DE1-SoC Computer System with ARM Cortex-A9

For Quartus Prime 16.0

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 the *Hard Processor System* (HPS) and FPGA inside the Cyclone® V SoC chip. The HPS comprises an ARM Cortex A9 dual-core processor, a DDR3 memory port, and a set of peripheral devices. The FPGA implements two Altera Nios II processors, 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.

2.1 Hard Processor System

The hard processor system (HPS), as shown in Figure 1, includes an ARM Cortex A9 dual-core processor. The A9 dual-core processor features two 32-bit CPUs and associated subsystems that are implemented as hardware circuits in the Altera Cyclone V SoC chip. An overview of the ARM A9 processor can be found in the document *Introduction to the ARM Processor*, which is provided in Altera's University Program web site. 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 this document. A summary of the address map can be found in Section 7.

A good way to begin working with the DE1-SoC Computer and the ARM A9 processor is to make use of a utility called the *Altera Monitor Program*. It provides an easy way to assemble/compile ARM A9 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 the ARM A9, list (and 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 pre-designed 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. Section 8 describes how the DE1-SoC Computer is integrated with the Monitor Program. An overview of the Monitor Program is available in the document *Altera Monitor Program Tutorial*, which is provided in the University Program web site.

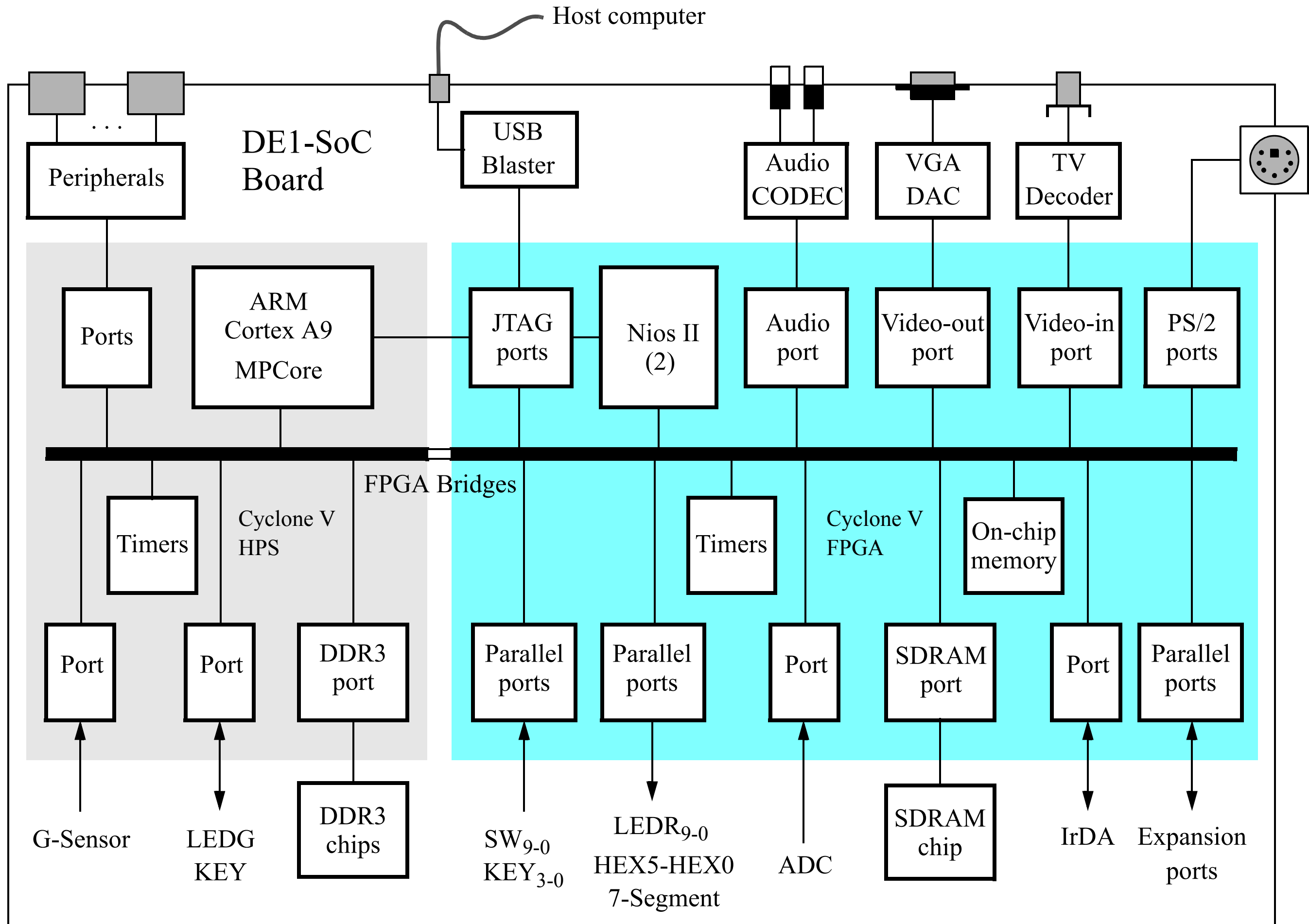


Figure 1. Block diagram of the DE1-SoC Computer.

2.2 Memory

The HPS includes a memory port that connects the ARM MPCORE to a 1 GB DDR3 memory. This memory is normally used as the storage location of programs and data used by the ARM processors. The memory is organized as 256M x 32-bits, and is accessible using word accesses (32 bits), halfwords, and bytes. The DDR3 memory is mapped to the address space 0x00000000 to 0x3FFFFFFF. There is also a 64 KB on-chip memory available inside each ARM A9 processor. This small memory is organized as 16K x 32-bits, and is mapped to the address space FFFF0000 to FFFFFFFF.

2.3 Pushbutton KEY and LED Port

The HPS includes a general purpose I/O port, called *GPIO1*, that is accessible by the ARM A9 processor. As illustrated in Figure 2, this parallel port is assigned the *Base* address 0xFF709000, and includes several 32-bit registers. These registers can be read or written usign word accesses. Only two bit locations in GPIO1 are used for

the DE1-SoC computer. Bit 24 of the data register (DR) is connected to a green light, LEDG, and bit 25 is connected to a pushbutton switch, KEY. To use these devices, the *data direction register* (DDR) shown in the figure has to be configured such that bit 24 is an output and bit 25 is an input. Writing a 1 into a corresponding bit position in the DDR sets this bit as an output, while writing a 0 sets the bit as an input. After the direction bits have been set, the green light LEDG can be turned on/off by writing to bit 24 in the data register. Similarly, the value of the pushbutton switch KEY can be obtained by reading the data register and checking the value of bit 25. An example program for the ARM A9 processor that uses GPIO1 is given in Section 2.4.

As indicated in Figure 2, the GPIO1 port includes several other registers in addition to the DR and DDR registers. These other registers are mostly used for setting characteristics of input pins, which affects only the KEY input in our system. Detailed information about these registers can be found in the *Altera Cyclone V Hard Processor System* documentation, which is available on Altera's website.

Address	31 ···	25	24	23 ··· 0	
0xFF709000	Unused			Unused	Data register
0xFF709004					Data direction register
0xFF709030					Interrupt enable register
	... not shown				
0xFF709060					Level sync register

Figure 2. Parallel port GPIO1.

2.4 Timer Modules

The HPS includes several hardware timer modules that can be used to keep track of time intervals. The ARM A9 MPCore includes one *private* timer module for each A9 core, and the HPS provides four other timer modules that can be used by either A9 core. The timers are described in more detail below.

2.4.1 ARM A9 MPCore Timers

Figure 3 shows the registers in the programmer's interface for each A9 core private timer. These registers have the base address 0xFFFEC600, as shown in the figure, and can be read or written using word accesses. To use the timer, it is necessary to first write an initial count value into the *Load* register. The timer can then be started by setting the enable bit *E* in the *Control* register to 1, and it can be stopped by setting *E* back to 0. Once enabled the timer decrements its count value until reaching 0. When it reaches 0, the timer sets the *F* bit in the *Interrupt status* register. The *F* bit can be checked by software using polled-I/O to determine when the timer period has expired. The *F* bit can be reset to 0 by writing a 1 into it. Also, if bit *I* in the *Control* register is set to 1, then a processor interrupt can be generated when the timer reaches 0. Using interrupts with the timer is discussed in Section 3.

When it reaches 0, the timer will stop if the auto bit (*A*) in the control register is set to 0. But if bit *A* is set to 1, then the timer will automatically reload the value in the *Load* register and continue decrementing. The current value of

the timer is available to software in the *Counter* register shown in Figure 3. The timer uses a clock frequency of 200 MHz. The *Prescaler* field in the *Control* register can be used to slow down the counting rate, as follows. The timer decrements each *Prescaler* +1 clock cycle. Therefore, if *Prescaler* = 0, then the timer decrements every clock cycle, if *Prescaler* = 1, the timer decrements every second clock cycle, and so on.

Address	31 ... 16	15 ... 8	7 3	2	1	0	Register name
0xFFFEC600	Load value						Load
0xFFFEC604	Current value						Counter
0xFFFEC608	Unused	Prescaler	Unused	I	A	E	Control
0xFFFEC60C	Unused					F	Interrupt status

Figure 3. ARM A9 private timer port.

2.4.2 HPS Timers

Figure 4 shows the registers in the programmer's interface for one of the HPS timers. These registers have the base address 0xFFC08000, as shown in the figure, and can be read or written using word accesses. To configure the timer, it is necessary to ensure that it is stopped by setting the enable bit *E* in the *Control* register to 0. A starting count value for the timer can then be written into the *Load* register. To instruct the timer to use the specified starting count value, the *M* in the *Control* register should be set to 1, and the timer can be started by setting *E* = 1. The timer counts down to 0, and then sets both bit *F* in the *End-of-interrupt* register and bit *S* in the *Interrupt status* register to 1. Software can poll the value of *S* to determine when the timer period has expired. The *S* bit, and the *F* bit can be reset to 0 by reading the contents of the *End-of-Interrupt* register. Also, if bit *I*, the interrupt mask bit, in the *Control* register is set to 0, then an interrupt can be generated when the timer reaches 0 (note that bit *I* in the ARM A9 private timer shown in Figure 3 has the opposite polarity). The use of interrupts with the timer is discussed in Section 3.

The current value of the timer is available to software in the *Counter* register shown in Figure 4. The timer uses a clock frequency of 100 MHz.

There are three other identical timers in the HPS, with the following base addresses: 0xFFC09000, 0xFFD00000, and 0xFFD01000. The first of these timers uses a 100 MHz clock, and the last two timers use a 25 MHz clock.

We should mention that other timer modules also exist in the HPS. The ARM A9 MPCore has a *global* timer that is shared by both A9 cores, as well as a *watchdog* timer for each processor. Also, the HPS has two additional watchdog timers. Documentation about the global timer and watchdog timers is available in the *ARM Cortex A9 MPCore Technical Reference Manual*, and in the *Altera Cyclone V Hard Processor System Technical Reference Manual*.

