EMAC GPIO Class

The EMAC GPIO Class is a generic programming interface for General Purpose Input Output (GPIO) devices in the Linux Kernel. The GPIO class driver has two main interfaces under standard Linux: Linux sysfs and character device. The GPIO class is extremely flexible and uses a set of function pointers at the kernel level to allow for specialized read/write functions for each device. EMAC utilizes the GPIO class to create devices that use the same simple interface, such as GPIO registers, simple configuration variables, hardware registers, and Analog-to-Digital converter drivers where the user needs to retrieve a single value from the device.

At this point the EMAC GPIO Class is only available in EMAC patched kernels for select devices.

Components

An EMAC GPIO Class Device has three main components. A clear understanding of these components is important for proper use and implementation.

Data

The `data` member of a GPIO device is the primary component. In the most simple devices, this is the only component accessible from userspace. The `data` member is read and/or written to access or set the current value of the GPIO device. The actual function of the `data` member is implementation specific. For a GPIO device in its purest form, such as the GPIO ports present in many of EMAC's CPLD designs, the `data` member is a register value with each of the 8 least significant bits representing the digital state of a single line in the GPIO port. In contrast, AtoD devices implemented with the EMAC GPIO Class will typically provide the analog value of the currently selected channel in the `data` member.

The `data` member is allocated as a 32-bit unsigned integer regardless of implementation. Many devices use only the least significant byte or the least significant bit.

DDR

The `ddr` or Data Direction Register is a member of the EMAC GPIO Class that is used primarily for direction-configurable GPIO devices. Many devices do not register this aspect of the interface. The `ddr` member is used to determine if a particular device should act as an input or output. Depending on the implementation, many devices are bit-configurable GPIO devices, meaning that each bit/line in the GPIO device can be configured as an input or output by its respective value in the `ddr`. Typically, a '1' value in the `ddr` configures the device as an output and a '0' value in the `ddr` configures the device as an input. Bidirectional GPIO devices are generally configured as inputs by default until explicitly configured as an output to prevent hardware contention.

For example, an EMAC GPIO Class device `porta` is a one-byte bit-configurable bidirectional GPIO device. Each bit in the `data` register corresponds to the digital status of the corresponding line `porta[0-7]`. On reset, the `ddr` member is read at `0x00`, indicating that the device is configured with all lines as inputs. If the `ddr` member is set to `0xAA` (binary `1010 1010`) bits `0`, `2`, `4`, and `6` will remain configured as inputs while the rest of the bits will be configured as outputs. Reading and setting the value in the `data` member will react according to the configuration set in the `ddr`. The `ddr` is allocated as a 32-bit unsigned integer regardless of implementation.

Index

Many EMAC GPIO Class devices are implemented as indexed GPIO Devices. These devices utilize a member named `index` which impacts the device according to the implementation. The `index` member is typically used to specify a memory offset from some base for the `data` member or some type of channel setting or controller selection. Many devices ignore the `index` value or do not register this member at all.

An example of an indexed GPIO Class device is the `indexed_atod` device (and other AtoD devices) found on many EMAC boards. In this case, the `index` member is used to set the current channel of the AtoD to read from the `data` member. When the `index` is set to `0`, channel `0` will be accessed. When the index is set to `3`, the `data` register will reflect the current value on channel `3` of the device. Another example would be two identical counter registers in a device where setting the index would control which counter was accessed when reading the `data` register. This

would be an alternative to registering a separate device for each counter.

Indexed GPIO devices should check the applicable range for the `index` when the user attempts to set the device. For example, an 8-channel AtoD device implemented as an EMAC GPIO Class device would have a valid range for the `index` member of `0-7`. Attempting to set the `index` to anything greater than 7 would cause the `index` to be set to the maximum allowed value of 7.

Lock

Locking support is currently not available on all EMAC boards. This feature was introduced in the later versions of the 2.6.28 kernel for the AT91-based boards. It will be implemented in other kernel trees in the future.

Many GPIO devices may be accessed by multiple processes simultaneously, which leads to the possibility of race conditions. The EMAC GPIO Class includes a locking mechanism to prevent these cases when used correctly. Without the locking mechanism, synchronization needs to be performed in userspace.

