Using SDCC
for
Embedded System Design
(ECEN 5613/4613)

Authors: Kimarie Hoot, Troy Curtiss, Linden McClure

Getting Started:

- Download SDCC from http://sdcc.sourceforge.net/
  - If you are developing on a Windows platform I strongly recommend you get http://prdownloads.sourceforge.net/sdcc/sdcc-2.5.0-setup.exe (or whatever is the latest revision) because it has an install wizard which will copy the files and will ask you if you want SDCC added to your path (HINT: I recommend you add SDCC to your path!)
  - The most up-to-date documentation is at http://sdcc.sourceforge.net/doc/sdccman.html/. SDCC also comes with an older revision of the same documentation which is installed in C:\Program Files\SDCC\doc\sdccman.html\index.html by default.
  - **Students have reported experiencing problems with rev 2.3.0 and rev 2.4.0 of SDCC, so make sure you are using rev 2.5.0 or newer.**
- Download GNU make from http://ece.colorado.edu/~mcclurel/make.exe.
  - You can add make to your path as well.
    - If you are developing on Windows XP:
      - Right click on "My Computer"
      - Select the "Advanced" tab
      - Click on "Environment Variables"
      - Select "PATH" and "Edit" if it already exists. Otherwise click "New" and create a PATH variable.
      - Add the location of make.exe to the "Variable Value"
    - If you are developing on another version of Windows you will need to investigate further because the Environment variables may not be in the same location. For example, in Windows NT look in Start/Settings/Control Panel/System and select the Environment tab to modify the PATH variable.

SDCC Memory Models

- SDCC basically has two memory models: Small and Large
  - The large memory model will allocate all variables in external RAM by default
    - Variables stored in internal RAM must be declared with the "data" or "near" keyword
  - The small memory model will allocate all variables in internal, directly-addressable RAM by default
    - Variables stored in external RAM must be declared with the "xdata" or "far" keyword
- SDCC recommends the use of the small memory model for more efficient code. However, for this class, since we are using combined program and data memory spaces, I think it is safer to use the large memory model.

- Be aware that, regardless of the memory model you choose, if you do not explicitly declare a pointer as data/near or xdata/far it will be 3 bytes!

SDCC Basics

- Assuming that the location of SDCC is defined in your path, you can use the following syntax for your header files:

  #include <stdio.h>

- To use SDCC on the command line, use a command line syntax similar to the following (note: a more complete list of flags is shown in the example makefile later):

  sdcc --code-loc 0x6000 --xram-loc 0xB000 file.c

  - SDCC will generate the following output files:

    file.asm – Assembler file created by the compiler
    file.lst – Assembler listing file created by the assembler
    file.rst – Assembler listing file updated by the linkage editor
    file.sym – Symbol listing created by the assembler
    file.rel – Object file created by the assembler, Input to the linkage editor
    file.map – Memory map for the load module created by the linker
    file.mem – Summary of the memory usage
    file.ihx – This is the load module in Intel hex format

  - By default SDCC uses the small memory model
  - The assembler is given the memory locations as .area directives instead of ORG statements.
    - You must remember to use the --code-loc and --xram-loc directives because this tells the linker where to place things in memory!
    - You can examine the file.rst and file.map output files to verify that your code and data are assigned to the correct location.
- SDCC standard library routines
  - Most standard routines are present (printf, malloc, etc…) However:
    - printf depends on putchar() which is not implemented.
      - You must implement putchar()
      - This allows you to decide where printf is displayed (on a terminal via serial port, on an LCD, etc…)
      - The putchar() function must have the following format:

        void putchar(char c);

      - If you need a getchar() function, the format is:

        char getchar();

- malloc depends on having heap space created but SDCC does not automatically create heap space for your program.
  - You must provide heap space for malloc to allocate memory from.
  - This can be done by:

```
#include <malloc.h>
#define HEAP_SIZE 4000
unsigned char xdata heap[HEAP_SIZE];
void main()
{
        init_dynamic_memory((MEMHEADER
                xdata *)heap, HEAP_SIZE);
}
```

