PSoC® 3 - 8051 Code And Memory Optimization

Author: Mark Ainsworth

Associated Project: No

Associated Part Family: All PSoC 3 parts

Software Version: N/A

Related Application Notes: None

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/go/AN60630.

AN60630 shows how to increase the efficiency of 8051 code in PSoC® 3 by making greater use of the 8051 core internal features. This can result in smaller code size in flash memory, as well as faster code. The efficiency gains can be realized without writing any 8051 assembler code. Instead, keywords for the Keil 8051 C compiler are used. Several coding techniques are shown.

Contents

Introduction

One common misconception when programming the PSoC® 3 8051 is that the only way to get optimal code is to use 8051 assembly language. This is not true, mainly because of the high performance capabilities of the Keil 8051 C compiler. This compiler is included free with PSoC Creator™, the development tool for PSoC 3, PSoC 4, and PSoC 5LP.

Because of the compiler's capabilities, most if not all PSoC 3 8051 code can be written in C, and it can be made to be small, fast, and efficient. The cost is that you must use Keil-specific keywords, and C code containing these keywords may not be easily portable to other processors, such as the Cortex CPUs in PSoC 4 and PSoC 5LP. However, PSoC Creator offers equivalent macros that make porting easier.

In any case, by using these keywords or macros, and with knowledge of some code architecture issues, you can make your 8051 code faster and smaller, and avoid using the PSoC 3 8051 in its slowest and least efficient mode.

It is assumed that the reader has a basic knowledge of C programming. Knowledge of 8051 assembler is recommended but not required.

Note All of the code shown in this application note was compiled using Keil optimization for size, level 3 (size, level 2 is the PSoC Creator default). Level 3 deletes redundant MOV operations, which can have a significant impact on code size and speed.

The 8051 "Inner Space"

The 8051 core is a 256-byte address space that contains 256 bytes of SRAM plus a large set of registers called Special Function Registers (SFRs), as Figure 1 shows. A lot of functionality is packed into this "internal space" and the 8051 is most efficient when it works in this space.

The lower 128 bytes of this space is all SRAM, and is accessible both directly and indirectly (more on these terms later).

The upper 128 bytes contains another 128 bytes of SRAM that can only be accessed indirectly. The same upper address space also contains a set of SFRs that can only be accessed directly.

In addition to normal SRAM access, some of the bytes in the lower address space can be accessed in other modes, as Table 1 shows.

The 8 "registers" `R0–R7` are a useful set of auxiliary registers that can be accessed quickly with single-byte, single-cycle 8051 assembler instructions such as:

```
ADD A,Rn
```

Only one register bank can be active at a time; usually it is register bank 0.

Each of the 128 bits in the bit-addressable space 20–2F can be accessed individually with bit-level assembler instructions such as:

```
SETB nn
```

where nn is the bit number. If nn is 00, then bit 0 of address 20 is accessed; if nn is 01, then bit 1 of address 20 is accessed, and so on.

Figure 1. 8051 Internal Space Map

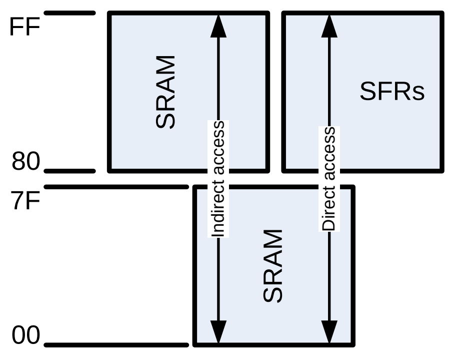

Table 1. 8051 Lower Internal Address Space Functions

Addresses	Function
20–2F	Bit-addressable space
10–1F	Register bank 3 (R0–R7)
10–17	Register bank 2 (R0–R7)
08–0F	Register bank 1 (R0–R7)
00–07	Register bank 0 (R0–R7)

Direct and Indirect Access

With direct access, the address is part of the assembler instruction; for example:

```
    INC  nn
```

where nn is the address of either the first 128 bytes of internal SRAM or an SFR.

For indirect access, register `R0` or `R1` is used as a pointer; for example:

```
    DEC  @Ri
```

where i is 0 or 1. Using indirect access, the full 256 bytes of internal SRAM is accessible.

The 8-bit stack pointer register `SP` is also a pointer to all 256 bytes of SRAM; pushing and popping the stack are considered indirect accesses. The stack pointer grows upward. Because the stack size is always less than 256 bytes, stack operations must be managed carefully.

SFR Space

As noted previously, direct addresses of addresses 80–FF access the SFRs. Almost all registers in the 8051, including the accumulator (`ACC`), program status word (`PSW`), and stack pointer (`SP`), are actually SFRs. Also, some PSoC 3 I/O port registers can be accessed as SFRs. Check the PSoC 3 datasheet and Technical Reference Manual for details on these SFRs; see also Table 2. Note that many of the SFRs are unpopulated; reading or writing to them yields unpredictable results.

Table 2. PSoC 3 8051 SFR Map

Address	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0xF8	SFRPRT15DR	SFRPRT15PS	SFRPRT15SEL					
0xF0	B		SFRPRT12SEL					
0xE8	SFRPRT12DR	SFRPRT12PS	MXAX					
0xE0	ACC							
0xD8	SFRPRT6DR	SFRPRT6PS	SFRPRT6SEL					
0xD0	PSW							
0xC8	SFRPRT5DR	SFRPRT5PS	SFRPRT5SEL					
0xC0	SFRPRT4DR	SFRPRT4PS	SFRPRT4SEL					
0xB8								
0xB0	SFRPRT3DR	SFRPRT3PS	SFRPRT3SEL					
0xA8	IE							
0xA0	P2AX		SFRPRT1SEL					
0x98	SFRPRT2DR	SFRPRT2PS	SFRPRT2SEL					
0x90	SFRPRT1DR	SFRPRT1PS		DPX0	DPX1			
0x88		SFRPRT0PS	SFRPRT0SEL					
0x80	SFRPRT0DR	SP	DPL0	DPH0	DPL1	DPH1	DPS	

As noted previously, bits 00–7F access a region in lower SRAM. Bits 80–FF access some of the SFRs, in the following manner: Bits 80–87 access the individual bits in SFR 80, `SFRPRT0DR`. Bits 88–8F access the individual bits in SFR 88, which is unpopulated, and so on. So individual bits can be accessed in SFRs at addresses 80, 88, 90, 98, …, F0, F8. The most frequently used PSoC 3 / 8051 registers are located at these SFR addresses.

Keil 8051 Memory Models

Based on the 8051 architecture and instruction set, the Keil C compiler defines three memory models: small, compact, and large. These memory models control the addressing mode in the 8051 assembly language output, and thus allow precise control of code size and execution speed, as Code 1 shows.

Code 1. C Code with Keil Keywords and Corresponding 8051 Assembler, for Different Keil Memory Models

```
    /* C variable definitions, in different memory spaces */
     data char small_direct_var;
    idata char small_indirect_var;
    pdata char compact_var;
          char large_var;  /* large memory model default */

    /* usage of the variables: simple increment operations in C */
    small_direct_var++;
    small_indirect_var++;
    compact_var++;
    large_var++;

    ; assembler equivalents of the above lines of C code
            ; small_direct_var++;
0500        INC     small_direct_var             ; 2 bytes, 3 cycles

            ; small_indirect_var++;
7800        MOV     R0,#LOW small_indirect_var   ; 3 bytes, 5 cycles
06          INC     @R0

            ; compact_var++;
7800        MOV     R0,#LOW compact_var          ; 5 bytes, 8 cycles
E2          MOVX    A,@R0
04          INC     A
F2          MOVX    @R0,A

            ; large_var++;
900000      MOV     DPTR,#large_var              ; 6 bytes, 9 cycles
E0          MOVX    A,@DPTR
04          INC     A
F0          MOVX    @DPTR,A
```

In Code 1, you can see that successively larger memory models require more flash bytes and more CPU cycles. The default model for PSoC Creator is large (to maintain compatibility with PSoC 5LP), but that default can be overridden for individual variables, functions, and even entire modules.