The lock is implemented as a `struct mutex` in the GPIO Class device structure. The lock is obtained by the GPIO driver interface through a call to `mutex_lock()` and freed using `mutex_unlock()`. This means that any attempts to lock an already locked device will cause the caller to sleep until the lock is freed by the process holding the lock. A nonblocking method using `mutex_trylock()` may be implemented in the future.

All access to a device must be synchronized using the lock. Further details on this implementation are described in later sections.

User Interfaces

This section describes the two user interfaces available for the GPIO class. The sysfs layer is very handy for scripting and testing. For higher performance and more flexible access, the character driver interface is used, generally from a C application. This provides a simple `ioctl()` interface.

Sysfs

The `sysfs` filesystem is a virtual filesystem provided by the Linux kernel that allows for a file-based interface to kernel driver parameters. The EMAC GPIO Class utilizes `sysfs` to provide a simple read/write interface to the GPIO Class device. This interface is easily accessible from the Linux shell and through shell scripting.

The GPIO Class driver registers a directory at `/sys/class/gpio` assuming that sysfs has been mounted on the system. Under this directory, every GPIO Class device registered with the system will have a directory (i.e. `/sys/class/gpio/porta`). Within this directory, there will be some files that are a part of the sysfs heirarchy as well as the GPIO Class interface files `data`, `ddr`, and `index`. Depending on the type of device and its implementation, the `ddr` and `index` files may or may not exist; `data` should always be present.

Note that the `/sys/class/gpio` directory is used by the new I/O driver code in the standard Linux kernel tree. This location will be changed by EMAC in the near future to avoid conflict.

Reading

Reading the current value of a device's settings is as simple as reading the respective file. The `cat` command is typically used for this. The output is formatted as ASCII hexadecimal characters. For example, to read the value of the data member of the `porta` device, run the command:

```
emac-oe$ cat /sys/class/gpio/porta/data
```

The implementation of the read function for the `data`, `ddr`, and `index` members using the sysfs interface automatically acquires and frees the lock on the respective GPIO device. The actual value returned depends on the device implementation.

Writing

Writing a new setting to a member of the GPIO device through the sysfs interface is accomplished by writing a new value to the respective file. The `echo` command is typically used along with file redirection to set the new value directly, although the output of any command could be used as long as it is formatted correctly. For example, to set the `porta` device to `0xFF` (all on) the user would first set all bits to output using the `ddr` register and then write `0xFF` to the data register as shown below:

```
emac-oe$ echo 0xff > /sys/class/gpio/porta/ddr
emac-oe$ echo 0xff > /sys/class/gpio/porta/data
```

The implementation of the write function for the `data`, `ddr`, and `index` members using the sysfs interface automatically acquires and frees the lock on the respective GPIO device. The low-level effects of the write function depends on the device implementation.

Character Driver

The character driver interface is the preferred method for accessing EMAC GPIO Class devices from programmed applications. This interface is significantly more efficient than the sysfs layer and easier to program. In kernels that support the GPIO Class locking mechanism, the character driver interface offers user control over the lock that is not available using the sysfs interface. The character driver interface uses the `ioctl()` system call. Newer kernels also support an `fasync` method for signal-based event notification from the driver to the user application.

Available IOCTLs

Several different `ioctl` command are available for use. The user must know the supported features of the device before using an `ioctl` command. For example, many devices do not support the `ddr` or `index` functionality. All ioctl commands will return a negative error code on failure.

Locking

The ioctl commands in this section will acquire and free the GPIO lock on kernels that include support for the lock. These commands are the most efficient to use when a single GPIO read/write is required. On kernels that do not support the GPIO lock, these commands simply preform the advertised action.

- `DATAREAD`: Read the current value of the devices `data` member by calling the `data_read` function. Return value is stored in a 32-bit integer which is passed to the ioctl via a pointer.
- `DATAWRITE`: Write a new value to the device's `data` member by calling the `data_write` function. The value to write is passed to the ioctl call through a pointer to a 32-bit integer.
- `DDRREAD`: Read the current value of the device's `ddr` member by calling the `ddr_read` function. Return value is stored in a 32-bit integer which is passed to the ioctl via a pointer.
- `DDRWRITE`: Write a new value to the device's `ddr` member by calling the `ddr_write` function. The value to write is passed to the ioctl call through a pointer to a 32-bit integer.
- `INDEXREAD`: Read the current value of the devices `index` member by calling the `index_read` function. Return value is stored in a 32-bit integer which is passed to the ioctl via a pointer.
- `INDEXWRITE`: Write a new value to the device's `index` member by calling the `index_write` function. The value to write is passed to the ioctl call through a pointer to a 32-bit integer.

