# Using HAL Device Drivers with the Altera Monitor Program

## 1 Introduction

This tutorial shows how to develop C programs that use device driver functions for the I/O devices in a Nios II hardware system. The device driver functions used in the tutorial are provided as part of the Altera University Program IP cores, which are available from the University Program section of Altera's website. These functions are implemented using Altera's Hardware Abstraction Layer (HAL). In addition to providing support for device drivers, the HAL simplifies many programming tasks, such as the development of programs that use interrupts. A detailed description of the features provided by the HAL can be found in the *Nios II Software Developer's Handbook*, which is available from Altera.

The Nios II software programs shown in the tutorial are implemented by using the Altera Monitor Program development environment. This tutorial includes screen captures obtained using version 4.2 of the Altera Monitor Program; if other versions of the software are used, some of the images may be slightly different. The device driver functions used in the example programs in the tutorial are from version 9.1 of the University Program IP Cores.

We assume that the reader has access to the Altera DE2 Development and Education board (including DE2-70 and DE2-115), or the DE1 board. If other boards are used, then the design examples for the tutorial may be not usable, as they require the presence of specific I/O devices.

**Contents**:

- Examining the HAL device driver functions that are provided for University Program IP cores
- Writing C programs that use HAL device driver functions
- Compiling HAL code with the Monitor Program
- Running and debugging HAL code using the Monitor Program
- Finding HAL device driver source code
- Using Nios II interrupts in HAL code

## 2 Writing Programs that use HAL Device Drivers

For this tutorial, we assume that the reader is already familiar with the Altera Monitor Program. It is described in the tutorial *Introduction to the Altera Monitor Program*, which is available from the University Program section of Altera's website.

To see an example of using HAL device drivers, create a new Monitor Program project for the DE2 board called *HAL_tutorial*. Store the project in a directory of your choice. When creating the project, specify the hardware system to be the prebuilt *DE2 Media Computer*, specify the program type as Program with Device Driver Support, and select the sample program named *getting_started_HAL*. This sample program makes use of two types of I/O devices in the DE2 Media Computer: parallel ports, and an audio device.

The HAL device drivers for I/O devices consist of collections of functions that allow software programs to access hardware devices. To use these functions, it is first necessary to examine the documentation provided for the IP core that connects to each I/O device, to determine the names of its device driver functions, the number and types of arguments, and the specified use of the functions. The documentation for the IP cores that are included in the DE2 Media Computer can be found on the University Program section of Altera's website under the heading Educational Materials > Computer Organization > IP cores. As an example, the documentation file for the parallel ports provides a section called *Programming with the Parallel Ports*. A small part of this section is displayed in Figure 1. A number of device driver functions are listed in the figure, including *alt_up_parallel_port_open_dev(…)*, which is used in C code to open a parallel port device, as well as *alt_up_parallel_port_read_data(…)* and *alt_up_parallel_port_write_data(…)*, which can be used to read/write data from/to a parallel port. For each function, the documentation specifies the data types of arguments and return values.

A complete example of C code that uses HAL device drivers is given in Figure 3. This code uses two parallel ports, connected to switches and LEDs on the DE2 board, and an audio port. The code performs the following operations:

1. When the switch $KEY_1$ is pressed on the DE2 board, audio input from the microphone jack is recorded for about 10 seconds. $LEDG_0$ is illuminated while recording.
2. When the switch $KEY_2$ is pressed, the recorded audio is played back on the line-out jack. $LEDG_1$ is illuminated during playback.

As shown in Figure 3(*a*), the C code first includes the necessary header files for the parallel port and audio devices. In lines 7 – 9 a pointer is declared for each of the three I/O devices to be used in the code. The pointers have a special type, which is the type of return value for the device driver function that opens the device–for example, in Figure 1 the data type of the function *alt_up_parallel_port_open_dev(…)* is *alt_up_parallel_port_dev*. Lines 10 – 12 of the code open the two parallel ports and the audio device needed in the program. As shown, each I/O device is referenced using a unique name. The names of the two parallel port devices are *Pushbuttons* and *Green_LEDs*, and the audio device is named *Audio*. These names are prefixed with the string */dev*, and then passed to the device driver function. The unique name of each device is assigned by the designer when the Nios II hardware system is created by using Altera's SOPC Builder tool. Figure 2 shows part of the *Systems Contents* that can be displayed in the SOPC Builder tool for the DE2 Media Computer, with the module names (without the */dev/ prefix*) displayed in the column labeled *Module Name*.

### 4.2 Programming with the Parallel Ports

The Parallel Port core is packaged with C-language device drivers accessible through the hardware abstraction layer (HAL). These functions implement basic operations for the Parallel Port.