The keywords ‘data’ and ‘idata’ are used to designate small model variables in direct and indirect modes, respectively. The keyword ‘pdata’ is used to designate the compact model, and ‘xdata’ (or default) is used for the large model. For details see Table 5 on page 24.

The small model accesses the 8051 internal space described previously.

The compact and large models access the “external” space, which is “external” to the 8051 core but of course is “internal” to the PSoC 3 device. All of the PSoC 3 SRAM, registers, EMIF space, and so on are in this “external” space. The size of this space is 16 Mbytes, so three address bytes are required to access this space. For more information see Topic #2: Pointers.

You can also see that in the compact (pdata) model, the “external” space is accessed using `R0` or `R1`. The other two bytes come from the SFRs `MXAX` and `P2AX`, so that the three-byte address, formed from the three registers, is:

```
[ MXAX : P2AX : Ri ]
```

So before accessing pdata variables the SFRs `MXAX` and `P2AX` must be loaded with appropriate values. For more information see Topic #8: Compact Data Space.

And, finally, you can see that in the large (xdata or default) model, the "external" space is accessed using the 16-bit `DPTR` register (which is composed of the SFRs `DPH` and `DPL`). The third byte comes from the SFR `DPX`, so that the three-byte address, formed from the three registers, is:

```
[ DPX : DPTR ]
```

Before accessing xdata variables, the SFR `DPX` must be loaded with an appropriate value. Note that because the PSoC 3 8051 has two `DPTR` registers, there are actually six SFRs: `DPX0`, `DPH0`, `DPL0`, `DPX1`, `DPH1`, and `DPL1`. The SFR `DPS` controls which `DPTR` is currently active.

The Guidelines

Now that you understand the PSoC 3 8051 memory layout and the Keil memory models, consider some guidelines to optimize your C code. Using these guidelines will, in most cases, yield improvements in both code size and execution time. The guidelines are in prioritized order of most effective first.

Guideline #1: Use Bit Variables

The simplest and best way to get dramatic improvements in efficiency is to look for all variables that will have only binary values (0 and not 0), and define them as type 'bit':

```
bit  myBitVar;
```

Variables of type bit are treated similarly to standard C variables:

```
/* assign a value to the variable */
myBitVar = 1;
myBitVar = 0;
/* toggle the variable */
myBitVar = ~myBitVar;
/* do bit-level operations */
bitVar1 |= bitVar2;
/* test the variable */
if (myBitVar)
{
    . . .
}
```

Function parameters and return values can be of type bit:

```
bit myFunction(bit x, bit y);
```

There are some limitations – you cannot have arrays of type bit, and you cannot have pointers to variables of type bit:

```
/* illegal statements */
bit myBitArray[10];
bit *myBitPointer;
```

With bit variables, the extensive set of 8051 bit-level assembler instructions can be used to generate very fast and compact code. For example, the following C code:

```
myVar = ~myVar;
if (!myVar)
{
    ...
}
```

Generates only two assembler instructions:

```
B200      CPL   myVar
200006    JB    myVar,?C0002
```

When you use bit variables, you can frequently implement a nontrivial C statement with just a single assembler instruction.

The above example uses only 5 flash bytes and 8 CPU cycles. Compare it to the assembler code that is generated if you change the variable type 'bit' to type 'char':

```
900000    MOV    DPTR,#myVar
E0        MOVX   A,@DPTR
F4        CPL    A
F0        MOVX   @DPTR,A

E0        MOVX   A,@DPTR
7002      JNZ    ?C0001
```

The code now uses 9 flash bytes and 15 CPU cycles, almost a 2x increase.

You are limited to a total of 128 bit variables in your code; this is the number of bits in the 8051 bit-addressable space. (You get a linker error if you overflow the bit space.)

Finally, it is easy to port this code to PSoC 5LP by using the CYBIT macro provided in PSoC Creator, instead of the 'bit' keyword:

```
CYBIT myBitVar;
```

PSoC Creator has a complete set of macros to ease portability of PSoC 3 C code to PSoC 5LP. For details, see the auto-generated file *cytypes.h*, in the cyboot folder.

Guideline #2: Do Not Call Functions from ISRs

When compiling C code for an interrupt service routine (ISR), the Keil compiler attempts to push onto the stack only those registers that it thinks will be changed by the ISR code. However, if the ISR code includes a function call, the compiler cannot tell which registers are modified by the called function, and therefore pushes *everything* onto the stack. For this reason, the C code in a very simple ISR:

```
CY_ISR(myISR)
{
    UART_1_ReadRxStatus();
}
```

Generates a massive amount of push/pop overhead in the corresponding assembler code:

```
C0F0      PUSH    B
C083      PUSH    DPH
C082      PUSH    DPL
C085      PUSH    DPH1
C084      PUSH    DPL1
C086      PUSH    DPS
758600    MOV     DPS,#00H
C000      PUSH    ?C?XPAGE1SFR
750000    MOV     ?C?XPAGE1SFR,#?C?XPAGE1RST
C0D0      PUSH    PSW
75D000    MOV     PSW,#00H
C000      PUSH    AR0
C001      PUSH    AR1
C002      PUSH    AR2
C003      PUSH    AR3
C004      PUSH    AR4
C005      PUSH    AR5
C006      PUSH    AR6
C007      PUSH    AR7
120000    LCALL   UART_1_ReadRxStatus
D007      POP     AR7
D006      POP     AR6
D005      POP     AR5
D004      POP     AR4
D003      POP     AR3
D002      POP     AR2
D001      POP     AR1
D000      POP     AR0
D0D0      POP     PSW
D000      POP     ?C?XPAGE1SFR
D086      POP     DPS
D084      POP     DPL1
D085      POP     DPH1
D082      POP     DPL
D083      POP     DPH
D0F0      POP     B
D0E0      POP     ACC
32        RETI
```

This code uses 79 flash bytes and 101 CPU cycles just to make one function call.

Now, this particular function call just reads a register, so we can modify the C code to read the register directly:

```
CY_ISR(myISR)
{
    /* copied from UART_1.c */
    UART_1_RXSTATUS;
}
```

This yields some reduction in push/pop overhead, as the following assembler code shows:

```
C0E0      PUSH    ACC
C083      PUSH    DPH
C082      PUSH    DPL
C085      PUSH    DPH1
C084      PUSH    DPL1
C086      PUSH    DPS
758600    MOV     DPS,#00H
C000      PUSH    ?C?XPAGE1SFR
750000    MOV     ?C?XPAGE1SFR,#?C?XPAGE1RST
C0D0      PUSH    PSW
75D000    MOV     PSW,#00H
C007      PUSH    AR7
906465    MOV     DPTR,#06465H
E0        MOVX    A,@DPTR
FF        MOV     R7,A
D007      POP     AR7
D0D0      POP     PSW
D000      POP     ?C?XPAGE1SFR
D086      POP     DPS
D084      POP     DPL1
D085      POP     DPH1
D082      POP     DPL
D083      POP     DPH
D0E0      POP     ACC
32        RETI
```

This code uses 51 bytes and 65 cycles, a reduction of 36% in the number of cycles, and the code is still easily portable to PSoC 5LP. This is good, but you can get even more improvement by using flags.

