DELL EMC

CONTROL THE GPIO PINS ON THE DELL EMBEDDED BOX PC 3000 IN UBUNTU LINUX

ABSTRACT

The Dell Embedded Box PC 3000 provides General Purpose I/O (GPIO) pins for customization. Due to the unavailability of the corresponding kernel driver, users who run the Linux system on the Box PC face a problem in controlling the GPIO pins. This white paper describes how to control the GPIO pins from a user-space program through a device file. The audience of this document should have general programming skills, knowledge of the GRUB boot loader and the Ubuntu operating system.

March, 2017

TABLE OF CONTENTS

GPIO ON DELL EMBEDDED BOX PC 3000

The Dell Embedded Box PC 3000 provides twelve General Purpose I/O (GPIO) pins, where six of the pins are designed for output use and the other six are designed for input use.

The GPIO pins are routed to the multi-function connector (standard DB44P male connector) on the back of the Embedded Box PC 3000. You can further extend the pins to the CN2 connector (standard DB25 female connector) with a multi-function cable.
See Figure 1 for examples of the connectors .

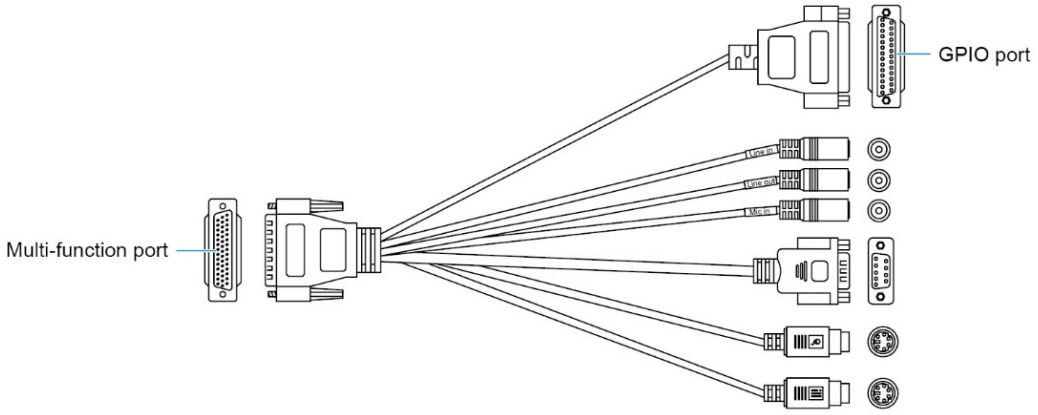

Figure 1

The GPIO pins are organized according to:

- Output pins: DO0, DO1,… and DO5
- Input pins: DI0, DI1,… and DI5.

See Figure 2 for an illustration of the GPIO pins on the CN2 connector.

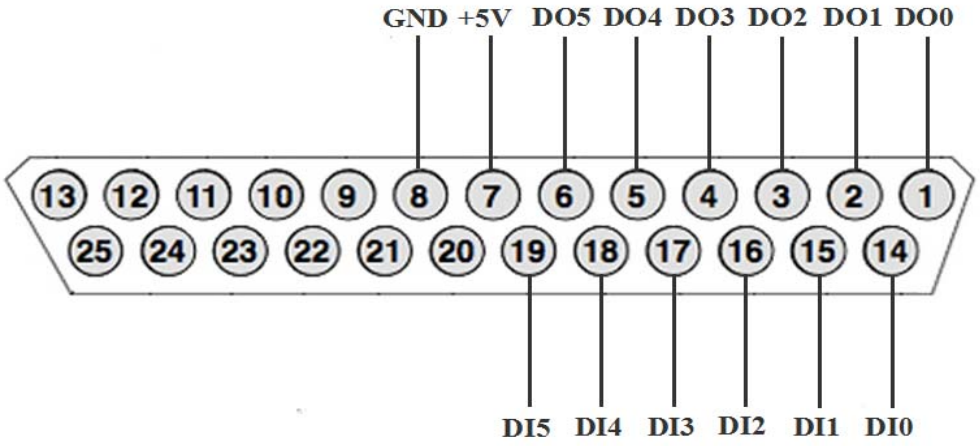

Figure 2

The GPIO pins are implemented by the on-board GPIO expander controller chip, the TI TCA9555. It interfaces with the host processor through the two-line bidirectional I2C bus. This means it behaves as an I2C slave and has to be attached to the same bus with an I2C master.