Lock Control

The ioctl commands in this section are only available on kernels that support the GPIO Class locking mechanism. They are used to control the current state of the lock.

- `GPIOLOCK`: Lock the device's mutex through a call to `mutex_lock()`. This command will sleep until the lock has been acquired.
- `GPIOUNLOCK`: Release the device's mutex through a call to `mutex_unlock()`. This command should not be called unless the lock has already been acquired using `GPIOLOCK`.

No-Lock

The ioctl commands in this section do not affect the lock of the GPIO device and are only available in kernels supporting the GPIO Class locking feature. They are simple versions of the standard commands listed in the Locking section. Before calling one of these commmands, the process must acquire the lock using the `GPIOLOCK` command. The purpose of these commands is to allow synchronization between multiple processes or threads when accessing the same device. For example, when preforming a read followed by a write, the lock would need to be held for the duration of both commands to ensure that incorrect values are not written to the device.

- `DATAREAD_NL`: Read the current value of the device's `data` member.
- `DATAWRITE_NL`: Write a new value to the device's `data` member.
- `DDRREAD_NL`: Read the current value of the device's `ddr` member.
- `DDRWRITE_NL`: Write a new value to the device's `ddr` member.
- `INDEXREAD_NL`: Read the current value of the device's `index` member.
- `INDEXWRITE_NL`: Write a new value to the device's `index` member.

Asynchronous Notification

The ioctl commands in this section are for use with the asynchronous notification feature of the EMAC GPIO Class. Most devices do not use this feature and these commands are only available on kernel's that support this feature. Low-level implementation is highly device-specific.

- `SETNOTIFY`: Set a new notification bitmask for the current file descriptor. The value is passed to the ioctl call as a pointer to a 32-bit integer. The gpio lock is acquired and released during this command.
- `GETNOTIFY`: Get the current notification bitmask for this file descriptor. The value is returned via a pointer to a 32-bit integer passed to the ioctl call. The gpio lock is not acquired and does not need to be held for this command.
- `DATAREADQ`: Read the first value from the data queue. The value is returned via a pointer to a 32-bit integer passed to the ioctl call. The gpio lock is not acquired and does not need to be held for this command.
- `GETQUEUESIZE`: Read the current number of values in the data queue for this file descriptor. The value is returned via a pointer to a 32-bit integer passed to the ioctl call. The gpio lock is not acquired and does not need to be held for this command.

Programming

The following general examples demonstrate the character driver programming interface.

The examples below do not check the return value of calls for errors. Be sure to check for and handle errors in a production system.

To access a device from a C application, it must first be opened using `open()`:

```
int fd;
fd = open(DEVICE, O_RDWR);
....
close(fd);
```

If the open command is successful, any supported ioctl command may be used to access the device. The example below sets the value of the `data` member to 0xFF:

```
__u32 arg;
...
arg = 0xff;
ioctl(fd, DATAWRITE, &arg);
```

The example below uses the locking mechanism to perform a sequence of accesses to a device:

```
#define CONTROLBIT (1 << 3)
__u32 arg;
...
ioctl(fd, GPIOLOCK, NULL);
arg = 1;
ioctl(fd, INDEXWRITE_NL, &arg);
ioctl(fd, DATAREAD_NL, &arg);
/* set control bit */
arg &= CONTROLBIT;
ioctl(fd, DATAWRITE_NL, &arg);
ioctl(fd, GPIOUNLOCK, NULL);
```

Asynchronous Notification Example

Asynchronous notification for the EMAC GPIO Class is a new feature in some kernels. This was added to the driver at the same time as locking support and utilizes the `fasync` system call. Essentially, this feature allows the kernel driver to notify the user application of some event through a signal. Typically this is used for devices which support hardware interrupts. Currently, no standard EMAC devices utilize the asynchronous notification feature.

For each open file descriptor of a certain device, it is possible to set a notification mask indicating which types of events will trigger notification as well as which signal will be sent to the process from the driver. The default signal is `SIGIO`. When an event occurs, a value will be written to a queue for each file descriptor that is registered for notification of the event. The notification mask, events, and queue functionality are all implementation-specific.