2.4.3 Using a Timer with Assembly Language Code

An example of ARM A9 assembly language code is shown in Figure 5. The code configures the private timer for the A9 core so that it produces one-second timeouts. An infinite loop is used to flash the green light connected to GPIO1, discussed in Section 2.3. The light is turned on for one second, then off, and so on.

Address	31 . . . 16	15 . . .	2	1	0	Register name
0xFFC08000	Load value					Load
0xFFC08004	Current value					Counter
0xFFC08008	Unused		I	M	E	Control
0xFFC0800C	Unused				F	End-of-Interrupt
0xFFC08010	Unused				S	Interrupt status

Figure 4. HPS timer port.

```
            .equ        bit_24_pattern, 0x01000000
/* This program provides a simple example of code for the ARM A9. The program performs
 * the following:
 *      1. starts the ARM A9 private timer
 *      2. loops forever, toggling the HPS green light LEDG when the timer expires
 */
            .text
            .global     _start
_start:
            LDR         R0, =0xFF709000             // GPIO1 base address
            LDR         R1, =0xFFFEC600             // MPCore private timer base address

            LDR         R2, =bit_24_pattern         // value to turn on the HPS green light LEDG
            STR         R2, [R0, #0x4]              // write to the data direction register to set
                                                    // bit 24 (LEDG) of GPIO1 to be an output
            LDR         R3, =200000000              // timeout = 1/(200 MHz) x 200 × 10^6 = 1 sec
            STR         R3, [R1]                    // write to timer load register
            MOV         R3, #0b011                  // set bits: mode = 1 (auto), enable = 1
            STR         R3, [R1, #0x8]              // write to timer control register
LOOP:
            STR         R2, [R0]                    // turn on/off LEDG
WAIT:       LDR         R3, [R1, #0xC]              // read timer status
            CMP         R3, #0
            BEQ         WAIT                        // wait for timer to expire

            STR         R3, [R1, #0xC]              // reset timer flag bit
            EOR         R2, R2, #bit_24_pattern     // toggle LEDG value
            B           LOOP
            .end
```

Figure 5. An example of assembly language code that uses a timer.

2.4.4 Using a Timer with C Code

An example of C code is shown in Figure 6. This code performs the same actions as the assembly language program in Figure 5—it flashes on/off the green light connected to GPIO1 at one-second intervals.

```
#define bit_24_pattern 0x01000000
/* This program provides a simple example of code for the ARM A9. The program performs
 * the following:
 *      1. starts the ARM A9 private timer
 *      2. loops forever, toggling the HPS green light LEDG when the timer expires
 */
int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that the locations will not be cached,
     * even in registers) */
    volatile int * HPS_GPIO1_pt             = (int *) 0xFF709000;     // GPIO1 base address
    volatile int * MPcore_private_timer_ptr = (int *) 0xFFFEC600;     // timer base address

    int HPS_LEDG = bit_24_pattern;                      // value to turn on the HPS green light LEDG
    int counter = 200000000;                            // timeout = 1/(200 MHz) x 200 × 10^6 = 1 sec

    *(HPS_GPIO1_ptr + 1) = bit_24_pattern;              // write to the data direction register to set
                                                        // bit 24 (LEDG) of GPIO1 to be an output
    *(MPcore_private_timer_ptr) = counter;              // write to timer load register
    *(MPcore_private_timer_ptr + 2) = 0b011;            // mode = 1 (auto), enable = 1

    while (1)
    {
        *HPS_GPIO1_ptr = HPS_LEDG;                      // turn on/off LEDG
        while (*(MPcore_private_timer_ptr + 3) == 0)
            ;                                           // wait for timer to expire
        *(MPcore_private_timer_ptr + 3) = 1;            // reset timer flag bit
        HPS_LEDG ^= bit_24_pattern;                     // toggle bit that controls LEDG
    }
}
```

Figure 6. An example of C code that uses a timer.

The source code files shown in Figures 6 and 5 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 *Timer Lights*.

2.4.5 FPGA Bridges

The *FPGA bridges* depicted in Figure 1 provide connections between the HPS and FPGA in the Cyclone V SoC device. The bridges are enabled, or disabled, by using the *Bridge reset* register, which is illustrated in Figure 7 and has the address 0xFFD0501C. Three distinct bridges exist, called *HPS-to-FPGA*, *lightweight HPS-to-FPGA*, and *FPGA-to-HPS*. In the DE1-SoC Computer the first two of these bridges are used to connect the ARM A9 processor to the FPGA. As indicated in Figure 7 the bridges are enabled/disabled by bits 0 – 2 of the *Bridge reset* register. To use the memory-mapped peripherals in the FPGA, software running on the ARM A9 must enable the HPS-to-FPGA and lightweight HPS-to-FPGA bridges by setting bits #0 and #1 of the *Bridge reset* register to 0. We should note that if a user program is downloaded and run on the ARM A9 by using the Altera Monitor Program, described in Section 8, then these bridges are automatically enabled before the user program is started.

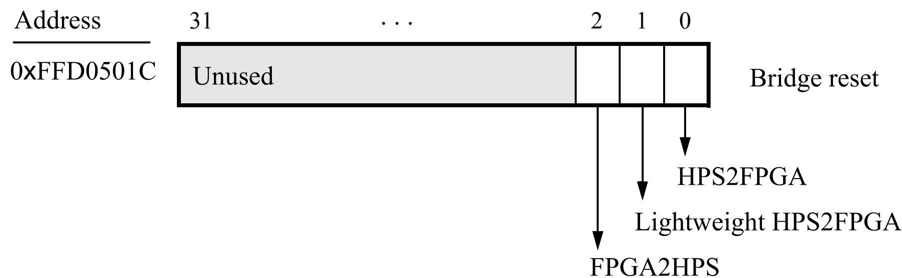


Figure 7. FPGA bridge reset register.

In addition to the components described above, the HPS also provides a number of other peripheral devices, such as USB, Ethernet, and a 3-D accelerometer (G-sensor). The G-sensor is described in the tutorial *Using the DE1-SoC Accelerometer with ARM*, available from Altera's University Program website. Documentation about the other devices connected to the HPS can be found in the *Altera Cyclone V Hard Processor System Technical Reference Manual*, as well as in the *DE1-SoC Board User Manual*.

2.5 FPGA Components

As shown in Figure 1 a number of 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.

2.5.1 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 two instances of the Nios II/f version, configured with floating-point hardware support. Instructions for using the Nios II processors in the DE1-SoC Computer are provided in a separate document, called *DE1-SoC Computer System with Nios II*.

2.5.2 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.

2.5.3 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 A9 processor using word (32-bit), halfword (16-bit), or byte operations, and is mapped to the address space 0xC0000000 to 0xC3FFFFFF.

2.5.4 On-Chip Memory

The DE1-SoC Computer includes a 256 KB memory that is implemented inside the FPGA. This memory is organized as 64K x 32 bits, and spans addresses in the range 0xC8000000 to 0xC803FFFF. The memory is used as a pixel buffer for the video-out and video-in ports.

2.5.5 On-Chip Memory Character Buffer

The DE1-SoC Computer includes an 8 KB 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 0xC9000000 to 0xC9001FFF.

2.5.6 Parallel Ports

There are several parallel ports implemented in the FPGA that support input, output, and bidirectional transfers of data between the ARM A9 processor and I/O peripherals. As illustrated in Figure 8, 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.

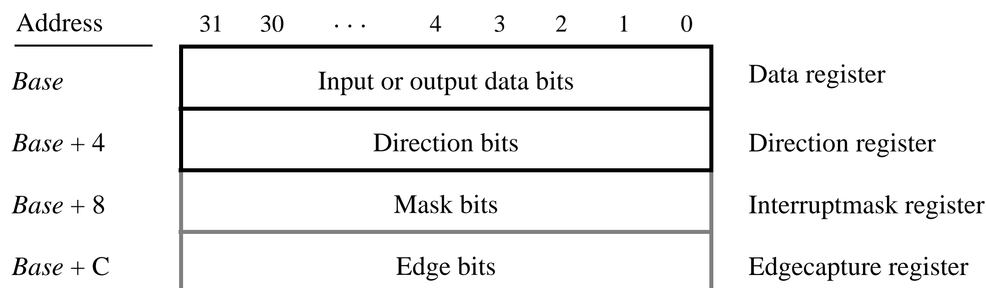


Figure 8. Parallel port registers in the DE1-SoC Computer.

Some of the parallel ports in the DE1-SoC Computer have registers at addresses *Base* + 8 and *Base* + C, as indicated

in Figure 8. These registers are discussed in Section 3.

2.5.7 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 9. The port contains a 10-bit *Data* register, which has the address 0xFF200000. This register can be written or read by the processor using word accesses, and the upper bits not used in the registers are ignored.

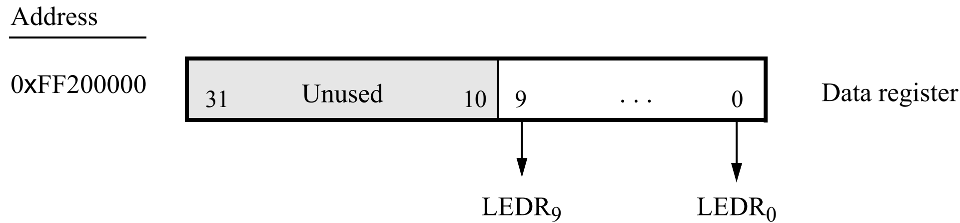


Figure 9. Output parallel port for *LEDR*.

2.5.8 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 10, 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, and read back, by using word operations. This data directly controls the segments of each display, according to the bit locations given in Figure 10. 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.

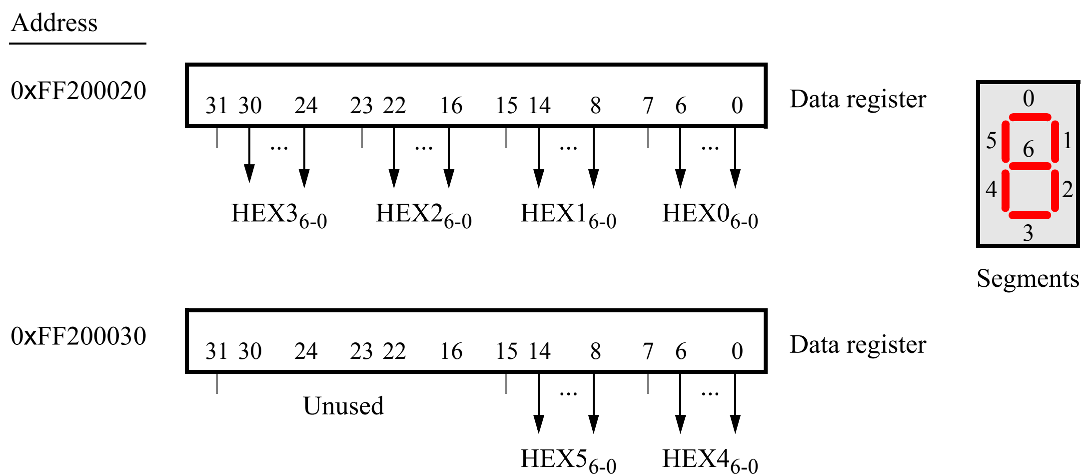


Figure 10. Bit locations for the 7-segment displays parallel ports.