The I2C device (i.e. slave) is typically controlled by a kernel driver, but the TI TCA9555 does not have corresponding kernel drivers.

If the Linux I2C subsystem supports the I2C adapter (i.e. master) in kernel, it will support the alternative method to interface with the device on the I2C bus from user space through the device file.

The TI TCA9555 uses the Intel SMBUS controller, the I2C-compatible adapter on “PCI bus 0, Dev 31, Fun 3”. This SMBUS controller is supported by the kernel driver, I2c-I801. In this document, the methods to access the controller will be illustrated by several source code examples.

USE THE i2c-i801 DRIVER

Before you start, it is important to ensure that the driver of the Intel SMBUS controller has loaded without errors. Occasionally, when the i2c-i801 driver is claiming resources during the OS boot sequence, driver initialization can fail.

Check that the driver is loaded successfully.

- If the driver has not started, load the driver by typing “`sudo modprobe i2c-i801`” and use “`dmesg`” to ‘dump’ the kernel message to the console screen.
- If the driver has started, ensure it is running with the correct parameters.
- If the setup is not right, then stop and start the driver manually.

If conflict occurs, the error message in Figure 3 is displayed.

```
ACPI Warning: SystemIO range 0x000000000000F040-0x000000000000F05F conflicts with
OpRegion 0x000000000000F040-0x000000000000F04F (\_SB.PCI0.SBUS.SMBI)
(20160108/utaddress-255)
ACPI: If an ACPI driver is available for this device, you should use it instead of the
native driver
```

Figure 3

If the above failure occurs, disable the conflict check. After the conflict check is disabled, the i2c-i801 driver will load successfully.

To activate the option,

- Add the argument “`acpi_enforce_resources=lax`” to the kernel command line, which is configured by GRUB, the boot loader of Ubuntu Linux.

To inspect the command line,

1. Type the command “`cat /proc/cmdline`”. If the argument does not appear in the command line, edit the default configuration file of GRUB with the command “`sudo gedit /etc/default/grub`”.
2. Add the argument to the recommended location in the file shown in Figure 4.

```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
```

```
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash acpi_enforce_resources=lax"
GRUB_CMDLINE_LINUX=""

# ...
```

Figure 4

3. After the file is edited and saved, run the command "`sudo update-grub`" to update the GRUB configuration on the system.
4. Reboot to make the new kernel command effective.
5. After reboot, check that the error has been eliminated.

Note: Canonical's Ubuntu release for the Embedded Box PCs has enabled the option in kernel by default. The user should not have any problem in loading the i2c-i801 driver on the release. (This is not applicable on the general public release of the Ubuntu operating system!).

OPEN THE INTERFACE OF I2C ADAPTER

The TI TCA9555 GPIO expander is not supported at the kernel level. This white paper details the alternative approach to access the device on an I2C adapter from user space, through the device file (i.e. /dev).

Each registered adapter on the system will get a unique number. The Intel SMBUS controller (the adapter which masters the GPIO expander) is enumerated as i2c-7 on the Embedded PC 3000. The slave address of GPIO expander is 0x20 on this bus.

To view the information of all registered adapters, run the command "`i2cdetect -l`".

Refer to the following source code example for opening the interface and specifying the address of the slave device.

```
#include <linux/i2c-dev.h>
#include <linux/i2c.h>

#define I2C_ADAPTER "/dev/i2c-7"
#define TCA9555_ADDRESS 0x20

int fd;
...

fd = open(I2C_ADAPTER, O_RDWR);
if ( fd < 0 )  {
      fprintf(stderr, "failed to open i2c adapter file \"%s\": %s\n",
                  I2C_ADAPTER, strerror(errno));
      exit(-1);
}

if (ioctl(fd, I2C_SLAVE, TCA9555_ADDRESS) < 0) {
      fprintf(stderr, "failed to change slave address\n",
                  TCA9555_ADDRESS, strerror(errno));
      Eixt(-1);
}
```

ACCESS THE GPIO CONTROLLER

The TI TCA9555 provides 16 GPIO pins, with 12 connected to the Embedded PC 3000 Input and Output pins (DI0-5, DO0-5) as described in previous sections. The remaining 4 pins are unconnected. Each can be configured as an input or an output pin. The TI TCA9555 presents several couples of two 8-bit registers (i.e. 16 bits in each couple) with each couple dedicated to input, output and more. To simplify the programming model, the read/write byte operations on a couple of 8-bit register will be combined into a single 16-bit operation.