The following example utilizes the asynchronous notification interface for a device that supports it. This particular device is an interrupt-enabled register that will notify a process if the type of interrupt that occurs matches the notification mask of the particular file descriptor. When an applicable interrupt occurs, the data register will immediately be queued into the file descriptor specific data queue.

The device utilized in the example below does not exist on any EMAC systems. The example is provided purely for the purpose of illustrating the asynchronous notification interface.

gpio-fasync-example.c

```
#define _GNU_SOURCE

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <asm/types.h>
#include <signal.h>
#include "gpio_char.h"

static sig_atomic_t caught = 0;
static sig_atomic_t run = 1;

void sig_handle(int sig)
{
        if (sig == SIGIO)
                caught = 1;
        else /* termination */
                run = 0;
}

#define DEVICE      "/dev/gpio_async"
#define NOTIFY_MASK (1 << 4)

/**
 * function to enable asynchronous notification on the given file descriptor
 * for the specified signal
 * @param fd the open GPIO device file descriptor to enable notification for
 * @param sig the signal to be sent by the kernel for event notification
 */
static void set_fasync(int fd, int sig);

/**
 * function to set the event notification flags for the driver on a given
 * file descriptor.
 * @param fd the open GPIO device file descriptor for notification
 * @param notify bitmask specifying which events this file descriptor should
 *   be notified on
 * @return the result of the ioctl() call to set the notification flags
 */
static int set_notify_flags(int fd, __u32 notify);

/**
 * function to get the queue size for the given file descriptor
 * @param fd the open GPIO device file descriptor to query
 * @return the current queue size as reported by the driver
 */
static __u32 get_queue_size(int fd);

int main(int argc, char *argv[])
{
        int fd;
        __u32 data;
        struct sigaction sact;
        sigset_t block_mask;
        sigset_t suspend_mask;
        sigset_t old_mask;

        /* set up signal handlers */
        sact.sa_handler = sig_handle;
        sigemptyset(&sact.sa_mask);
        sact.sa_flags = 0;
```

gpio_char.h

```
#ifndef GPIO_CHAR_H_
#define GPIO_CHAR_H_

/* Userspace version of kernel header file */

#include <linux/ioctl.h>

#define RTDM_CLASS_GPIO 0x80

/* standard read/write functions */
#define DDRREAD         _IOR(RTDM_CLASS_GPIO,0,char)
#define DDRWRITE        _IOW(RTDM_CLASS_GPIO,0,char)
#define DATAREAD        _IOR(RTDM_CLASS_GPIO,1,char)
#define DATAWRITE       _IOW(RTDM_CLASS_GPIO,1,char)
#define INDEXREAD       _IOR(RTDM_CLASS_GPIO,2,char)
#define INDEXWRITE      _IOW(RTDM_CLASS_GPIO,2,char)

/* nolock functions */
#define DDRREAD_NL      _IOR(RTDM_CLASS_GPIO, 3, char)
#define DDRWRITE_NL     _IOW(RTDM_CLASS_GPIO, 3, char)
#define DATAREAD_NL     _IOR(RTDM_CLASS_GPIO, 4, char)
#define DATAWRITE_NL    _IOW(RTDM_CLASS_GPIO, 4, char)
#define INDEXREAD_NL    _IOR(RTDM_CLASS_GPIO, 5, char)
#define INDEXWRITE_NL   _IOW(RTDM_CLASS_GPIO, 5, char)

/* lock/unlock */
#define GPIOLOCK        _IO( RTDM_CLASS_GPIO, 6)
#define GPIOUNLOCK      _IO( RTDM_CLASS_GPIO, 7)

#define DATAREADQ       _IOR(RTDM_CLASS_GPIO, 8, char)
#define GETNOTIFY       _IOR(RTDM_CLASS_GPIO, 9, char)
#define SETNOTIFY       _IOW(RTDM_CLASS_GPIO, 10, char)
#define GETQUEUESIZE    _IOR(RTDM_CLASS_GPIO, 11, char)

#endif /*GPIO_CHAR_H_*/
```

Kernel Driver Implementation

The EMAC GPIO Class makes it easy to add new devices to the kernel. Function pointers are used to define the implementation of the read and write functions for `data`, `ddr`, and `index` members. This allows for an efficient definition of custom implementations. Standard read and write functions for accessing memory addresses are defined for use if they match the needs of the interface. This section describes the process of creating and registering an EMAC GPIO Class device in the Linux kernel.