To use the functions, the C code must include the statement:

```
#include "altera_up_avalon_parallel_port.h"
```

#### 4.2.1 alt_up_parallel_port_open_dev

**Prototype:** `alt_up_parallel_port_dev* alt_up_parallel_port_open_dev(const char *name)`
**Include:** `<altera_up_avalon_parallel_port.h>`
**Parameters:** `name` – the parallel port name. For example, if the parallel port name in SOPC Builder is "green_leds", then *name* should be "/dev/green_leds"
**Returns:** The corresponding device structure, or NULL if the device is not found.
**Description:** Open the parallel port device specified by *name* .

#### 4.2.2 alt_up_parallel_port_read_data

**Prototype:** `unsigned int alt_up_parallel_port_read_data(alt_up_parallel_port_dev *parallel_port)`
**Include:** `<altera_up_avalon_parallel_port.h>`
**Parameters:** `parallel_port` – struct for the parallel port device .
**Returns:** data – The data read for the parallel port.
**Description:** Read from the data register of the parallel port.

#### 4.2.3 alt_up_parallel_port_write_data

**Prototype:** `void alt_up_parallel_port_write_data(alt_up_parallel_port_dev *parallel_port, unsigned data)`

Figure 1. A part of the documentation file for the parallel port.

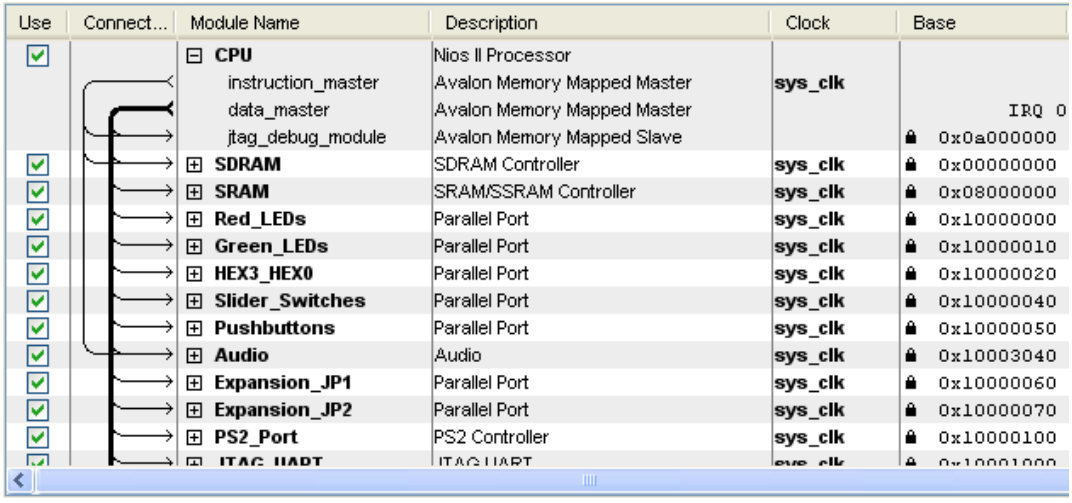


Figure 2. Module names for the DE2 Media Computer shown in the SOPC Builder.

```
1   #include "altera_up_avalon_parallel_port.h"
2   #include "altera_up_avalon_audio.h"
    /* globals */
3   #define BUF_SIZE 500000          // about 10 seconds of buffer (@ 48K samples/sec)
4   #define BUF_THRESHOLD 96         // 75% of 128 word buffer
    /* function prototypes */
5   void check_KEYs( int *, int *, int *, alt_up_parallel_port_dev *, alt_up_audio_dev * );
    /************************************************************************************
     * This program demonstrates use of the media ports in the DE2 Media Computer
     *
     * It performs the following:
     *      1. records audio for about 10 seconds when KEY[1] is pressed. LEDG[0] is
     *         lit while recording
     *      2. plays the recorded audio when KEY[2] is pressed. LEDG[1] is lit while
     *         playing

    ************************************************************************************/
6   int main(void)
    {
        /* declare variables to point to devices that are opened */
7       alt_up_parallel_port_dev *KEY_dev;
8       alt_up_parallel_port_dev *green_LEDs_dev;
9       alt_up_audio_dev *audio_dev;
        // open the pushbutton KEY parallel port
10      KEY_dev = alt_up_parallel_port_open_dev ("/dev/Pushbuttons");
        // open the green LEDs parallel port
11      green_LEDs_dev = alt_up_parallel_port_open_dev ("/dev/Green_LEDs");
        // open the audio port
12      audio_dev = alt_up_audio_open_dev ("/dev/Audio");
        /* used for audio record/playback */
13      int record = 0, play = 0, buffer_index = 0;
14      unsigned int l_buf[BUF_SIZE];
15      unsigned int r_buf[BUF_SIZE];
16      int num_read; int num_written;
```

