



DE1-SoC Computer System with Nios II

For Quartus II 13.1

1 Introduction

This document describes a computer system that can be implemented on the Altera DE1-SoC development and education board. This system, called the *DE1-SoC Computer*, is intended for use in experiments on computer organization and embedded systems. To support such experiments, the system contains embedded processors, memory, audio and video devices, and some simple I/O peripherals. The FPGA programming file that implements this system, as well as its design source files, can be obtained from the University Program section of Altera's web site.

2 DE1-SoC Computer Contents

A block diagram of the DE1-SoC Computer system is shown in Figure 1. As indicated in the figure, the components in this system are implemented utilizing both the FPGA and the *Hard Processor System (HPS)* inside Altera's Cyclone[®]V SoC chip. The FPGA implements a Nios II processor and several peripheral ports: memory, timer modules, audio-in/out, video-in/out, PS/2, analog-to-digital, infrared receive/transmit, and parallel ports connected to switches and lights. The HPS comprises an ARM Cortex A9 dual-core processor and a set of peripheral devices. Instructions for using the HPS and ARM processor are provided in an separate document, called *DE1-SoC Computer System with ARM Cortex-A9*.

2.1 FPGA Components

As shown in Figure 1 many of the components in the DE1-SoC Computer are implemented inside the FPGA in the Cyclone V SoC chip. Several of these components are described in this section, and the others are presented in Section 4.

2.2 Nios II Processor

The Altera Nios[®]II processor is a 32-bit CPU that can be implemented in an Altera FPGA device. Three versions of the Nios II processor are available, designated economy (/e), standard (/s), and fast (/f). The DE1-SoC Computer includes the Nios II/f version, configured with floating-point hardware support.

An overview of the Nios II processor can be found in the document *Introduction to the Altera Nios II Processor*, which is provided in the University Program's web site. An easy way to begin working with the DE1-SoC Computer and the Nios II processor is to make use of a utility called the *Altera Monitor Program*. It provides an easy way to assemble/compile Nios II programs written in either assembly language or the C language. The Monitor Program, which can be downloaded from Altera's web site, is an application program that runs on the host computer connected to the DE1-SoC board. The Monitor Program can be used to control the execution of code on Nios II, list (and

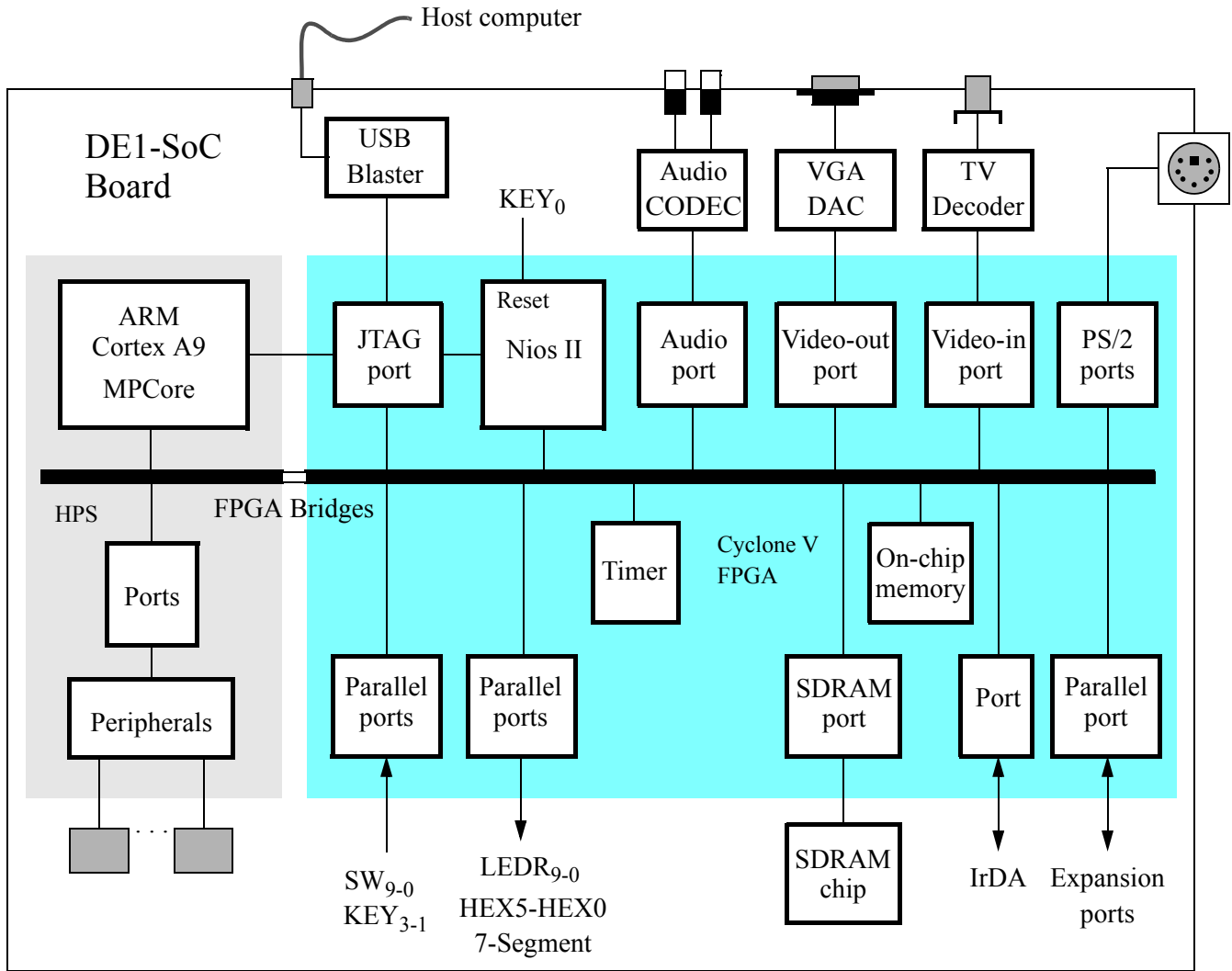


Figure 1. Block diagram of the DE1-SoC Computer.

edit) the contents of processor registers, display/edit the contents of memory on the DE1-SoC board, and similar operations. The Monitor Program includes the DE1-SoC Computer as a predesigned system that can be downloaded onto the DE1-SoC board, as well as several sample programs in assembly language and C that show how to use the DE1-SoC Computer's peripherals. Some images that show how the DE1-SoC Computer is integrated with the Monitor Program are described in Section 8. An overview of the Monitor Program is available in the document *Altera Monitor Program Tutorial*, which is provided in the University Program web site.

As indicated in Figure 1, the Nios II processor can be reset by pressing KEY₀ on the DE1-SoC board. The reset mechanism is discussed further in section 3. All of the I/O peripherals in the DE1-SoC Computer are accessible by the processor as memory mapped devices, using the address ranges that are given in the following subsections.

2.3 Memory Components

The DE1-SoC Computer has an SDRAM port, as well as two memory modules implemented using the on-chip memory inside the FPGA. These memories are described below.

2.3.1 SDRAM

An SDRAM Controller in the FPGA provides an interface to the 64 MB synchronous dynamic RAM (SDRAM) on the DE1-SoC board, which is organized as 32M x 16 bits. It is accessible by the Nios II processor using word (32-bit), halfword (16-bit), or byte operations, and is mapped to the address space 0x00000000 to 0x03FFFFFF.

2.3.2 On-Chip Memory

The DE1-SoC Computer includes a 16-Kbyte memory that is implemented inside the FPGA. This memory is organized as 16K x 8 bits, and spans addresses in the range 0x08000000 to 0x08003FFF.

2.3.3 On-Chip Memory Character Buffer

The DE1-SoC Computer includes an 8-Kbyte memory implemented inside the FPGA that is used as a character buffer for the video-out port, which is described in Section 4.2. The character buffer memory is organized as 8K x 8 bits, and spans the address range 0x09000000 to 0x09001FFF.

2.4 Parallel Ports

There are several parallel ports implemented in the FPGA that support input, output, and bidirectional transfers of data between the Nios II processor and I/O peripherals. As illustrated in Figure 2, each parallel port is assigned a *Base* address and contains up to four 32-bit registers. Ports that have output capability include a writable *Data* register, and ports with input capability have a readable *Data* register. Bidirectional parallel ports also include a *Direction* register that has the same bit-width as the *Data* register. Each bit in the *Data* register can be configured as an input by setting the corresponding bit in the *Direction* register to 0, or as an output by setting this bit position to 1. The *Direction* register is assigned the address *Base* + 4.

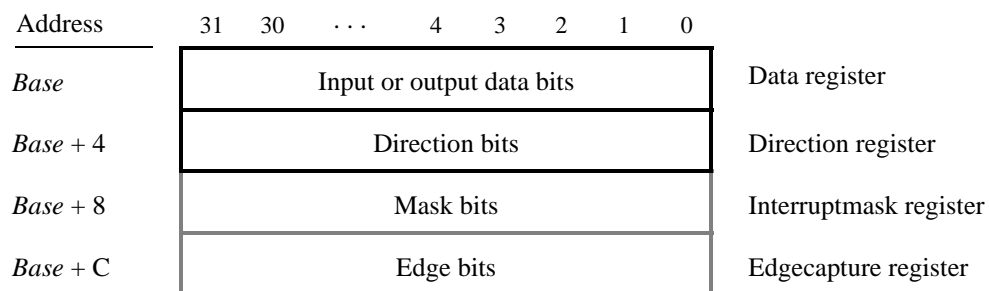


Figure 2. Parallel port registers in the DE1-SoC Computer.

Some of the parallel ports in the DE1-SoC Computer have registers at addresses *Base* + 8 and *Base* + C, as indicated in Figure 2. These registers are discussed in Section 3.

2.4.1 Red LED Parallel Port

The red lights $LEDR_{9-0}$ on the DE1-SoC board are driven by an output parallel port, as illustrated in Figure 3. The port contains a 10-bit write-only *Data* register, which has the address $0xFF200000$. This register can be written using word accesses, and the upper bits not used in the registers are ignored.

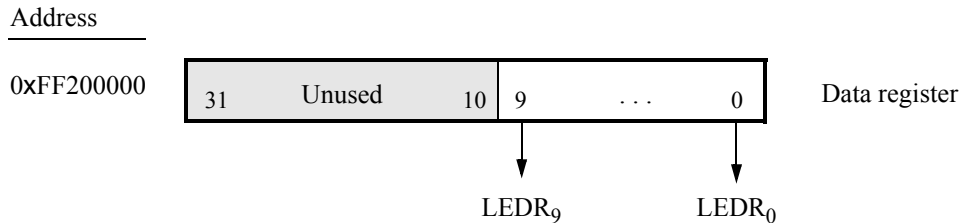


Figure 3. Output parallel port for $LEDR$.

