C PROGRAMMING FOR DP #2

with the output of a logic level decoder as we did for the static 7-segment LED display in Figure 7-17. The reason for this is that LCD's rapidly and irreversibly deteriorate if a steady DC voltage of more than about 50 mV is applied between a segment and the backplane. For character type LCDs, both the backplane and the segments must be driven by square wave pulses with a frequency of 30-150 Hz as shown in Figure 7-20b. Note that for an off segment, the resultant voltage between the backplane and the segment is zero and for an on segment the resultant voltage is a non-zero value. To the driving circuitry, the segment-backplane sandwich appears just as a somewhat leaky capacitor, so CMOS gates can easily provide the required drive signals. However, providing the required drive signals and multiplexing would be difficult to accurately produce directly with standard microprocessor output ports, so most LCD displays are available as modules that have an integrated controller which implements these functions.

As an example for this section, suppose that you have decided to add a character display to a portable instrument you are developing to measure and display the distance to an object using an ultrasonic rangefinder. From the product specification you know that you need a display consisting of two character rows of 16-20 characters each. The lowest-cost choices are an LED display module and an LCD display module. Since the instrument is portable, power dissipation is a major consideration. Assuming the instrument will be used mostly in daylight conditions where backlighting is not required most of the time, the LCD module is the best choice based on the criterion of minimum power dissipation. However, since the instrument may also be used in low light conditions, a transflective display with backlight capability is needed.

As the LCD module for this design example, we will use the 2-line x20 character NHD-C0220BiZ-FSW-FBW-3VM device LCD transflective display module from Newhaven Display International. This module has a built-in Sitronix ST7036 controller that handles all the user commands and data sent from the processor by a user. Figure 7-21 shows how one of these LCDs modules can be connected with a microprocessor. Commands and data are sent from the processor to the module over a simple, two wire I^2C serial interface that consists of the SCL and SDA lines. As we will discuss in much greater detail later in the chapter, I^2C is used to interface with many different types of modules, so most current processors have built-in programmable I^2C controllers. The TI Sitara AM335X processors, for example, have three I^2C controllers, I2C (0-2).

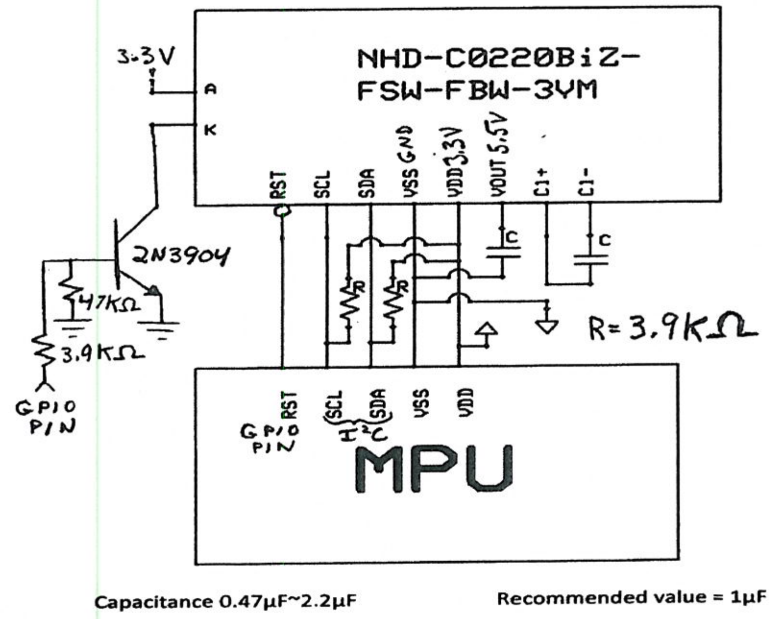


Figure 7-21. Interfacing an NHD-C0220BiZ-FSW-FBW-3VM LCD module with a microprocessor.

returned value handled in main(). Then in the next section we will briefly discuss the BCD2BIN procedure in 4-32b.