The following source code example implements the functions for reading and writing 16-bit words from and to the I2C bus via IOCTL.

```
int smbus_read_word(int fd, int reg, __u16 *word)
{
      union i2c_smbus_data data;
      struct i2c_smbus_ioctl_data ioctl_data;
      int rc;

      ioctl_data.read_write = I2C_SMBUS_READ;
      ioctl_data.command = reg;
      ioctl_data.size = I2C_SMBUS_WORD_DATA;
      ioctl_data.data = &data;
      rc = ioctl(fd, I2C_SMBUS, &ioctl_data);
      if (rc)
            fprintf(stderr, "Failed to read word from SMBUS\n");
      else
            *word = data.word;
      return rc;
}

int smbus_write_word(int fd, int reg, __u16 word)
{
      union i2c_smbus_data data;
      struct i2c_smbus_ioctl_data ioctl_data;
      int rc;

      ioctl_data.read_write = I2C_SMBUS_WRITE;
      ioctl_data.command = reg;
      ioctl_data.size = I2C_SMBUS_WORD_DATA;
      ioctl_data.data = &data;
      data.word = word;
      rc = ioctl(fd, I2C_SMBUS, &ioctl_data);
      if (rc)
            fprintf(stderr, "Failed to write word to SMBUS\n"”);
      return rc;
}
```

THE REGISTERS ON GPIO CONTROLLER

The TI TCA9555 consists of four couples of 8-bit registers. From the programmer’s perspective, there are four 16-bit registers comprising of configuration, input, output, and polarity registers.

Each bit in the configuration register represents the direction, the input or output of a GPIO pin.

- Each bit in the output register specifies the level of an output pin to be high or low.
- Each bit in the input register indicates the read value of the current level of a GPIO pin.

The following source code example implements three functions to set the direction, set and read the level of a specific GPIO pin.

```
#define TCA9555_REG_INPUT     0
#define TCA9555_REG_OUTPUT    2
#define TCA9555_REG_DIRECTION 6

#define GPIO_MODE_OUTPUT      1
#define GPIO_MODE_INPUT       2

int gpio_pin_set_direction(int pin, int dir)
{
      __u16 bmp;
      int rc = smbus_read_word(fd, TCA9555_REG_DIRECTION, &bmp);
      if ( rc )
            return rc;

      if ( dir == GPIO_MODE_INPUT )
            bmp |= 1 << chip_pin;
      else if ( dir == GPIO_MODE_OUTPUT )
            bmp &= ~(1 << chip_pin);

      rc = smbus_write_word(fd, TCA9555_REG_DIRECTION, bmp);
      return rc;
}

int gpio_pin_set_level(int pin, int level)
{
      __u16 bmp;
      int rc = smbus_read_word(fd, TCA9555_REG_OUTPUT, &bmp);
      if ( rc )
            return rc;

      if ( level )
            bmp |= 1 << chip_pin;
      else
            bmp &= ~(1 << chip_pin);

      rc = smbus_write_word(fd, TCA9555_REG_OUTPUT, bmp);
      return rc;
}

int gpio_pin_get_level(int pin, int *level)
{
      __u16 bmp;
      int rc = smbus_read_word(fd, TCA9555_REG_INPUT, &bmp);
      if ( !rc )
            *level = !!(bmp & (1 << chip_pin));
      return rc;
}
```

APPENDIX: FULL SOURCE CODE OF AN EXAMPLE PROGRAM

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>
#include <sys/ioctl.h>
#include <fcntl.h>

// Embedded PC 3000: board-specific data and functions
#define I2C_ADAPTER "/dev/i2c-7"
#define TCA9555_ADDRESS 0x20

#define PIN_DO0     1
#define PIN_DO5     6
#define PIN_DI0     14
#define PIN_DI5     19

static int _smbus_write_word(int fd, int reg, __u16 word)
{
       union i2c_smbus_data data;
       struct i2c_smbus_ioctl_data ioctl_data;
       int rc;

       ioctl_data.read_write = I2C_SMBUS_WRITE;
       ioctl_data.command = reg;
       ioctl_data.size = I2C_SMBUS_WORD_DATA;
       ioctl_data.data = &data;
       data.word = word;
       rc = ioctl(fd, I2C_SMBUS, &ioctl_data);
       if (rc)
              fprintf(stderr, "Failed to write word to SMBUS: %s\n", strerror(errno));