2.4.2 7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE1-SoC board, each of which comprises a 32-bit write-only *Data* register. As indicated in Figure 4, the register at address $0xFF200020$ drives digits $HEX3$ to $HEX0$, and the register at address $0xFF200030$ drives digits $HEX5$ and $HEX4$. Data can be written into these two registers by using word operations. This data directly controls the segments of each display, according to the bit locations given in Figure 4. The locations of segments 6 to 0 in each seven-segment display on the DE1-SoC board is illustrated on the right side of the figure.

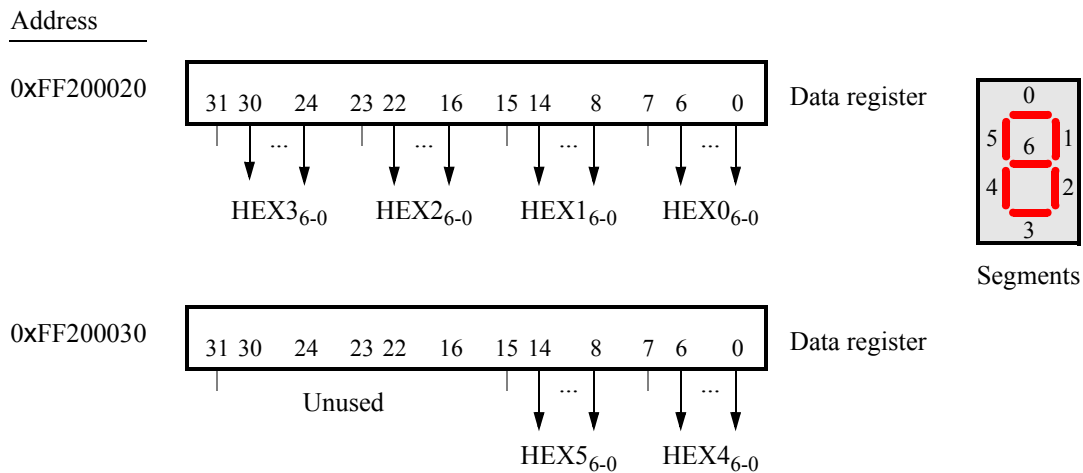


Figure 4. Bit locations for the 7-segment displays parallel ports.

2.4.3 Slider Switch Parallel Port

The SW_{9-0} slider switches on the DE1-SoC board are connected to an input parallel port. As illustrated in Figure 5, this port comprises a 10-bit read-only *Data* register, which is mapped to address $0xFF200040$.

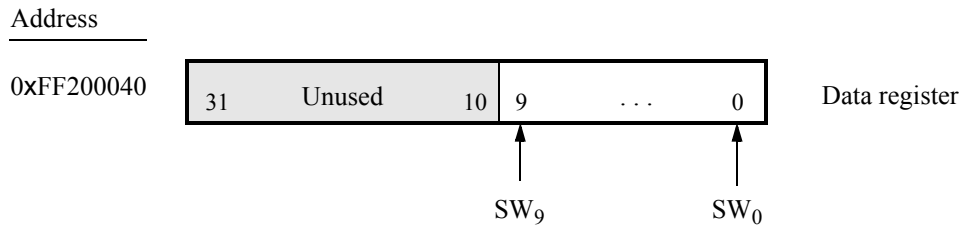


Figure 5. Data register in the slider switch parallel port.

2.4.4 Pushbutton Key Parallel Port

The parallel port connected to the KEY_{3-1} pushbutton switches on the DE1-SoC board comprises three 3-bit registers, as shown in Figure 6 (bit KEY_0 is the Nios II reset signal). These registers have the base address 0xFF200050 and can be accessed using word operations. The read-only *Data* register provides the values of the switches KEY_{3-1} . The other two registers shown in Figure 6, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

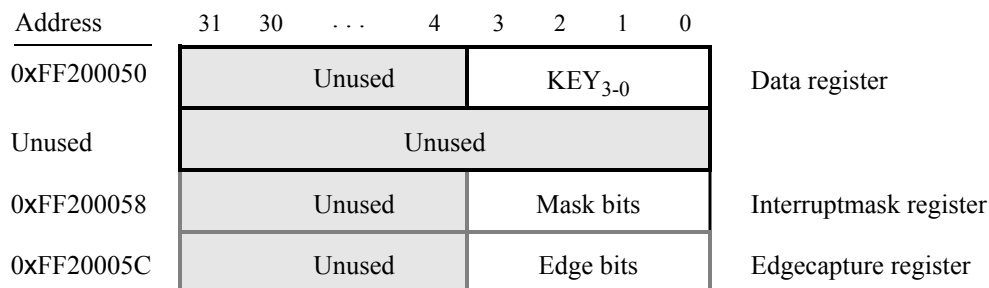


Figure 6. Registers used in the pushbutton parallel port.

2.4.5 Expansion Parallel Port

The DE1-SoC Computer includes two bidirectional parallel ports that are connected to the $JP1$ and $JP2$ 40-pin headers on the DE1-SoC board. These parallel ports include the four 32-bit registers that were described previously for Figure 2. The base address of the port for $JP1$ is 0xFF200060, and for $JP2$ is 0xFF200070. Figure 7 gives a diagram of the 40-pin connectors on the DE1-SoC board, and shows how the respective parallel port *Data* register bits, D_{31-0} , are assigned to the pins on the connector. The figure shows that bit D_0 of the parallel port is assigned to the pin at the top right corner of the connector, bit D_1 is assigned below this, and so on. Note that some of the pins on the 40-pin header are not usable as input/output connections, and are therefore not used by the parallel ports. Also, only 32 of the 36 data pins that appear on each connector can be used.

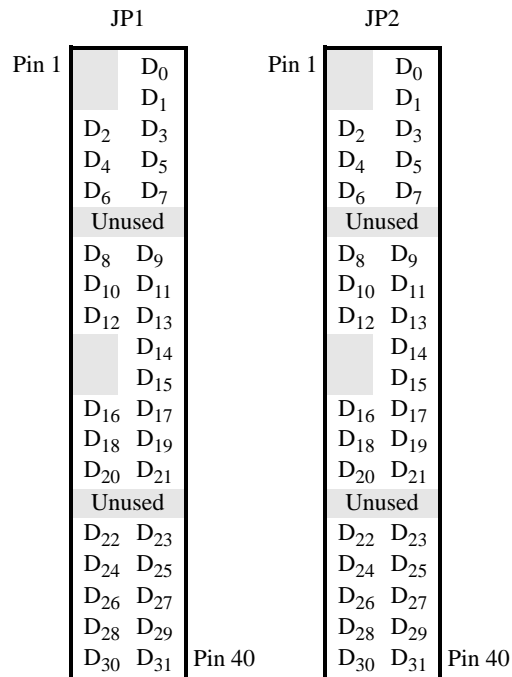


Figure 7. Assignment of parallel port bits to pins on *JP1* and *JP2*.

2.4.6 Using the Parallel Ports with Assembly Language Code and C Code

The DE1-SoC Computer provides a convenient platform for experimenting with Nios II assembly language code, or C code. A simple example of such code is provided in Figures 8 and 9. Both programs perform the same operations, and illustrate the use of parallel ports by using either assembly language or C code.

The code in the figures displays the values of the SW switches on the red lights *LEDR*. It also displays a rotating pattern on 7-segment displays *HEX3*, ..., *HEX0*. This pattern is rotated to the left by using a Nios II *rotate* instruction, and a delay loop is used to make the shifting slow enough to observe. The pattern on the HEX displays can be changed to the values of the SW switches by pressing any of pushbuttons *KEY₃₋₁*. When a pushbutton key is pressed, the program waits in a loop until the key is released.

The source code files shown in Figures 8 and 9 are distributed as part of the Altera Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Getting Started*.

```

/*****
* This program demonstrates the use of parallel ports in the DE1-SoC Computer:
*   1. displays the SW switch values on the red LEDR
*   2. displays a rotating pattern on the HEX displays
*   3. if KEY[3..1] is pressed, uses the SW switches as the pattern
*****/

        .text                /* executable code follows */
        .global    _start

_start:
        /* initialize base addresses of parallel ports */
        movia    r15, 0x10000040    /* SW slider switch base address */
        movia    r16, 0x10000000    /* red LED base address */
        movia    r17, 0x10000050    /* pushbutton KEY base address */
        movia    r20, 0x10000020    /* HEX3_HEX0 base address */
        movia    r19, HEX_bits
        ldwio    r6, 0(r19)        /* load pattern for HEX displays */

DO_DISPLAY:
        ldwio    r4, 0(r15)        /* load input from slider switches */
        stwio    r4, 0(r16)        /* write to red LEDs */
        ldwio    r5, 0(r17)        /* load input from pushbuttons */
        beq     r5, r0, NO_BUTTON
        mov     r6, r4            /* copy SW switch values onto HEX displays */

WAIT:
        ldwio    r5, 0(r17)        /* load input from pushbuttons */
        bne     r5, r0, WAIT        /* wait for button release */

NO_BUTTON:
        stwio    r6, 0(r20)        /* store to HEX3 ... HEX0 */
        roli    r6, r6, 1          /* rotate the displayed pattern */
        movia    r7, 500000        /* delay counter */

DELAY:
        subi    r7, r7, 1
        bne     r7, r0, DELAY
        br     DO_DISPLAY

        .data                /* data follows */
HEX_bits:
        .word    0x0000000F
        .end

```

Figure 8. An example of assembly language code that uses parallel ports.

```

/*****
* This program demonstrates the use of parallel ports in the DE1-SoC Computer
* It performs the following:
*   1. displays the SW switch values on the red lights LEDR
*   2. displays a rotating pattern on the HEX displays
*   3. if KEY[3..0] is pressed, uses the SW switches as the pattern
*****/
int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that the locations will not be cached,
    * even in registers) */
    volatile int * LED_ptr          = (int *) 0xFF200000;    // red LED address
    volatile int * HEX3_HEX0_ptr   = (int *) 0xFF200020;    // HEX3_HEX0 address
    volatile int * SW_switch_ptr   = (int *) 0xFF200040;    // SW slider switch address
    volatile int * KEY_ptr         = (int *) 0xFF200050;    // pushbutton KEY address

    int HEX_bits = 0x0000000F;                // initial pattern for HEX displays
    int SW_value;
    volatile int delay_count;                // volatile so C compiler does not remove loop

    while (1)
    {
        SW_value = *(SW_switch_ptr);         // read the SW slider switch values
        *(LED_ptr) = SW_value;               // light up the red LEDs

        if (*KEY_ptr != 0)                   // check if any KEY was pressed
        {
            HEX_bits = SW_value;             // set pattern using SW values
            while (*KEY_ptr != 0);          // wait for pushbutton KEY release
        }
        *(HEX3_HEX0_ptr) = HEX_bits;        // display pattern on HEX3 ... HEX0

        /* rotate the pattern shown on the HEX displays */
        if (HEX_bits & 0x80000000)
            HEX_bits = (HEX_bits << 1) | 1;
        else
            HEX_bits = HEX_bits << 1;

        for (delay_count = 100000; delay_count != 0; --delay_count);    // delay loop
    }
}

```

Figure 9. An example of C code that uses parallel ports.