A flag is a global variable that is used to signal state changes between multiple independent functions.

It is simple to implement a flag – in the ISR, set a global variable (of type bit), and then have the background code read the register when the variable is set:

```
CYBIT flag;

CY_ISR(myISR)
{
    flag = 1;
}

void main()
{
    /* Wait for the ISR to set the
     * flag, then reset it before
     * taking any action.
     */
    if (flag)
    {
        flag = 0;
        UART_1_ReadRxStatus();
        . . .
```

The ISR portion of this C code generates the following assembler code:

```
D200      SETB    flag
32        RETI
```

which uses 3 bytes and 7 cycles, for a 93% reduction in number of cycles from the original ISR code.

The cost of having a flag-based design is that you need to make sure that the status register is read by the background code in a timely fashion, which may be difficult in some cases.

Making the flag type uint8 instead of bit does not yield any similar reductions, because the variable is in the same default xdata space as the register. However, this can be solved by placing the variable in an 8051 internal memory space, as explained in the next section.

Guideline #3: Place Your Variables in the Correct Memory Spaces

As shown previously, significant efficiencies can be gained when a variable is placed in one of the 8051 internal memory spaces. Therefore, in order of frequency of access, variables should be of type 'data', then 'idata', 'pdata', and lastly 'xdata' (xdata is the PSoC Creator default). See Table 5 on page 24 for details.

Also, because of limited stack space, the Keil compiler does not save local variables on the stack as is normally done in C. Instead, it uses fixed memory locations to store local variables and shares those locations among local variables in functions that don't call each other. See Topic #1: Variable Overlay on page 10 for details.

So this guideline is actually twofold:

1. As much as possible, make variables local within functions. Not only is it good programming practice to have as few global variables as possible, but the Keil compiler can try to store locals in auxiliary registers `R0-R7`, which further improves efficiency.

2. Make as many local variables as possible of type 'data' or 'idata'. (You get a linker error if you overflow the data space.) Check your function / ISR calling depth to make sure that you don't run out of stack space, which is shared with the data and idata spaces in internal SRAM.

Guideline #4: Decrement Loop Variables

Try to make your loop variables decrement instead of increment, because it's faster to test for equality to zero than for less than a constant. For example, the following C code:

```
void main()
{
    data uint8 i;

    /* loop 10 times */
    for (i = 10; i != 0; i--)
    {
        ...
    }
```

Generates the following small amount of assembler:

```
75000A    MOV     i,#0AH   ; i = 10
        ?C0002:
E500      MOV     A,i      ; i != 0
6006      JZ      ?C0003
          . . .
1500      DEC     i        ; i--
80EF      SJMP    ?C0002
        ?C0003:
```

If you write the C code such that the loop variable increments instead of decrements:

```
void main()
{
    data uint8 i;

    /* loop 10 times */
    for (i = 0; i < 10; i++)
    {
        ...
    }
```

You get a larger amount of assembler:

```
E4          CLR     A         ; i = 0
F500        MOV     i,A
        ?C0002:
E500        MOV     A,i       ; i < 10
C3          CLR     C
940A        SUBB    A, #0AH
5006        JNC     ?C0003
            . . .
0500        INC     i         ; i++
80EF        SJMP    ?C0002
     ?C0003
```

Guideline #5: Use Bits for Bitwise Operations

As Guideline #1 shows, defining bit variables can greatly increase code efficiency by generating bit-level assembler instructions. Bit-level assembler instructions can also be used to implement C bitwise operations. Consider a variable with a bit that you want to set or test. In C, you would write the following:

```
uint8 x;

x |=  0x10;    /* set bit 4 */
x &= ~0x10;    /* clear bit 4 */
x ^=  0x10;    /* toggle bit 4 */
if (x & 0x10)  /* test bit 4 */
{
    . . .
}
```

To implement C bitwise operations using 8051 bit-level assembler instructions, you must use the ‘sbit’ keyword and the special operator ‘^’ (which in this case does NOT do a C exclusive-or operation).

There are two ways to do this. The first is to place the variable in the internal bit-addressable space 20–2F, using the ‘bdata’ keyword. Then, define the bit of interest using sbit and ^:

```
/* This places myVar in the 8051
 * internal data space, in 20-2F.
 */
bdata uint8 myVar;

/* this is bit 4 of myVar */
sbit mybit4 = myVar^4;

/* set bit 4 of myVar */
mybit4 = 1;
/* clear bit 4 of myVar */
mybit4 = 0;
/* toggle bit 4 of myVar */
mybit4 = ~mybit4;
/* test bit 4 of myVar */
if (mybit4)
{
    . . .
}
```

This technique yields all of the efficiencies noted in Guideline #1. It even works for variables larger than 8 bits; for example, uint16, int32, and so on. Note that the bdata and sbit definitions must be done on global or static variables, not locally within a function.

The second method, which is useful only for testing bits, is to temporarily place the value of interest in one of the bit-addressable SFRs. As an example, the bits in the SFR `PSW` are defined in the PSoC Creator generated source file *PSoC3_8051.h*, in the cy_boot folder:

```
sfr PSW   = 0xD0;

sbit P    = PSW^0;
sbit F1   = PSW^1;
sbit OV   = PSW^2;
sbit RS0  = PSW^3;
sbit RS1  = PSW^4;
sbit F0   = PSW^5;
sbit AC   = PSW^6;
sbit CY   = PSW^7;
```

Because the Program Status Word (`PSW`) is in SFR D0, its bits are directly accessible. Each of the bits in the `PSW` are defined using the sbit keyword. For this reason, each bit can be accessed in the same manner as Guideline #1 shows. For example:

```
F0 = ~F0;
```

Note `PSW` bits `F0` and `F1` are flag bits that are conveniently available for general-purpose use.

Two SFRs are usually available for temporary use in bitwise operations – the accumulator (`ACC`) and an auxiliary register called `B`. However, only the SFRs themselves are defined in *PSoC3_8051.h*; you must define the bits within those SFRs yourself:

```
/* bit 4 of ACC SFR */
sbit A4 = ACC^4;
/* bit 3 of B SFR */
sbit B3 = B^3;
```

Then, for faster testing of a bit you can do the following:

```
/* assume return value is 8 bits */
ACC = UART_1_ReadRxStatus();
if (A4) /* test bit 4 */
{
    . . .
}
```

You can also test multiple bits quickly. Try to rewrite the following using traditional C bitwise instructions:

```
/* assume return value is 8 bits */
ACC = UART_1_ReadRxStatus();
/* test if bit 4 == 1 AND
 *         bit 3 == 0
 */
if (A4 && !A3)
{
    . . .
}
```

This second method is useful only for testing a bit (or bits), however it has the advantage that you don't have to use any SRAM in the bit-addressable space.

IO Port Control SFRs

Also, as Table 2 on page 3 shows, certain registers for every PSoC 3 IO port are available as SFRs. You can read the input port pins' states by reading the corresponding `SFRPRTxPS` SFR, and you can then test individual pin states using the bit-level techniques described above.