Cforasm.c

```
/* Example of C program that calls asm program */
/* Copyright Doug Hall                         */

unsigned short BCD = 0x47;
unsigned short BIN;

extern unsigned short _BCD2BIN(unsigned short b);

int main() {
  BIN = _BCD2BIN(BCD);
 }
```

(a)

asmforc.s

```
@ ARM ASM procedure to convert BCD byte to Binary
@ Doug Hall

.text
.global _BCD2BIN

_BCD2BIN:
        MOV     R11,R0              @ BCD VALUE PASSED IN R0
        MOV     R12,R0              @ COPIES IN R11 AND R12 (scratch registers)
        AND     R11,R11,#0x0F       @ MASK ALL BUT LS NIBBLE
        AND     R12,R12,#0xF0       @ MASK ALL BUT SECOND MS NIBBLE
        MOV     R12,R12,LSR #4      @ MOVE TO LS NIBBLE POSITION
        ADD     R12,R12,R12,LSL #2  @ MULTIPLY BY 5
        MOV     R12,R12,LSL #1      @ MULTIPLY BY 2
        ADD     R12,R12,R11         @ ADD LS NIBBLE FROM R11
        MOV     R0,R12              @ RESULT TO R0 FOR RETURN
        MOV     PC,LR               @ RETURN

.end
```

(b)

Figure 4-32. Program with C and Assembly modules. (a) C mainline. (b) Assembly language procedure.

write procedures that are callable from your C programs if you just follow the basic rules. It is also possible to call C library functions from an assembly language program but, since you will usually write the main program for an application in C or some other high-level language, it is much more likely that you will want to call an assembly language procedure from a C program as we have shown you how to do in this section. In the next and last section of this chapter, we briefly show you how it is possible to write short sections of assembly language code directly in C programs

IN-LINE ASSEMBLY LANGUAGE PROGRAMMING

Most current C compilers such have a built-in assembler that makes it possible for you to embed short sections of assembly language directly in your C programs to perform simple initializations or other bit-twiddling operations. Figure 4-34 shows how the BCD2BIN operation that we did with a procedure call in Figure 4-32 can be done within a C program as *in-line assembly code*. The assembly language section starts with two underscores followed by the keyword asm. The assembly language statements are then enclosed in parentheses marks as shown. Note that the quotation marks and the \n\t are required on each assembly language instruction. Also note the terminating ; after the close parenthesis at the end of the assembly language instructions. Further note that the comments for assembly language instructions in a C program must use the C language comment format as shown. Also, a section of in-line assembly code in a C program must be within a C function. For this simple example, we just put it directly in the main() function.

```
                                                        inline.c
/* Example of C program that uses in-line assembly */
/* Doug Hall              */

static unsigned short BCD = 0x47;
static unsigned short BIN;

int main(void)
{
__asm
(
        "LDR     R11,=BCD\n\t"
        "LDRH R11,[R11]\n\t"                  /* BCD VALUE INTO R11 */
        "MOV  R12,R11\n\t"                    /* R11,R12 ARE scratch registers*/
        "AND     R11,R11,#0x0F\n\t"           /* MASK ALL BUT LS NIBBLE */
        "AND     R12,R12,#0xF0\n\t"           /* MASK ALL BUT SECOND MS NIBBLE*/
        "MOV     R12,R12,LSR #4\n\t"          /* MOVE TO LS NIBBLE POSITION */
        "ADD     R12,R12,R12,LSL #2\n\t"      /* MULTIPLY BY 5 */
        "MOV     R12,R12,LSL #1\n\t"          /* MULTIPLY BY 2 */
        "ADD     R12,R12,R11\n\t"             /* ADD LS NIBBLE FROM R11*/
        "LDR     R11,=BIN\n\t"                /* LOAD ADDRESS OF BIN*/
        "STRH    R12,[R11]");                 /* WRITE RESULT TO BIN */
}
```

Figure 4-34. Example of C program that uses inline assembly code.

In the program in Figure 4-34, we first use the LDR R11,=BCD instruction to load the address of BCD into R11 then use the LDRH R11, [R11] instruction to read the value from the