2.5.9 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 11, this port comprises a 10-bit read-only *Data* register, which is mapped to address 0xFF200040.

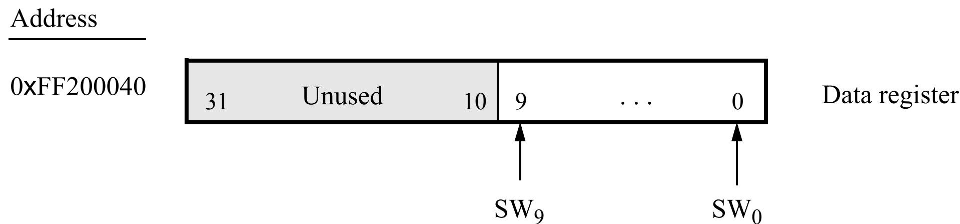


Figure 11. *Data* register in the slider switch parallel port.

2.5.10 Pushbutton Key Parallel Port

The parallel port connected to the KEY_{3-0} pushbutton switches on the DE1-SoC board comprises three 4-bit registers, as shown in Figure 12. 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_{3-0}. The other two registers shown in Figure 12, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

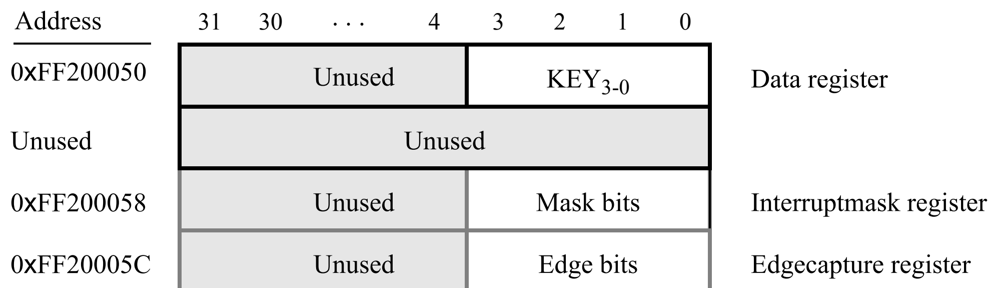


Figure 12. Registers used in the pushbutton parallel port.

2.5.11 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 8. The base address of the port for JP1 is 0xFF200060, and for JP2 is 0xFF200070. Figure 13 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.

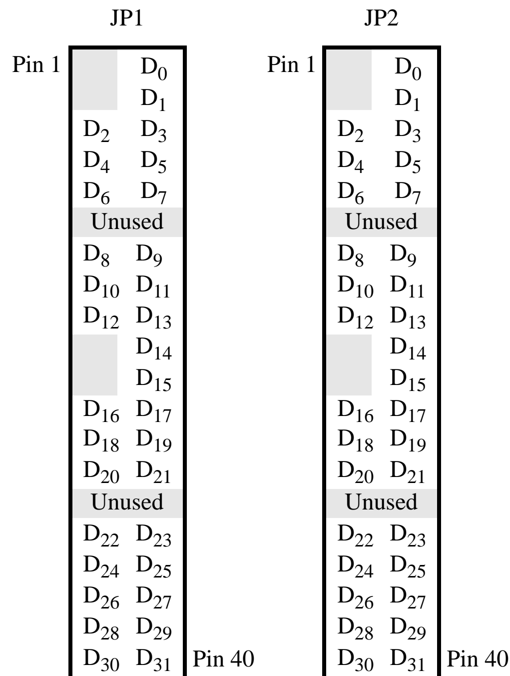


Figure 13. Assignment of parallel port bits to pins on *JP1* and *JP2*.

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

The DE1-SoC Computer provides a convenient platform for experimenting with ARM A9 assembly language code, or C code. A simple example of such code is provided in Figures 14 and 15. 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 an ARM *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 KEY_{3-0}. When a pushbutton key is pressed, the program waits in a loop until the key is released.

The source code files shown in Figures 14 and 15 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
 * 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
********************************************************************************/
                .text                           /* executable code follows */
                .global _start
_start:
                MOV     R0, #31                 // used to rotate a bit pattern: 31 positions to the
                                                // right is equivalent to 1 position to the left
                LDR     R1, =0xFF200000         // base address of LEDR lights
                LDR     R2, =0xFF200020         // base address of HEX3_HEX0 7-segs
                LDR     R3, =0xFF200040         // base address of SW switches
                LDR     R4, =0xFF200050         // base address of KEY pushbuttons
                LDR     R5, HEX_bits            // load the initial pattern for the HEX displays

DO_DISPLAY:     LDR     R6, [R3]                // load SW switches
                STR     R6, [R1]                // write to red LEDs

                LDR     R7, [R4]                // load pushbutton keys
                CMP     R7, #0                  // check if any key is presssed
                BEQ     NO_BUTTON
                MOV     R5, R6                  // copy SW switch values onto HEX displays
WAIT:
                LDR     R7, [R4]                // load pushbuttons
                CMP     R7, #0
                BNE     WAIT                    // wait for KEY release

NO_BUTTON:      STR     R5, [R2]                // store to HEX3 ... HEX0
                ROR     R5, R0                  // rotate the displayed pattern to the left

                LDR     R6, =50000000           // delay counter
SUB_LOOP:       SUBS    R6, R6, #1
                BNE     SUB_LOOP

                B       DO_DISPLAY

HEX_bits:
                .word   0x0000000F              // initial pattern for the HEX displays
                .end
```

Figure 14. An example of ARM 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 = 500000; delay_count != 0; --delay_count);     // delay loop
    }
}
```

Figure 15. An example of C code that uses parallel ports.

2.5.13 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 ARM A9 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 16. 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 16. 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 16. 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.

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

Figures 17 and 19 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 17 and 19 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*.

```
            .equ        DDR_HIGH_WORD, 0x3FFFFFFC
/********************************************************************************
 * This program demonstrates use of the JTAG UART port in the DE1-SoC Computer
 * It performs the following:
 *      1. sends an example text string to the JTAG UART
 *      2. reads and echos character data from/to the JTAG UART
********************************************************************************/
            .text                                   /* executable code follows */
            .global     _start
_start:
            /* set up stack pointer */
            MOV         SP, #DDR_HIGH_WORD          // highest memory word address

            /* print a text string */
            LDR         R4, =TEXT_STRING
LOOP:
            LDRB        R0, [R4]
            CMP         R0, #0
            BEQ         CONT                        // string is null-terminated

            BL          PUT_JTAG                    // send the character in R0 to UART
            ADD         R4, R4, #1
            B           LOOP

            /* read and echo characters */
CONT:       BL          GET_JTAG                    // read from the JTAG UART
            CMP         R0, #0                      // check if a character was read
            BEQ         CONT
            BL          PUT_JTAG
            B           CONT

            .end
```

Figure 17. An example of assembly language code that uses the JTAG UART (Part *a*).

```
/********************************************************************************
 * Subroutine to send a character to the JTAG UART
 * R0 = character to send
********************************************************************************/
            .global  PUT_JTAG
PUT_JTAG:
            LDR      R1, =0xFF201000      // JTAG UART base address
            LDR      R2, [R1, #4]         // read the JTAG UART control register
            LDR      R3, =0xFFFF
            ANDS     R2, R2, R3           // check for write space
            BEQ      END_PUT              // if no space, ignore the character
            STR      R0, [R1]             // send the character
END_PUT:
            BX       LR

/********************************************************************************
 * Subroutine to get a character from the JTAG UART
 * Returns the character read in R0
********************************************************************************/
            .global  GET_JTAG
GET_JTAG:
            LDR      R1, =0xFF201000      // JTAG UART base address
            LDR      R0, [R1]             // read the JTAG UART data register
            ANDS     R2, R0, #0x8000      // check if there is new data
            BEQ      RET_NULL             // if no data, return 0
            AND      R0, R0, #0x00FF      // return the character
            B        END_GET
RET_NULL:   MOV      R0, #0
END_GET:    BX       LR

TEXT_STRING:
            .asciz   "\nJTAG UART example code\n> "

            .end
```

Figure 17. An example of assembly language code that uses the JTAG UART (Part *b*).