EMAC GPIO Class Definition

The EMAC GPIO Class is implemented in several important files. First, the `gpio.h` header file defines the structures used for the GPIO class. This file is located in `include/linux/class/gpio.h` in the kernel source. It defines the following types:

```
typedef u32 gpio_data;

typedef struct gpio_s {
        const char *name;
        int subclass;
        gpio_data index;
        gpio_data range;
        void *ddr;
        void *data;
        gpio_data shadow;
         gpio_data(*data_read) (struct gpio_s * gpio);
        int (*data_write) (struct gpio_s * gpio, gpio_data data);
         gpio_data(*ddr_read) (struct gpio_s * gpio);
        int (*ddr_write) (struct gpio_s * gpio, gpio_data data);
         gpio_data(*index_read) (struct gpio_s * gpio);
        int (*index_write) (struct gpio_s * gpio, gpio_data data);
        struct mutex lock;
        struct gpio_irq_data irq_data;
} gpio_t;

struct gpio_irq_data {
        int irq;
        unsigned int irq_flags;
        gpio_data (*handler)(struct gpio_s *gpio);
        gpio_data (*get_queue)(struct gpio_s *gpio, gpio_data notify);
        void (*irq_config)(struct gpio_s *gpio);
        void (*change_notify)(struct gpio_s *gpio, gpio_data curr_notify,
                        gpio_data new_notify);
};
```

The `struct gpio_s` structure is the main GPIO device definition. The `data_read`, `data_write`, `ddr_read`, `ddr_write`, `index_read`, and `index_write` function pointers define the low-level actions that are performed when the device is accessed. The `struct gpio_irq_data` structure is used only for IRQ-enabled GPIO devices as required for the asynchronous notification feature.

In addition, standard low-level functions are declared for use in accessing 8-bit ports. These functions utilize the `ioread8` and `iowrite8` functions for accessing the memory locations associated with the device.

```
int gpio_ddr_write8(gpio_t *gpio, gpio_data data);
int gpio_data_write8(gpio_t *gpio, gpio_data data);
int gpio_index_write(gpio_t *gpio, gpio_data data);
int gpio_empty_write(gpio_t *gpio, gpio_data data);
gpio_data gpio_ddr_read8(gpio_t *gpio);
gpio_data gpio_data_read8(gpio_t *gpio);
gpio_data gpio_index_read(gpio_t *gpio);
gpio_data gpio_shadow_read8(gpio_t *gpio);
gpio_data gpio_ff_read(gpio_t *gpio);
gpio_data gpio_zero_read(gpio_t *gpio);
```

Additional functions are provided for registering GPIO devices in the kernel. These will be discussed in the example implementation below.

The EMAC GPIO Class is implemented in the file `drivers/misc/classes/gpio.c` in the kernel source tree. This file implements the functions described above as declared in `gpio.h` and provides the implementation for the `sysfs` interface. More detailed information can be obtained by reviewing the source code.

The character driver interface for the EMAC GPIO Class is defined in `drivers/misc/classes/char/gpio_char.c`. This file provides the `ioctl` interface, character device registration, and asynchronous notification support. All GPIO class devices create a character device interface upon registration.

Example GPIO Devices

GPIO class devices may be defined and registered at any location in the kernel code. Often times, these devices are hardware-specific and are initialized in the machine initialization code. The example code in this section defines an EMAC GPIO Class device that is used to control several GPIO lines on an AT91-family processor. This code is implemented in a separate C file and initialized from the board-specific implementation.