Figure 3. An example of C code that uses HAL device drivers (Part *a*).

```
        /* read and echo audio data */
17      record = 0;
18      play = 0;
19      while(1)
        {
20          check_KEYs (&record, &play, &buffer_index, KEY_dev, audio_dev);
21          if (record)
            {
22              alt_up_parallel_port_write_data (green_LEDs_dev, 0x1);   // set LEDG[0] on
                // record data until the buffer is full
23              if (buffer_index < BUF_SIZE)
                {
24                  num_read = alt_up_audio_record_r (audio_dev, &(r_buf[buffer_index]),
                        BUF_SIZE − buffer_index);
                    /* assume we can read same # words from the left and right */
25                  (void) alt_up_audio_record_l (audio_dev, &(l_buf[buffer_index]), num_read);
26                  buffer_index += num_read;
27                  if (buffer_index == BUF_SIZE)
                    {
                        // done recording
28                      record = 0;
29                      alt_up_parallel_port_write_data (green_LEDs_dev, 0x0);   // turn off LEDG
                    }
                }
            }
```

Figure 3. An example of C code that uses HAL device drivers (Part *b*).

The rest of the code in Figure 3 performs the recording and playback of audio, and controls the pushbutton and green lights parallel ports. Device driver functions are called in the main program in lines 22, 24 – 25, 29, 31, 33 – 34, and 38, and in the function *check_KEYs*, which examines the values of the pushbutton switches, in lines 41 – 42, 45, and 49. The operation of each of these functions is described in the documentation file for the corresponding IP core, as discussed previously for Figure 1.

## 3 Compiling Programs that use Device Drivers

The *HAL_tutorial* project can be compiled in the normal way by using the Monitor Program commands Actions > Compile, or Actions > Compile & Load. The first time this is done, the Monitor Program compiles not only the file *getting_started_HAL.c*, but also a number of C library functions that are a part of the HAL system, and all of the device driver functions that are provided for every device in the Nios II hardware system. Although this process is somewhat time consuming, it is only done once, and subsequent compilations only compile the source file *getting_started_HAL.c*. If it is necessary to recompile all of the HAL functions at a later time, this can be accomplished by using the command Actions > Regenerate Device Drivers (BSP). This action might be necessary, for example, if a new version of IP cores is installed at a later time.

```
30          else if (play)
            {
31              alt_up_parallel_port_write_data(green_LEDs_dev, 0x2);   // set LEDG[1] on
                // output data until the buffer is empty
32              if (buffer_index < BUF_SIZE)
                {
33                  num_written = alt_up_audio_play_r (audio_dev, &(r_buf[buffer_index]),
                        BUF_SIZE − buffer_index);
                    /* assume that we can write the same # words to the left and right */
34                  (void) alt_up_audio_play_l (audio_dev, &(l_buf[buffer_index]), num_written);
35                  buffer_index += num_written;

36                  if (buffer_index == BUF_SIZE)   // done playback
                    {
37                      play = 0;
38                      alt_up_parallel_port_write_data(green_LEDs_dev, 0x0);   // turn off LEDG
                    }
                }
            }
        }
    }
```

Figure 3. An example of C code that uses HAL device drivers (Part *c*).

### 3.1 Running the Program

Programs that use HAL device drivers can be run and debugged in the Monitor Program in the same way as other C or assembly language programs. Figure 4 shows an example of a breakpoint set in the code of Figure 3. The figure shows the value read from the pushbutton parallel port by the device driver function *alt_up_parallel_port_read_data*(…). This particular device driver is executed without using a *call* assembly language instruction, which means that it is implemented as a macro, rather than a subroutine. The value returned by the function is shown in the Nios II register *r5*. The value is `0x00000002`, which means that $KEY_1$ on the DE2 board was pressed when the function was called. Other device drivers are implemented as subroutines, such as *alt_up_audio_read_fifo_avail*(…), as indicated in Figure 5. In the figure, a breakpoint has been set at this function call. The subroutine can be executed in a debugging session either by using Actions > Step Over Subroutine or by single-stepping into the device driver code if desired.

## 4 Examining the HAL Device Driver Source Code

It is possible to examine the source code of the device driver functions used in the HAL. These functions are installed in the same filesystem directory as the Quartus II software, as illustrated on the left side of Figure 6. In this figure, `QUARTUS_ROOTDIR` [1] represents the installation directory of the Quartus II software. The device driver code is