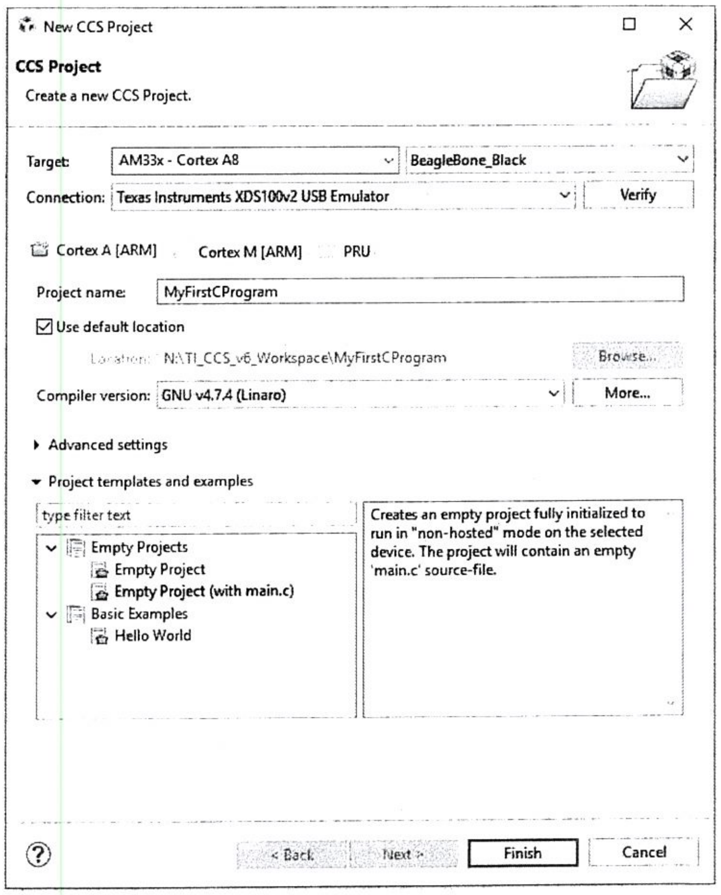
Getting Started with C in CCS

Charles Stoll

February 2018

1 Setting up the Project

Setting up the project is nearly identical to setting up and assembly process. Follow the tutorial from ECE 371 until you are about to finish creating the project. At this point, this tutorial says to "make sure that you are not selecting "Empty Project (with main.c)" option. However, this project is in C so make sure to select the correct empty project like so:



Target: AM33x - Cortex A8 | BeagleBone_Black

Connection: Texas Instruments XDS100v2 USB Emulator

Cortex A [ARM] Cortex M [ARM] PRU

Project name: MyFirstCProgram

☑ Use default location

Location: N:\TI_CCS_v6_Workspace\MyFirstCProgram

Compiler version: GNU v4.7.4 (Linaro)

- Empty Projects
  - Empty Project
  - Empty Project (with main.c)
- Basic Examples
  - Hello World

Creates an empty project fully initialized to run in "non-hosted" mode on the selected device. The project will contain an empty 'main.c' source-file.

Figure 1: Everything is the same as in the tutorial, just make sure to highlight the "Empty Project (with main.c)"

Now, follow the rest of the tutorial in order to set the correct properties and configuration targets for the project.

2 Test Program

In order to make sure the project is working, copy and paste the given LED blinker C code into the main.c file. You must then change a few lines in the startup_ARMCA8.s file (as shown below). After this, save both files, rebuild the project, connect the beagle bone black, and load your program–all exactly how you would do it in assembly.

Now, go to the startup_ARMCA8.s file and change the following sections:

```
Enter_BootLoader:
         LDR   r10,= _start                     @ Get the address of _start
         MOV   lr,pc                            @ Dummy return
         BX    r10                              @ Branch to main
         SUB   pc, pc, #0x08                    @ looping
```

To

```
Enter_BootLoader:
         LDR   r10,= main                       @ Get the address of main
         MOV   lr,pc                            @ Dummy return
         BX    r10                              @ Branch to main
         SUB   pc, pc, #0x08                    @ looping
```

and change

```
__isr_vector:
     ldr     pc, [pc,#24]       @ 0x00 Reset
     ldr     pc, [pc,#-8]       @ 0x04 Undefined Instruction
     ldr     pc, [pc,#24]       @ 0x08 Supervisor Call
     ldr     pc, [pc,#-8]       @ 0x0C Prefetch Abort
     ldr     pc, [pc,#-8]       @ 0x10 Data Abort
     ldr     pc, [pc,#-8]       @ 0x14 Not used
     ldr     pc, [pc,#-8]       @ 0x18 IRQ interrupt
     ldr     pc, [pc,#-8]       @ 0x1C FIQ interrupt
```

To

```
__isr_vector:
     ldr     pc, [pc,#24]       @ 0x00 Reset
     ldr     pc, [pc,#-8]       @ 0x04 Undefined Instruction
     ldr     pc, [pc,#24]       @ 0x08 Supervisor Call
     ldr     pc, [pc,#-8]       @ 0x0C Prefetch Abort
     ldr     pc, [pc,#-8]       @ 0x10 Data Abort
     ldr     pc, [pc,#-8]       @ 0x14 Not used
     ldr     pc, =int_handler    @ 0x18 IRQ interrupt
     ldr     pc, [pc,#-8]       @ 0x1C FIQ interrupt
```

These will correctly hook the startup file into the given C program. It should be noted that the given C program does not have any interrupts occurring. This is just to show the method to hook an interrupt into C code.

