SDCC Programming Tips for the Atmel AT89C51RC2

1. Data memory (i.e. XRAM) is quite limited in the AT89C51RC2, so it is crucial that data memory is used most efficiently. To use code and data memory more efficiently, utilize a smaller library function if possible.
2. Consider using printf_tiny() instead of using printf_small or printf(). You don't have to include any additional header files to use the printf_tiny() function. (only 'stdio.h; is required)
3. The printf() and printf_small() function calls use some of the available XRAM, so it may be a challenge to write the Lab #3 code using the printf() or printf_small() functions given only 1024 bytes of XRAM.
4. printf_tiny() is an optimized version of printf_small(), implemented in assembly. It doesn't occupy any memory in XRAM.
5. Use 'unsigned char' or 'char' instead of 'int' or 'long' if possible to save memory space.
6. Don't declare large arrays in functions. It may overrun your stack as the stack is implemented in the internal memory of 128 bytes.

How to check memory usage?

- You can check the code and data memory usage by examining the *.mem and *.map files that are generated in the release folder after each successful build. Below, a brief overview is given regarding the .mem and .map file formats.

*.map example

```
  Value  Global
  --------  --------------------------------
0D:0000    _mem_heap
0D:01A4    _b_size
0D:01A8    _b_add
0D:01AC    _j
.          .
.          .
.          .
0D:0233    _print_statistics_PARM_2
0D:023D    __sdcc_prev_memheader
0D:0247    __sdcc_first_memheader
0D:0249    _init_dynamic_memory_PARM_2
0D:0255    __gptrput_PARM_2
0D:0256    __mulint_PARM_2
0D:0258    __moduint_PARM_2

  Value  Global
  --------  --------------------------------
0C:005F    __sdcc_program_startup
0C:0064    _main
0C:026C    _putchar
0C:028E    _getchar
0C:0298    _port_init
0C:02A5    _serial_init
```

You will find similar sections in your *.map file. It shows the details of your data and code memory mapping.

Data mapping format:
OD:Address (in hex) *_global_variable_name*
OD:Address (in hex) *__library_variable_name*

0D: in address fields stands for data segment
Address (in hex) is the exact address where the local variable is mapped or stored.
All the global variables names that are declared in your file are preceded by a single underscore (**'_'**),.
All the library variable names that are included in your files are preceded by double underscores (**'__'**).

```
e.g. 0D:0000    _mem_heap    (mem_heap array is located at 0x0000 and it
                               occupies memory up to 0x01A3)
     0D:01A4    _b_size      (b_size variable is located at 0x0aA4 and
                               takes 4 bytes up to 0x01A7)
     0D:01A8    _b_add
```

Code mapping format:
OC:Address (in hex) *_global_symbol_name*
OC:Address (in hex) *__library_symbol_name*

0C: in address fields stands for code segment
Address (in hex) is the exact address where the symbol is located.
All the global names that are declared in your file are preceded by a single underscore (**'_'**),.
All the library names that are included in your files are preceded by double underscores (**'__'**).

```
e.g.
0C:005F    __sdcc_program_startup  (this function is located at 0x005F)

0C:0064    _main       (main() function is located at 0x0064 and it
                        takes code space up to 0x026b)
0C:026C    _putchar    (putchar() is located at 0x026c)
0C:028E    _getchar
```

You can verify the above mapping from the *.lst file as well. Note that the *.lst file also contains symbol information, which means it shows the assembly implementation of each 'C' line in your code. You can also use this information along with your assembly knowledge to debug your code in detail if required.

*.mem example

```
Internal RAM layout:
      0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00:|0|0|0|0|0|0|0|0|a|a|b|Q|Q| | | |
0x10:| | | | | | | | | | | | | | | | |
0x20:|B|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x30:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x40:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x50:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x60:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x70:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xc0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xd0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
```

```
0xe0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xf0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0-3:Reg Banks, T:Bit regs, a-z:Data, B:Bits, Q:Overlay, I:iData, S:Stack,
A:Absolute

Stack starts at: 0x21 (sp set to 0x20) with 223 bytes available.

Other memory:
   Name             Start    End      Size     Max
   ---------------- -------- -------- -------- --------
   PAGED EXT. RAM                        0      256
   EXTERNAL RAM     0x0000   0x028d     654    65536
   ROM/EPROM/FLASH  0x0000   0x1349    4938    65536
```

The .mem file shows the usage of internal RAM, XRAM and Flash memory. The table above shows the amount of memory reserved for the stack (shown using 'S'), Data (shown using 'a-z') and register banks (shown using '0-3')

Notice the starting location of your stack. It starts from 0x21 in the above example. That means the stack is limited from 0x21 up to 0xff giving only 223 bytes for the stack. If you have a large series of nested function calls or large data declaration within a function, then you can overrun your stack. Stack overrun can cause some serious errors, including the loss of return addresses of function calls.

The table shown in red shows the summary of memory usage. Note that the AT89C51RC2 has only 1024 bytes (0x0400) of XRAM memory and 32KB (0x8000) of code memory. After a successful build, you can also check your *.mem file to ensure that your code and data fit in these memory bounds by examining the end address in the table shown in red. If your XRAM usage exceeds 0x0400 then it will give you unpredicted results when you run your code on the system.
SDCC can automatically check whether the entire code and data memory fit within a specified limit. Add "--code-size 0x8000 --xram-size 0x0400" to the linker command line option. (Linker command is under project_properties->C/C++ Build->Tool Setting->SDCC Linker->Command). SDCC will generate an error if the memory usage of your program exceeds these limits. Both the location and the size of each segment must be defined, similar to the example below:

1. Specify the code and XRAM locations on the linker command line
   (e.g. --code-loc 0x0000 and --xram-loc 0x0000)
2. Specify the code and XRAM sizes on the linker command line
   (e.g. --code-size 0x8000 and --xram-size 0x0400)

Some students have had some difficulties with the SDCC syntax for printing out pointers and working with malloc.

This document suggests that printf_tiny() may be used instead of printf(), and may conserve XRAM space. However, printf_tiny() is quite limited and cannot handle all types of formatting styles.

Some SDCC sample code has been created so that you can see some syntax that might help you in your lab assignments. The example code file is not polished, and is meant just to point out some things that might work and might not work for you. Please refer to the "**sdcc_syntax_examples.c**" file available on the course web site.

If you compile this sample code, you can also examine the SDCC output assembly code to learn more about how SDCC handles things like pointers and interrupts. The putchar() and getchar() functions include redundant checks for TI and RI, just so you can see different forms of assembly generated by the compiler for these different methods of checking the serial controller flags. While examining your .asm, .lst, or .rst files, note the "Peephole" comments that indicate optimizations performed by the peephole optimizer. In some cases, the peephole optimizer may have removed some redundant code, and in those cases you'll see the C code in the comments, but no corresponding assembly code.

This example code with printf() was built on the professor's system using SDCC version 2.6.0, and it ran with no problems on a board loaded with a C501 processor. With a heap of size 0x300, the program still used only 861 out of 1024 bytes of available XRAM.