You can also control the output port pins by writing to the corresponding `SFRPRTxDR` SFR. Because the `SFRPRTxDR` registers are located in bit-addressable SFRs, pin outputs can be quickly changed using bit-level assembler instructions.

All of the port SFR definitions are available in *PSoC3_8051.h*, but, again, you must create your own sbit definitions for individual pins. You must also set the SFR `SFRPRTxSEL` to control whether the pin is changed by CPU / DMA register access as is normally done, or by SFR access.

The following example shows a very fast toggle of GPIO pin P1.6 using SFR bit-level access:

```
/* port 1 pin 6 DR */
sbit P1_6 = SFRPRT1DR^6;

void main()
{
    /* P1.6 to be changed by SFR
     * access
     */
    SFRPRT1SEL = 0x40;

    for(;;) /* do forever */
    {
        /* toggle P1.6 by SFR/sbit
         * access
         */
        P1_6 = ~P1_6;
    }
}
```

The for loop is implemented using only two assembler instructions:

```
       ?C0001
B296    CPL      P1_6
80D3    SJMP     ?C0001
```

Guideline #6: Use the B Register for Temporary Storage

In the 8051 architecture, the `B` register (in SFR address F0) is used by the assembler instructions MUL and DIV. At all other times, it's just an auxiliary register and is usually not used. But as an auxiliary register, it can be handy, for example, when swapping two 8-bit variables:

```
uint8 x, y;

B = x;
x = y;
y = B;
```

The `B` register can also be used for rapid bit-level testing, as noted previously.

Advanced Topics

The guidelines shown previously introduced some of the Keil C keywords and showed some simple C coding techniques that, using the keywords, yield increased efficiency.

The following topics build on the guidelines, but are more on an architecture level. They show how to design C code for the 8051 to get further reductions in code size and CPU cycles.

Topic #1: Variable Overlay

As seen previously in Code 1 and Guideline #3, you get the greatest amount of code efficiency by using the 8051 internal data spaces (data, idata, bit, bdata, SFRs).

Also, because of limited stack space, the Keil compiler does not save local variables on the stack as is normally done in C. Instead, it uses fixed memory locations to store local variables and shares those locations among local variables in functions that don’t call each other. Keil calls this “data overlaying”.

The following example has two functions, olTest1() and olTest2(), that are called only from main(). Each function, plus main(), manipulates two automatic 32-bit variables.

```
void olTest1()
{
    uint32 x = 1;
    uint32 y = x + 2;
    x = y - 1;
}

void olTest2()
{
    uint32 a = 3;
    uint32 b = a + 5;
    a = b - 1;
}

void main()
{
    uint32 m = 10;
    uint32 n = m + 20;
    m = n - 7;

    olTest1();
    olTest2();
```

An 8-bit processor requires a lot of code to handle 32-bit variables. To increase the efficiency of that code, you could move the automatic variables from the (default) xdata space to the data space, but that would use up a lot of valuable bytes in the data space. To solve this problem, Keil automatically shares storage for the variables x, y, a, and b in the two test functions. (Variables m and n in main() must have dedicated storage.)

In PSoC Creator, Workspace Explorer window, click the **Results** tab and find the *.map* file for your project. (PSoC Creator projects have map file creation enabled by default. If you don’t see a *.map* file, check your project build settings, under Linker.) In the *.map* file, find a line like this:

```
START     STOP      LENGTH    ALIGN  RELOC    MEMORY CLASS   SEGMENT NAME
=========================================================================
. . .
000051H   000060H   000010H   BYTE   UNIT     XDATA          _XDATA_GROUP_
```

The segment `_XDATA_GROUP_` includes the space that is shared by all overlaid variables. The segment occupies 16 bytes: 8 for the overlaid variables in the test functions and 8 for the non-overlaid variables in main(). Build a PSoC Creator project with the code shown above, run the debugger, and bring up a memory window to monitor this segment. Step into both test functions and see that their automatic variables share the same memory.

Now exit the debugger and look in the *.map* file for a line like this:

```
START      STOP       LENGTH     ALIGN   RELOC     MEMORY CLASS    SEGMENT NAME
=========================================================================
. . .
000008H    00000FH    000008H    BYTE    UNIT      DATA            _DATA_GROUP_
```

This line shows that 8 bytes in the data space are already being used by some other functions. You should be able to reuse the same space for the overlaid automatic variables. Do this by adding the ‘small’ (or the PSoC Creator macro CYSMALL) keyword to the two test functions:

```
void olTest1() small
void olTest2() small
```

When applied to a function, the ‘small’ keyword causes that function’s arguments and automatic variables to be placed in the data space. You can instead add the ‘data’ (or CYDATA) keyword to the automatic variable declarations, but this does not affect storage of function arguments.

Now rebuild and check the *.map* file – there is no increase in the use of data space bytes. The debugger also shows that these bytes are being shared by the test functions. You have gained the efficiencies of the 8051 internal data space without using any additional bytes within that space.

Fewer than 128 data space bytes are available; if you run out, you can add the ‘idata’ keyword to automatic variables. This allows you to use some of the other 128 bytes in the 8051 internal space (leave room for stack growth), and creates an overlay segment `_IDATA_GROUP_` for the idata space. Similarly, automatic variables of type ‘bit’ can be placed in an overlay segment `_BIT_GROUP_`.

The significance of this topic is that your code should be constructed such that the further up in the calling tree a function is, the fewer local variables and arguments it should have. Ideally, main() should have none. Your code’s calling tree depth should be as small as possible; this also reduces stack usage. Functions at the bottom of the calling tree can be declared ‘small’ to maximize efficient use of their local variables. Finally, there should be as few global and static variables as possible, as these cannot be overlaid.

This brings up another issue, C library functions. For example, try adding to one of the test functions a call to memset() to clear one of the variables:

```
memset((void *)&a, (char)0, sizeof(a));
```

Examine the *.map* file before and after adding the call. You should see no difference in the amount of data or xdata memory being used (the code size increases, of course). Keil does not supply the source for most library functions. However, because no additional SRAM is used, and from a review of the assembler code (in the PSoC Creator debugger’s Disassembler window), you can infer that the function is using registers, variable overlay, or both. This is generally true of Keil library functions, although some may behave differently.

Another point from this topic is the need to check the *.map* file to understand how your code is being implemented in the 8051 architecture. The *.map* file provides a wealth of information on usage of the different memory spaces, in addition to many other subjects. For more information, see the Keil LX51 Linker User’s Guide.

Topic #2: Pointers

Most CPUs have a single linear address space, and so the size of C pointer variables for these CPUs is determined by the size of the address space. For example, a CPU with a 64 K address space has 2-byte pointer variables, while a 32-bit CPU (such as the ARM Cortex in PSoC 4 and PSoC 5LP) has 4-byte pointer variables.

The 8051 CPU is different in that it has multiple address spaces, ranging from 128 bytes to 64 K bytes in size. To handle this, the Keil C compiler defines two types of pointers – generic and memory-specific.

