y) /*Pointer to structures as parameters */
{
        if (x->no_of_lines >= y->no_of_lines)
        return x;

        return y; /*Pointer to structures as return values */
}
void main (void)
{
        X x1, x2, *m;

        x1.no_of_lines = 100; /* Members of a structure can be
                                 assigned values directly */
        x2.no_of_lines = 200;

        m = max(&x1, &x2);/*Pointer to structures as parameters*/
}
```

12 Conclusion

SDCC has support for all major features that Keil supports. The migration from Keil to SDCC will need some effort in bit addressable and in a few language constructs and mostly depends on how extensively the Keil specific language extensions are used.

13 Acknowledgements

My sincere acknowledgements go to Rizal Prasetyokusuma, Manoj Palat and Christober Rayappan for giving the idea to write this application note and their valuable feedback and also to Mike Copeland, Sonali nath, Kasinath and Nagappan Rethinam for their suggestions.

14 References

- SDCC User Manual for Infineon XC800 family (sdcc_xc800_usermanual.pdf, which is part of the package, SDCC_XC800_2.5.4B4_R1.6_Setup.zip)
- SDCC User Manual for Assembler and Linker for Infineon XC800 family (sdcc_xc800_asm_lnk.pdf, which is part of the package, SDCC_XC800_2.5.4B4_R1.6_Setup.zip)
- Keil C51 Compiler User Guide

- Using the Free SDCC Compiler to develop firmware for DS89C420 family of microcontrollers (2005JUN07_EMS_CTRLD_AN.pdf)

http://www.infineon.com

Published by Infineon Technologies AG