2.5.15 Second JTAG UART

The DE1-SoC Computer includes a second JTAG UART that is accessible by the ARM A9 MPCORE. This second UART is mapped to the base address `0xFF201008`, and operates as described above. The reason that two JTAG UARTs are provided is to allow each processor in the ARM A9 MPCORE to have access to a separate UART.

```
/* 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 18. 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 19. An example of C code that uses the JTAG UART (Part *b*).

2.5.16 Interval Timers

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 a 100-MHz clock. The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 20. 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*) is 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 100 MHz $\times$ 125 msec = 12.5×10^6. 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.

A second interval timer, which has an identical interface to the one described above, is also available in the FPGA, starting at the base address 0xFF202020.

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

Figure 20. Interval timer registers.

3 Exceptions and Interrupts

The A9 processor supports eight types of exceptions, including the *reset* exception and the *interrupt request* (IRQ) exception, as well a number of exceptions related to error conditions. All of the exception types are described in the document *Introduction to the ARM Processor*, which is provided in Altera's University Program web site. Exception processing uses a table in memory, called the *vector table*. This table comprises eight words in memory and has one entry for each type of exception. The contents of the vector table have to be set up by software, which typically places a branch instruction in each word of the table, where the branch target is the desired exception service routine. When an exception occurs, the A9 processor stops the execution of the program that is currently running, and then fetches the instruction stored at the corresponding vector table entry. The vector table usually starts at the address 0x00000000 in memory. The first entry in the table corresponds to the reset vector, and the IRQ vector uses the seventh entry in the table, at the address 0x00000018.

The IRQ exception allows I/O peripherals to generate interrupts for the A9 processor. All interrupt signals from the peripherals are connected to a module in the processor called the *generic interrupt controller* (GIC). The GIC allows individual interrupts for each peripheral to be either enabled or disabled. When an enabled interrupt happens, the GIC causes an IRQ exception in the A9 processor. Since the same vector table entry is used for all interrupts, the software for the interrupt service routine must determine the source of the interrupt by querying the GIC. Each peripheral is identified in the GIC by an interrupt identification (ID) number. Table 1 gives the assignment of interrupt IDs for each of the I/O peripherals in the DE1-SoC Computer. The rest of this section describes the interrupt behavior associated with the timers and parallel ports, while interrupts for the other devices are discussed in Section 4.

3.1 Interrupts from the ARM A9 Private Timer

Figure 3, in Section 2.4.1, shows four registers that are associated with the A9 private timer. As we said in Section 2.4.1, bit *F* in the *Interrupt status* register is set to 1 when the timer reaches a count value of 0. It is possible to generate an A9 interrupt when this occurs, by using bit *I* of the *Control* register. Setting bit *I* to 1 causes the timer to send an interrupt signal to the GIC whenever the timer reaches a count value of 0. The *F* bit can be cleared to 0 by writing writing a 1 into the *Interrupt status* register.

I/O Peripheral	Interrupt ID #
A9 Private Timer	29
HPS GPIO1	197
HPS Timer 0	199
HPS Timer 1	200
HPS Timer 2	201
HPS Timer 3	202
FPGA Interval Timer	72
FPGA Pushbutton KEYs	73
FPGA Second Interval Timer	74
FPGA Audio	78
FPGA PS/2	79
FPGA JTAG	80
FPGA Infrared (IrDA)	81
FPGA JP1 Expansion	83
FPGA JP2 Expansion	84
FPGA PS/2 Dual	89

Table 1. Interrupt IDs in the DE1-SoC Computer.

3.2 Interrupts from the HPS Timers

Figure 4, in Section 2.4.2, shows five registers that are associated with each HPS timer. As we said in Section 2.4.2, when the timer reaches a count value of zero, bit *F* in the *End-of-Interrupt* register is set to 1. The value of the *F* bit is also reflected in the *S* bit in the *Interrupt status* register. It is possible to generate an A9 interrupt when the *F* bit becomes 1, by using the *I* bit of the *Control* register. Setting bit *I* to 0 *unmasks* the interrupt signal, and causes the timer to send an interrupt signal to the GIC whenever the *F* bit is 1. After an interrupt occurs, it can be cleared by reading the *End-of-Interrupt* register.

3.3 Interrupts from the FPGA Interval Timer

Figure 20, in Section 2.5.16, shows six registers that are associated with the interval timer. As we said in Section 2.5.16, 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 GIC whenever *TO* becomes 1. After an interrupt occurs, it can be cleared by writing any value into the *Status* register.

3.4 Interrupts from Parallel Ports

Parallel ports implemented in the FPGA in the DE1-SoC Computer were illustrated in Figure 8, which is reproduced as Figure 21. 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 GIC 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 21. Registers used for interrupts from the parallel ports.

3.4.1 Interrupts from the Pushbutton Switches

Figure 12, reproduced as Figure 22, shows the registers associated with the pushbutton parallel port. The *Interrupt-mask* 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 GIC and sets all bits of the *Edgecapture* register to zero.

Address	31 30 . . . 4	3 2 1 0	
0x10000050	Unused	$KEY_{3\text{-}0}$	Data register
Unused	Unused		
0x10000058	Unused	Mask bits	Interruptmask register
0x1000005C	Unused	Edge bits	Edgecapture register

Figure 22. Registers used for interrupts from the pushbutton parallel port.

3.5 Interrupts from the JTAG UART

Figure 16, reproduced as Figure 23, shows the data and *Control* registers of the JTAG UART. As we said in Section 2.5.13, *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 23 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.

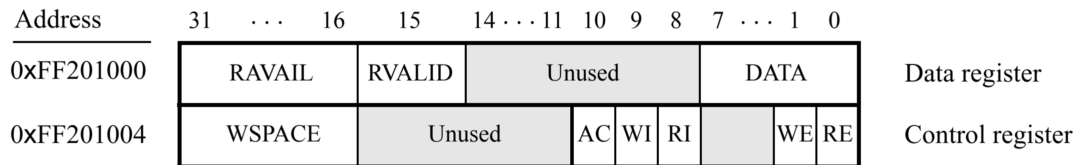


Figure 23. Interrupt bits in the JTAG UART registers.

3.6 Using Interrupts with Assembly Language Code

An example of assembly language code for the DE1-SoC Computer that uses interrupts is shown in Figure 24, which has three main parts. The beginning part of the code, in Figure 24*a*, sets up the exception vector table. This code must be in a special assembler section called .**section**, as shown. The entries in the table provide branches to the various exception service routines; they are discussed later in this section.

When the program is executed it flashes a green LED at one-second intervals, and also displays a rotating pattern on the HEX3 – 0 seven-segment displays. The pattern rotates to the right if pushbutton KEY_1 is pressed, to the left if KEY_2 is pressed, and stops rotating if KEY_3 is pressed. Pressing KEY_0 causes the pattern to be set using the SW switch values. Three types of interrupts are used in the code. The flashing green light is controlled by interrupts from an HPS timer, the HEX displays are controlled by interrupts from the FPGA interval timer, and the KEYs are also handled through interrupts.

```
/********************************************************************************
 * Initialize the exception vector table
********************************************************************************/
                .section        .vectors, "ax"
                LDR             PC, =SERVICE_RESET          // reset vector
                LDR             PC, =SERVICE_UND            // undefined instruction vector
                LDR             PC, =SERVICE_SVC            // software interrrupt vector
                LDR             PC, =SERVICE_ABT_INST       // aborted prefetch vector
                LDR             PC, =SERVICE_ABT_DATA       // aborted data vector
                .word 0         // unused vector
                LDR             PC, =SERVICE_IRQ            // IRQ interrupt vector
                LDR             PC, =SERVICE_FIQ            // FIQ interrupt vector
```

Figure 24. An example of assembly language code that uses interrupts (Part *a*).

The main program is shown in part *b* of Figure 24. It first initializes the A9 banked stack pointer (sp) registers for interrupt (IRQ) mode and supervisor (SVC) mode, because these are the processor modes that are used in the program. The code then calls subroutines to initialize the HPS timer, FPGA interval timer, and FPGA pushbutton KEYs. Finally, the code initializes the HPS GPIO1 port, enables IRQ interrupts in the A9 processor, and then enters an infinite loop. The loop code turns on and off a green light whenever the global variable named *tick* is set to 1. This variable is set to 1 by the exception service routine for the HPS timer, which is described later in this section.

```
/********************************************************************************
 * Main program
 ********************************************************************************/
            .text
            .global     _start
_start:                                         /* set up stack pointers
            MOV         R1, #0b11010010
            MSR         CPSR_c, R1              // change to IRQ mode with interrupts disabled
            LDR         SP, =0xFFFFFFFF − 3     // set IRQ stack to top of A9 on-chip memory

            MOV         R1, #0b11010011
            MSR         CPSR_c, R1              // change to SVC mode with interrupts disabled
            LDR         SP, =0x3FFFFFFF − 3     // set SVC stack to top of DDR3 memory

            BL          CONFIG_GIC              // configure the ARM generic interrupt controller
            BL          CONFIG_HPS_TIMER        // configure the HPS timer
            BL          CONFIG_KEYS             // configure the pushbutton KEYs
            BL          CONFIG_INTERVAL_TIMER   // configure the FPGA interval timer

            /* initialize the GPIO1 port */
            LDR         R0, =0xFF709000         // GPIO1 base address
            MOV         R4, #0x01000000         // value to turn on the HPS green light LEDG
            STR         R4, [R0, #0x4]          // write to the data direction register to set
                                                // bit 24 (LEDG) to be an output
            /* enable IRQ interrupts in the processor */
            MOV         R1, #0b01010011         // IRQ unmasked, MODE = SVC
            MSR         CPSR_c, R1

            LDR         R1, =0xFF200040         // slider switch base address
            LDR         R2, =0xFF200000         // LEDR base address
            LDR         R3, =TICK               // global variable
LOOP:
            LDR         R5, [R1]                // read the SW port
            STR         R5, [R2]                // light up the red lights
            LDR         R5, [R3]                // read tick variable
            CMP         R5, #0                  // HPS timer expired?
            BEQ         LOOP
            MOV         R5, #0
            STR         R5, [R3]                // reset tick variable
            STR         R4, [R0]                // turn on/off LEDG
            EOR         R4, R4, #0x01000000     // toggle bit that controls LEDG
            B           LOOP
```

Figure 24. An example of assembly language code that uses interrupts (Part *b*).

Figure 24*c* shows the subroutine that initializes the GIC. This code performs the minimum-required steps needed to configure the three interrupts used in the program, by writing to the *processor targets* (ICDIPTRn) registers in the GIC, and the *set enable* (ICDISERn) registers. For the HPS timer, the registers used have addresses 0xFFFED8C4 and 0xFFFED118, as shown in the figure. For the FPGA interval timer and KEYs, the register addresses are 0xFFFED848 and 0xFFFED108. Instructions for calculating these addresses, and determining the bit patterns to write into them can be found in the tutorial *Using the Generic Interrupt Controller*, available in Altera's University Program website. The last part of the code in Figure 24*c* enables the CPU Interface and Distributor in the GIC.

```
/* Configure the Generic Interrupt Controller (GIC) */
CONFIG_GIC:
            /* configure the HPS timer interrupt */
            LDR     R0, =0xFFFED8C4         // ICDIPTRn: processor targets register
            LDR     R1, =0x01000000         // set target to cpu0
            STR     R1, [R0]
            LDR     R0, =0xFFFED118         // ICDISERn: set enable register
            LDR     R1, =0x00000080         // set interrupt enable
            STR     R1, [R0]

            /* configure the FPGA interval timer and KEYs interrupts */
            LDR     R0, =0xFFFED848         // ICDIPTRn: processor targets register
            LDR     R1, =0x00000101         // set targets to cpu0
            STR     R1, [R0]
            LDR     R0, =0xFFFED108         // ICDISERn: set enable register
            LDR     R1, =0x00000300         // set interrupt enable
            STR     R1, [R0]

            /* configure the GIC CPU interface */
            LDR     R0, =0xFFFEC100         // base address of CPU interface
            /* Set Interrupt Priority Mask Register (ICCPMR) */
            LDR     R1, =0xFFFF             // enable interrupts of all priorities levels
            STR     R1, [R0, #0x04]         // ICCPMR
            /* Set the enable bit in the CPU Interface Control Register (ICCICR). This bit allows
            * interrupts to be forwarded to the CPU(s) */
            MOV     R1, #1
            STR     R1, [R0, #0x00]         // ICCICR

            /* Set the enable bit in the Distributor Control Register (ICDDCR). This bit allows
            * the distributor to forward interrupts to the CPU interface(s) */
            LDR     R0, =0xFFFED000
            STR     R1, [R0, #0x00]         // ICDDCR

            BX      LR
```

Figure 24. An example of assembly language code that uses interrupts (Part *c*).

Figure 24*d* shows the subroutines used to initialize the timers and pushbutton KEYs. The CONFIG_HPS_TIMER routine sets up the HPS timer 0 so that it will produce an interrupt every one second. Since this timer uses a 100 MHz clock, the timer *load* register is initialized to the value 100×10^6. The CONFIG_INTERVAL_TIMER routine configures the FPGA interval timer to produce interrupts every 50 msec. Since this timer uses a 100 MHz clock, the required starting count value is 5×10^6. The CONFIG_KEYS routine sets up the FPGA KEYs parallel port to produce an interrupt when any KEY is pressed.