- SDCC interrupt support
  - To write an ISR in C, create a function similar to the following format:

```
void isr_foo() interrupt 1
{
}
```

    - This format tells SDCC to generate an interrupt vector (at offset 0x0B from the --code-loc address) that calls isr_foo in response to interrupt 1.
    - It also tells SDCC to generate a RETI instruction instead of a RET instruction to return from a call to isr_foo().
  - The standard code generated for the interrupt is not very efficient. SDCC takes a conservative view and will save registers on the stack before executing any of your code in the ISR and it will restore those registers before executing the RETI instruction.
    - You can use the keyword "_naked" to make your interrupt faster. This keyword will prevent SDCC from generating any entry/exit code to save registers for your ISR.
      - WARNING: If you use the _naked keyword you must save and restore any registers that are modified by your ISR or you must guarantee that no registers are used by your ISR.
      - I would only recommend using the _naked keyword if your ISR only contains inline assembly in which case you know explicitly which registers are used or you are setting a single bit (such as a port pin) in which case no registers are used.
      - You can use the _naked keyword on any function, not just for ISRs. In non-ISR routines you must be aware of the calling convention used and save/restore the registers you used within your function as appropriate.
- SDCC serial port initialization
  - There is no support routine built into SDCC to initialize the serial port
  - If you want to use the serial port with your C program that you burn into EPROM you will have to initialize the hardware first.

- SDCC pointers and internal/external memory access
  - SDCC provides a mechanism where you can assign an absolute address to a symbol in your code.
    - The following syntax means that lcd_reg is pointing to an 8 bit location in the external memory map that corresponds to the address 0xE000:

      ```
      xdata at 0xE000 unsigned char lcd_reg;
      ```

  - You can also use the absolute addressing syntax to define symbols for special function registers and bit addressable locations in order to make your code easier to read and maintain.
    - A special function register:

      ```
      sfr at 0x8A TL0; /* Timer 0 low byte */
      ```

    - A bit addressable location:

      ```
      bit at  0xB1 TXD; /* Transmit */
      ```

    - I strongly suggest that you make use of this feature because it will make your life significantly easier when you are debugging!

- Other miscellaneous tips and hints
  - Search Google for "SDCC 8051"

  - Make sure that you provide function prototypes or define the function before its use. The SDCC compiler does not always seem to warn you if there is a mismatch and the resulting behavior is undefined.

  - The SDCC compiler by default does not put local variables on the stack. This makes local variables “static” by default. To specify that you want SDCC to place local variables on the stack use --stack-auto when you compile. However, the amount of internal RAM space available for a stack is limited so you’ll have fewer worries if you use the large memory model and don’t use the stack for local variables. If you explicitly specify a storage class (e.g. xdata/idata/near/far/etc…) then SDCC will allocate your variable in that location regardless of your memory model or stack options.

  - Many times, when you have a variable that is pointing to an external memory or I/O location, SDCC will optimize it away unless you use the volatile keyword. Anytime you create a C variable in a function and only use it in inline assembly, the SDCC optimizer will remove the symbol. If the assembler complains about a missing symbol, use volatile and the problem will go away.

- **SDCC open knowledge resource: http://sdccokr.dl9sec.de/**
  This web site contains library code that is useful with SDCC.
  Students in ECEN 4613/5613 have successfully used the I2C routines that are available from this site (zip file contains lib24c16.c).

- **SDCC MCS51 Information - SDCC Memory Map with the 8051**
  http://www.turbobit.com/mem51.html

- Using the Free **SDCC** C Compiler to Develop Firmware for the **...**
  This application **note** will focus on using the **SDCC** to develop firmware for the DS89C420/430/440/450 family of ultra-high-speed 8051-compatible **...**
  www.maxim-ic.com/AN3477 - 45k

- Using the **SDCC** Compiler for the DS80C400 - Maxim/Dallas
  This **application note** describes how to use the **SDCC** tools to build **applications** for the DS80C400. It begins with a HelloWorld **application**, **...**
  www.maxim-ic.com/an3346 - 44k