The GPIO device in this example is named `ege` (EMAC GPIO Example). The following variables and setup functions are defined:

```
/**
 * static array defining GPIO set
 */
static unsigned ege_gpio_set[] = {
        AT91_PIN_PC0,
        AT91_PIN_PC1,
        AT91_PIN_PC2,
        AT91_PIN_PC3,
        AT91_PIN_PB0,
        AT91_PIN_PB1,
        AT91_PIN_PB2,
        AT91_PIN_PB3,
};

#define EGE_NUM_GPIO ARRAY_SIZE(ege_gpio_set)

gpio_data ddr_value_shadow = 0x0;

/**
 * function to initialize the digital inputs and outputs
 */
static int ege_gpio_init(void)
{
        int i;

        /*
         * set all GPIOs to inputs by default
         */
        for (i = 0; i < EGE_NUM_GPIO; i++)
                at91_set_gpio_input(ege_gpio_set[i], 0);

        return 0;
}
```

Each value in the `ege_gpio_set` array represents one bit in the GPIO device `data` register. Reading or writing bit 0 will access `AT91_PIN_PC0` on the processor, while bit 7 will access `AT91_PIN_PB3`. The `ege_gpio_init()` function is required to initialize all of the GPIO pins to inputs prior to the first access.

The first step in creating the GPIO class device is defining the access functions for the `data`, `ddr`, and `index` members. This device will not use an `index` member, so the `index_write` and `index_read` variables will be set to NULL. This prevents them from being registered with the device. There will be no `index` file created for the `sysfs` interface, and attempting to access the `index` through the character interface `ioctl` will return an error. The `ege_gpio_data_read`, `ege_gpio_data_write`, `ege_gpio_ddr_read`, and `ege_gpio_ddr_write` functions are defined as follows:

```
/**
 * function to read the current value of each GPIO line
 */
static gpio_data ege_gpio_data_read(struct gpio_s *gpio)
{
        gpio_data value;
        int i;

        value = 0;
        for (i = 0; i < EGE_NUM_GPIO; i++)
                value |= (at91_get_gpio_value(ege_gpio_set[i]) << i);

        return value;
}

/**
 * function to set the value of the GPIO lines
 */
static int ege_gpio_data_write(struct gpio_s *gpio, gpio_data data)
{
        int i;

        for (i = 0; i < EGE_NUM_GPIO; i++)
                at91_set_gpio_value(ege_gpio_set[i], (data >> i) & 1);

        return 0;
}

/**
 * data direction configuration read
 */
static gpio_data ege_gpio_ddr_read(struct gpio_s *gpio)
{
        return ddr_value_shadow;
}

/**
 * data direction configuration write
 */
static int ege_gpio_ddr_write(struct gpio_s *gpio, gpio_data data)
{
        int i;

        ddr_value_shadow = data;

        for (i = 0; i < EGE_NUM_GPIO; i++) {
                if(data & (1 << i)) {
                        /* set to output */
                        at91_set_gpio_output(ege_gpio_set[i], 0);
                        at91_set_multi_drive(ege_gpio_set[i], 0);
                } else  {
                        /* set to input */
                        at91_set_gpio_input(ege_gpio_set[i], 0);
                }
        }
        return 0;
}
```

Note that the code to read the current `ddr` configuration of each line is replaced by a shadow value that is stored during the `ege_gpio_ddr_write` function. After defining these functions, the device must be created and registered. The `ege_gpio_class_create` function performs this operation and is shown below:

```
struct device *ege_gpio_class_create(void)
{
        gpio_t *gpio = kmalloc(sizeof(gpio_t), GFP_KERNEL);

        memset(gpio, 0, sizeof(gpio_t));
        gpio->name = "ege_gpio";
        gpio->subclass = GPIO_SUBCLASS;
        gpio->data_write = ege_gpio_data_write;
        gpio->data_read = ege_gpio_data_read;
        gpio->ddr_write = ege_gpio_ddr_write;
        gpio->ddr_read = ege_gpio_ddr_read;
        gpio->index_write = NULL;
        gpio->index_read = NULL;
        printk("registering gpio device: %s\n", gpio->name);

        return gpio_register_device(gpio);
}
```

Once the `ege_gpio_init` function is called followed by the `ege_gpio_class_create` function, the `ege_gpio` device will be created and registered with the system.

Adding Asynchronous Notification Support

If required, asynchronous notification support could be added to the `ege_gpio` device from the example above. This section provides a simple example of the code required to add this feature.

The `irq_data` member of the GPIO class device is used to specify the action used for asynchronous notification and interrupt handling. The IRQ will be requested, configured, and registered in the GPIO character interface (`gpio_char.c`) if `irq_data` is valid.