```
/* Configure the HPS timer to create interrupts at one-second intervals */
CONFIG_HPS_TIMER:
            LDR     R0, =0xFFC08000         // HPA timer 0 base address
            MOV     R1, #0                  // used to stop the timer
            STR     R1, [R0, #0x8]          // write to timer control register
            LDR     R1, =100000000          // period = 1/(100 MHz) × (100x10^6) = 1 sec
            STR     R1, [R0]                // write to timer load register
            MOV     R1, #0b011              // int mask = 0, mode = 1, enable = 1
            STR     R1, [R0, #0x8]          // write to timer control register
            BX      LR

/* Configure the FPGA interval timer to create interrupts at 50-msec intervals */
CONFIG_INTERVAL_TIMER:
            LDR     R0, =0xFF202000         // Interval timer base address
            LDR     R1, =5000000            // 1/(100 MHz) ×(5000000) = 50 msec
            STR     R1, [R0, #0x8]          // store the low half word of counter start value
            LSR     R1, R1, #16
            STR     R1, [R0, #0xC]          // high half word of counter start value

            // start the interval timer, enable its interrupts
            MOV     R1, #0x7                // START = 1, CONT = 1, ITO = 1
            STR     R1, [R0, #0x4]
            BX      LR

/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
            // write to the pushbutton port interrupt mask register
            LDR     R0, =0xFF200050         // pushbutton key base address
            MOV     R1, #0xF                // set interrupt mask bits
            STR     R1, [R0, #0x8]          // interrupt mask register is (base + 8)
            BX      LR
```

Figure 24. An example of assembly language code that uses interrupts (Part *d*).

Part *e* of Figure 24 shows the global data used by the program. It includes the *tick* variable that was discussed for the code in Figure 24*b*, and three other variables. The *pattern* variable holds the bit-pattern that is written to the

HEX3 − 0 seven-segment displays, the *key_pressed* variable indicates which FPGA KEY has been recently pressed, and the *shift_dir* variable specifies the direction of shifting for the HEX displays.

```
            /* Global variables */
            .global     TICK
TICK:
            .word       0x0                     // used by HPS timer
            .global     PATTERN
PATTERN:
            .word       0x0000000F              // initial pattern for HEX displays
            .global     KEY_PRESSED
KEY_PRESSED:
            .word       1                       // recent pushbutton KEY pressed
            .global     SHIFT_DIR
SHIFT_DIR:
            .word       1                       // pattern shifting direction
            .end
```

Figure 24. An example of assembly language code that uses interrupts (Part *e*).

The exception service routines for the main program in Figure 24 are given in Figure 25. Part *a* of the figure gives the IRQ exception handler. This routine first reads from the *interrupt acknowledge* register in the GIC to determine the interrupt ID of the peripheral that caused the interrupt. The code then checks which of the three possible sources of interrupt has occurred, and calls the corresponding interrupt service routine for the HPS timer, FPGA interval timer, or FPGA KEY parallel port. These interrupt service routine are shown in Figures 26 to 28.

Finally, the exception handler in Figure 25 writes to the *end-of-interrupt* register in the GIC to clear the interrupt, and then returns to the main program by using the instruction "SUBS PC, LR, #4".

Figure 25*b* shows handlers for exceptions that correspond to the reset exception, various types of error conditions, and the FIQ interrupt. The reset handler shows a branch to the start of the main program in Figure 24. This handler is just an indicator of the result of performing a reset of the A9 processor—the actual reset process involves executing code from a special boot ROM on the processor, and then executing a program called the *pre-loader* before actually starting the main program. More information about the reset process for the A9 processor can be found in the document "Using the Pre-loader Software for the A9 Processor," which is available from Altera's University Program website. The other handlers in Figure 25*b* , which are just loops that branch to themselves, are intended to serve as placeholders for code that would handle the corresponding exceptions. More information about each of these types of exceptions can be found in the document *Introduction to the ARM Processor*, also available in Altera's University Program web site.

```
/******************************************************************************
 * Define the IRQ exception handler
******************************************************************************/
            .global     SERVICE_IRQ
SERVICE_IRQ:
            PUSH        R0-R7, LR                   // save registers

            /* get the interrupt ID from the GIC */
            LDR         R4, =0xFFFEC100             // GIC CPU interface base address
            LDR         R5, [R4, #0x0C]             // read the ICCIAR

HPS_TIMER_CHECK:
            CMP         R5, #199                    // check for HPS timer interrupt
            BNE         INTERVAL_TIMER_CHECK
            BL          HPS_TIMER_ISR
            B           EXIT_IRQ

INTERVAL_TIMER_CHECK:
            CMP         R5, #72                     // check for FPGA timer interrupt
            BNE         KEYS_CHECK
            BL          TIMER_ISR
            B           EXIT_IRQ

KEYS_CHECK:
            CMP         R5, #73                     // check for KEYs interrupt

UNEXPECTED:
            BNE         UNEXPECTED                  // if not recognized, stop here

            BL          KEY_ISR
EXIT_IRQ:
            STR         R5, [R4, #0x10]             // Write to end-of-interrupt register (ICCEOIR)

            POP         R0-R7, LR
            SUBS        PC, LR, #4
```

Figure 25. Exception handlers assembly language code (Part *a*).

```
/* Define the remaining exception handlers */
                .global     SERVICE_RESET                /* Reset */
SERVICE_RESET:
                B           _start
                .global     SERVICE_UND                  /* Undefined instructions */
SERVICE_UND:
                B           SERVICE_UND
                .global     SERVICE_SVC                  /* Software interrupts */
SERVICE_SVC:
                B           SERVICE_SVC
                .global     SERVICE_ABT_DATA             /* Aborted data reads */
SERVICE_ABT_DATA:
                B           SERVICE_ABT_DATA
                .global     SERVICE_ABT_INST             /* Aborted instruction fetch */
SERVICE_ABT_INST:
                B           SERVICE_ABT_INST
                .global     SERVICE_FIQ                  /* FIQ */
SERVICE_FIQ:
                B           SERVICE_FIQ
                .end
```

Figure 25. Exception handlers assembly language code (Part *b*).

```
/*******************************************************************************
 * HPS timer interrupt service routine
 *
 * This code increments the TICK global variable, and clears the interrupt
 ******************************************************************************/
                .extern     TICK                         // externally-defined variable
                .global     HPS_TIMER_ISR
HPS_TIMER_ISR:
                LDR         R0, =0xFFC08000              // base address of timer
                LDR         R1, =TICK                    // used by main program

                LDR         R2, [R1]
                ADD         R2, R2, #1
                STR         R2, [R1]                     // ++tick
                LDR         R0, [R0, #0xC]               // read timer end-of-interrupt
                                                         // register to clear the interrupt
                BX          LR
                .end
```

Figure 26. Interrupt service routine for the HPS timer.

```
/**************************************************************************************
 * Pushbutton KEY interrupt service routine
 *
 * This routine checks which KEY has been pressed. It writes this value to the global variable
 * KEY_PRESSED.
 *************************************************************************************/
            .extern     KEY_PRESSED             // externally defined variable
            .global     KEY_ISR
KEY_ISR:
            LDR         R0, =0xFF200050         // base address of KEYs
            LDR         R1, [R0, #0xC]          // read edge capture register
            STR         R1, [R0, #0xC]          // clear the interrupt

            LDR         R0, =KEY_PRESSED        // global variable to return the result
CHECK_KEY0:
            MOVS        R3, #0x1
            ANDS        R3, R1                  // check for KEY0
            BEQ         CHECK_KEY1
            MOVS        R2, #0
            STR         R2, [R0]                // return KEY0 value
            B           END_KEY_ISR
CHECK_KEY1:
            MOVS        R3, #0x2
            ANDS        R3, R1                  // check for KEY1
            BEQ         CHECK_KEY2
            MOVS        R2, #1
            STR         R2, [R0]                // return KEY1 value
            B           END_KEY_ISR
CHECK_KEY2:
            MOVS        R3, #0x4
            ANDS        R3, R1                  // check for KEY2
            BEQ         IS_KEY3
            MOVS        R2, #2
            STR         R2, [R0]                // return KEY2 value
            B           END_KEY_ISR
IS_KEY3:
            MOVS        R2, #3
            STR         R2, [R0]                // return KEY3 value
END_KEY_ISR:
            BX          LR

            .end
```

Figure 27. Interrupt service routine for the pushbutton KEYs.

```
/*******************************************************************************
 * Interval timer interrupt service routine
 *
 * Shifts a PATTERN being displayed on the HEX displays. The shift direction is set by the
 * external variable KEY_PRESSED.
 ******************************************************************************/
            .extern     KEY_PRESSED
            .extern     SHIFT_DIR
            .extern     PATTERN
            .global     TIMER_ISR
TIMER_ISR:
            PUSH        R4-R7
            LDR         R1, =0xFF202000         // interval timer base address
            MOVS        R0, #0
            STR         R0, [R1]                // clear the interrupt

            LDR         R1, =0xFF200020         // HEX3_HEX0 base address
            LDR         R2, =PATTERN            // set up a pointer to the pattern for HEX displays
            LDR         R3, =KEY_PRESSED        // set up a pointer to the key pressed
            LDR         R7, =SHIFT_DIR          // set up a pointer to the shift direction variable

            LDR         R6, [R2]                // load pattern for HEX displays
            STR         R6, [R1]                // store to HEX3 ... HEX0

            LDR         R4, [R3]                // check which key has been pressed
CHK_KEY0:   CMP         R4, #0                  // KEY0
            BNE         CHK_KEY1
            LDR         R1, =0xFF200040         // SW switches base address
            LDR         R6, [R1]                // load a new pattern from the SW switches
            B           SHIFT
CHK_KEY1:   CMP         R4, #0                  // KEY1
            BNE         CHK_KEY2
            MOVS        R5, #1                  // set rotation direction to the right (1)
            STR         R5, [R7]
            B           SHIFT
CHK_KEY2:   CMP         R4, #0                  // KEY2
            BNE         CHK_KEY3
            MOVS        R5, #2                  // set rotation direction to the left (2)
            STR         R5, [R7]
            B           SHIFT
```

Figure 28. Interrupt service routine for the interval timer (Part *a*).