[1] In Windows operating systems, the environment variable `QUARTUS_ROOTDIR` points to the folder where Quartus II software is installed.

```
/*****************************************************************************
 * Subroutine to read KEYs
*****************************************************************************/
void check_KEYs(int *KEY1, int *KEY2, int *counter, alt_up_parallel_port_dev *KEY_dev,
                alt_up_audio_dev *audio_dev)
{
    int KEY_value;
    KEY_value = alt_up_parallel_port_read_data (KEY_dev);     // read the pushbutton KEY values
    while (alt_up_parallel_port_read_data (KEY_dev));         // wait for pushbutton KEY release
    if (KEY_value == 0x2)                                     // check KEY1
    {
        *counter = 0;                                         // reset counter to start recording
        alt_up_audio_reset_audio_core (audio_dev);            // reset audio port
        *KEY1 = 1;
    }
    else if (KEY_value == 0x4)                                // check KEY2
    {
        *counter = 0;                                         // reset counter to start playback
        alt_up_audio_reset_audio_core (audio_dev);            // reset audio port
        *KEY2 = 1;
    }
}
```

Figure 3. An example of C code that uses HAL device drivers (Part *d*).

stored in two directories: the *inc* directory contains device driver subroutine prototypes and device driver macros, and the *src* directory contains subroutine definitions. During the process of compiling a project in the Monitor Program, the *inc* and *src* directories for all IP cores are copied into the project's directory. As shown on the right side of Figure 6, the directories are copied into a folder called *drivers*, which is a subfolder of the folder called *BSP* (the acronym stands for *Board Support Package*). The figure shows the directory structure of the parallel port and audio IP cores as an example, but the source files for all of the IP cores in the hardware system are copied in this way.

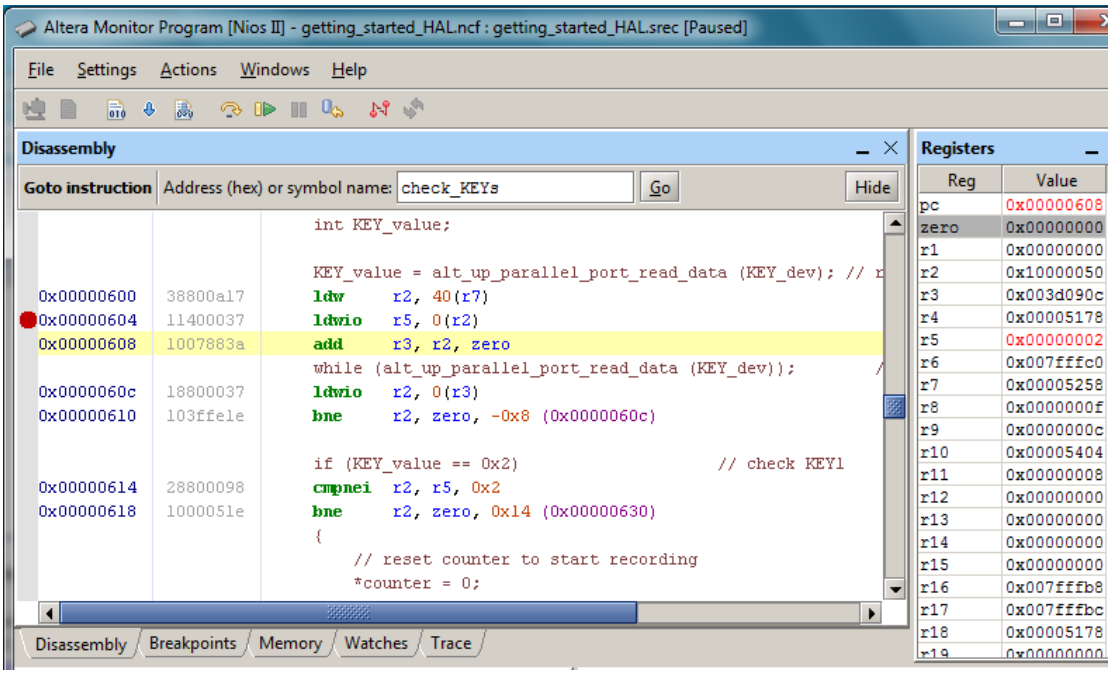


Figure 4. An example of a HAL device driver function that is implemented as a macro.