       return rc;
}

static int _smbus_read_word(int fd, int reg, __u16 *word)
{
       union i2c_smbus_data data;
       struct i2c_smbus_ioctl_data ioctl_data;
       int rc;

       ioctl_data.read_write = I2C_SMBUS_READ;
       ioctl_data.command = reg;
       ioctl_data.size = I2C_SMBUS_WORD_DATA;
       ioctl_data.data = &data;
       rc = ioctl(fd, I2C_SMBUS, &ioctl_data);
       if (rc)
              fprintf(stderr, "Failed to read word from SMBUS: %s\n", strerror(errno));
       else
              *word = data.word;

       return rc;
}

static int map_to_chip_pin(int pin)
{
// "pin" specifies the pin number on connector
       int chip_pin = -1;
       if ( pin >= PIN_DO0 && pin <= PIN_DO5 )
              chip_pin = pin - 1;
```

```
	else if ( pin >= PIN_DI0 && pin <= PIN_DI5 )
		chip_pin = pin - 14 + 8;
	return chip_pin;
}

// TI TCA9555: chip-specfic data
#define TCA9555_REG_INPUT	0
#define TCA9555_REG_OUTPUT	2
#define TCA9555_REG_DIRECTION	6

#define GPIO_CAP_OUTPUT		1
#define GPIO_CAP_INPUT		2
#define GPIO_CAP_INOUT		(GPIO_CAP_OUTPUT|GPIO_CAP_INPUT)

#define GPIO_MODE_UNKNOWN	0
#define GPIO_MODE_OUTPUT	1
#define GPIO_MODE_INPUT		2

typedef struct {
	int capability;
	int mode;
} gpio_pin;

typedef struct {
	int fd;
	gpio_pin pins[16];
} gpio_controller;

gpio_controller *gpio_initialize(char *platform)
{
	gpio_controller *gc;
	char *adapter_name = I2C_ADAPTER;
	int slave_addr = TCA9555_ADDRESS;
	int i, pin, rc;

	gc = malloc(sizeof(*gc));
	if ( !gc )  {
		fprintf(stderr, "%s: no memory error\n", __FUNCTION__);
		return NULL;
	}

	gc->fd = open(adapter_name, O_RDWR);
	if ( gc->fd < 0 )  {
		fprintf(stderr, "%s: failed to open i2c adapter file \"%s\": %s\n",
			__FUNCTION__, adapter_name, strerror(errno));
		goto exit_on_open_error;
	}

	if (ioctl(gc->fd, I2C_SLAVE, slave_addr) < 0) {
		fprintf(stderr, "%s: failed to change slave address to 0x%X: %s\n",
			__FUNCTION__, slave_addr, strerror(errno));
		goto exit_on_ioctl_error;
	}

	// Setup controller-level pin variables
	for ( i = 0 ; i < 16 ; i++ )  {
		gc->pins[i].mode = GPIO_MODE_UNKNOWN;
		gc->pins[i].capability = GPIO_CAP_INOUT;
	}

	// Apply board-level contraints to pins

	// Pin 1~6 on connector are output pins
	int bmp_dir = 0;
	for ( i = PIN_DO0 ; i <= PIN_DO5 ; i++ )  {
```

```
                pin = map_to_chip_pin(i);
                gc->pins[pin].mode = GPIO_MODE_OUTPUT;
          // we do not allow user to change mode so restrict the pin cap to
          // be output only
                gc->pins[pin].capability = GPIO_CAP_OUTPUT;
        }
        // Pin 14~19 on connector are input pins
        for ( i = PIN_DI0 ; i <= PIN_DI5 ; i++ )  {
                pin = map_to_chip_pin(i);
                gc->pins[pin].mode = GPIO_MODE_INPUT;
                bmp_dir |= (1 << pin);
          // we do not allow user to change mode so restrict the pin cap to
          // be input only
                gc->pins[pin].capability = GPIO_CAP_INPUT;
        }

        if ( _smbus_write_word(gc->fd, TCA9555_REG_DIRECTION, bmp_dir) ) {
                fprintf(stderr, "%s: failed to set pin directions", __FUNCTION__);
                goto exit_on_access_error;
        }

        return gc;

exit_on_access_error:
exit_on_ioctl_error:
        close(gc->fd);

exit_on_open_error:
        free(gc);