2.5 JTAG Port

The JTAG port implements a communication link between the DE1-SoC board and its host computer. This link can be used by the Altera Quartus II software to transfer FPGA programming files into the DE1-SoC board, and by the Altera Monitor Program, discussed in Section 8. The JTAG port also includes a UART, which can be used to transfer character data between the host computer and programs that are executing on the Nios II processor. If the Altera Monitor Program is used on the host computer, then this character data is sent and received through its *Terminal Window*. The programming interface of the JTAG UART consists of two 32-bit registers, as shown in Figure 10. The register mapped to address 0xFF201000 is called the *Data* register and the register mapped to address 0xFF201004 is called the *Control* register.

Address	31	...	16	15	14	...	11	10	9	8	7	...	1	0	
0xFF201000	RAVAIL			RVALID	Unused				DATA				Data register		
0xFF201004	WSPACE			Unused				AC	WI	RI			WE	RE	Control register

Figure 10. JTAG UART registers.

When character data from the host computer is received by the JTAG UART it is stored in a 64-character FIFO. The number of characters currently stored in this FIFO is indicated in the field *RAVAIL*, which are bits 31–16 of the *Data* register. If the receive FIFO overflows, then additional data is lost. When data is present in the receive FIFO, then the value of *RAVAIL* will be greater than 0 and the value of bit 15, *RVALID*, will be 1. Reading the character at the head of the FIFO, which is provided in bits 7–0, decrements the value of *RAVAIL* by one and returns this decremented value as part of the read operation. If no data is present in the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7–0 is undefined.

The JTAG UART also includes a 64-character FIFO that stores data waiting to be transmitted to the host computer. Character data is loaded into this FIFO by performing a write to bits 7–0 of the *Data* register in Figure 10. Note that writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the transmit FIFO is provided in bits 31–16 of the *Control* register. If the transmit FIFO is full, then any characters written to the *Data* register will be lost.

Bit 10 in the *Control* register, called *AC*, has the value 1 if the JTAG UART has been accessed by the host computer. This bit can be used to check if a working connection to the host computer has been established. The *AC* bit can be cleared to 0 by writing a 1 into it.

The *Control* register bits *RE*, *WE*, *RI*, and *WI* are described in Section 3.

2.5.1 Using the JTAG UART with Assembly Language Code and C Code

Figures 11 and 13 give simple examples of assembly language and C code, respectively, that use the JTAG UART. Both versions of the code perform the same function, which is to first send an ASCII string to the JTAG UART, and then enter an endless loop. In the loop, the code reads character data that has been received by the JTAG UART, and echoes this data back to the UART for transmission. If the program is executed by using the Altera Monitor Program, then any keyboard character that is typed into the *Terminal Window* of the Monitor Program will be echoed

back, causing the character to appear in the *Terminal Window*.

The source code files shown in Figures 11 and 13 are made available as part of the Altera Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *JTAG UART*.

```

/*****
* This program demonstrates use of the JTAG UART port in the DE1-SoC Computer
*
* It performs the following:
*   1. sends a text string to the JTAG UART
*   2. reads character data from the JTAG UART
*   3. echos the character data back to the JTAG UART
*****/

        .text                /* executable code follows */
        .global    _start
_start:
        /* set up stack pointer */
        movia    sp, 0x03FFFFFF    /* put stack at top of SDRAM */

        movia    r6, 0x10001000    /* JTAG UART base address */

        /* print a text string */
        movia    r8, TEXT_STRING
LOOP:
        ldb     r5, 0(r8)
        beq    r5, zero, GET_JTAG    /* string is null-terminated */
        call   PUT_JTAG
        addi   r8, r8, 1
        br     LOOP

        /* read and echo characters */
GET_JTAG:
        ldwio   r4, 0(r6)            /* read the JTAG UART Data register */
        andi   r8, r4, 0x8000        /* check if there is new data */
        beq    r8, r0, GET_JTAG      /* if no data, wait */
        andi   r5, r4, 0x00ff        /* the data is in the least significant byte */

        call   PUT_JTAG            /* echo character */
        br     GET_JTAG
        .end

```

Figure 11. An example of assembly language code that uses the JTAG UART (Part a).

```

/*****
* Subroutine to send a character to the JTAG UART
*   r5 = character to send
*   r6 = JTAG UART base address
*****/

        .global    PUT_JTAG
PUT_JTAG:
        /* save any modified registers */
        subi      sp, sp, 4          /* reserve space on the stack */
        stw       r4, 0(sp)         /* save register */

        ldwio     r4, 4(r6)         /* read the JTAG UART Control register */
        andhi     r4, r4, 0xffff    /* check for write space */
        beq       r4, r0, END_PUT    /* if no space, ignore the character */
        stwio     r5, 0(r6)         /* send the character */

END_PUT:
        /* restore registers */
        ldw       r4, 0(sp)
        addi      sp, sp, 4

        ret

        .data                                /* data follows */
TEXT_STRING:
        .asciz   "\nJTAG UART example code\n> "

        .end

```

Figure 11. An example of assembly language code that uses the JTAG UART (Part b).

```

/* function prototypes */
void put_jtag(char);
char get_jtag(void);
/*****
 * This program demonstrates use of the JTAG UART port in the DE1-SoC Computer
 * It performs the following:
 *   1. sends a text string to the JTAG UART
 *   2. reads and echos character data from/to the JTAG UART
 *****/
int main(void)
{
    char text_string[] = "\nJTAG UART example code\n> \0";
    char *str, c;

    /* print a text string */
    for (str = text_string; *str != 0; ++str)
        put_jtag (*str);
    /* read and echo characters */
    while (1)
    {
        c = get_jtag ( );
        if (c != '\0')
            put_jtag (c);
    }
}

/*****
 * Subroutine to send a character to the JTAG UART
 *****/
void put_jtag( char c )
{
    volatile int * JTAG_UART_ptr = (int *) 0xFF201000; // JTAG UART address
    int control;
    control = *(JTAG_UART_ptr + 1); // read the JTAG_UART control register
    if (control & 0xFFFF0000) // if space, echo character, else ignore
        *(JTAG_UART_ptr) = c;
}

```

Figure 12. An example of C code that uses the JTAG UART (Part a).

```

/*****
* Subroutine to read a character from the JTAG UART
* Returns \0 if no character, otherwise returns the character
*****/
char get_jtag( void )
{
    volatile int * JTAG_UART_ptr = (int *) 0xFF201000; // JTAG UART address
    int data;

    data = *(JTAG_UART_ptr);           // read the JTAG_UART data register
    if (data & 0x00008000)              // check RVALID to see if there is new data
        return ((char) data & 0xFF);
    else
        return ('\0');
}

```

Figure 13. An example of C code that uses the JTAG UART (Part b).

2.6 Interval Timer

The DE1-SoC Computer includes a timer module implemented in the FPGA that can be used by the A9 processor. This timer can be loaded with a preset value, and then counts down to zero using the 50-MHz clock signal provided on the DE1-SoC board. The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 14. The 16-bit register at address 0xFF202000 provides status information about the timer, and the register at address 0xFF202004 allows control settings to be made. The bit fields in these registers are described below:

- *TO* provides a timeout signal which is set to 1 by the timer when it has reached a count value of zero. The *TO* bit can be reset by writing a 0 into it.
- *RUN* is set to 1 by the timer whenever it is currently counting. Write operations to the status halfword do not affect the value of the *RUN* bit.
- *ITO* is used for generating interrupts, which are discussed in section 3.
- *CONT* affects the continuous operation of the timer. When the timer reaches a count value of zero it automatically reloads the specified starting count value. If *CONT* is set to 1, then the timer will continue counting down automatically. But if *CONT* = 0, then the timer will stop after it has reached a count value of 0.
- (*START/STOP*) can be used to commence/suspend the operation of the timer by writing a 1 into the respective bit.

The two 16-bit registers at addresses 0xFF202008 and 0xFF20200C allow the period of the timer to be changed by setting the starting count value. The default setting provided in the DE1-SoC Computer gives a timer period of 125 msec. To achieve this period, the starting value of the count is $50 \text{ MHz} \times 125 \text{ msec} = 6.25 \times 10^6$. It is possible to capture a snapshot of the counter value at any time by performing a write to address 0xFF202010. This write operation causes the current 32-bit counter value to be stored into the two 16-bit timer registers at addresses 0xFF202010 and 0xFF202014. These registers can then be read to obtain the count value.

Address	31	...	17	16	15	...	3	2	1	0		
0xFF202000							Unused			RUN	TO	Status register
0xFF202004							Unused	STOP	START	CONT	ITO	Control register
0xFF202008	Not present (interval timer has 16-bit registers)						Counter start value (low)					
0xFF20200C							Counter start value (high)					
0xFF202010							Counter snapshot (low)					
0xFF202014							Counter snapshot (high)					

Figure 14. Interval timer registers.

3 Exceptions and Interrupts

The reset address of the Nios II processor in the DE1-SoC Computer is set to 0x00000000. The address used for all other general exceptions, such as divide by zero, and hardware IRQ interrupts is 0x00000020. Since the Nios II processor uses the same address for general exceptions and hardware IRQ interrupts, the Exception Handler software must determine the source of the exception by examining the appropriate processor status register. Table 1 gives the assignment of IRQ numbers to each of the I/O peripherals in the DE1-SoC Computer. The rest of this section describes the interrupt behavior associated with the interval timer, parallel ports, and serial ports in the DE1-SoC Computer. Interrupts for other devices listed in Table 1 are discussed in section 4.

I/O Peripheral	IRQ #
Interval timer	0
Pushbutton switch parallel port	1
Audio port	6
PS/2 port	7
JTAG port	8
Serial port	10
JP1 Expansion parallel port	11
JP2 Expansion parallel port	12

Table 1. Hardware IRQ interrupt assignment for the DE1-SoC Computer.

3.1 Interrupts from Parallel Ports

Parallel ports implemented in the FPGA in the DE1-Soc Computer were illustrated in Figure 2, which is reproduced as Figure 15. As the figure shows, parallel ports that support interrupts include two related registers at the addresses $Base + 8$ and $Base + C$. The *Interruptmask* register, which has the address $Base + 8$, specifies whether or not an interrupt signal should be sent to the processor when the data present at an input port changes value. Setting a bit location in this register to 1 allows interrupts to be generated, while setting the bit to 0 prevents interrupts. Finally, the parallel port may contain an *Edgecapture* register at address $Base + C$. Each bit in this register has the value 1 if the corresponding bit location in the parallel port has changed its value from 0 to 1 since it was last read. Performing a write operation to the *Edgecapture* register sets all bits in the register to 0, and clears any associated interrupts.