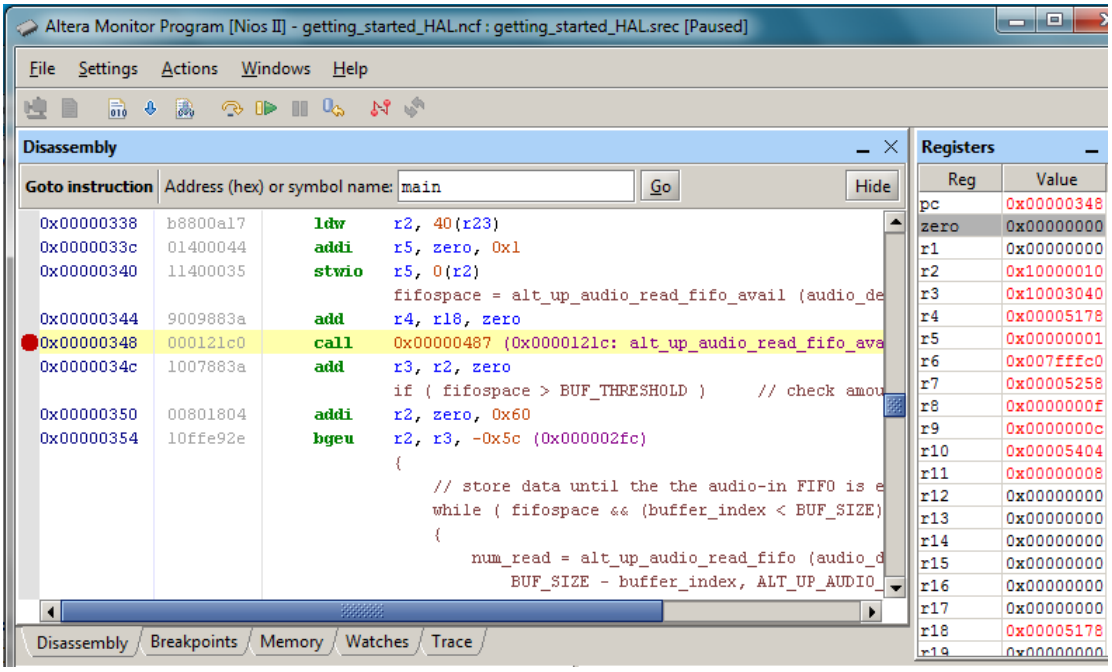


Figure 5. An example of a HAL device driver function implemented as a subroutine.

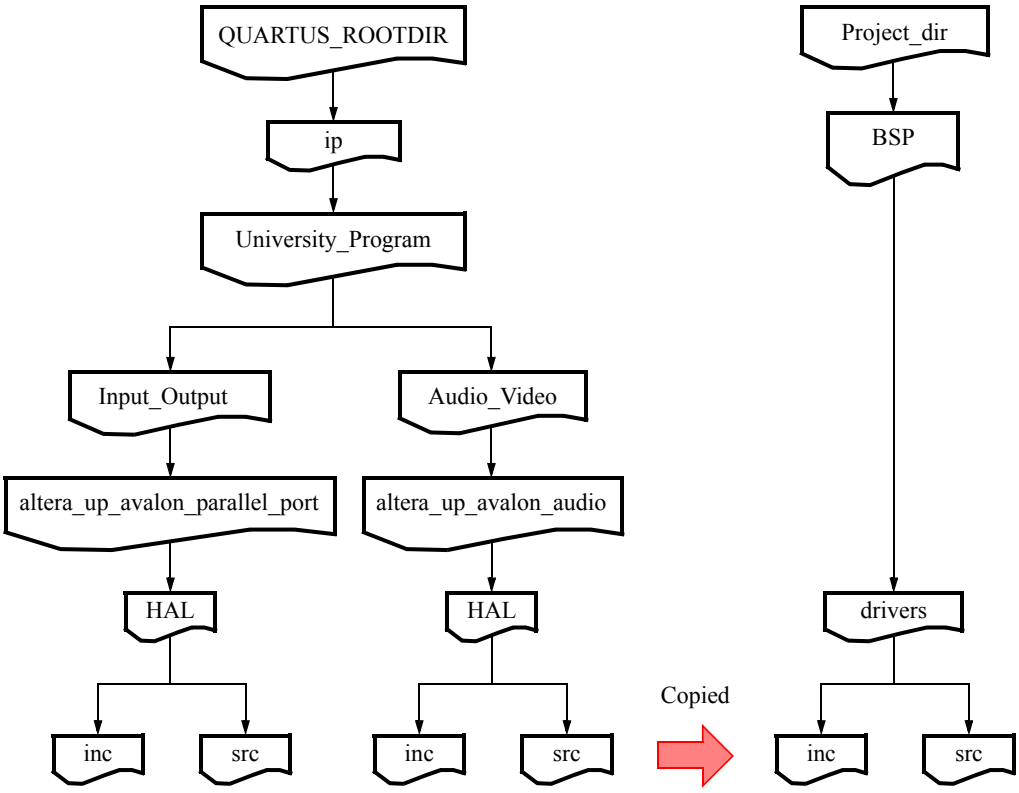


Figure 6. Finding the HAL device driver source code files.