A generic pointer accesses data regardless of the memory in which it is stored. It uses 3 bytes – the first is the memory type, the second is the high-order byte of the address, and the third is the low-order byte of the address. A memory-specific pointer uses only one or two bytes depending on the specified memory type. The following example demonstrates each type:

```
char idata *ip = &ival;   /* memory-specific pointer to idata space */
char xdata *xp = &xval;   /* memory-specific pointer to xdata space */
char *p = &xval;          /* generic pointer (to xdata space) */

char val = *ip;           /* read a value from the idata space */
val = *xp;                /* read a value from the xdata space */
val = *p;                 /* read a value using the generic pointer */


; Corresponding assembler

750000     MOV     ip,#LOW ival      ; 1-byte ptr to idata space
750000     MOV     xp,#HIGH xval     ; 2-byte ptr to xdata space
750000     MOV     xp+01H,#LOW xval
750001     MOV     p,#01H            ; 3-byte generic ptr (01H = xdata)
750000     MOV     p+01H,#HIGH xval
750000     MOV     p+02H,#LOW xval
A800       MOV     R0,ip             ; read from the idata space
E6         MOV     A,@R0             ;  5 bytes, 7 cycles
F500       MOV     val,A
850082     MOV     DPL,xp+01H        ; read from the xdata space
850083     MOV     DPH,xp            ;  9 bytes, 11 cycles
E0         MOVX    A,@DPTR
F500       MOV     val,A
AB00       MOV     R3,p              ; read using the generic ptr
AA00       MOV     R2,p+01H          ;  11 bytes, 19+ cycles
A900       MOV     R1,p+02H
120000     LCALL   ?C?CLDPTR         ; Keil library function
F500       MOV     val,A
```

The main point of this topic is that memory-specific pointers are more efficient. Generic pointers should be used only when the memory type is unknown. Note that most Keil library functions take generic pointers as arguments; memory-specific pointers are automatically cast to generic pointers.

Function Pointers

As Topic #4 describes, the Keil compiler does not pass C function arguments on the stack. Instead, it uses either registers or fixed memory locations. This can cause problems with function pointers, because the linker cannot predict where the pointed-to function resides in the calling tree and therefore may not put the parameters in a safe location in memory.

To address this issue, Keil provides an OVERLAY linker directive. Keil also provides a detailed application note on this topic, see http://www.keil.com/appnotes/files/apnt_129.pdf for details.

Topic #3: Constants and Flash

C has a qualifier keyword ‘const’ that can be added to a variable or array declaration. The keyword tells the compiler that the variable may not be changed; code that tries to change the variable gives a compile error. However, the ‘const’ qualifier says nothing about where the variable is stored, as this example shows:

```
const char testvar = 37;

void main()
{
    char testvar2 = testvar;
```

The corresponding 8051 assembler shows that the const variable ‘testvar’ is stored in SRAM.

```
900000      MOV     DPTR,#testvar
E0          MOVX    A,@DPTR              ; MOVX accesses xdata space
900000      MOV     DPTR,#testvar2
F0          MOVX    @DPTR,A
```

And, to initialize the SRAM the value is also stored in flash, and copied to the correct SRAM location in the C startup code.

PSoC 3 has 8 times more flash than SRAM. If SRAM is being used up, it may make sense to keep some const variables, strings, or arrays in flash. In the Keil C compiler, to force storage of a variable or array in flash, you must use the keyword ‘code’ (or CYCODE) in the declaration:

```
code const char testvar = 37;

void main()
{
    char testvar2 = testvar;
```

The corresponding 8051 assembler shows that the const variable ‘testvar’ is now stored in flash:

```
900000      MOV     DPTR,#testvar
E4          CLR     A
93          MOVC    A,@A+DPTR            ; MOVC accesses code space
900000      MOV     DPTR,#testvar2
F0          MOVX    @DPTR,A
```

Because of the nature of the MOVC instruction, this actually costs at least one extra byte to set up the index in the accumulator. For this reason, use this method only when truly necessary.

Be careful about the syntax. The ‘const’ is not necessary but may be needed for portability, and one declaration results in a compile error:

```
code const char testvar = 37;  /* stores in flash */
code char testvar = 37;        /* stores in flash */
const char code testvar = 37;  /* stored in flash */
char code testvar = 37;        /* stores in flash */
const char testvar = 37;       /* stores in SRAM  */
const code char testvar = 37;  /* compile error   */
```

The syntax for keeping arrays and strings in flash is similar:

```
const float code array[512] = { . . . };
code const char hello[] = "Hello World";
```

Finally, by forcing location of a variable or array in flash, you can use memory-specific pointers, which increases code efficiency (see Topic #2: Pointers). Of course, the same is true if you force a variable into any other memory space.

Also, with regard to the differing syntax in the previous examples, the following is from Keil's documentation on declaring memory space:

```
/* older method, may not be supported in future versions of the compiler
   [Memory space]  [Qualifier and Data type]  variable_name  */
code const int testvar; // example
/* preferred method
   [Qualifier and Data type]  [Memory space]  variable_name  */
int idata testvar; // example

/* preferred method for declaring pointer variables
   [Qualifier and Data type] [Data type memory space]
     * [Variable memory space] var_name */
/* pointer stored in xdata points to an integer stored in data */
int data * xdata p;
```

Topic #4: Passing Arguments to Functions

C function arguments are usually passed on a CPU's hardware stack. But because the 8051 hardware stack size is limited to less than 256 bytes, the Keil compiler does not pass arguments on the stack. Instead, it uses either registers `R1–R7` or fixed memory locations. The preferred method is to use registers because it's faster and uses fewer code bytes. However, this method has some limitations, as Table 3 shows.

Table 3. Keil Scheme for Passing Function Arguments in Registers

Argument Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R7, R6 (MSB)	R7–R4 (MSB)	R3 (mem type), R2 (MSB), R1
2	R5	R5, R4 (MSB)	R7–R4 (MSB)	R3 (mem type), R2 (MSB), R1
3	R3	R3, R2 (MSB)	-	R3 (mem type), R2 (MSB), R1

If an argument does not fit into the scheme in Table 3 then it is passed in a fixed memory location. So as much as possible, functions should be limited to three arguments, but even then the compiler may not pass all three arguments in registers. For example, the following C code might be written to search an array:

```
/* search function, with three arguments */
int search(char *addr, int nbytes, char c);

char array[300];

void main()
{
    search(array, sizeof(array), 'X');
```

And the following is the corresponding 8051 assembler:

```
7B01       MOV     R3,#01H                 ; first argument in regs,
7A00       MOV     R2,#HIGH array          ;  generic pointer
7900       MOV     R1,#LOW array
900000     MOV     DPTR,#?_search?BYTE+05H ; third argument,
7458       MOV     A,#058H                 ;  in memory
F0         MOVX    @DPTR,A
7D00       MOV     R5,#02CH                ; second argument in regs
7C02       MOV     R4,#01H
120000     LCALL   _search                 ; 19 bytes, 23 cycles
```

According to Table 3, the third argument must be passed in a fixed memory location even though it's just a char and R6 and R7 are not being used. The code can be more efficient if all arguments are passed in registers. Two methods are available for achieving that.

First, note that in Table 3, the third argument and generic pointer arguments both have only one placement option, that is, R1–R3. (See Topic #2: Pointers for a discussion of generic pointers.) If you simply change the order of the arguments, to make the generic pointer the third argument, then all of the arguments can be passed in registers:

```
/* search function, with three arguments */
int search(int nbytes, char c, char *addr);

7B01        MOV     R3,#01H         ; third argument
7A00        MOV     R2,#HIGH array
7900        MOV     R1,#LOW array
7D58        MOV     R5,#058H        ; second argument
7F00        MOV     R7,#02CH        ; first argument
7E02        MOV     R6,#01H
120000      LCALL   _search         ; 15 bytes, 16 cycles
```

Another solution is to use a memory-specific pointer (see Topic #2: Pointers), which requires only two bytes instead of three:

```
/* search function to be called,
   with three arguments */
int search(char xdata *addr, int nbytes, char c);

7E00        MOV     R6,#HIGH array  ; first argument
7F00        MOV     R7,#LOW array
7B58        MOV     R3,#058H        ; third argument
7D00        MOV     R5,#02CH        ; second argument
7C02        MOV     R4,#01H
120000      LCALL   _search         ; 13 bytes, 14 cycles
```

In this case, you must have the array in external SRAM. If you move it to flash or internal SRAM then the function must be changed.