The `handler` member of `irq_data` will be called by the generic GPIO class IRQ handler. This will be scheduled in a workqueue from the interrupt handler outside of interrupt context and called with the GPIO device lock held. This allows for the feature to be utilized by interfaces such as SPI devices which may sleep during hardware access. The function returns a bit mask for comparison with the notification mask set on any open devices. This reflects which type of hardware event has occurred. Typically, the `handler` function would read an interrupt status register from the device associated with the GPIO interface, such as a PLD register. In the most basic form where only one type of interrupt can occur the `handler` would simply return 1. The example `handler` function shown below calls a function `read_interrupt_status()` for which the implementation is not discussed here.

```
static gpio_data ege_gpio_handler(struct gpio_s *gpio)
{
        return read_interrupt_status();
}
```

The `get_queue` member of `irq_data` is called by the GPIO class interface to determine what data to add to the queue for any file-descriptor matching the notify mask of the interrupt type that has occurred. This function will be called with the device mutex held. The most common implementation of this function is to simply utilize the `data_read` function of the associated device and store the current value. This is useful to capture the current hardware state as instantaneously as possible after the interrupt occurs before the userspace process is notified of the event. It may be used for other purposes, however, such as taking a timestamp or reading a different hardware register. The current notify mask returned by `handler` will be passed to the `get_queue` function, so it is possible to customize the action based on the type of event(s) that occurred. The `get_queue` function shown below simply reads the current value returned by `data_read`.

```
static gpio_data ege_gpio_get_queue(struct gpio_s *gpio, gpio_data notify)
{
        return atomic_gpio_data_read(gpio);
}
```

The irq_config member of irq_data is called by the GPIO class interface before calling request_irq. The purpose of this function is to configure and enable the interrupt in the associated hardware for the GPIO device. In many cases, this function will have few tasks to perform. In the example irq_config function shown below, the set_irq_wake() function is called to note that the interrupt should be allowed to wake the hardware from APM suspend mode.

```
static void ege_gpio_irq_config(struct gpio_s *gpio)
{
        set_irq_wake(gpio->irq_data.irq, 1);
}
```

The change_notify member of irq_data is used to make any changes to the hardware when the user interface changes the notification mask. This may or may not be necessary depending on the desired operation. The primary design function is to keep track of which types of interrupts should be enabled and disabled in the associated hardware. For example, if no file-descriptors are registered for notification on a certain type of event, the associated bit should be disabled in the interrupt enable register. The bit should be enabled in the interrupt enable register as soon as one or more file-descriptors are set up to be notified on this type of event. The current notify mask for the file-descriptor as well as the new notify mask will be passed as arguments so that change_notify can determine which bits have been altered. The function for this simple interface does nothing.

```
static void ege_gpio_change_notify(struct gpio_s *gpio, gpio_data curr_notify,
                gpio_data new_notify)
{
}
```

The final step for adding IRQ handling and asynchronous notification to the interface is to specify the irq_data structure as part of the device initialization. The irq and irq_flags members are used to denote the specific interrupt and its features. The updated ege_gpio_class_create function is as follows:

```
struct device *ege_gpio_class_create(void)
{
        gpio_t *gpio = kmalloc(sizeof(gpio_t), GFP_KERNEL);

        memset(gpio, 0, sizeof(gpio_t));
        gpio->name = "ege_gpio";
        gpio->subclass = GPIO_SUBCLASS;
        gpio->data_write = ege_gpio_data_write;
        gpio->data_read = ege_gpio_data_read;
        gpio->ddr_write = ege_gpio_ddr_write;
        gpio->ddr_read = ege_gpio_ddr_read;
        gpio->index_write = NULL;
        gpio->index_read = NULL;

        gpio->irq_data.irq = AT91SAM9260_ID_IRQ1;
        gpio->irq_data.irq_flags = IRQF_SHARED | IRQF_TRIGGER_RISING;
        gpio->irq_data.handler = ege_gpio_handler;
        gpio->irq_data.get_queue = ege_gpio_get_queue;
        gpio->irq_data.irq_config = ege_gpio_irq_config;
        gpio->irq_data.change_notify = ege_gpio_change_notify;

        printk("registering gpio device: %s\n", gpio->name);

        return gpio_register_device(gpio);
}
```

After adding the code above, the EMAC GPIO Class driver will register, enable, and handle the interrupt exactly as specified. Notification can be enabled for userspace processes according to the asynchronous notification example.