```
CHK_KEY3:   CMP     R4, #0              // KEY3
            BNE     SHIFT
            MOVS    R5, #4              // set rotation direction to none (4)
            STR     R5, [R7]
SHIFT:
            MOVS    R5, #4              // R5 = NONE (4)
            STR     R5, [R3]            // key press handled, so clear
            LDR     R5, [R7]            // get shift direction
            CMP     R5, #1              // RIGHT
            BNE     SHIFT_L
            MOVS    R5, #1              // used to rotate right by 1 position
            RORS    R6, R5              // rotate right for KEY1
            B       END_TIMER_ISR
SHIFT_L:
            CMP     R5, #2              // LEFT
            BNE     END_TIMER_ISR
            MOVS    R5, #31             // used to rotate left by 1 position
            RORS    R6, R5

END_TIMER_ISR:
            STR     R6, [R2]            // store HEX display pattern
            POP     R4-R7
            BX      LR

            .end
```

Figure 28. Interrupt service routine for the interval timer (Part *b*).

3.7 Using Interrupts with C Code

An example of C code for the DE1-SoC Computer that uses interrupts is shown in Figure 29. This code performs exactly the same operations as the code described in Figure 24.

Before it call subroutines to configure the generic interrupt controller (GIC), timers, and pushbutton KEY port, the main program first initializes the IRQ mode stack pointer by calling the routine *set_A9_IRQ_stack()*. The code for this routine uses in-line assembly language instructions, as shown in Part *b* of the figure. This step is necessary because the C compiler generates code to set only the supervisor mode stack, which is used for running the main program, but the compiler does not produce code for setting the IRQ mode stack. To enable IRQ interrupts in the A9 processor the main program uses the in-line assembly code shown in the subroutine called *enable_A9_interrupts()*.

The exception handlers for the main program in Figure 29 are given in Figure 30. These routines have unique names that are meaningful to the C compiler and linker tools, and they are declared with the special type of **__attribute__** called **interrupt**. These mechanisms cause the C compiler and linker to use the addresses of these routines as the contents of the exception vector table.

The function with the name *__cs3_isr_irq* is the IRQ exception handler. As discussed for the assembly language code in Figure 25 this routine first reads from the *interrupt acknowledge* register in the GIC to determine the interrupt ID of the peripheral that caused the interrupt, and then calls the corresponding interrupt service routine for either the HPS timer, FPGA interval timer, or FPGA KEY parallel port. These interrupt service routines are shown in Figures 31 to 33.

Figure 30 also shows handlers for exceptions that correspond to the various types of error conditions and the FIQ interrupt. These handlers are just loops that are meant to serve as place-holders for code that would handle the corresponding exceptions.

The source code files shown in Figure 24 to Figure 32 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*.

```
void set_A9_IRQ_stack (void);
void config_GIC (void);
void config_HPS_timer (void);
void config_interval_timer (void);
void config_KEYs (void);
void enable_A9_interrupts (void);

/* These global variables are written by interrupt service routines; we have to declare these as volatile
 * to avoid the compiler caching their values in registers */
volatile int tick = 0;                    // set to 1 every time the HPS timer expires
volatile int key_pressed = 4;             // stores a KEY value when pressed (4 = NONE)
volatile int pattern = 0x0000000F;        // pattern for HEX displays
volatile int shift_dir = 2;               // direction to shift the pattern (2 = LEFT)
/*******************************************************************************
 * Main program
 ******************************************************************************/
int main(void)
{
    volatile int * HPS_GPIO1_ptr = (int *) 0xFF709000;     // GPIO1 base address
    volatile int * LEDR_ptr = (int *) 0xFF200000;          // LEDR base address
    volatile int * slider_switch_ptr = (int *) 0xFF200040; // SW base address
    volatile int HPS_timer_LEDG = 0x01000000;      // value to turn on the HPS green light LEDG

    set_A9_IRQ_stack ();        // initialize the stack pointer for IRQ mode
    config_GIC ();              // configure the general interrupt controller
    config_HPS_timer ();        // configure the HPS timer
    config_KEYs ();             // configure pushbutton KEYs to generate interrupts
    config_interval_timer ();   // configure Altera interval timer to generate interrupts

    *(HPS_GPIO1_ptr + 0x1) = HPS_timer_LEDG;      // write to the data direction register to set
                                                  // bit 24 (LEDG) to be an output
    enable_A9_interrupts ();    // enable interrupts in the A9 processor
    while (1)
    {
        *(LEDR_ptr) = *(slider_switch_ptr);             // light up the red lights
        if (tick)
        {
            tick = 0;
            *HPS_GPIO1_ptr = HPS_timer_LEDG;            // turn on/off the green light LEDG
            HPS_timer_LEDG ^= 0x01000000;               // toggle the bit that controls LEDG
        }
    }
}
```

Figure 29. An example of C code that uses interrupts (Part *a*).

```
/* Initialize the banked stack pointer register for IRQ mode */
void set_A9_IRQ_stack(void)
{
    int stack, mode;
    stack = 0xFFFFFFFF − 7;      // top of A9 on-chip memory, aligned to 8 bytes
    /* change processor to IRQ mode with interrupts disabled */
    mode = 0b11010010;
    asm("msr cpsr, %[ps]" : : [ps] "r" (mode));
    /* set banked stack pointer */
    asm("mov sp, %[ps]" : : [ps] "r" (stack));

    /* go back to SVC mode before executing subroutine return! */
    mode = 0b11010011;
    asm("msr cpsr, %[ps]" : : [ps] "r" (mode));
}

/* Turn on interrupts in the ARM processor */
void enable_A9_interrupts(void)
{
    int status = 0b01010011;
    asm("msr cpsr, %[ps]" : : [ps]"r"(status));
}

/* Configure the Generic Interrupt Controller (GIC) */
void config_GIC(void)
{
    /* configure the HPS timer interrupt */
    *((int *) 0xFFFED8C4) = 0x01000000;
    *((int *) 0xFFFED118) = 0x00000080;

    /* configure the FPGA interval timer and KEYs interrupts */
    *((int *) 0xFFFED848) = 0x00000101;
    *((int *) 0xFFFED108) = 0x00000300;

    // Set Interrupt Priority Mask Register (ICCPMR). Enable interrupts of all priorities
    *((int *) 0xFFFEC104) = 0xFFFF;

    // Set CPU Interface Control Register (ICCICR). Enable signaling of interrupts
    *((int *) 0xFFFEC100) = 1;      // enable = 1

    // Configure the Distributor Control Register (ICDDCR) to send pending interrupts to CPUs
    *((int *) 0xFFFED000) = 1;      // enable = 1
}
```

Figure 29. An example of C code that uses interrupts (Part *b*).

```
/* setup HPS timer */
void config_HPS_timer()
{
    volatile int * HPS_timer_ptr = (int *) 0xFFC08000;     // timer base address

    (HPS_timer_ptr + 0x2) = 0;          // write to control register to stop timer
    /* set the timer period */
    int counter = 100000000;            // period = 1/(100 MHz) x (100x10^6) = 1 sec
    *(HPS_timer_ptr) = counter;         // write to timer load register

    /* write to control register to start timer, with interrupts */
    *(HPS_timer_ptr + 2) = 0b011;       // interrupt mask = 0, mode = 1, enable = 1
}

/* setup the interval timer interrupts in the FPGA */
void config_interval_timer()
{
    volatile int * interval_timer_ptr = (int *) 0xFF202000;     // interal timer base address

    /* set the interval timer period for scrolling the HEX displays */
    int counter = 5000000;       // 1/(100 MHz)×(5000000) = 50 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
}

/* setup the KEY interrupts in the FPGA */
void config_KEYs()
{
    volatile int * KEY_ptr = (int *) 0xFF200050;      // pushbutton KEY address

    *(KEY_ptr + 2) = 0xF;       // enable interrupts for all four KEYs
}
```

Figure 29. An example of C code that uses interrupts (Part *c*).

```
void HPS_timer_ISR (void);
void interval_timer_ISR (void);
void pushbutton_ISR (void);

/* Define the IRQ exception handler */
void __attribute__ ((interrupt)) __cs3_isr_irq (void)
{
    // Read the ICCIAR from the processor interface
    int int_ID = *((int *) 0xFFFEC10C);

    if (int_ID == 199)              // check if interrupt is from the HPS timer
        HPS_timer_ISR ();
    else if (int_ID == 72)          // check if interrupt is from the Altera timer
        interval_timer_ISR ();
    else if (int_ID == 73)          // check if interrupt is from the KEYs
        pushbutton_ISR ();
    else
        while (1)                   // if unexpected, then stay here

    // Write to the End of Interrupt Register (ICCEOIR)
    *((int *) 0xFFFEC110) = int_ID;
    return;
}

// Define the remaining exception handlers */
void __attribute__ ((interrupt)) __cs3_isr_undef (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_swi (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_pabort (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_dabort (void)
{
    while (1);
}
void __attribute__ ((interrupt)) __cs3_isr_fiq (void)
{
    while (1);
}
```

Figure 30. Exception handlers C code.

```
/********************************************************************************
 * HPS timer interrupt service routine
 * This code increments the TICK global variable, and clears the interrupt
********************************************************************************/
extern volatile int tick;
void HPS_timer_ISR( )
{
    volatile int * HPS_timer_ptr = (int *) 0xFFC08000;    // HPS timer base address
    ++tick;                         // used by main program

    *(HPS_timer_ptr + 3);           // Read timer end of interrupt register to
                                    // clear the interrupt
    return;
}
```

Figure 31. Interrupt service routine for the HPS timer.

```
/********************************************************************************
 * Pushbutton - Interrupt Service Routine
 * This routine checks which KEY has been pressed. It writes this value to the global
 * variable key_pressed.
********************************************************************************/
extern volatile int key_pressed;
void pushbutton_ISR( void )
{
    volatile int * KEY_ptr = (int *) 0xFF200050;
    int press;

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

    if (press & 0x1)                // KEY0
        key_pressed = 0;
    else if (press & 0x2)           // KEY1
        key_pressed = 1;
    else if (press & 0x4)           // KEY2
        key_pressed = 2;
    else                            // press & 0x8, which is KEY3
        key_pressed = 3;

    return;
}
```

Figure 32. Interrupt service routine for the pushbutton KEYs.

```
/********************************************************************************
 * 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.
********************************************************************************/
extern volatile int key_pressed;
extern volatile int pattern;
extern volatile int shift_dir;
void interval_timer_ISR( )
{
	volatile int * interval_timer_ptr = (int *) 0xFF202000;	// Altera timer address
	volatile int * slider_switch_ptr  = (int *) 0xFF200040;	// SW base address
	volatile int * HEX3_HEX0_ptr = (int *) 0xFF200020;	// 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 == 0)				// for KEY0 read new pattern
		pattern = *(slider_switch_ptr);		// read a new pattern from the SW slider switches
	else if (key_pressed == 1)			// for KEY1 rotate right
		shift_dir = 1;				// 1 = RIGHT
	else if (key_pressed == 2)			// for KEY2 rotate right
		shift_dir = 2;				// 2 = LEFT
	else if (key_pressed == 3)			// for KEY3 don't rotate
		shift_dir = 4;				// 4 = NONE

	key_pressed = 4;				// key press handled, so clear (4 = NONE)