In this example, by using one of the two techniques, you can save as much as 31% bytes and 40% cycles on a function call, depending on the function's arguments.

Note that arguments of type 'bit' cannot be passed in a register; they are always passed in a fixed memory location in the bit space in the 8051 internal memory. This is generally acceptable because very little code is needed to access a bit variable. However bit variables should be declared at the end of a function's argument list, to keep the other arguments within the Table 3 scheme.

A similar concept applies to function return values, as Table 4 shows.

Table 4. Keil Scheme for Passing Function Return Values in Registers

Return Type	Register
bit	Carry flag
char, 1-byte pointer	R7
int, 2-byte pointer	R7, R6 (MSB)
long, float	R7–R4 (MSB)
generic pointer	R3 (mem type), R2 (MSB), R1

Function return values, including those of type 'bit', are always passed in registers. This can, in turn, affect the order of function arguments. For example, suppose you want to use the search function's return value as an argument for another search. The following code finds in an array the last instance of a character before the first instance of another character:

```
int search(char xdata *addr, int nbytes, char c);   /* search forward */
int searchb(char xdata *addr, int nbytes, char c);  /* search backward */

char array[300];

void main()
{
    searchb(array, search(array, sizeof(array), 'X'), 'A');
```

Here's the corresponding assembler:

```
7E00        MOV     R6,#HIGH array
7F00        MOV     R7,#LOW array
7B58        MOV     R3,#058H
7D00        MOV     R5,#00H
7C02        MOV     R4,#02H
120000      LCALL   _search
AC06        MOV     R4,AR6          ; move return value to argument 2
AD07        MOV     R5,AR7
7E00        MOV     R6,#HIGH array
7F00        MOV     R7,#LOW array
7B41        MOV     R3,#041H
120000      LCALL   _searchb
```

It costs an extra 4 bytes and 6 cycles to move the return value, which you can avoid if you reorder the arguments:

```
int search(int nbytes, char xdata *addr, char c);   /* search forward */
int searchb(int nbytes, char xdata *addr, char c);  /* search backward */

char array[300];

void main()
{
  searchb(search(sizeof(array), array, 'X'), array, 'A');

7E00        MOV     R4,#HIGH array
7F00        MOV     R5,#LOW array
7B58        MOV     R3,#058H
7D00        MOV     R7,#00H
7C02        MOV     R6,#02H
120000      LCALL   _search
7E00        MOV     R4,#HIGH array   ; return value is already in R6, R7
7F00        MOV     R5,#LOW array
7B41        MOV     R3,#041H
120000      LCALL   _searchb
```

The main lesson from this example is that if a function argument may be the return value of another function, put that argument first in the argument list whenever possible.

When you write a C function, you usually don't need to care about the order of the function's arguments. With Keil 8051 C, if you pay attention to the argument order, you may gain significant reductions in code size.

Topic #5: Passing Structures

In C, it is possible to pass a structure to a function. You can either pass the structure directly (if it is small) or pass a pointer to a structure. The following is a simple example of passing a structure directly:

```
/* structure definition and instance */
struct myStruct
{
    char x, y;
} testvar = {1, 2};

/* function with structure passed in directly,
   returns the sum of the structure’s members
*/
char doAdd(struct myStruct str)
{
    return str.x + str.y;
}

void main()
{ /* test call for the above function */
    char testvar2 = doAdd(testvar); /* structure passed directly */
```

Note that the resulting assembler is large:

```
    ; doAdd:
    900000      MOV     DPTR,#str+01H           ; get the structure members from
    E0          MOVX    A,@DPTR                 ; the fixed memory location,
    FF          MOV     R7,A
    900000      MOV     DPTR,#str
    E0          MOVX    A,@DPTR
    2F          ADD     A,R7                    ; and add them
    FF          MOV     R7,A
    22          RET

    ; main:
    7B01        MOV     R3,#01H                 ; pass structure members in memory
    7A00        MOV     R2,#HIGH testvar
    7900        MOV     R1,#LOW testvar
    7800        MOV     R0,#LOW ?doAdd?BYTE
    7C00        MOV     R4,#HIGH ?doAdd?BYTE
    7D01        MOV     R5,#01H
    7E00        MOV     R6,#00H
    7F02        MOV     R7,#02H
    120000      LCALL   ?C?COPYAMD              ; Keil library function
    120000      LCALL   doAdd
    900000      MOV     DPTR,#testvar2
    EF          MOV     A,R7
    F0          MOVX    @DPTR,A
```

The alternative, passing by reference (passing a pointer to the structure), results in less code in main() but more code in doAdd(), which makes the two methods approximately equal in this case:

```
/* function with structure passed in by reference,
   returns the sum of the structure's members
*/
char doAdd(struct myStruct *str) {
    return str->x + str->y;
}

void main()
{ /* test call for the above function */
    char testvar2 = doAdd(&testvar);  /* structure passed by reference */

; doAdd:
900000      MOV     DPTR,#str            ; store the pointer in memory
120000      LCALL   ?C?PSTXDATA
900000      MOV     DPTR,#str            ; get the structure members from memory
120000      LCALL   ?C?PLDXDATA
E9          MOV     A,R1
2401        ADD     A,#01H
F9          MOV     R1,A
E4          CLR     A
3A          ADDC    A,R2
FA          MOV     R2,A
120000      LCALL   ?C?CLDPTR
FF          MOV     R7,A
900000      MOV     DPTR,#str
120000      LCALL   ?C?PLDXDATA
120000      LCALL   ?C?CLDPTR
2F          ADD     A,R7                 ; and add them
FF          MOV     R7,A
22          RET

; main:
7B01        MOV     R3,#01H              ; pass structure pointer in registers
7A00        MOV     R2,#HIGH testvar
7900        MOV     R1,#LOW testvar
120000      LCALL   doAdd
900000      MOV     DPTR,#testvar2
EF          MOV     A,R7
F0          MOVX    @DPTR,A
```

The lesson from this topic is simple: try to avoid passing structures to functions, regardless of method. Consider passing structure members instead, or just make the structures static or even global. Also, be careful when using the C operator '->'; it is costly to implement.