        return NULL;
}

void gpio_exit(gpio_controller *gc)
{
        if ( gc == NULL )
                return;

        if ( gc->fd >= 0 )  {
                close(gc->fd);
                gc->fd = -1;
        }
        free(gc);
}

int gpio_pin_set_level(gpio_controller *gc, int pin, int level)
{
        int rc, chip_pin;
        __u16 bmp;

        chip_pin = map_to_chip_pin(pin);
        if ( chip_pin < 0 )  {
                fprintf(stderr, "%s: pin number %d is invalid\n", __FUNCTION__, pin);
                return -EINVAL;
        }

        if (gc->pins[chip_pin].mode != GPIO_MODE_OUTPUT)  {
                fprintf(stderr, "%s: pin %d not a output pin\n", __FUNCTION__, pin);
                return -EINVAL;
        }

        rc = _smbus_read_word(gc->fd, TCA9555_REG_OUTPUT, &bmp);
        if ( rc )  {
                fprintf(stderr, "%s: failed to access controller\n", __FUNCTION__);
```

```
                return rc;
        }

        if ( level )
                bmp |= 1 << chip_pin;
        else
                bmp &= ~(1 << chip_pin);

        rc = _smbus_write_word(gc->fd, TCA9555_REG_OUTPUT, bmp);
        if ( rc )  {
                fprintf(stderr, "%s: failed to access controller\n", __FUNCTION__);
                return rc;
        }

        return 0;
}

int gpio_pin_get_level(gpio_controller *gc, int pin, int *level)
{
        int rc, chip_pin;
        __u16 bmp;

        chip_pin = map_to_chip_pin(pin);
        if ( chip_pin < 0 )  {
                fprintf(stderr, "%s: pin number %d is invalid\n", __FUNCTION__, pin);
                return -EINVAL;
        }

        // We don't check the pin mode (i.e. input or output) here because the level
        // could be read in both input and output mode

        rc = _smbus_read_word(gc->fd, TCA9555_REG_INPUT, &bmp);
        if ( rc )  {
                fprintf(stderr, "%s: failed to access controller\n", __FUNCTION__);
                return rc;
        }

        *level = !!(bmp & (1 << chip_pin));
        return 0;
}

int gpio_pin_set_direction(gpio_controller *gc, int pin, int dir)
{
        int chip_pin = map_to_chip_pin(pin);
        if ( chip_pin < 0 )  {
                fprintf(stderr, "%s: pin number %d is invalid\n", __FUNCTION__, pin);
                return -EINVAL;
        }

// On Embedded PC 3000 system, we do not allow user to change the directions of GPIO pins
        return -ENOSYS;
}

int gpio_pin_get_direction(gpio_controller *gc, int pin, int *dir)
{
        int chip_pin = map_to_chip_pin(pin);
        if ( chip_pin < 0 )  {
                fprintf(stderr, "%s: pin number %d is invalid\n", __FUNCTION__, pin);
                return -EINVAL;
        }

        *dir = gc->pins[chip_pin].mode;
        return 0;
}
```

```
int main(int argc, char** argv)
{
	int rc, level;
	__u16 data;

	gpio_controller *epc3000_gc = gpio_initialize("Embedded PC 3000");

	if ( !epc3000_gc )  {
		fprintf(stderr, "Failed to initialize Embedded PC 3000 GPIO controller\n");
		return -1;
	}

	// Set the level of DO0 (pin 1 on connector)
	printf("Setting the level of DO0 to low...\n");
	rc = gpio_pin_set_level(epc3000, 1, 0);
	if ( !rc )
		printf("Successful\n");
	printf("\n");

	// Get the level of DI0 (pin 14 on connector)
	printf("Reading the level of DI0...\n");
	rc = gpio_pin_get_level(epc3000, 14, &level);
	if ( !rc )
		printf("Level of DI0 = %d\n", level);
	printf("\n");

	// Set the level of DO0 (pin 1 on connector)
	printf("Setting the level of DO0 to high...\n");
	rc = gpio_pin_set_level(epc3000, 1, 1);
	if ( !rc )
		printf("Successful\n");
	printf("\n");

	// Get the level of DI0 (pin 14 on connector)
	printf("Reading the level of DI0...\n");
	rc = gpio_pin_get_level(epc3000, 14, &level);
	if ( !rc )
		printf("Level of DI0 = %d\n", level);
	printf("\n");

	gpio_exit(epc3000);

	return 0;
}
```