	if (shift_dir == 2)				// LEFT
		if (pattern & 0x80000000)
			pattern = (pattern << 1) | 1;
		else
			pattern = pattern << 1;
	else if (shift_dir == 1)			// RIGHT
		if (pattern & 0x00000001)
			pattern = (pattern >> 1) | 0x80000000;
		else
			pattern = (pattern >> 1) & 0x7FFFFFFF;
	// else don't shift
	return;
}
```

Figure 33. Interrupt service routine for the interval timer.

4 Media Components

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

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 34. 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 that 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 34 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 34. Audio port registers.

The read-only *Fifospace* register in Figure 34 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 35. 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 35. 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 holds the data (color) for each pixel that is displayed by the VGA controller. As illustrated in Figure 36, the pixel 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 37*a* shows that each pixel color 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 37, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE1-SoC Computer the default address of the pixel buffer is 0xC8000000, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at location 0,0 has the address 0xC8000000, the pixel 1,0 has the address *base* + $(00000000\ 000000001\ 0)_2$

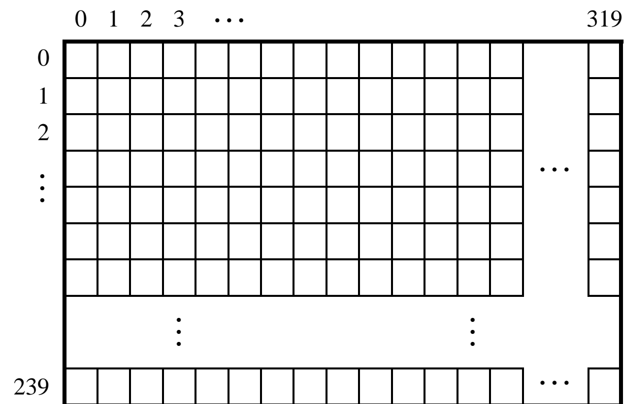
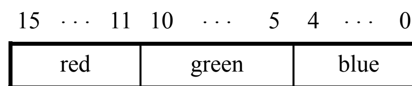
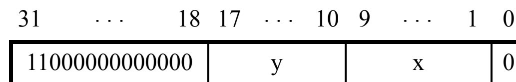


Figure 36. Pixel buffer coordinates.

= 0xC8000002, the pixel 0,1 has the address *base* + $(00000001\ 000000000\ 0)_2$ = 0xC8000400, and the pixel at location 319,239 has the address *base* + $(11101111\ 100111111\ 0)_2$ = 0xC803BE7E.



(a) Pixel values



(b) Pixel buffer addresses

Figure 37. Pixel values and addresses.

You can create an image by writing color values into the pixel addresses as described above. A dedicated *pixel buffer controller* continuously reads this pixel data from sequential addresses in the corresponding memory for display on the VGA screen. You can modify the pixel data at any time, simply by writing to the pixel addresses. Thus, an image can be changed even when it is in the process of being displayed. However, it is also possible to avoid making changes to the pixel buffer while it is being displayed, by using the concept of *double-buffering*. In this scheme, two pixel buffers are involved, called the *front* and *back* buffers, described below.

4.2.2 Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the VGA screen. This pixel buffer controller includes a programming interface in the form of a set of registers, as illustrated in Figure 38. The register at address 0xFF203020 is called the *Buffer* register, and the register at address

0xFF203024 is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the VGA screen. As mentioned above, in the default configuration of the DE1-SoC Computer this Buffer register is set to the address 0xC8000000, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also 0xC8000000, which means that there is only one pixel buffer. But software can modify the address stored in the Backbuffer register, thereby creating a second pixel buffer. The pixel buffer can be located in the SDRAM memory in the DE1-SoC Computer, which has the base address 0xC0000000. Note that the pixel buffer cannot be located in the DDR3 memory in the DE1-SoC Computer, because the pixel buffer controller is not connected to the DDR3 memory. An image can be drawn into the second buffer by writing to its pixel addresses. This image is not displayed on the VGA monitor until a pixel buffer *swap* is performed, as explained below.

A pixel buffer swap is caused by writing the value 1 to the Buffer register. This write operation does not directly modify the content of the Buffer register, but instead causes the contents of the Buffer and Backbuffer registers to be swapped. The swap operation does not happen right away; it occurs at the end of a VGA screen-drawing cycle, after the last pixel in the bottom-right corner has been displayed. This time instance is referred to as the *vertical synchronization* time, and occurs every 1/60 seconds. Software can poll the value of the S bit in the *Status* register, at address 0xFF20302C, to see when the vertical synchronization has happened. Writing the value 1 into the Buffer register causes S to be set to 1. Then, when the swap of the Buffer and Backbuffer registers has been completed S is reset back to 0.

Address	31 . . . 24	23 . . . 16	15 . . . 8	7 . . . 4	3	2	1	0	
0xFF203020	front buffer address								Buffer register
0xFF203024	back buffer address								Backbuffer register
0xFF203028	Y		X						Resolution register
0xFF20302C	m	n	Unused	B	Unused		A	S	Status register

Figure 38. Pixel buffer controller registers.

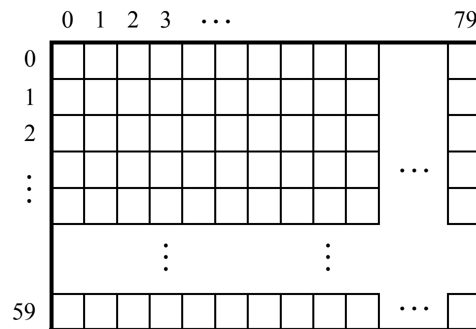
In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, a pixel buffer swap is performed. Then, the pixel buffer that is now pointed to by the Backbuffer register, which was already displayed, is cleared and the next new image is drawn. In this way, the next image to be displayed is always drawn in the "back" pixel buffer, and the two pixel buffer pointers are swapped when the new image is ready to be displayed. Each time a swap is performed software has to synchronize with the VGA controller by waiting until the S bit in the Status register becomes 0.

As shown in Figure 38 the *Status* register contains additional information other than the S bit. The fields n and m give the number of address bits used for the X and Y pixel coordinates, respectively. The B field specifies the number of bytes used for each pixel, with the minimum being 1 and the maximum 4. The A field allows the selection of two different ways of forming pixel addresses. If configured with $A = 0$, then the pixel controller expects addresses to contain X and Y fields, as we have used in this section. But if $A = 1$, then the controller expects addresses to be consecutive values starting from 0 and ending at the total number of pixels-1.

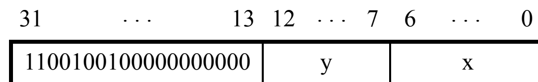
In Figure 37*b* the default values of the status register fields in the DE1-SoC Computer are used when forming pixel addresses. The defaults are $n = 9$, $m = 8$, $B = 2$, and $A = 0$. If the pixel buffer controller is changed to provide different values of these fields, then the way in which pixel addresses are formed has to be modified accordingly. The programming interface also includes a *Resolution* register, shown in Figure 38, that contains the X and Y resolution of the pixel buffer(s).

4.2.3 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 39*a*, 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 39*a* 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 39 shows that characters are addressed in the memory by using the combination of a *base* address, which has the value $(C9000000)_{16}$, and an *x,y* offset. Using this scheme, the character at location 0,0 has the address $(C9000000)_{16}$, the character 1,0 has the address *base* + $(000000\ 0000001)_2 = (C9000001)_{16}$, the character 0,1 has the address *base* + $(000001\ 0000000)_2 = (C9000080)_{16}$, and the character at location 79,59 has the address *base* + $(111011\ 1001111)_2 = (C9001DCF)_{16}$.



(a) Character buffer coordinates



(b) Character buffer addresses

Figure 39. Character buffer coordinates and addresses.

4.2.4 Using the Video-out Port with C code

A fragment of C code that uses the pixel and character buffers is shown in Figure 40. The first **for** 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 40 is included in the sample program called *Media* that is distributed with the Altera Monitor Program.

```
int pixel_ptr, row, col;
...
/* Draw a box with corners (x1, y1) and (x2, y2). Assume that the box coordinates are valid */
for (row = y1; row <= y2; row++)
    for (col = x1; col <= x2; ++col)
    {
        pixel_ptr = 0xC8000000 | (row << 10) | (col << 1);
        *(short *)pixel_ptr = pixel_color;                  // set pixel color
    }
}

...

int offset;
char *text_ptr;
...
/* Display a null-terminated text string at coordinates x, y. Assume that the text fits on one line */
offset = (y << 7) + x;
while ( *(text_ptr) )
{
    *(0xC9000000 + offset) = *(text_ptr);                   // write to the character buffer
    ++text_ptr;
    ++offset;
}
```

Figure 40. An example of code that uses the video-out port.

4.3 Video-in Port

The DE1-SoC Computer includes a video-in port for use with the composite video-in connector on the DE1-SoC board. The video analog-to-digital converter (ADC) connected to this port is configured to support an NTSC video source. The video-in port provides frames of video at a resolution of 320 x 240 pixels. These video frames can be displayed on a VGA monitor by using the video-out port described in Section 4.2. The video-in port writes each frame of the video-in data into the pixel buffer described in Section 4.2.1. The video-in port can be configured to provide two types of images: either the "raw" image provided by the video ADC, or a version of this image in which only "edges" that are detected in the image are drawn.

The video-in port has a programming interface that consists of two registers, as illustrated in Figure 41. The *Control* register at the address 0xFF20306C is used to enable or disable the video input. If the *EN* bit in this register is set to 0, then the video-in core does not store any data into the pixel buffer. Setting *EN* to 1 and then changing *EN* to 0 can be used to capture a still picture from the video-in port.

The register at address 0xFF203070 is used to enable or disable edge detection. Setting the *E* bit in this register

to 1 causes the input video to passed through hardware circuits that detect edges in the images. The image stored in the pixel buffer will then consist of dark areas that are puctuated by lighter lines along the edges that have been detected. Setting $E = 0$ causes a normal image to be stored into the pixel buffer.

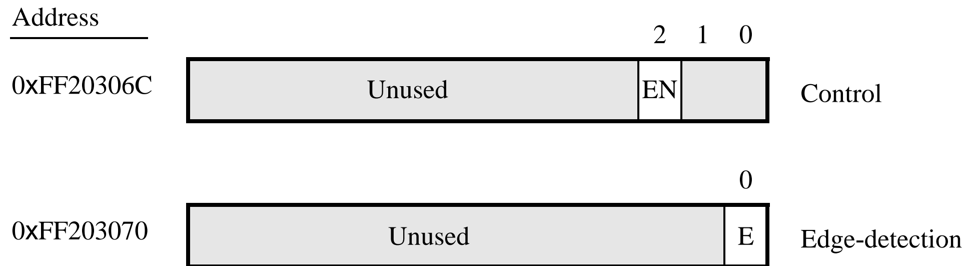


Figure 41. The video-in port programming interface.

4.4 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.

4.5 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 42. 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 43. 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 $byte_1$, $byte_2$, and $byte_3$. This code is included as part of a larger sample program called *Media* that is distributed with the Altera Monitor Program.

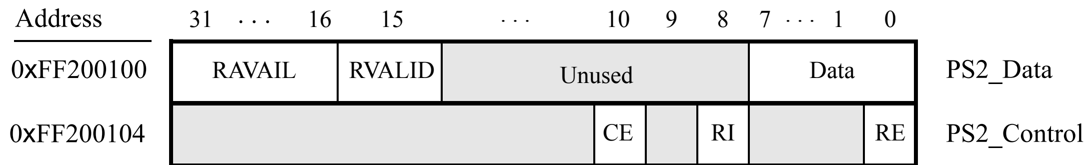


Figure 42. PS/2 port registers.

```
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 43. An example of code that uses the PS/2 port.

4.5.1 PS/2 Port Dual

The DE0-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 DE0-SoC board through this cable. The PS/2 port dual has the same registers as the PS/2 port shown in Figure 43, except that the base address of its *PS2_Data* register is 0xFF200108 and the base address of its *PS2_Control* register is 0xFF20010C.

4.6 IrDA Infrared Serial Port

The IrDA port in the DE0-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 44. 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 44, 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 44. 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 44. 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.