Address	31	30	...	4	3	2	1	0	
$Base$	Input or output data bits								Data register
$Base + 4$	Direction bits								Direction register
$Base + 8$	Mask bits								Interruptmask register
$Base + C$	Edge bits								Edgecapture register

Figure 15. Registers used for interrupts from the parallel ports.

3.1.1 Interrupts from the Pushbutton Switches

Figure 6 showed the registers associated with the pushbutton parallel port. The *Interruptmask* register allows interrupts to be generated when a key is pressed. Each bit in the *Edgecapture* register is set to 1 by the parallel port when the corresponding key is pressed. An interrupt service routine can read this register to determine which key has been pressed. Writing any value to the *Edgecapture* register deasserts the interrupt signal being sent to the processor and sets all bits of the *Edgecapture* register to zero.

3.2 Interrupts from the JTAG UART

Figure 10 showed the data and *Control* registers of the JTAG UART. As we said in Section 2.5, *RAVAIL* in the *Data* register gives the number of characters that are stored in the receive FIFO, and *WSPACE* gives the amount of unused space that is available in the transmit FIFO. The *RE* and *WE* bits in Figure 10 are used to enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when *RAVAIL* for the receive FIFO, or *WSPACE* for the transmit FIFO, exceeds 7. Pending interrupts are indicated in the *Control* register's *RI* and *WI* bits, and can be cleared by writing or reading data to/from the JTAG UART.

3.3 Interrupts from the FPGA Interval Timer

Figure 14, in Section 2.6, shows six registers that are associated with the interval timer. As we said in Section 2.6, the *TO* bit in the *Status* register is set to 1 when the timer reaches a count value of 0. It is possible to generate an interrupt when this occurs, by using the *ITO* bit in the *Control* register. Setting the *ITO* bit to 1 causes an interrupt

request to be sent to the processor whenever *TO* becomes 1. After an interrupt occurs, it can be cleared by writing any value into the *Status* register.

3.4 Using Interrupts with Assembly Language Code

An example of assembly language code for the DE1-SoC Computer that uses interrupts is shown in Figure 16. When this code is executed on the DE1-SoC board it displays a rotating pattern on the HEX 7-segment displays. The pattern rotates to the right if pushbutton *KEY*₁ is pressed, and to the left if *KEY*₂ is pressed. Pressing *KEY*₃ causes the pattern to be set using the SW switch values. Two types of interrupts are used in the code. The HEX displays are controlled by an interrupt service routine for the interval timer, and another interrupt service routine is used to handle the pushbutton keys. The speed at which the HEX displays are rotated is set in the main program, by using a counter value in the interval timer that causes an interrupt to occur every 33 msec.

```
.equ      KEY1, 0
.equ      KEY2, 1
/*****
* This program demonstrates use of interrupts in the DE1-SoC Computer. It first starts the
* interval timer with 33 msec timeouts, and then enables interrupts from the interval timer
* and pushbutton KEYS
*
* The interrupt service routine for the interval timer displays a pattern on the HEX displays, and
* shifts this pattern either left or right. The shifting direction is set in the pushbutton
* interrupt service routine, as follows:
*   KEY[1]: shifts the displayed pattern to the right
*   KEY[2]: shifts the displayed pattern to the left
*   KEY[3]: changes the pattern using the settings on the SW switches
*****/
.text          /* executable code follows */
.global       _start
_start:
/* set up stack pointer */
movia        sp, 0x03FFFFFFC /* stack starts from highest memory address in SDRAM */

movia        r16, 0x10002000 /* internal timer base address */
/* set the interval timer period for scrolling the HEX displays */
movia        r12, 0x190000 /* 1/(50 MHz) × (0x190000) = 33 msec */
sthio        r12, 8(r16) /* store the low halfword of counter start value */
srli         r12, r12, 16
sthio        r12, 0xC(r16) /* high halfword of counter start value */
```

Figure 16. An example of assembly language code that uses interrupts (Part a).


```

/* start interval timer, enable its interrupts */
movi    r15, 0b0111          /* START = 1, CONT = 1, ITO = 1 */
sthio   r15, 4(r16)

/* write to the pushbutton port interrupt mask register */
movia   r15, 0x10000050      /* pushbutton key base address */
movi    r7, 0b01110         /* set 3 interrupt mask bits (bit 0 is Nios II reset) */
stwio   r7, 8(r15)         /* interrupt mask register is (base + 8) */

/* enable Nios II processor interrupts */
movi    r7, 0b011          /* set interrupt mask bits for levels 0 (interval */
wrcctl  ienable, r7        /* timer) and level 1 (pushbuttons) */
movi    r7, 1
wrcctl  status, r7        /* turn on Nios II interrupt processing */

IDLE:
br     IDLE                /* main program simply idles */

.data
/* The two global variables used by the interrupt service routines for the interval timer and the
* pushbutton keys are declared below */

.global  PATTERN
PATTERN:
.word  0x0000000F          /* pattern to show on the HEX displays */

.global  KEY_PRESSED
KEY_PRESSED:
.word  KEY2                /* stores code representing pushbutton key pressed */

.end

```

Figure 16. An example of assembly language code that uses interrupts (Part *b*).

The reset and exception handlers for the main program in Figure 16 are given in Figure 17. The reset handler simply jumps to the `_start` symbol in the main program. The exception handler first checks if the exception that has occurred is an external interrupt or an internal one. In the case of an internal exception, such as an illegal instruction opcode or a trap instruction, the handler simply exits, because it does not handle these cases. For external exceptions, it calls either the interval timer interrupt service routine, for a level 0 interrupt, or the pushbutton key interrupt service routine for level 1. These routines are shown in Figures 18 and 19, respectively.

```

/*****
* RESET SECTION
* The Monitor Program automatically places the ".reset" section at the reset location
* specified in the CPU settings in Qsys.
* Note: "ax" is REQUIRED to designate the section as allocatable and executable.
*/
    .section    .reset, "ax"
    movia      r2, _start
    jmp        r2                /* branch to main program */

/*****
* EXCEPTIONS SECTION
* The Monitor Program automatically places the ".exceptions" section at the
* exception location specified in the CPU settings in Qsys.
* Note: "ax" is REQUIRED to designate the section as allocatable and executable.
*/
    .section    .exceptions, "ax"
    .global    EXCEPTION_HANDLER
EXCEPTION_HANDLER:
    subi       sp, sp, 16        /* make room on the stack */
    stw        et, 0(sp)

    rdctl      et, ct14
    beq        et, r0, SKIP_EA_DEC /* interrupt is not external */

    subi       ea, ea, 4         /* must decrement ea by one instruction */
                                /* for external interrupts, so that the */
                                /* interrupted instruction will be run after eret */

SKIP_EA_DEC:
    stw        ea, 4(sp)        /* save all used registers on the Stack */
    stw        ra, 8(sp)        /* needed if call inst is used */
    stw        r22, 12(sp)

    rdctl      et, ct14
    bne        et, r0, CHECK_LEVEL_0 /* exception is an external interrupt */

NOT_EI:
    br         END_ISR          /* exception must be unimplemented instruction or TRAP */
                                /* instruction. This code does not handle those cases */

```

Figure 17. Reset and exception handler assembly language code (Part a).

```

CHECK_LEVEL_0:                /* interval timer is interrupt level 0 */
    andi    r22, et, 0b1
    beq    r22, r0, CHECK_LEVEL_1
    call   INTERVAL_TIMER_ISR
    br     END_ISR

CHECK_LEVEL_1:                /* pushbutton port is interrupt level 1 */
    andi    r22, et, 0b10
    beq    r22, r0, END_ISR    /* other interrupt levels are not handled in this code */
    call   PUSHBUTTON_ISR

END_ISR:
    ldw    et, 0(sp)          /* restore all used register to previous values */
    ldw    ea, 4(sp)
    ldw    ra, 8(sp)         /* needed if call inst is used */
    ldw    r22, 12(sp)
    addi   sp, sp, 16

    eret
    .end

```

Figure 17. Reset and exception handler assembly language code (Part *b*).

```

.include    "key_codes.s"    /* includes EQU for KEY1, KEY2 */
.extern    PATTERN          /* externally defined variables */
.extern    KEY_PRESSED

/*****
* Interval timer interrupt service routine
*
* Shifts a PATTERN being displayed on the HEX displays. The shift direction
* is determined by the external variable KEY_PRESSED.
*
*****/

.global    INTERVAL_TIMER_ISR
INTERVAL_TIMER_ISR:
    subi    sp, sp, 36        /* reserve space on the stack */
    stw    ra, 0(sp)
    stw    r4, 4(sp)
    stw    r5, 8(sp)
    stw    r6, 12(sp)

```

Figure 18. Interrupt service routine for the interval timer (Part *a*).

```

stw    r8, 16(sp)
stw    r10, 20(sp)
stw    r20, 24(sp)
stw    r21, 28(sp)
stw    r22, 32(sp)

movia  r10, 0x10002000    /* interval timer base address */
sthio  r0, 0(r10)        /* clear the interrupt */

movia  r20, 0x10000020    /* HEX3_HEX0 base address */
addi   r5, r0, 1          /* set r5 to the constant value 1 */
movia  r21, PATTERN       /* set up a pointer to the pattern for HEX displays */
movia  r22, KEY_PRESSED  /* set up a pointer to the key pressed */

ldw    r6, 0(r21)         /* load pattern for HEX displays */
stwio  r6, 0(r20)        /* store to HEX3 ... HEX0 */

ldw    r4, 0(r22)         /* check which key has been pressed */
movi   r8, KEY1           /* code to check for KEY1 */
beq    r4, r8, LEFT      /* for KEY1 pressed, shift right */
rol    r6, r6, r5        /* else (for KEY2) pressed, shift left */
br     END_INTERVAL_TIMER_ISR
LEFT:
ror    r6, r6, r5        /* rotate the displayed pattern right */

END_INTERVAL_TIMER_ISR:
stw    r6, 0(r21)         /* store HEX display pattern */
ldw    ra, 0(sp)         /* Restore all used register to previous */
ldw    r4, 4(sp)
ldw    r5, 8(sp)
ldw    r6, 12(sp)
ldw    r8, 16(sp)
ldw    r10, 20(sp)
ldw    r20, 24(sp)
ldw    r21, 28(sp)
ldw    r22, 32(sp)
addi   sp, sp, 36        /* release the reserved space on the stack */
ret
.end