Topic #6: Switch Statements

When making a multipath decision based on the state of a variable (for example, when implementing a state machine) you can use either a series of if-else if-else statements or a switch-case construct. To test which method is less costly, first examine the switch option:

```
/* basic state machine
   note that C coding best practices
   require having:
   - a break statement at the end of
     each case, and
   - a default case
*/
char state = 0;

switch (state)
{
case 0:
    state++;
    break;
case 1:
    state--;
    break;
default:
    state = 0;
    break;
}
```

The resulting assembler uses a sequentially scanned jump table with a library function:

```
900000      MOV     DPTR,#state
E0          MOVX    A,@DPTR
120000      LCALL   ?C?CCASE
0000        DW      ?C0002
00          DB      00H
0000        DW      ?C0003
01          DB      01H
0000        DW      00H
0000        DW      ?C0004
```

Now, examine the if-else if-else construct:

```
/* basic state machine */
if (state == 0)
{
    state++;
}
else if (state == 1)
{
    state--;
}
else
{
    state = 0;
}
```

The resulting assembler has a series of compares and jumps:

```
900000      MOV     DPTR,#state         ; if state == 0
E0          MOVX    A,@DPTR
7005        JNZ     ?C0001
E0          MOVX    A,@DPTR             ; state++
04          INC     A
F0          MOVX    @DPTR,A
8010        SJMP    ?C0005
?C0001
900000      MOV     DPTR,#state         ; else if state == 1
E0          MOVX    A,@DPTR
B40104      CJNE    A,#01H,?C0003
14          DEC     A                   ; state--
F0          MOVX    @DPTR,A
8005        SJMP    ?C0005
?C0003:
E4          CLR     A                   ; else state = 0
900000      MOV     DPTR,#state
F0          MOVX    @DPTR,A
?C0005:
```

This code is smaller for a state machine of this size, but for larger state machines it grows at a faster rate than the jump table in the previous code. The general rule for code of any complexity is to use the switch statement.

This is also a perfect example for using a more efficient memory space (see Guideline #3). Moving the variable ‘state’ to the data space, results in large reductions in code size:

```
data char state;

; switch code
E500        MOV     A,state          ; switch state
120000      LCALL   ?C?CCASE
0000        DW      ?C0002           ; jump table
00          DB      00H
0000        DW      ?C0003
01          DB      01H
0000        DW      00H
0000        DW      ?C0004
?C0002:
0500        INC     state            ; case 0: state++
8007        SJMP    ?C0005
?C0003:
1500        DEC     state            ; case 1: state--
8003        SJMP    ?C0005
?C0004:
E4          CLR     A                ; default: state = 0
F500        MOV     state,A
?C0005:

; if – else if – else code
E500        MOV     A,state          ; if state == 0
7004        JNZ     ?C0001
0500        INC     state            ; state++
80DE        SJMP    ?C0005
?C0001:
E500        MOV     A,state          ; else if state == 1
B40104      CJNE    A,#01H,?C0003
1500        DEC     state            ; state--
80D5        SJMP    ?C0005
?C0003:
E4          CLR     A                ; else state = 0
F500        MOV     state,A
?C0005:
```

Further optimizations are available; for example, Keil compiler optimization level 4 optimizes switch / case statements. The previous example has a sequentially scanned table to decide where to jump to, which means that it may take longer to reach the case statement for some values than for others. Keil compiler optimization for speed as opposed to size may change that to a true jump table, where the time to reach a case statement is the same regardless of switch value.

Topic #7: Large Arrays and Structures

Large arrays and structures are handled efficiently by the Keil compiler. If you need to access a structure member or an array element directly, the corresponding address is simply accessed, as the following example shows:

```
/* complex structure with multiple members */
struct myStruct
{
    char  m1;
    int   m2;
    long  m3;
    float m4;
    long  m5[256]; /* including an array member */
} testvar;

/* access one element of the array member */
testvar.m5[3] = 20;

E4           CLR     A                    ; create and store a 32-bit value
7F14         MOV     R7,#014H
FE           MOV     R6,A
FD           MOV     R5,A
FC           MOV     R4,A
900000       MOV     DPTR,#testvar+017H
120000       LCALL   ?C?LSTXDATA          ; library function
```

The code gets more complicated, however, when array indices are calculated:

```
int i = 3;
testvar.m5[i] = 20;

900000       MOV     DPTR,#i             ; set i = 3
E4           CLR     A
F0           MOVX    @DPTR,A
A3           INC     DPTR
7403         MOV     A,#03H
F0           MOVX    @DPTR,A
E4           CLR     A                   ; create and save a 32-bit value
7F14         MOV     R7,#014H
FE           MOV     R6,A
FD           MOV     R5,A
FC           MOV     R4,A
C004         PUSH    AR4
C005         PUSH    AR5
C006         PUSH    AR6
C007         PUSH    AR7
900000       MOV     DPTR,#i             ; calculate offset based on i,
E0           MOVX    A,@DPTR             ; offset should be i * 4
FE           MOV     R6,A
A3           INC     DPTR
E0           MOVX    A,@DPTR
7802         MOV     R0,#02H
?C0004
C3           CLR     C
33           RLC     A
CE           XCH     A,R6
33           RLC     A
CE           XCH     A,R6
D8F9         DJNZ    R0,?C0004
2400         ADD     A,#LOW testvar+0BH  ; load value to address + offset
F582         MOV     DPL,A
7400         MOV     A,#HIGH testvar+0BH
3E           ADDC    A,R6
```

```
F583         MOV     DPH,A
D007         POP     AR7
D006         POP     AR6
D005         POP     AR5
D004         POP     AR4
120000       LCALL   ?C?LSTXDATA            ; library function
```

A lot of code is required to calculate the offset. You can reduce the amount of code by making three changes, to only the index variable:

- Size index variables appropriately. If the number of elements in the array is 256 or less, you need only a 1-byte index. Don’t use a 2-byte index variable unless absolutely necessary.
- Make sure that index variables are unsigned. The previous example highlights a common problem in C for the 8051, which is the use of the ‘int’ type:

  ```
  int i;

  for (i = 0; i < 100; i++)
  { /* do something with testvar[i] */
  ```

  Although using ‘int’ is common practice, it causes variables to be 16-bit with the Keil compiler, which reduces code efficiency. A better method is to use one of the macros supplied by PSoC Creator, and explicitly define the size of the variable and whether it is signed: int8, uint8, int16, uint16, int32, uint32.
- To make offset calculations more efficient, keep index variables in the data space. Index variables are usually automatic and can be overlaid (see Topic #1: Variable Overlay).

  ```
  uint8 data i = 3;
  testvar.m5[i] = 20;


  975003      MOV     i,#03H              ; set i = 3
  E4          CLR     A                   ; load 32-bit value into registers
  7F14        MOV     R7,#014H
  FE          MOV     R6,A
  FD          MOV     R5,A
  FC          MOV     R4,A
  75F004      MOV     B,#04H              ; calculate offset and store the value
  E500        MOV     A,i                 ; using library functions
  900000      MOV     DPTR,#testvar+0BH
  120000      LCALL   ?C?OFFXADD
  120000      LCALL   ?C?LSTXDATA
  ```

Proper declaration and placement of index variables can greatly reduce the amount of code needed to process large structures and arrays.

Topic #8: Compact Data Space

The previous guidelines and topics have shown that the best place to store variables is in the 8051 internal data space or idata space. But if you run out of room in these spaces (even with data overlaying), you don't have to settle for the external (xdata) space / large memory model. There is a more efficient way to use the xdata space: the 'pdata' space or 'compact' memory model. To understand how it works, consider some 8051 assembler, specifically the two forms of the MOVX instruction. The compact form uses R0 or R1 as a pointer into the external data space, and the large form uses DPTR:

```
; compact form
E2          MOVX    A,@R0      ; 3 cycles
F2          MOVX    @R0,A      ; 4 cycles
E3          MOVX    A,@R1      ; 3 cycles
F3          MOVX    @R1,A      ; 4 cycles

; large form
E0          MOVX    A,@DPTR    ; 2 cycles
F0          MOVX    @DPTR,A    ; 3 cycles
```

Although the compact form uses one more cycle than the large form, when you include the bytes to load the pointer register, the number of cycles is the same and one less byte is used:

```
; compact form
A800        MOV     R0,#testvar     ; 3 bytes, 5 cycles
E2          MOVX    A,@R0
A800        MOV     R0,#testvar2    ; 3 bytes, 6 cycles
F2          MOVX    @R0,A

; large form
900000      MOV     DPTR,#testvar   ; 4 bytes, 5 cycles
E0          MOVX    A,@DPTR
900000      MOV     DPTR,#testvar2  ; 4 bytes, 6 cycles
F0          MOVX    @DPTR,A
```

Note that DPTR is a 16-bit "register" (formed from the DPL and DPH registers) and so the large form can address 64 K bytes in the xdata space. R0 and R1 are 8-bit registers and can access only 256 bytes, so how is 64 K in the xdata space accessed using the compact form?

