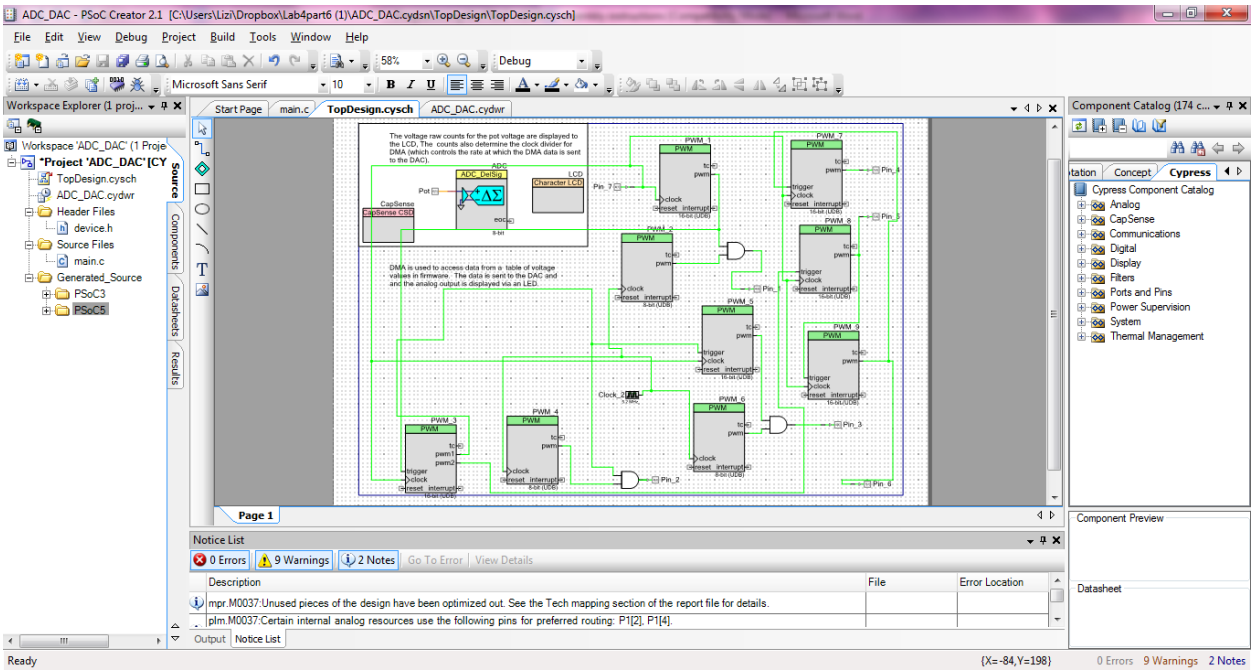
Interfacing C with 8051 assembly code in PSoC® Creator for PSoC3 User's Guide

6.115 Power Electronics Laboratory
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Introduction

Cypress' PSoC (Programmable System on Chip) is a microcontroller with internal peripherals (timers, PWM generators, DACs, etc.) as well as configurable digital blocks. Its design environment PSoC Creator is free software that provides you with a space to link your C code and your configurations of the digital peripherals, GPIO pins, clocking resources, etc. See the powerpoint Introduction to PSoC Creator for more information on getting started with Creator.



In this document, you will learn how to include 8051 assembly routines in your C code for a PSoC 3 device (8051 core). This is useful if:

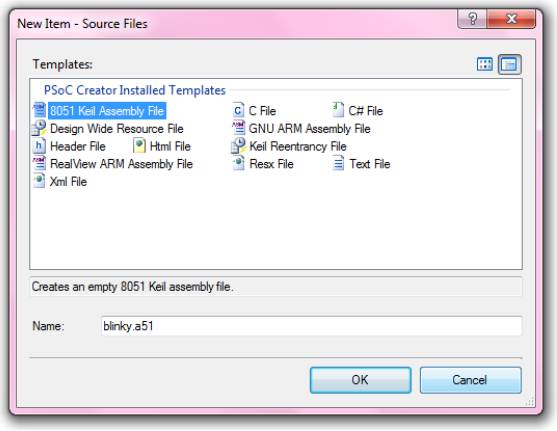
a) You have assembly code already written that you wish to use
b) You need to improve the speed of a particular function
c) You want to manipulate SFRs or memory-mapped I/O devices directly from assembly
d) You are a 6.115 student studying 8051 assembly, and want to use it in the bulk of your final project!