```

Figure 18. Interrupt service routine for the interval timer (Part b).

```

.include      "key_codes.s"          /* includes EQU for KEY1, KEY2 */
.extern      PATTERN                /* externally defined variables */
.extern      KEY_PRESSED
/*****
* Pushbutton - Interrupt Service Routine
*
* This routine checks which KEY has been pressed. If it is KEY1 or KEY2, it writes this value
* to the global variable KEY_PRESSED. If it is KEY3 then it loads the SW switch values and
* stores in the variable PATTERN
*****/

.global      PUSHBUTTON_ISR
PUSHBUTTON_ISR:
    subi     sp, sp, 20              /* reserve space on the stack */
    stw     ra, 0(sp)
    stw     r10, 4(sp)
    stw     r11, 8(sp)
    stw     r12, 12(sp)
    stw     r13, 16(sp)

    movia   r10, 0x10000050          /* base address of pushbutton KEY parallel port */
    ldwio   r11, 0xC(r10)           /* read edge capture register */
    stwio   r0, 0xC(r10)           /* clear the interrupt */

    movia   r10, KEY_PRESSED        /* global variable to return the result */
CHECK_KEY1:
    andi    r13, r11, 0b0010       /* check KEY1 */
    beq     r13, zero, CHECK_KEY2
    movi    r12, KEY1
    stw     r12, 0(r10)             /* return KEY1 value */
    br      END_PUSHBUTTON_ISR

CHECK_KEY2:
    andi    r13, r11, 0b0100       /* check KEY2 */
    beq     r13, zero, DO_KEY3
    movi    r12, KEY2
    stw     r12, 0(r10)            /* return KEY2 value */
    br      END_PUSHBUTTON_ISR

DO_KEY3:
    movia   r13, 0x10000040         /* SW slider switch base address */
    ldwio   r11, 0(r13)            /* load slider switches */
    movia   r13, PATTERN           /* address of pattern for HEX displays */
    stw     r11, 0(r13)            /* save new pattern */

```

Figure 19. Interrupt service routine for the pushbutton keys (Part a).

```
END_PUSHBUTTON_ISR:
    ldw    ra, 0(sp)           /* Restore all used register to previous values */
    ldw    r10, 4(sp)
    ldw    r11, 8(sp)
    ldw    r12, 12(sp)
    ldw    r13, 16(sp)
    addi   sp, sp, 20

    ret
.end
```

Figure 19. Interrupt service routine for the pushbutton keys (Part b).

3.5 Using Interrupts with C Language Code

An example of C language code for the DE1-SoC Computer that uses interrupts is shown in Figure 20. This code performs exactly the same operations as the code described in Figure 16.

To enable interrupts the code in Figure 20 uses *macros* that provide access to the Nios II status and control registers. A collection of such macros, which can be used in any C program, are provided in Figure 21.

The reset and exception handlers for the main program in Figure 20 are given in Figure 22. The function called *the_reset* provides a simple reset mechanism by performing a branch to the main program. The function named *the_exception* represents a general exception handler that can be used with any C program. It includes assembly language code to check if the exception is caused by an external interrupt, and, if so, calls a C language routine named *interrupt_handler*. This routine can then perform whatever action is needed for the specific application. In Figure 22, the *interrupt_handler* code first determines which exception has occurred, by using a macro from Figure 21 that reads the content of the Nios II interrupt pending register. The interrupt service routine that is invoked for the interval timer is shown in 23, and the interrupt service routine for the pushbutton switches appears in Figure 24.

The source code files shown in Figure 16 to Figure 24 are distributed as part of the Altera Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Interrupt Example*.

```

#include "nios2_ctrl_reg_macros.h"
#include "key_codes.h"           // defines values for KEY1, KEY2

/* key_pressed and pattern are written by interrupt service routines; we have to declare
 * these as volatile to avoid the compiler caching their values in registers */
volatile int key_pressed = KEY2;    // shows which key was last pressed
volatile int pattern = 0x0000000F;  // pattern for HEX displays
/*****

 * This program demonstrates use of interrupts in the DE1-SoC Computer. It first starts the
 * interval timer with 33 msec timeouts, and then enables interrupts from the interval timer
 * and pushbutton KEYs
 *
 * The interrupt service routine for the interval timer displays a pattern on the HEX displays, and
 * shifts this pattern either left or right. The shifting direction is set in the pushbutton
 * interrupt service routine, as follows:
 *   KEY[1]: shifts the displayed pattern to the right
 *   KEY[2]: shifts the displayed pattern to the left
 *   KEY[3]: changes the pattern using the settings on the SW switches
 *****/

int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that IO load and store instructions
     * will be used to access these pointer locations instead of regular memory loads and stores) */
    volatile int * interval_timer_ptr = (int *) 0x10002000;  // interval timer base address
    volatile int * KEY_ptr = (int *) 0x10000050;             // pushbutton KEY address

    /* set the interval timer period for scrolling the HEX displays */
    int counter = 0x190000;          // 1/(50 MHz) × (0x190000) = 33 msec
    *(interval_timer_ptr + 0x2) = (counter & 0xFFFF);
    *(interval_timer_ptr + 0x3) = (counter >> 16) & 0xFFFF;

    /* start interval timer, enable its interrupts */
    *(interval_timer_ptr + 1) = 0x7;    // STOP = 0, START = 1, CONT = 1, ITO = 1

    *(KEY_ptr + 2) = 0xE;               /* write to the pushbutton interrupt mask register, and
     * set 3 mask bits to 1 (bit 0 is Nios II reset) */

    NIOS2_WRITE_IENABLE( 0x3 );        /* set interrupt mask bits for levels 0 (interval timer)
     * and level 1 (pushbuttons) */
    NIOS2_WRITE_STATUS( 1 );           // enable Nios II interrupts

    while(1);                          // main program simply idles
}

```

Figure 20. An example of C code that uses interrupts.

```

#ifndef __NIOS2_CTRL_REG_MACROS__
#define __NIOS2_CTRL_REG_MACROS__

/*****
/* Macros for accessing the control registers.
*****/

#define NIOS2_READ_STATUS(dest) \
    do { dest = __builtin_rdctl(0); } while (0)

#define NIOS2_WRITE_STATUS(src) \
    do { __builtin_wrctl(0, src); } while (0)

#define NIOS2_READ_ESTATUS(dest) \
    do { dest = __builtin_rdctl(1); } while (0)

#define NIOS2_READ_BSTATUS(dest) \
    do { dest = __builtin_rdctl(2); } while (0)

#define NIOS2_READ_IENABLE(dest) \
    do { dest = __builtin_rdctl(3); } while (0)

#define NIOS2_WRITE_IENABLE(src) \
    do { __builtin_wrctl(3, src); } while (0)

#define NIOS2_READ_IPENDING(dest) \
    do { dest = __builtin_rdctl(4); } while (0)

#define NIOS2_READ_CPUID(dest) \
    do { dest = __builtin_rdctl(5); } while (0)

#endif

```

Figure 21. Macros for accessing Nios II status and control registers.


```

#include "nios2_ctrl_reg_macros.h"

/* function prototypes */
void main(void);
void interrupt_handler(void);
void interval_timer_isr(void);
void pushbutton_ISR(void);

/* global variables */
extern int key_pressed;

/* The assembly language code below handles Nios II reset processing */
void the_reset(void) __attribute__((section(".reset")));
void the_reset(void)
/*****
 * Reset code; by using the section attribute with the name ".reset" we allow the linker program
 * to locate this code at the proper reset vector address. This code just calls the main program
 *****/
{
    asm (".set    noat");           // magic, for the C compiler
    asm (".set    nobreak");       // magic, for the C compiler
    asm ("movia  r2, main");       // call the C language main program
    asm ("jmp    r2");
}
/* The assembly language code below handles Nios II exception processing. This code should not be
 * modified; instead, the C language code in the function interrupt_handler() can be modified as
 * needed for a given application. */
void the_exception(void) __attribute__((section(".exceptions")));
void the_exception(void)
/*****
 * Exceptions code; by giving the code a section attribute with the name ".exceptions" we allow
 * the linker to locate this code at the proper exceptions vector address. This code calls the
 * interrupt handler and later returns from the exception.
 *****/
{
    asm (".set    noat");           // magic, for the C compiler
    asm (".set    nobreak");       // magic, for the C compiler
    asm ("subi   sp, sp, 128");
    asm ("stw    et, 96(sp)");
    asm ("rdctl  et, ct14");
    asm ("beq    et, r0, SKIP_EA_DEC"); // interrupt is not external
    asm ("subi   ea, ea, 4");       /* must decrement ea by one instruction for external
 * interrupts, so that the instruction will be run */
}

```

Figure 22. Reset and exception handler C code (Part a).

```

asm ( "SKIP_EA_DEC:" );
asm ( "stw  r1, 4(sp)" );           // save all registers
asm ( "stw  r2, 8(sp)" );
asm ( "stw  r3, 12(sp)" );
asm ( "stw  r4, 16(sp)" );
asm ( "stw  r5, 20(sp)" );
asm ( "stw  r6, 24(sp)" );
asm ( "stw  r7, 28(sp)" );
asm ( "stw  r8, 32(sp)" );
asm ( "stw  r9, 36(sp)" );
asm ( "stw  r10, 40(sp)" );
asm ( "stw  r11, 44(sp)" );
asm ( "stw  r12, 48(sp)" );
asm ( "stw  r13, 52(sp)" );
asm ( "stw  r14, 56(sp)" );
asm ( "stw  r15, 60(sp)" );
asm ( "stw  r16, 64(sp)" );
asm ( "stw  r17, 68(sp)" );
asm ( "stw  r18, 72(sp)" );
asm ( "stw  r19, 76(sp)" );
asm ( "stw  r20, 80(sp)" );
asm ( "stw  r21, 84(sp)" );
asm ( "stw  r22, 88(sp)" );
asm ( "stw  r23, 92(sp)" );
asm ( "stw  r25, 100(sp)" );        // r25 = bt (skip r24 = et, because it was saved above)
asm ( "stw  r26, 104(sp)" );      // r26 = gp
// skip r27 because it is sp, and there is no point in saving this
asm ( "stw  r28, 112(sp)" );      // r28 = fp
asm ( "stw  r29, 116(sp)" );      // r29 = ea
asm ( "stw  r30, 120(sp)" );      // r30 = ba
asm ( "stw  r31, 124(sp)" );      // r31 = ra
asm ( "addi fp, sp, 128" );

asm ( "call  interrupt_handler" ); // call the C language interrupt handler

asm ( "ldw  r1, 4(sp)" );         // restore all registers
asm ( "ldw  r2, 8(sp)" );
asm ( "ldw  r3, 12(sp)" );
asm ( "ldw  r4, 16(sp)" );
asm ( "ldw  r5, 20(sp)" );
asm ( "ldw  r6, 24(sp)" );
asm ( "ldw  r7, 28(sp)" );