- **SDCC** 8051 TSL IDE
  This **application note** describes how to integrate the. **SDCC** 8051 Tools into the Silicon Laboratories IDE. (Integrated Development Environment). It applies to **...**
  www.silabs.com/public/documents/ tpub_doc/a**note**/Microcontrollers/en/an198.pdf

- **MSC12xx Programming with SDCC**
  http://focus.ti.com/lit/an/sbaa109/
  The MSC12xx is a "system-on-a-chip," which embeds an 8051 microcontroller, a 24-bit ADC, 4-32K bytes of flash memory, and a set of peripherals in one package. Programming the 8051 is an important part of a complete product development effort. The "Small Device C-Compiler," or SDCC, is one of the options available for 8051 software development. This application note discusses how to program the MSC12xx by using open-source SDCC code (available for download with the document).
  Download .pdf (sbaa109.pdf, 303 Kbytes) (Requires **Acrobat Reader 4.x**)
  SBAA109.zip; contains two ZIP files, "ADC" and "Hello World" (sbaa109.zip, 5 Kbytes)

- Atmel include files for SDCC
  http://www.atmel.com/dyn/products/tools.asp?family_id=604

- Downloader for Atmel with support for SDCC and Dunfield
  http://chaokhun.kmitl.ac.th/~kswichit/easy1/easy.htm

- Other 8051 code resources
  http://www.smartdata.com.au/8051/default.htm

GNU Makefile example using SDCC

```
################################################################################
# GNU Makefile demonstrating combination of C and assembly source files
# File Name: makefile
# All targets in the makefile are processed sequentially by SDCC
# To create the file 'file.hex' using GNU make, just execute 'make'
################################################################################
# The following lines defines additional directories to search for include files
INCLUDES := -I"C:\Your Directory\Lab3\"

# The following line defines flags given to the SDCC C compiler
CFLAGS := -c –model-large $(INCLUDES)

# The following line defines flags given to the SDCC linker
# Non-specific: –code-loc 0x6000 –xram-loc 0xB000 –model-large
LFLAGS := --code-loc 0x6000 --code-size 0x5000 --xram-loc 0xB000 --xram-size
0x3000 --model-large

# The following line specifies the default target(s) to build
all: file.hex

# The following line specifies the object files that are to be linked together
OBJECTS := file.rel another_file.rel assembly_file.rel

# The following lines define a rule that sends the object files through the linker to
# create file.ihx which then has to be processed by packihx to create file.hex
file.hex : $(OBJECTS)
        sdcc $(LFLAGS) $^
        packihx file.ihx > file.hex

# The following rule sends each C file through the preprocessor and creates the asm file
# that is then assembled to create the rel file.
%.rel : %.c
        sdcc $(CFLAGS) $<

# The following rule sends each asm file (Not the asm files created by SDCC as an
# intermediate output of the compilation process.) through the assembler to create a rel
# file.
%.rel : %.asm
        asx8051 –losa $<

# The following rule will clean all the derived objects from your directory.  This will
# save you from accidentally tying 'rm *' if you are developing on a UNIX platform.
# Note: To execute this rule type 'make clean'
clean:
        del *.rel *.lst *.rst *.hex *.ihx
```

Addendum/Update by Troy Curtiss

Getting Started:

- In Fall 2006, I used version 2.6.0 successfully, finding no 'compiler bugs.'
- Please read the SDCC documentation. It is very good, and gives many examples of how to make SDCC generate optimal 8051 code.
- The Windows installer version is handiest way to get SDCC onto your Windows system: http://downloads.sourceforge.net/sdcc/sdcc-2.6.0-setup.exe
- I also have (the latest) Cygwin tools installed, for 'make' and other handy command-line utilities. If you only need 'make', it's probably easier just to download the instructor's version, but if you want a more Unix-like environment (bash, vim, grep, cat, etc.) Cygwin is definitely the easiest path if you aren't already running Linux/FreeBSD/etc. http://www.cygwin.com/

SDCC Memory Model:

- Using the reference C501 design in ECEN 5613, the 'large' memory model makes the most sense, due to the abundance of external RAM. All variables are placed in external ram (implicit storage class of __xdata) unless otherwise specified. Since internal ram (storage class __data) is very limited in the 8051 architecture, it is best used selectively for code optimizations later.