# 5 Using Nios II Interrupts in the HAL

The HAL provides a simple interface for using Nios II interrupts in C programs. To use interrupts programs must specify the statement **#include** "sys/alt_irq.h". This include file provides a function named *alt_irq_register (…)*, which is used to specify the hardware interrupt levels and associated interrupt service routines for any Nios II interrupts that are being used. To see an example that uses interrupts, create a new Monitor Program project for the DE2 board called *HAL_tutorial_int*. Store the project in a directory of your choice. When creating the project, specify the hardware system to be the *DE2 Media Computer*, specify the program type as Program with Device Driver Support, and select the sample program provided in the Monitor Program named *interrupts_HAL*. This sample program has the exact same functionality as the code in Figure 3, except that the pushbutton *KEY* parallel port and audio device are handled using interrupts. There are three source files in this program: *interrupts_HAL.c* contains the main program, and the interrupt service routines are found in the files *pushbutton_ISR.c* and *audio_ISR.c*.

Figure 7 shows the C code for the main program. It first includes the file *globals.h*, which, as illustrated in Figure 8, includes the parallel port and audio HAL header files *altera_up_avalon_parallel_port.h* and *altera_up_avalon_audio.h*. The *globals.h* file also includes *sys/alt_stdio.h* and *alt_irq.h*. As mentioned above, *alt_irq.h* is needed to use interrupts with the HAL. The file *alt_stdio.h* defines some simplified versions of the functions in the standard C library *stdio.h*. The purpose of *alt_stdio.h* is to conserve memory space in the hardware system by providing functions that have limited capability but also produce less machine code when compiled. In this example, we will use a function called *alt_printf*, which is a simplified version of *printf*. The use of such functions is not necessary for the example program, and is provided only for illustrative purposes. The *alt_stdio.h* library, and other C libraries provided by Altera, is described in the *Nios II Software Developer's Handbook*, available from Altera's website.

The last few lines of code in Figure 8 declare a C structure named *alt_up_dev*. As indicated in the code, this structure is used to hold a pointer to the I/O devices for the two parallel ports and audio port. We will explain the purpose of this structure shortly. Referring back to the main program in Figure 7, line 2 defines an instance, named *up_dev*, of the *alt_up_dev* structure. Line 3 defines two variables used with the audio device. These variables have to be declared using the keyword *volatile*, because their values are written by interrupt service routines. If this keyword is not used, then the C compiler may choose to save the value of the variable in a Nios II general-purpose register and to retrieve the value of this variable, when needed in a program, from this register, rather than from memory. In this case, changes to the variable's value that are written into memory by an interrupt service routine would not be seen in the main program. Using the *volatile* keyword prohibits the C compiler from saving the value of the variable in a CPU register, and causes the Nios II processor to access the variable using *load I/O* and *store I/O* instructions [2].

[2] Even if a version of the Nios II processor that has a *data cache* is being used, the load I/O instruction causes the processor to bypass this cache and access the associated variable at its address in memory.

```
1   #include "globals.h"
    /* global variables */
2   struct alt_up_dev up_dev;                          // holds a pointer to each open I/O device
    /* The globals below are written by interrupt service routines, so we have to declare
     * them as volatile to avoid the compiler caching their values in registers */
3   volatile int buf_index_record, buf_index_play;     // audio variables
    /* function prototypes */
4   void pushbutton_ISR(void *, unsigned int);
5   void audio_ISR(void *, unsigned int);
    /*****************************************************************************
     * This program demonstrates use of HAL functions and interrupts
     *
     * It performs the following:
     *      1. records audio for about 10 seconds when an interrupt is generated by
     *         pressing KEY[1]. LEDG[0] is lit while recording. Audio recording is
     *         controlled by using interrupts
     *      2. plays the recorded audio when an interrupt is generated by pressing
     *         KEY[2]. LEDG[1] is lit while playing. Audio playback is controlled by
     *         using interrupts

    *****************************************************************************/
6   int main(void)
    {
        /* declare device driver pointers for devices */
7       alt_up_parallel_port_dev *KEY_dev;
8       alt_up_parallel_port_dev *green_LEDs_dev;
9       alt_up_audio_dev *audio_dev;
        // open the pushbutton KEY parallel port
10      KEY_dev = alt_up_parallel_port_open_dev ("/dev/Pushbuttons");
11      if (KEY_dev == NULL)
        {
12          alt_printf ("Error: could not open pushbutton KEY device\n");
13          return -1;
        }
14      else
        {
15          alt_printf ("Opened pushbutton KEY device\n");
16          up_dev.KEY_dev = KEY_dev;       // store for use by ISRs
        }
```