(PSoC 3 actually has a 16 Mbyte xdata space; 3 bytes are used to address this space. The MS bytes, stored in SFRs DPX for large model and MXAX for compact model, have default reset values of zero, so the first 64 K is always available as a default. All PSoC 3 SRAM and most registers are addressed within the first 64 K of the xdata space. See the xdata memory map and discussion in the device datasheet for details.)

For the compact form, the most significant byte is stored in the P2AX register, SFR #A0H. In the compact memory model, the external space is split into 256-byte pages, where P2AX is the page register and R0 or R1 is the index into the page. Although you can, in theory, access the entire 64K of xdata space using the compact form, usually just the first 256 bytes are used for accessing data in a more efficient mode.

With the Keil C compiler, you can define a global, static or automatic variable, structure or array to be in the compact space by using the Keil keyword 'pdata':

```
char pdata testvar[5];

void main()
{
    char pdata testvar2 = testvar[3];
    testvar[1] = 44;
```

And, similar to Topic #1: Variable Overlay, an overlay space exists for the compact data space, called _PDATA_GROUP_. Test the example in Topic #1 with the 'small' keywords changed to 'compact' (or CYCOMPACT), and observe the shared usage of the pdata space.

The main takeaway from this topic is the same as in Guideline #3 – for maximum efficiency, place your data in the appropriate memory space. Although your code may vary, you should generally follow the recommendations in Table 5.

Table 5. 8051 Memory Spaces and Recommended Usage

8051 memory space	Keil compiler keywords	Usage
Internal	data, idata, bit, bdata, small	Bit variables. Automatic variables, especially those used for complex calculations.
External, compact mode	pdata, compact	Frequently accessed variables (global, static or automatic, depending on program).
External, large mode	xdata, large (PSoC Creator default)	Large arrays or structures. Variables accessed less often.

Note that accessing multiple pages in compact mode, by changing `P2AX`, is possible but is not recommended. One reason is that an interrupt handler might use a different page than the background thread. Unless `P2AX` is carefully managed (for example through push / pop operations), your code may end up accessing a different page than intended and a hard-to-find defect may result.

Topic #9: Use All of the Resources in Your PSoC

There is one final method available for reducing code size. It is based on the fact that PSoC is designed to be a flexible device that enables you to build custom functions in programmable analog and digital blocks. For example, in PSoC 3 you have the following peripherals that can act as “co-processors”:

- DMA Controller. Note that the most common CPU assembler instructions are MOV and MOVX, which implies that the CPU spends a lot of cycles just moving bytes around. Let the DMA controller do that instead.
- Digital Filter Block (DFB) – a sophisticated 24-bit sum of products calculator
- Universal Digital Blocks (UDBs). There are as many as 24 UDBs, and each UDB has an 8-bit datapath that can add, subtract, and do bitwise operations, shifts, and cyclic redundancy check (CRC). The datapaths can be chained for word-wide calculations. Consider offloading CPU calculations to the datapaths.
- The UDBs also have programmable logic devices (PLDs) which can be used to build state machines, c.f. the Lookup Table (LUT) Component datasheet. LUTs can be an effective alternative to programming state machines in the CPU using C switch / case statements.
- Analog components including ADCs, DACs, comparators, opamps, as well as programmable switched capacitor / continuous time (SC/CT) blocks from which you can create programmable gain amplifiers (PGAs), transimpedance amplifiers (TIAs), and mixers. Consider doing your processing in the analog domain instead of the digital domain.

PSoC Creator offers a large number of Components to implement various functions in these peripherals. This allows you to develop an effective multiprocessing system in a single chip, significantly offloading functionality from the CPU. This in turn can not only reduce code size but by reducing the number of tasks that the CPU must perform you can reduce CPU speed and thereby reduce power.

For example, with PSoC 3 a digital system can be designed to control multiplexed ADC inputs, and interface with DMA to save the data in SRAM, to create an advanced analog data collection system with zero usage of the CPU.

Cypress offers extensive application note support for PSoC peripherals, as well as detailed data in the device datasheets and technical reference manuals (TRMs). For more information see the PSoC 3 home page at www.cypress.com.

Summary

This application note has demonstrated that:

- The 8051 CPU can be made to work very efficiently when its core internal features are used. These resources must be used carefully because they are limited.
- The efficiency gains can be realized without writing any 8051 assembler code. Keywords for the Keil 8051 C compiler must be used; portability issues can be mitigated by the use of macros provided by PSoC Creator.
- The Keil C compiler provides a number of ways to make a C program work efficiently on the 8051.

After you compile your C code, you should review the resultant assembler and understand why the particular instructions are there. There are two ways to do that in PSoC Creator.

1. Bring up the list file corresponding to the compiled C file (*filename.lst*). The default PSoC Creator project build setting is to create a list file. To find it, in the PSoC Creator Workspace Explorer window click the **Results** tab.
2. Use the disassembly window in the debugger. That window shows mixed source and assembler, which helps in debugging. However, the disadvantage is that you must have working target hardware and a project that builds correctly before you can use the debugger.

Note that all of the techniques described in this application note were done using compiler optimization level 3. Further gains may be achievable by using higher levels of compiler optimization, at the cost of possible difficulties in debugging. For details, see the PSoC Creator Help topic “Compiler Build Settings” and the Keil help topic “OPTIMIZE Compiler Directive”.

The best way to learn more about coding for the 8051 is to review the Keil C keywords, which can be found in PSoC Creator menu Help, Documentation, Keil, Cx51 Compiler User’s Guide, and Language Extensions.

About the Author

Name:	Mark Ainsworth
Title:	Applications Engineer Principal
Background:	Mark Ainsworth has a BS in Computer Engineering from Syracuse University and an MSEE from University of Washington, as well as many years experience designing and building embedded systems.

Document History

Document Title: AN60630 - PSoC® 3 - 8051 Code and Memory Optimization

Document Number: 001-60630

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2901594	MKEA	03/30/10	New application note.
*A	3209904	MKEA	03/30/11	Changed title according new standards. Added clarifications for CY macros and compiler optimization, and other text and code.
*B	3248324	MKEA	05/04/11	Added Advanced Topics and updated all sections.
*C	3259272	MKEA	05/17/11	Fixed PDF
*D	3275139	MKEA	06/06/11	Template Fix
*E	3535835	MKEA	02/28/2012	Updated template. Modified Title and Abstract.
*F	3946665	MKEA	03/27/2013	Added text to the Keil 8051 Memory Models section. Added a section on function pointers. Added keywords. Other minor text and formatting updates.
*G	4282039	MKEA	02/14/2014	Added examples of bit variable usage. Broke out separate Guideline #4 and added Topic #9. Updated to *L template. Miscellaneous minor edits.
*H	5713610	AESATMP9	04/26/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709