```

Figure 22. Reset and exception handler C language code (Part b).

```

asm ( "ldw  r8, 32(sp)" );
asm ( "ldw  r9, 36(sp)" );
asm ( "ldw  r10, 40(sp)" );
asm ( "ldw  r11, 44(sp)" );
asm ( "ldw  r12, 48(sp)" );
asm ( "ldw  r13, 52(sp)" );
asm ( "ldw  r14, 56(sp)" );
asm ( "ldw  r15, 60(sp)" );
asm ( "ldw  r16, 64(sp)" );
asm ( "ldw  r17, 68(sp)" );
asm ( "ldw  r18, 72(sp)" );
asm ( "ldw  r19, 76(sp)" );
asm ( "ldw  r20, 80(sp)" );
asm ( "ldw  r21, 84(sp)" );
asm ( "ldw  r22, 88(sp)" );
asm ( "ldw  r23, 92(sp)" );
asm ( "ldw  r24, 96(sp)" );
asm ( "ldw  r25, 100(sp)" );           // r25 = bt
asm ( "ldw  r26, 104(sp)" );         // r26 = gp
// skip r27 because it is sp, and we did not save this on the stack
asm ( "ldw  r28, 112(sp)" );         // r28 = fp
asm ( "ldw  r29, 116(sp)" );         // r29 = ea
asm ( "ldw  r30, 120(sp)" );         // r30 = ba
asm ( "ldw  r31, 124(sp)" );         // r31 = ra

asm ( "addi sp, sp, 128" );
asm ( "eret" );
}

/*****
* Interrupt Service Routine: Determines the interrupt source and calls the appropriate subroutine
*****/
void interrupt_handler(void)
{
    int ipending;
    NIOS2_READ_IPENDING(ipending);
    if ( ipending & 0x1 )                // interval timer is interrupt level 0
        interval_timer_isr( );
    if ( ipending & 0x2 )                // pushbuttons are interrupt level 1
        pushbutton_ISR( );
    // else, ignore the interrupt
    return;
}

```

Figure 22. Reset and exception handler C code (Part c).

```

#include "key_codes.h"                // defines values for KEY1, KEY2

extern volatile int key_pressed;
extern volatile int pattern;
/*****
 * Interval timer interrupt service routine
 *
 * Shifts a pattern being displayed on the HEX displays. The shift direction is determined
 * by the external variable key_pressed.
 *
 *****/
void interval_timer_isr()
{
    volatile int * interval_timer_ptr = (int *) 0x10002000;
    volatile int * HEX3_HEX0_ptr = (int *) 0x10000020;    // HEX3_HEX0 address

    *(interval_timer_ptr) = 0;                // clear the interrupt

    *(HEX3_HEX0_ptr) = pattern;              // display pattern on HEX3 ... HEX0

    /* rotate the pattern shown on the HEX displays */
    if (key_pressed == KEY2)                // for KEY2 rotate left
        if (pattern & 0x80000000)
            pattern = (pattern << 1) | 1;
        else
            pattern = pattern << 1;
    else if (key_pressed == KEY1)          // for KEY1 rotate right
        if (pattern & 0x00000001)
            pattern = (pattern >> 1) | 0x80000000;
        else
            pattern = (pattern >> 1) & 0x7FFFFFFF;

    return;
}

```

Figure 23. Interrupt service routine for the interval timer.

```

#include "key_codes.h"           // defines values for KEY1, KEY2

extern volatile int key_pressed;
extern volatile int pattern;

/*****
 * Pushbutton - Interrupt Service Routine
 *
 * This routine checks which KEY has been pressed. If it is KEY1 or KEY2, it writes this value
 * to the global variable key_pressed. If it is KEY3 then it loads the SW switch values and
 * stores in the variable pattern
 *****/
void pushbutton_ISR( void )
{
    volatile int * KEY_ptr = (int *) 0x10000050;
    volatile int * slider_switch_ptr = (int *) 0x10000040;
    int press;

    press = *(KEY_ptr + 3);           // read the pushbutton interrupt register
    *(KEY_ptr + 3) = 0;              // clear the interrupt

    if (press & 0x2)                  // KEY1
        key_pressed = KEY1;
    else if (press & 0x4)              // KEY2
        key_pressed = KEY2;
    else                               // press & 0x8, which is KEY3
        pattern = *(slider_switch_ptr); // read the SW slider switch values; store in pattern

    return;
}

```

Figure 24. Interrupt service routine for the pushbutton keys.

4 Media Components

This section describes the audio in/out port, video-out port, video-in port, PS/2 port, and IrDA port.

4.1 Audio In/Out Port

The DE1-SoC Computer includes an audio port that is connected to the audio CODEC (COder/DECOder) chip on the DE1-SoC board. The default setting for the sample rate provided by the audio CODEC is 48K samples/sec. The audio port provides audio-input capability via the microphone jack on the DE1-SoC board, as well as audio output functionality via the line-out jack. The audio port includes four FIFOs that are used to hold incoming and outgoing data. Incoming data is stored in the left- and right-channel *Read* FIFOs, and outgoing data is held in the left- and right-channel *Write* FIFOs. All FIFOs have a maximum depth of 128 32-bit words.

The audio port's programming interface consists of four 32-bit registers, as illustrated in Figure 25. The *Control* register, which has the address 0xFF203040, is readable to provide status information and writable to make control settings. Bit *RE* of this register provides an interrupt enable capability for incoming data. Setting this bit to 1 allows the audio core to generate a Nios II interrupt when either of the *Read* FIFOs are filled 75% or more. The bit *RI* will then be set to 1 to indicate that the interrupt is pending. The interrupt can be cleared by removing data from the *Read* FIFOs until both are less than 75% full. Bit *WE* gives an interrupt enable capability for outgoing data. Setting this bit to 1 allows the audio core to generate an interrupt when either of the *Write* FIFOs are less than 25% full. The bit *WI* will be set to 1 to indicate that the interrupt is pending, and it can be cleared by filling the *Write* FIFOs until both are more than 25% full. The bits *CR* and *CW* in Figure 25 can be set to 1 to clear the *Read* and *Write* FIFOs, respectively. The clear function remains active until the corresponding bit is set back to 0.

Address	31	...	24	23	...	16	15	...	10	9	8	7	...	3	2	1	0	
0xFF203040	Unused										WI	RI		CW	CR	WE	RE	Control
0xFF203044	WSLC		WSRC		RALC				RARC				Fifospace					
0xFF203048	Left data																Leftdata	
0xFF20303C	Right data																Rightdata	

Figure 25. Audio port registers.

The read-only *Fifospace* register in Figure 25 contains four 8-bit fields. The fields *RARC* and *RALC* give the number of words currently stored in the right and left audio-input FIFOs, respectively. The fields *WSRC* and *WSLC* give the number of words currently available (that is, *unused*) for storing data in the right and left audio-out FIFOs. When all FIFOs in the audio port are cleared, the values provided in the *Fifospace* register are $RARC = RALC = 0$ and $WSRC = WSLC = 128$.

The *Leftdata* and *Rightdata* registers are readable for audio in, and writable for audio out. When data is read from these registers, it is provided from the head of the *Read* FIFOs, and when data is written into these registers it is loaded into the *Write* FIFOs.

A fragment of C code that uses the audio port is shown in Figure 26. The code checks to see when the depth of either the left or right *Read* FIFO has exceeded 75% full, and then moves the data from these FIFOs into a memory buffer. This code is part of a larger program that is distributed as part of the Altera Monitor Program. The source code can be found under the heading *sample programs*, and is identified by the name *Media*.

```

volatile int * audio_ptr = (int *) 0xFF203040;           // audio port address
int fifospace, int buffer_index = 0;
int left_buffer[BUF_SIZE];
int right_buffer[BUF_SIZE];
...
fifospace = *(audio_ptr + 1);                           // read the audio port fifospace register
if ( (fifospace & 0x000000FF) > 96)                    // check RARC, for > 75% full
{
    /* store data until the audio-in FIFO is empty or the memory buffer is full */
    while ( (fifospace & 0x000000FF) && (buffer_index < BUF_SIZE) )
    {
        left_buffer[buffer_index] = *(audio_ptr + 2);   //Leftdata
        right_buffer[buffer_index] = *(audio_ptr + 3);  //Rightdata
        ++buffer_index;
        fifospace = *(audio_ptr + 1);                   // read the audio port fifospace register
    }
}
...

```

Figure 26. An example of code that uses the audio port.

4.2 Video-out Port

The DE1-SoC Computer includes a video-out port with a VGA controller that can be connected to a standard VGA monitor. The VGA controller supports a screen resolution of 640×480 . The image that is displayed by the VGA controller is derived from two sources: a *pixel* buffer, and a *character* buffer.

4.2.1 Pixel Buffer

The pixel buffer for the video-out port reads stored pixel values from a memory buffer for display by the VGA controller. As illustrated in Figure 27, the memory buffer provides an image resolution of 320×240 pixels, with the coordinate 0,0 being at the top-left corner of the image. Since the VGA controller supports the screen resolution of 640×480 , each of the pixel values in the pixel buffer is replicated in both the *x* and *y* dimensions when it is being displayed on the VGA screen.

Figure 28a shows that each pixel value is represented as a 16-bit halfword, with five bits for the blue and red components, and six bits for green. As depicted in part *b* of Figure 28, pixels are addressed in the memory buffer by using the combination of a *base* address and an *x,y* offset. In the DE1-SoC Computer the pixel buffer uses the base address $(08000000)_{16}$, which corresponds to the starting address of the SRAM chip on the DE1-SoC board. Using this scheme, the pixel at location 0,0 has the address $(08000000)_{16}$, the pixel 1,0 has the address *base* + (00000000)

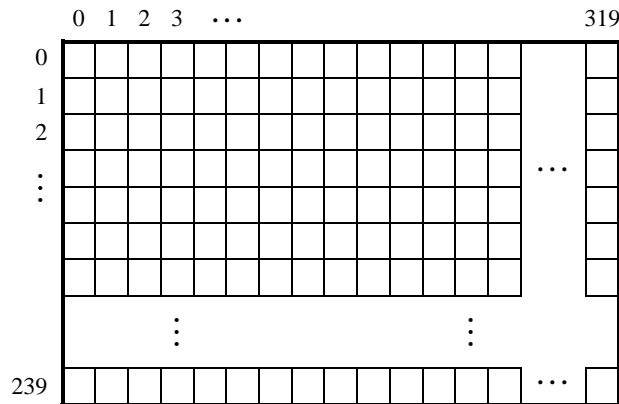
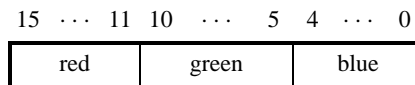


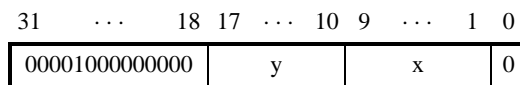
Figure 27. Pixel buffer coordinates.

$00000001\ 0)_2 = (08000002)_{16}$, the pixel 0,1 has the address $base + (00000001\ 00000000\ 0)_2 = (08000400)_{16}$, and the pixel at location 319,239 has the address $base + (11101111\ 10011111\ 1\ 0)_2 = (0803BE7E)_{16}$.