SDCC Basics:

- SDCC syntax error reporting is a little ambiguous at times. Pay attention to the _first_ error reported, as any latter errors might be cascade effects of the initial error.
- Note that the SDCC package has many vendor-specific system headers for various 8051 variants. Since the C501 is basically a vanilla 8052, the following include will get (most) everything you need (SFR's by their common mnemonics.) Take a look at the files 8052.h and 8051.h in the SDCC distribution for more information.
```
#include <8052.h>
```
- Suggested compiler flags are as follows (-mmcs51 is the default, but it's best to call it out explicitly):
```
-mmcs51 --std-sdcc99 --model-large
```
- Some initial RAM linker flags are as follows (again, -mmcs51 and –out-fmt-ihx are the defaults) allowing 0x6000 bytes for code and 0x2000 bytes for xdata:
```
-mmcs51 --code-loc 0x6000 --code-size 0x6000 --xram-loc 0xC000 --xram-size
0x2000 --model-large --out-fmt-ihx
```
- Some initial ROM linker flags are as follows:
```
-mmcs51 --code-loc 0x0000 --code-size 0x6000 --xram-loc 0x6000 --xram-size
0x8000 --model-large --out-fmt-ihx
```
- The standard C library functions are fairly complete. One subtle difference is the variations of printf() that exist. Standard printf() is very large and can handle both integer and floating point numbers. Instead of printf(), one may opt to use printf_tiny() or printf_small() to use less memory, but with various limitations. See the 'library routines' section of the SDCC documentation for more detail on these limitations.
- When implementing speed-critical ISRs, there are a few options that SDCC gives the programmer to significantly lower the ISR overhead. The most useful for this

is the 'using(…)' construct. When specified in the declaration of the ISR, using(…) dedicates another set of banked registers to that ISR, removing the need to save and restore any of the general purpose registers used. The SDCC documentation clearly details how to use this construct. Note that the optimization works best when only one ISR is using a particular register bank. If two or more routines share a register bank, the compiler must generate code that saves and restores all modified registers.

```
// Serial port ISR
void sio_isr(void) using(1) interrupt SI0_VECTOR {…}
```

- A handy trick when trying to optimize the assembly code generated by SDCC is to compile with the –S switch (note that the 'S' must be uppercase.) This tells SDCC to just generate the assembly code (.asm file) and stop. One can then look at the assembly to see how elegantly SDCC is generating optimized 8051 assembly code from the C code.
    - One common pitfall this reveals is C's integer promotion rules. For example, when doing an operation on one 8-bit variable and comparing with another 8-bit variable, both values will be promoted to an integer (16-bit) without the user's knowledge. Explicitly casting the operation on the variable to remain 8-bit removes C's implicit promotion.

```
unsigned char a,b; // a & b are 8-bit unsigned variables

if ( (a + 1) == b ) {…}  // bloated 8051 code due to int promotion

if ( (unsigned char)(a + 1) == b ) {…} // more optimal 8051 code
```

- Another optimization tactic is to place variables that need faster access (ie. less 8051 opcodes to access) in internal ram (assuming that the large memory model is in use.) This is especially useful for FIFOs used in ISRs, or other data that needs to be accessed efficiently.

```
unsigned char out_fifo[16]; // Slow FIFO

__data unsigned char out_fifo[16]; // Fast FIFO
```

- To conserve RAM, put any constant data in code space. This not only saves RAM, but also saves program startup time by not having to copy the data into RAM from ROM upon C runtime init.

```
char my_string = "This string will never change";

__code char my_string = "This string will never change";
```

- Read the SDCC manual section on 'Pointers to MCS51/DS390 specific memory spaces.' Normally, it's safest to simply use generic pointers, but if you are trying to optimize your code (for space or speed) by using pointer trickery, it's best to understand exactly how to declare the pointer you need for the desired results. Here is an excerpt from that section giving some pointer examples:

```
/* pointer physically in internal ram pointing to object in external ram */
__xdata unsigned char * __data p;

/* pointer physically in external ram pointing to object in internal ram */
__data unsigned char * __xdata p;
```

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

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

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

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

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