Figure 7. An example of C code that uses interrupts (Part *a*).

```
        /* write to the pushbutton interrupt mask register, and set 3 mask bits to 1
             * (we can't use pushbutton 0; it is the Nios II reset button) */
17      alt_up_parallel_port_set_interrupt_mask (KEY_dev, 0xE);
        // open the green LEDs parallel port
18      green_LEDs_dev = alt_up_parallel_port_open_dev ("/dev/Green_LEDs");
19      if (green_LEDs_dev == NULL)
        {
20          alt_printf ("Error: could not open green LEDs device\n");
21          return -1;
        }
22      else
        {
23          alt_printf ("Opened green LEDs device\n");
24          up_dev.green_LEDs_dev = green_LEDs_dev; // store for use by ISRs
        }
        // open the audio port
25      audio_dev = alt_up_audio_open_dev ("/dev/Audio");
26      if (audio_dev == NULL)
        {
27          alt_printf ("Error: could not open audio device\n");
28          return -1;
        }
29      else
        {
30          alt_printf ("Opened audio device\n");
31          up_dev.audio_dev = audio_dev; // store for use by ISRs
        }
        /* use the HAL facility for registering interrupt service routines. */
        /* Note: we are passing a pointer to up_dev to each ISR (using the HAL context argument) as
         * a way of giving the ISR a pointer to every open device. This is useful because some of the
         * ISRs need to access more than just one device (e.g. the pushbutton ISR accesses both
         * the pushbutton device and the audio device) */
32      alt_irq_register (1, (void *) &up_dev, (void *) pushbutton_ISR);
33      alt_irq_register (6, (void *) &up_dev, (void *) audio_ISR);
        /* the main program can now exit; further program actions are handled by interrupts */
    }
```

Figure 7. An example of C code that uses interrupts (Part *b*).

Lines 7−31 of Figure 7 open the three I/O devices needed in the program, and use *alt_printf* to display an appropriate error if a device cannot be properly opened. Although this check is not strictly needed in our example, it is a good practice to check the return value of functions for any errors that may occur. The pushbutton KEYs parallel port is configured to generate hardware interrupts by using the HAL function shown in line 17 of the code. Lines 16, 24, and 31 store the pointer to each opened I/O device in the *up_dev* structure. This structure is used in lines 32 and 33, which call the *alt_irq_register* function. The first argument to this function is the level of the hardware interrupt

being used. In the DE2 Media Computer hardware system, the pushbutton KEYs parallel port is assigned interrupt level 1 and the audio port has interrupt level 6. The second argument of *alt_irq_register* is a pointer of type *void* * called the *context* pointer. This pointer is simply passed on to the associated interrupt service routine (ISR) when the interrupt occurs; it can point to any type of object and can be used for any purpose needed in the ISR. In this case, we pass a pointer to the *up_dev* structure to the ISR for both the pushbutton parallel port and audio port. Since *up_dev* holds a pointer to all of the open I/O devices, it allows an ISR to execute device driver functions for any of the devices. The last argument to *alt_irq_register* is a pointer to the associated ISR. Each ISR has to have two arguments, as illustrated in lines 4 and 5 of Figure 7. The first of these argument is used for the context pointer, and the second argument gives the interrupt level.

The interrupt service routine for the pushbutton parallel port is given in Figure 9. As shown in the code, it uses the *up_dev* context pointer to access both the pushbutton parallel port and audio port. Line 8 uses a device driver function to determine which pushbutton KEY caused the interrupt, and line 9 clears this interrupt. If $KEY_1$ is pressed, then read interrupts are enabled for the audio device to begin recording, and if $KEY_2$ is pressed, then write interrupts are enabled for the audio device to perform playback.

Figure 10 shows the ISR for the audio device. It is very similar to the code in Figure 3, except that line 8 checks for audio read interrupts, and line 16 disables audio read interrupts when the recording buffer is full. Also, line 17 checks for audio write interrupts, and line 25 disables these interrupts when the playback buffer is empty.

```
/* include HAL device driver functions for the parallel port and audio device */
#include "altera_up_avalon_parallel_port.h"
#include "altera_up_avalon_audio.h"

#include "sys/alt_stdio.h"
#include "sys/alt_irq.h"

/* This structure holds a pointer to each open I/O device */
struct alt_up_dev {
    alt_up_parallel_port_dev *KEY_dev;
    alt_up_parallel_port_dev *green_LEDs_dev;
    alt_up_audio_dev *audio_dev;
};
```

Figure 8. Include files and a structure to hold pointers.

```
1   #include "globals.h"
    /* indices for audio record and playback; we reset them when pushbuttons are pressed */
2   extern volatile int buf_index_record;
3   extern volatile int buf_index_play;
    /*****************************************************************************
     * Pushbutton - Interrupt Service Routine
     *
     * This ISR checks which KEY has been pressed. If KEY1, then it enables audio-in
     * interrupts (recording). If KEY2, it enables audio-out interrupts (playback).
     *****************************************************************************/
4   void pushbutton_ISR(struct alt_up_dev *up_dev, unsigned int id)
    {
5       alt_up_audio_dev *audio_dev;
6       audio_dev = up_dev->audio_dev;
7       int KEY_value;
        /* read the pushbutton interrupt register */
8       KEY_value = alt_up_parallel_port_read_edge_capture (up_dev->KEY_dev);
9       alt_up_parallel_port_clear_edge_capture (up_dev->KEY_dev); // clear the interrupt
10      if (KEY_value == 0x2)                   // check KEY1
        {
            // reset the buffer index for recording
11          buf_index_record = 0;
            // clear audio FIFOs
12          alt_up_audio_reset_audio_core (audio_dev);
            // enable audio-in interrupts
13          alt_up_audio_enable_read_interrupt (audio_dev);
        }
14      else if (KEY_value == 0x4)              // check KEY2
        {
            // reset counter to start playback
15          buf_index_play = 0;
            // clear audio FIFOs
16          alt_up_audio_reset_audio_core (audio_dev);
            // enable audio-out interrupts
17          alt_up_audio_enable_write_interrupt (audio_dev);
        }
18      return;
    }
```