The pixel buffer includes a programming interface in the form of a set of registers. These registers allow the base address of the memory buffer used by the pixel buffer to be changed under software control, as well as providing status information. A detailed description of this programming interface is available in the online documentation for the Video-out port, which is available from Altera’s University Program web site.



(a) Pixel values



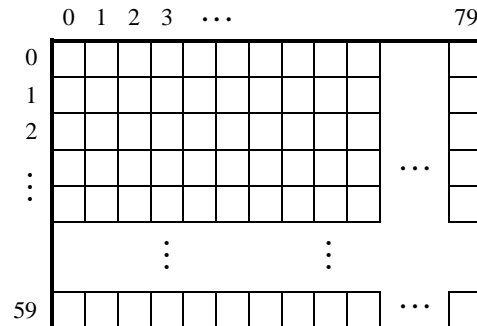
(b) Pixel buffer addresses

Figure 28. Pixel values and addresses.

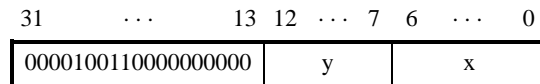
4.2.2 Character Buffer

The character buffer for the video-out port is stored in on-chip memory in the FPGA on the DE1-SoC board. As illustrated in Figure 29a, the buffer provides a resolution of 80×60 characters, where each character occupies an 8×8 block of pixels on the VGA screen. Characters are stored in each of the locations shown in Figure 29a using their ASCII codes; when these character codes are displayed on the VGA monitor, the character buffer automatically generates the corresponding pattern of pixels for each character using a built-in font. Part b of Figure 29

shows that characters are addressed in the memory by using the combination of a *base* address, which has the value $(09000000)_{16}$, and an x,y offset. Using this scheme, the character at location 0,0 has the address $(09000000)_{16}$, the character 1,0 has the address $base + (000000\ 0000001)_2 = (09000001)_{16}$, the character 0,1 has the address $base + (000001\ 0000000)_2 = (09000080)_{16}$, and the character at location 79,59 has the address $base + (111011\ 1001111)_2 = (09001DCF)_{16}$.



(a) Character buffer coordinates



(b) Character buffer addresses

Figure 29. Character buffer coordinates and addresses.

4.2.3 Using the Video-out Port with C code

A fragment of C code that uses the pixel and character buffers is shown in Figure 30. The first **while** loop in the figure draws a rectangle in the pixel buffer using the color *pixel_color*. The rectangle is drawn using the coordinates x_1, y_1 and x_2, y_2 . The second **while** loop in the figure writes a null-terminated character string pointed to by the variable *text_ptr* into the character buffer at the coordinates x, y . The code in Figure 30 is included in the sample program called *Media* that is distributed with the Altera Monitor Program.

4.3 Audio/Video Configuration Module

The audio/video configuration module controls settings that affect the operation of both the audio port and the video-out port. The audio/video configuration module automatically configures and initializes both of these ports whenever the DE1-SoC Computer is reset. For typical use of the DE1-SoC Computer it is not necessary to modify any of these default settings. In the case that changes to these settings are needed, the reader should refer to the audio/video configuration module's online documentation, which is available from Altera's University Program web site.

```

volatile short * pixel_buffer = (short *) 0x08000000;    // Pixel buffer
volatile char * character_buffer = (char *) 0x09000000; // Character buffer
int x1, int y1, int x2, int y2, short pixel_color;
int offset, row, col;
int x, int y, char * text_ptr;
...
/* Draw a box; assume that the coordinates are valid */
for (row = y1; row <= y2; row++)
    for (col = x1; col <= x2; ++col)
    {
        offset = (row << 9) + col;
        *(pixel_buffer + offset) = pixel_color;           // compute halfword address, set pixel
    }

/* Display a text string; assume that it fits on one line */
offset = (y << 7) + x;
while ( *(text_ptr) )
{
    *(character_buffer + offset) = *(text_ptr);           // write to the character buffer
    ++text_ptr;
    ++offset;
}

```

Figure 30. An example of code that uses the video-out port.

4.4 PS/2 Port

The DE1-SoC Computer includes two PS/2 ports that can be connected to a standard PS/2 keyboard or mouse. The port includes a 256-byte FIFO that stores data received from a PS/2 device. The programming interface for the PS/2 port consists of two registers, as illustrated in Figure 31. The *PS2_Data* register is both readable and writable. When bit 15, *RVALID*, is 1, reading from this register provides the data at the head of the FIFO in the *Data* field, and the number of entries in the FIFO (including this read) in the *RAVAIL* field. When *RVALID* is 1, reading from the *PS2_Data* register decrements this field by 1. Writing to the *PS2_Data* register can be used to send a command in the *Data* field to the PS/2 device.

The *PS2_Control* register can be used to enable interrupts from the PS/2 port by setting the *RE* field to the value 1. When this field is set, then the PS/2 port generates an interrupt when *RAVAIL* > 0. While the interrupt is pending the field *RI* will be set to 1, and it can be cleared by emptying the PS/2 port FIFO. The *CE* field in the *PS2_Control* register is used to indicate that an error occurred when sending a command to a PS/2 device.

A fragment of C code that uses the PS/2 port is given in Figure 32. This code reads the content of the *Data* register, and saves data when it is available. If the code is used continually in a loop, then it stores the last three bytes of data received from the PS/2 port in the variables *byte₁*, *byte₂*, and *byte₃*. This code is included as part of a larger sample program called *Media* that is distributed with the Altera Monitor Program.

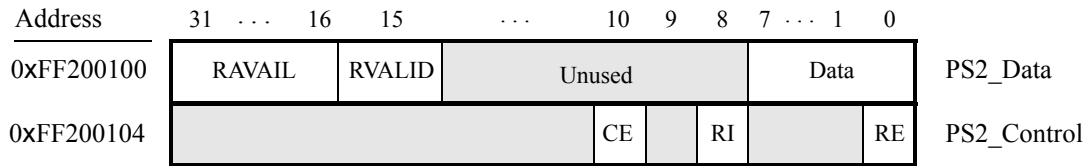


Figure 31. PS/2 port registers.

```

volatile int * PS2_ptr = (int *) 0xFF200100;           // PS/2 port address
int PS2_data, RVALID;
char byte1 = 0, byte2 = 0, byte3 = 0;
...
PS2_data = *(PS2_ptr);                               // read the Data register in the PS/2 port
RVALID = PS2_data & 0x8000;                          // extract the RVALID field
if (RVALID)
{
    /* save the last three bytes of data */
    byte1 = byte2;
    byte2 = byte3;
    byte3 = PS2_data & 0xFF;
}
...

```

Figure 32. An example of code that uses the PS/2 port.

4.4.1 PS/2 Port Dual

The DE1-SoC Computer includes a second PS/2 port that allows both a keyboard and mouse to be used at the same time. To use the dual port a Y-splitter cable must be used and the keyboard and mouse must be connected to the PS/2 connector on the DE1-SoC board through this cable. The PS/2 port dual has the same registers as the PS/2 port shown in Figure 32, except that the base address of its *PS2_Data* register is 0xFF200108 and the base address of its *PS2_Control* register is 0xFF20010C.

4.5 IrDA Infrared Serial Port

The IrDA port in the DE1-SoC Computer implements a UART that is connected to the infrared transmit/receive device on the DE1-SoC board. This UART is configured for 8-bit data, one stop bit, and no parity, and operates at a baud rate of 115,200. The serial port's programming interface consists of two 32-bit registers, as illustrated in Figure 33. The register at address 0xFF201020 is referred to as the *Data* register, and the register at address 0xFF201024 is called the *Control* register.

When character data is received from the IrDA chip it is stored in a 256-character FIFO in the UART. As illustrated in Figure 33, the number of characters *RAVAIL* currently stored in this FIFO is provided in bits 23–16 of the *Data* register. If the receive FIFO overflows, then additional data is lost. When the data that is present in the receive FIFO is available for reading, then the value of bit 15, *RVALID*, will be 1. Reading the character at the head of the FIFO,

Address	31	...	24	23	...	16	15	14	...	10	9	8	7	...	1	0		
0xFF201020	Unused			RAVAIL			RVALID			Unused			PE		DATA		Data register	
0xFF201024	Unused			WSPACE			Unused			WI		RI		WE		RE		Control register

Figure 33. IrDA serial port UART registers.

which is provided in bits 7–0, decrements the value of *RAVAIL* by one and returns this decremented value as part of the read operation. If no data is available to be read from the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7–0 is undefined.

The UART also includes a 256-character FIFO that stores data waiting to be sent to the IrDA device. Character data is loaded into this register by performing a write to bits 7–0 of the *Data* register. Writing into this register has no effect on received data. The amount of space *WSPACE* currently available in the transmit FIFO is provided in bits 23–16 of the *Control* register, as indicated in Figure 33. If the transmit FIFO is full, then any additional characters written to the *Data* register will be lost.

The *RE* and *WE* bits in the *Control* register are used to enable A9 processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when *RAVAIL* for the receive FIFO, or *WSPACE* for the transmit FIFO, exceeds 31. Pending interrupts are indicated in the *Control* register's *RI* and *WI* bits, and can be cleared by writing or reading data to/from the UART.

4.6 Floating-point Hardware

The Nios II processor in the DE1-SoC Computer includes hardware support for floating-point addition, subtraction, multiplication, and division. To use this support in a C program, variables must be declared with the type *float*. A simple example of such code is given in Figure 34. When this code is compiled, it is necessary to pass the special argument `-mcustom-fpu-cfg=60-2` to the C compiler, to instruct it to use the floating-point hardware support.

5 Modifying the DE1-SoC Computer

It is possible to modify the DE1-SoC Computer by using Altera's Quartus II software and Qsys tool. Tutorials that introduce this software are provided in the University Program section of Altera's web site. To modify the system it is first necessary to make an editable copy of the DE1-SoC Computer. The files for this system are installed as part of the Monitor Program installation. Locate these files, copy them to a working directory, and then use the Quartus II and Qsys software to make any desired changes.

```

/*****
 * This program demonstrates use of floating-point numbers in the DE1-SOC Computer
 *
 * It performs the following:
 *   1. reads two FP numbers from the Terminal window
 *   2. performs +, -, *, and / on the numbers, then prints results on Terminal window
 *****/
int main(void)
{
    float x, y, add, sub, mult, div;

    while(1)
    {
        printf ("Enter FP values X Y:\n");
        scanf ("%f", &x);
        printf ("%f ", x); // echo the typed data to the Terminal window
        scanf ("%f", &y);
        printf ("%f\n", y); // echo the typed data to the Terminal window
        add = x + y;
        sub = x - y;
        mult = x * y;
        div = x / y;
        printf ("X + Y = %f\n", add);
        printf ("X - Y = %f\n", sub);
        printf ("X * Y = %f\n", mult);
        printf ("X / Y = %f\n", div);
    }
}

```

Figure 34. An example of code that uses floating-point variables.