If you have done everything successfully, the LEDs on the Beagle Bone Black will now be blinking on and off in a pattern. If you choose to step through the program, you will notice that you now have to click the "assembly step into" button repeatedly for each line of code. This is because each line of C is converted into several lines of assembly. At this point, you should play around with the code until you feel comfortable understanding what is happening and how to write C code that interact with registers.

```
//Defines Section
#define HWREG(x) (*((volatile unsigned int *)(x)))

//GPIO defines
#define GPIO1BA 0x4804C000
#define GPIO_SET_DATA_OUT 0x194
#define GPIO_CLEAR_DATA_OUT 0x190

//INTC defines
#define INTCBA 0x48200000

//Timer 5 defines
#define TIMER5_BA 0x48046000

//other defines
#define CLKWKUPS 0x44E00000
#define LIGHT_BITS 0x01E00000

//Function Declarations

void IntMasterIRQEnable();
void int_handler();
void turn_off_leds();
void turn_on_leds();
void timer5_int();
void wait_loop();
void return_from_int();

//global variables
int current_state = 1;
int x;
volatile unsigned int USR_STACK[100];
volatile unsigned int INT_STACK[100];

int main(void) {
      //SET UP STACKS
      //init USR stack
      asm("LDR R13, =USR_STACK");
      asm("ADD R13, R13, #0x100");
      //init IRQ stack
      asm("CPS #0x12");
      asm("LDR R13, =INT_STACK");
      asm("ADD R13, R13, #0x100");
      asm("CPS #0x13");
      //LED INIT
      HWREG(CLKWKUPS + 0xAC) = 0x2;                    //GPIO1
initialization code
      HWREG(GPIO1BA + GPIO_CLEAR_DATA_OUT) = LIGHT_BITS;
      //Set initial GPIO values
      HWREG(GPIO1BA + 0x134) &= 0xFE1FFFFF;
      // set output enable
```

```
	//TIMER 5 INIT
	HWREG(CLKWKUPS + 0xEC) = 0x2;			//wakeup timer 5
	HWREG(CLKWKUPS + 0x518) = 0x2;			//set clock
speed
	HWREG(TIMER5_BA + 0x10) = 0x1;			//software reset
	HWREG(TIMER5_BA + 0x28) = 0x7;			//clear irqs
	HWREG(TIMER5_BA + 0x2C) = 0x2;			//enable
overflow IRQ

	//INTC INIT
	HWREG(INTCBA + 0x10) = 0x2;			//reset INTC
	HWREG(INTCBA + 0xC8) = 0x20000000;		//unmast
INTC_TINT5

	//ENABLE IRQ
	IntMasterIRQEnable();

	//INIT INTERNAL STATE
	HWREG(TIMER5_BA + 0x3C) = 0xFFFFE000; //set timer for 250
ms
	HWREG(TIMER5_BA + 0x38) = 0x1;			//start timer

	wait_loop();
	return 0;
}

void wait_loop(void)
{
	while(1)
	{
		//do nothing loop
	}
}

void int_handler(void)
{
	if(HWREG(0x482000D8) == 0x20000000)
	{
		timer5_int();
	}
	asm("LDMFD SP!, {LR}");
	asm("LDMFD SP!, {LR}");
	asm("SUBS PC, LR, #0x4");
}

void timer5_int(void)
{
	HWREG(TIMER5_BA + 0x28) = 0x7;		//clear timer5
interrupts
	HWREG(INTCBA + 0x48) = 0x1;		//clear NEWIRQ bit in
INTC
	if(current_state == 1)				//toggle current
```

```
state
      {
            current_state = 0;
            HWREG(TIMER5_BA + 0x3C) = 0xFFFFE000; //set timer for
250 ms
            HWREG(TIMER5_BA + 0x38) = 0x1;              //start
timer
            turn_off_leds();
      }
      else
      {
            current_state = 1;
            HWREG(TIMER5_BA + 0x3C) = 0xFFFFA002; //set timer for
750 ms
            HWREG(TIMER5_BA + 0x38) = 0x1;              //start
timer
            turn_on_leds();
      }                                                 //toggle
finished

}

void turn_on_leds(void)
{
      HWREG(GPIO1BA + GPIO_SET_DATA_OUT) = LIGHT_BITS; //turn on
leds
}

void turn_off_leds(void)
{
      HWREG(GPIO1BA + GPIO_CLEAR_DATA_OUT) = LIGHT_BITS;
      //turn off leds
}


void IntMasterIRQEnable(void)
{
      asm(" mrs         r0, CPSR\n\t"
            "     bic         r0, r0, #0x80\n\t"
            "     msr         CPSR_c, R0");
}
```