Figure 9. The pushbutton *KEY* interrupt service routine.

```
1   #include "globals.h"
2   #define BUF_SIZE 500000                 // about 10 seconds of audio buffer (@ 48K samples/sec)
    /* globals used for audio record/playback */
3   extern volatile int buf_index_record, buf_index_play;
4   unsigned int l_buf[BUF_SIZE];           // audio buffer
5   unsigned int r_buf[BUF_SIZE];           // audio buffer
    /*****************************************************************************
     * Audio - Interrupt Service Routine
     * This interrupt service routine records or plays back audio, depending on which type
     * interrupt (read or write) is pending in the audio device.
     *****************************************************************************/
6   void audio_ISR(struct alt_up_dev *up_dev, unsigned int id)
    {
7       int num_read, num_written;
8       if (alt_up_audio_read_interrupt_pending(up_dev->audio_dev))          // check for read interrupt
        {
9           alt_up_parallel_port_write_data (up_dev->green_LEDs_dev, 0x1);   // set LEDG[0] on
            // store data until the buffer is full
10          if (buf_index_record < BUF_SIZE)
            {
11              num_read = alt_up_audio_record_r (up_dev->audio_dev, &(r_buf[buf_index_record]),
                    BUF_SIZE − buf_index_record);
                /* assume we can read same # words from the left and right */
12              (void) alt_up_audio_record_l (up_dev->audio_dev, &(l_buf[buf_index_record]), num_read);
13              buf_index_record += num_read;
14              if (buf_index_record == BUF_SIZE)                            // done recording
                {
15                  alt_up_parallel_port_write_data (up_dev->green_LEDs_dev, 0);  // turn off LEDG
16                  alt_up_audio_disable_read_interrupt(up_dev->audio_dev);
                }
            }
        }
```

Figure 10. The audio device interrupt service routine (Part *a*).

```
17      if (alt_up_audio_write_interrupt_pending(up_dev->audio_dev))          // check for write interrupt
        {
18          alt_up_parallel_port_write_data (up_dev->green_LEDs_dev, 0x2);    // set LEDG[1] on
            // output data until the buffer is empty
19          if (buf_index_play < BUF_SIZE)
            {
20              num_written = alt_up_audio_play_r (up_dev->audio_dev, &(r_buf[buf_index_play]),
                    BUF_SIZE − buf_index_play);
                /* assume that we can write the same # words to the left and right */
21              (void) alt_up_audio_play_l (up_dev->audio_dev, &(l_buf[buf_index_play]), num_written);
22              buf_index_play += num_written;
23              if (buf_index_play == BUF_SIZE)                               // done playback
                {
24                  alt_up_parallel_port_write_data (up_dev->green_LEDs_dev, 0);  // turn off LEDG
25                  alt_up_audio_disable_write_interrupt(up_dev->audio_dev);
                }
            }
        }
26      return;
    }
```

Figure 10. The audio device interrupt service routine (Part *b*).

## 6 Final Remarks

In this tutorial we have introduced the use of HAL device drivers with C programs. We have shown that C code using device driver functions can be developed by following the steps below:

1. Obtain a Nios II hardware system for which the IP cores in the system include HAL device drivers. In this tutorial we used the DE2 Media Computer hardware system. It uses the University Program IP cores that are available from the University Program section of Altera's website.
2. Examine the documentation provided for the IP cores in the hardware system. This documentation includes a section that describes the available HAL device driver functions for the IP core.
3. Create an Altera Monitor Program project for the hardware system, with the program type set to Program with Device Driver Support.
4. Based on the functionality needed, write a program that calls the necessary HAL device driver functions. The include files that define the device driver functions must be included in the program. If interrupts are needed, then an interrupt service for each interrupt can be registered by using the *alt_irq_register* library function. Other mechanism for dealing with interrupts are also available, and are described in the document *Nios II Software Developer's Handbook*, which is provided by Altera.
5. Compile the program in the normal way by using the provided commands in the Monitor Program. Debug the program by setting breakpoints, single-stepping, examining memory, and so on, using the provided features of the Monitor Program.