6 Making the System the Default Configuration

The DE1-SoC Computer can be loaded into the nonvolatile FPGA configuration memory on the DE1-SoC board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE1-SoC board in this manner can be found in the tutorial *Introduction to the Quartus II Software*, which is available from Altera's University Program.

7 Memory Layout

Table 2 summarizes the memory map used in the DE1-SoC Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x03FFFFFF	SDRAM
0x08000000	0x08003FFF	FPGA On-chip Memory
0x09000000	0x09001FFF	FPGA On-chip Memory Character Buffer
0xFF200000	0xFF20000F	Red LEDs
0xFF200020	0xFF20002F	7-segment HEX3–HEX0 Displays
0xFF200030	0xFF20003F	7-segment HEX5–HEX4 Displays
0xFF200040	0xFF20004F	Slider Switches
0xFF200050	0xFF20005F	Pushbutton KEYs
0xFF200060	0xFF20006F	JP1 Expansion
0xFF200070	0xFF20007F	JP2 Expansion
0xFF200100	0xFF200107	PS/2
0xFF200108	0xFF20010F	PS/2 Dual
0xFF201000	0xFF201007	JTAG UART
0xFF201020	0xFF201027	Infrared (IrDA)
0xFF202000	0xFF20201F	Interval Timer
0xFF203000	0xFF20301F	Audio/video Configuration
0xFF203020	0xFF20302F	Pixel Buffer Control
0xFF203030	0xFF203037	Character Buffer Control
0xFF203040	0xFF20304F	Audio

Table 2. Memory layout used in the DE1-SoC Computer.

8 Altera Monitor Program Integration

As we mentioned earlier, the DE1-SoC Computer system, and the sample programs described in this document, are made available as part of the Altera Monitor Program. Figures 35 to 38 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 35, the user specifies a file system folder where the project will be stored, gives the project a name, and specifies the type of processor that is being used. Pressing Next opens the window in Figure 36. Here, the user can select the DE1-SoC Computer as a pre-designed system. The Monitor Program then fills in the relevant information in the *System details* box, which includes the files called *Computer_System.sopcinfo* and *DE1_SoC_Computer.sof*. The first of these files specifies to the Monitor Program information about the components that are available in the DE1-SoC Computer, such as the type of processor and memory components, and the address map. The second file is an FPGA programming bitstream for the DE1-SoC Computer, which can be downloaded by the Monitor Program into the DE1-SoC board.

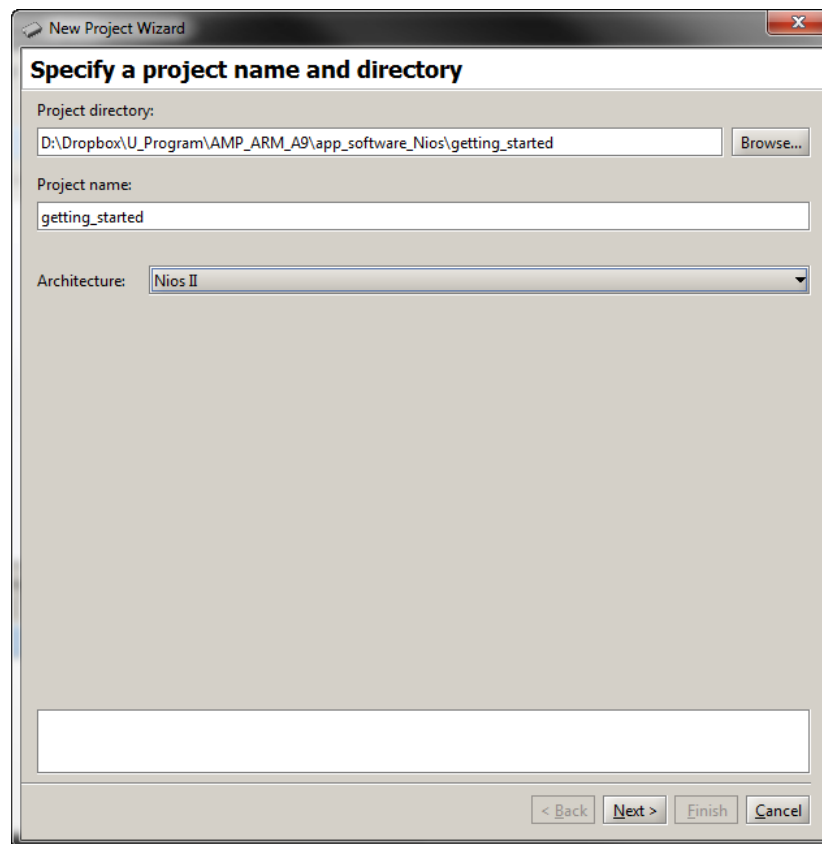


Figure 35. Specifying the project folder and project name.

Pressing **Next** again opens the window in Figure 37. Here the user selects the type of program that will be used, such as Assembly language, or C. Then, the check box shown in the figure can be used to display the list of sample programs for the DE1-SoC Computer that are described in this document. When a sample program is selected in this list, its source files, and other settings, can be copied into the project folder in subsequent screens of the Monitor Program.

Figure 38 gives the final screen that is used to create a new project in the Monitor Program. This screen shows the default addresses of compiler and linker sections that will be used for the assembly language or C program associated with the Monitor Program project. In the figure, the drop-down next to *Linker Section Presets* has been set to *Interrupts*. With this setting the Monitor Program uses a compiler and linker section for the A9 processor exception table, called *.vectors*, and another section for the main program, called *.text*. It also shows the initial value used to set the main stack pointer for C programs, which is the starting address of the *.stack* section.

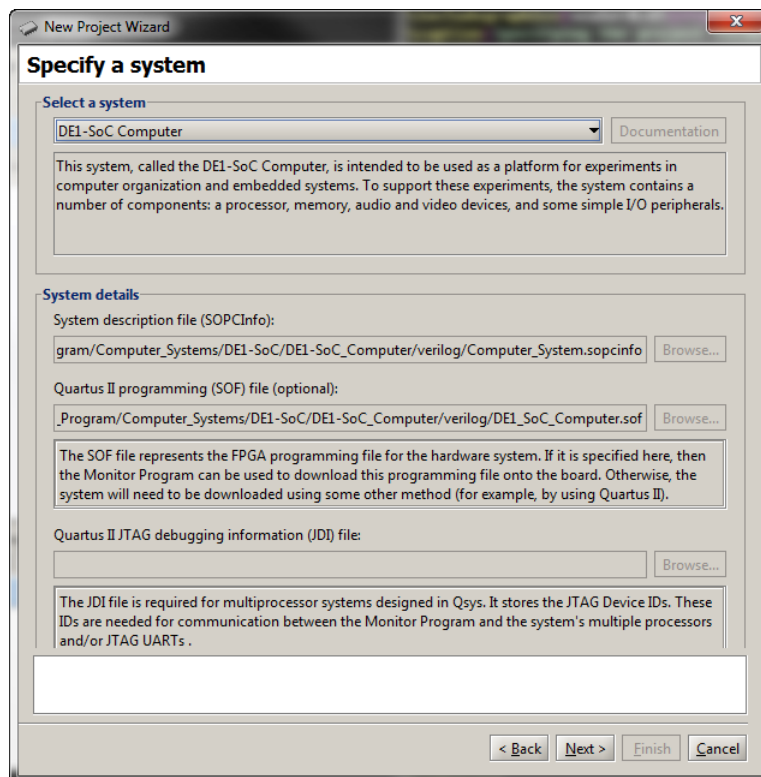


Figure 36. Specifying the DE1-SoC Computer system.

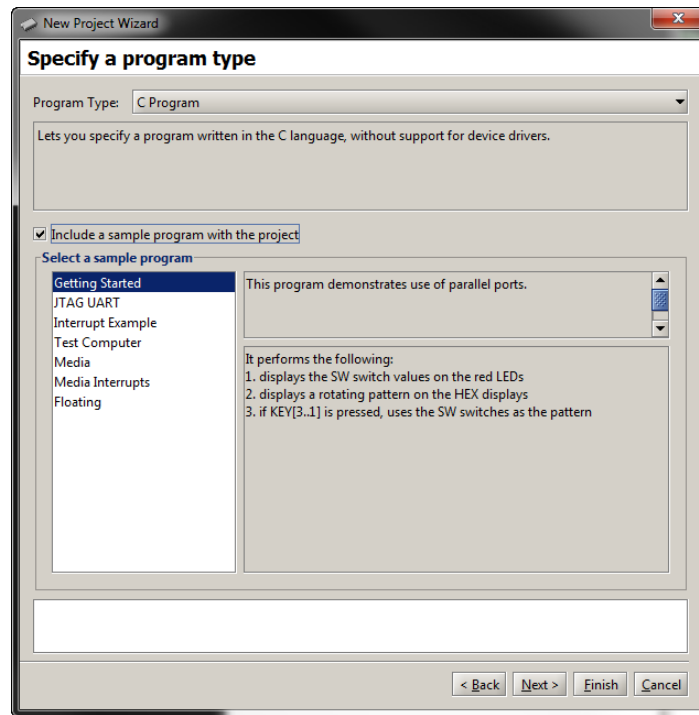


Figure 37. Selecting sample programs.

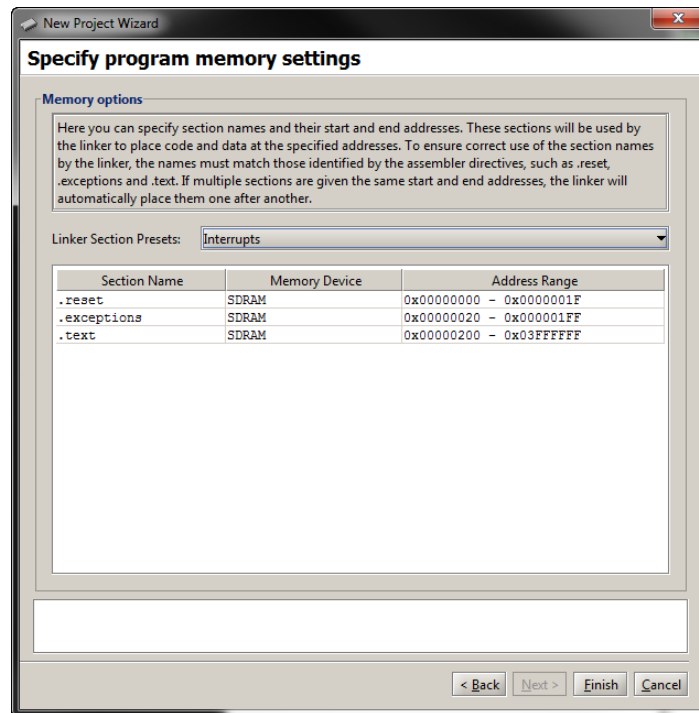


Figure 38. Setting offsets for .text and .data.