Writing your assembly file

Creator uses the C compiler Keil, so in order to write an 8051 routine, you have to obey Keil's compiler rules.

You can pass values to and from your assembly function with specific registers. Also, your assembly function will have access to any global variables defined in your C code and your C code will have access to any public variables defined in your assembly code.

First, I created a new 8051 source file: Right click on Source Files→New Item→8051 Keil Assembly File. Name the file blinky.a51 and click OK.

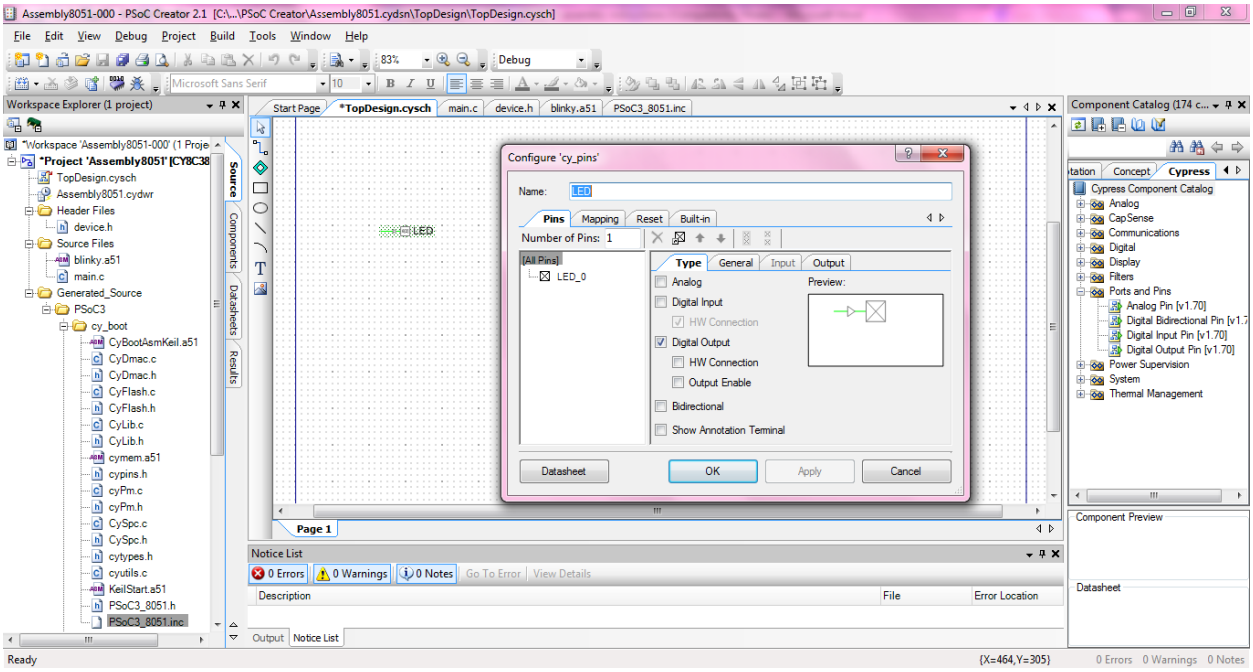


Here is an assembly function written in Creator. Several A51 assembler statements are needed and are included with comments indicating the function of each non-familiar line:

```
$NOMOD51                    ; Suppress SFR definitions for an 8051 device
$INCLUDE (PSoC3_8051.inc)   ; Use definitions in this include file

_COOL SEGMENT CODE          ; This line declares a segment of 8051 code.
                            ; Underscore before name of function with
                            ; arguments passed in registers, according
                            ; to Keil rules. The segment name is COOL
                            ; and it is stored in the CODE memory space.
RSEG _COOL                  ; This line selects this code segment.

PUBLIC _inc_count           ; Specifies that this function can be used in
                            ; other object modules (ie. your C code)
                            ; Again, the initial underscore means that this
                            ; function has arguments passed in registers.

_inc_count:                 ; Your assembly function here.
    mov A, R7
    CPL A
    mov R0, #40h
    loop_back0:
        mov R1, #0FFh
        loop_back1:
            mov R2, #0FFh
            loop_back2:
                djnz R2, loop_back2
            djnz R1, loop_back1
        djnz R0, loop_back0
      mov R7, A
      ret

END                         ; Marks the end of the input file for assembler
```