4.7 Analog-to-Digital Conversion Port

The Analog-to-Digital Conversion (ADC) Port provides access to the eight-channel, 12-bit analog-to-digital converter on the DE1-SoC board. As illustrated in Figure 45, the ADC port comprises eight 12-bit registers starting at the base address 0xFF204000. The first two registers have dual purposes, acting as both data and control registers. By default, the ADC port updates the A-to-D conversion results for all ports only when instructed to do so. Writing to the control register at address 0xFF204000 causes this update to occur. Reading from the register at address 0xFF204000 provides the conversion data for channel 0. Reading from the other seven registers provides the conversion data for the corresponding channels. It is also possible to have the ADC port continually request A-to-D conversion data for all channels. This is done by writing the value 1 to the control register at address 0xFF204004. The *R* bit of each channel register in Figure 45 is used in Auto-update mode. *R* is set to 1 when its corresponding channel is refreshed and set to 0 when the channel is read.

Figure 46 shows the connector on the DE1-SoC board that is used with the ADC port. Analog signals in the range of 0 V to the V_{CC5} power-supply voltage can be connected to the pins for channels 0 to 7.

Address	31 . . . 16	15	14 12	11 . . . 0	
0xFF204000	Unused	R	Unused		Channel 0 / Update
0xFF204004	Unused	R	Unused		Channel 1 / Auto-update
0xFF204008	Unused	R	Unused		Channel 2
	... not shown				
0xFF20401C	Unused	R	Unused		Channel 7

Figure 45. ADC port registers.

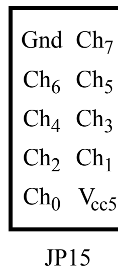


Figure 46. ADC connector.

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.

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	0x3FFFFFFF	DDR3 Memory
0xFFFF0000	0xFFFFFFFF	A9 On-chip Memory
0xC0000000	0xC3FFFFFF	SDRAM
0xC8000000	0xC803FFFF	FPGA On-chip Memory
0xC9000000	0xC9001FFF	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
0xFF201008	0xFF20100F	Second JTAG UART
0xFF201020	0xFF201027	Infrared (IrDA)
0xFF202000	0xFF20201F	Interval Timer
0xFF202020	0xFF20202F	Second Interval Timer
0xFF203000	0xFF20301F	Audio/video Configuration
0xFF203020	0xFF20302F	Pixel Buffer Control
0xFF203030	0xFF203037	Character Buffer Control
0xFF203040	0xFF20304F	Audio
0xFF203060	0xFF203070	Video-in
0xFF204000	0xFF20401F	ADC
0xFF709000	0xFF709063	HPS GPIO1
0xFFC04000	0xFFC040FC	HPS I2C0
0xFFC08000	0xFFC08013	HPS Timer0
0xFFC09000	0xFFC09013	HPS Timer1
0xFFD00000	0xFFD00013	HPS Timer2
0xFFD01000	0xFFD01013	HPS Timer3
0xFFD0501C	0xFFD0501F	FPGA Bridge
0xFFFEC100	0xFFFEC1FC	GIC CPU Interface
0xFFFED000	0xFFFEDFFC	GIC Distributor Interface
0xFFFEC600	0xFFFEC60F	ARM A9 Private Timer

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 47 to 50 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 47, 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 48. 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 downloaded by the Monitor Program into the DE1-SoC board.

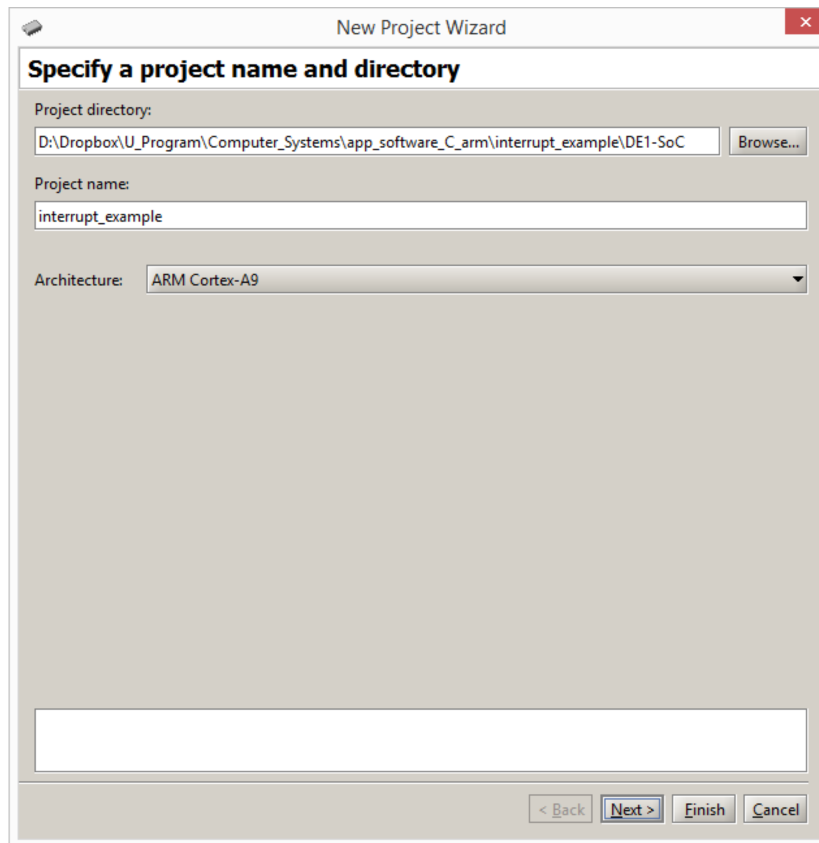


Figure 47. Specifying the project folder and project name.

Pressing Next again opens the window in Figure 49. 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 50 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 Exceptions. 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.

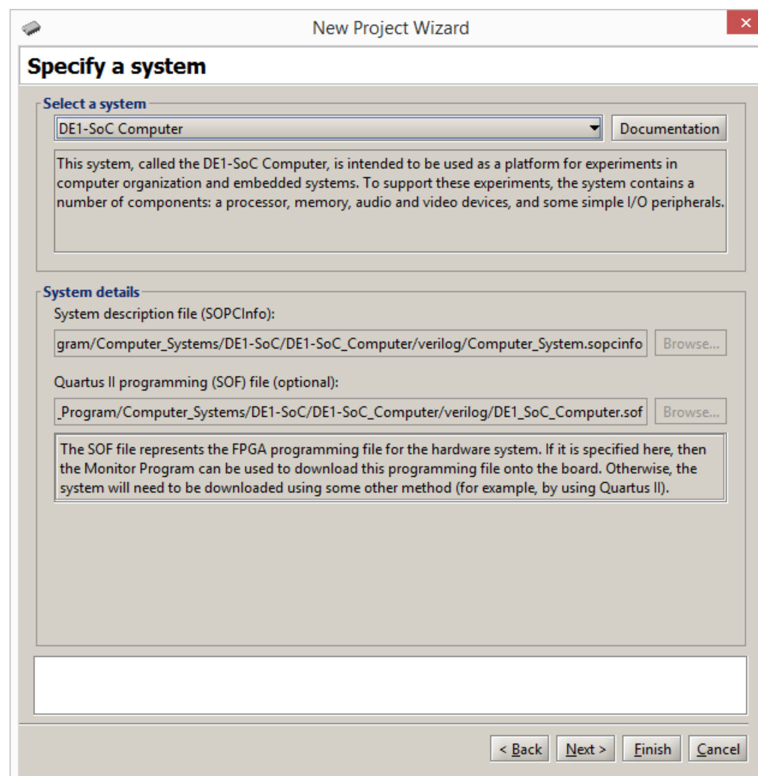


Figure 48. Specifying the DE1-SoC Computer system.

Copyright ©2015 Altera Corporation.

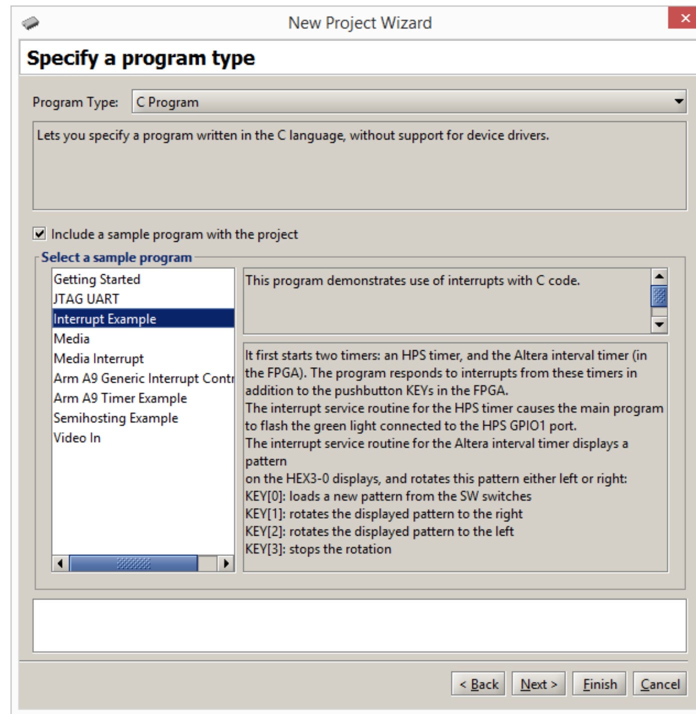


Figure 49. Selecting sample programs.

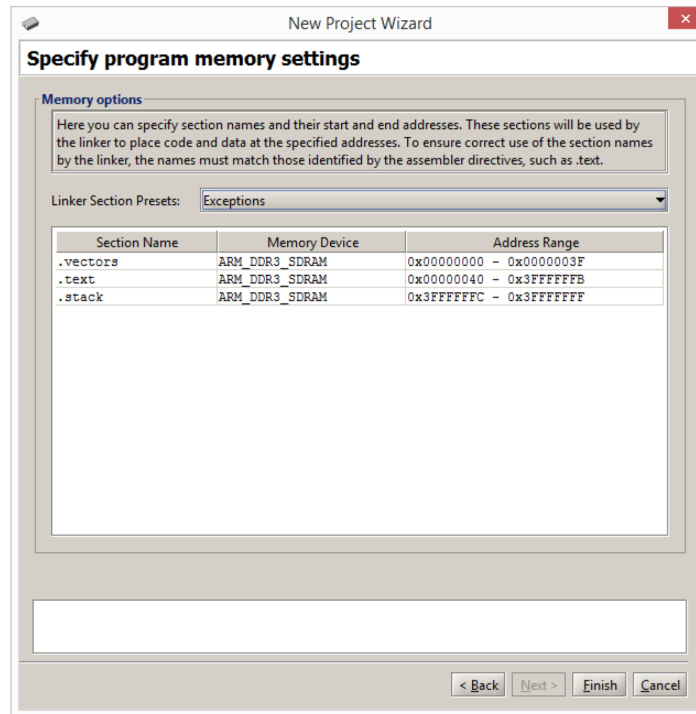


Figure 50. Setting offsets for *.text* and *.data*.