Writing your C code

Before you write any C code, you should place a digital output pin on the Cypress schematic sheet. Click on TopDesign.cysch, find the Component catalog on the right and click on Ports and Pins. Drag a digital output pin onto the schematic sheet and name it LED. We do not need a hardware connection.



By linking this internal pin to a physical pin on the chip, we can send our PWM signal to the pin attached to the LED on the kit. We can vary the duty cycle of this blinking as usual—by changing the values of the registers R0-R2 in the above code.

In the main.c source file, you should use your assembly function inc_count. You can pull up the Pins datasheet for the component APIs. There you will find the Pin_Write(x) function to write a value directly to an output pin.

```
#include <device.h>

{
    uint8 count = 0;

    while(1)
    {
        count = inc_count(count);
        LED_Write(count);
    }
}
```

The last thing we need to do is include a prototype of our function in the device.h header file. Open this file and add your routine prototype:

```
#ifndef DEVICE_H
#define DEVICE_H
#include <project.h>

uint8 inc_count(uint8 count);       /* inc_count prototype */

#endif
```

Dealing with Global Variables

Global variables you create in your C programs are stored in the memory area specified or in the default memory area implied by the memory model. The assembly label for the variable is the variable name. For example, for the following global variables:
unsigned int bob;
unsigned char jim;
the compiler generates the following assembler code:

```
?DT?MAIN          SEGMENT DATA
    PUBLIC jim
    PUBLIC bob

  RSEG  ?DT?MAIN
      bob:   DS   2
      jim:   DS   1
; unsigned int bob;
; unsigned char jim;
```

To access these variables in assembler, you must create an extern declaration that matches the original declaration. For example:
EXTERN DATA(jim)
If you use in-line assembler, you may simply use C extern variable declarations to generate the assembler EXTERN declarations.
You may access global variables in assembler using their label names. For example:
MOV A, jim

Note:
Type information is not transmitted to your assembler routines. Assembly code must explicitly know the type of the global variable and the order in which it is stored.
The EXTERN definitions for a C external variable are generated only if the variable is referenced by the C code in the module. If an external variable is only referenced by in-line assembly code, you must declare them in in-line assembly code within that module.

Passing in Registers

C functions may pass parameters in registers and fixed memory locations. A maximum of 3 parameters may be passed in registers. All other parameters are passed using fixed memory locations. The following tables define which registers are used for passing parameters.

Arg Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7 (MSB in R6,LSB in R7)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
2	R5	R4 & R5 (MSB in R4,LSB in R5)	R4—R7	R1—R3 (Mem type in R3, MSB in R2, LSB in R1)
3	R3	R2 & R3 (MSB in R2,LSB in R3)		R1—R3 (Mem type in R3, MSB in R2, LSB in R1)

The following examples clarify how registers are selected for parameter passing.

Declaration	Description
func1 (int a)	The first and only argument, **a**, is passed in registers R6 and R7.
func2 (int b, int c, int *d)	The first argument, **b**, is passed in registers R6 and R7. The second argument, **c**, is passed in registers R4 and R5. The third argument, **d**, is passed in registers R1, R2, and R3.
func3 (long e, long f)	The first argument, **e**, is passed in registers R4, R5, R6, and R7. The second argument, **f**, cannot be located in registers since those available for a second parameter with a type of long are already used by the first argument. This parameter is passed using fixed memory locations.
func4 (float g, char h)	The first argument, **g**, passed in registers R4, R5, R6, and R7. The second parameter, **h**, cannot be passed in registers and is passed in fixed memory locations.

Register Usage

Assembler functions may change all register contents in the currently selected register bank as well as the contents of the ACC, B, DPTR, and PSW registers.

When invoking a C function from assembly, assume that these registers are destroyed by the C function that is called.