CODE TIME TECHNOLOGIES

Abassi RTOS

GPIO Support

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the GPIO driver used by Abassi[1] [R1] (including mAbassi [R2] and μAbassi [R3]). The standalone version of the GPIO driver is also described here.

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`???_gpio.h`	Include file for the GPIO driver (??? is target dependent)
`???_gpio.c`	"C" file for the GPIO driver (??? is target dependent)
`SAL.h`	Include file for the standalone abstraction layer (supplied with standalone package only)
`SAL.c`	"C" file for the standalone abstraction layer (supplied with standalone package only)
`ISRhandler_???.s`	"ASM" add-on file for the standalone version. It contains support for both the driver and the demo application (supplied with standalone package only)

1.2 Features

The GPIO driver API and build options are kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself. All GPIO APIs, except the initialization, can be used in interrupt handlers. When the target platform is a SoC FPGA, it is likely the manufacturer's FPGA IP GPIO modules are also supported in extra of the core processor GPIOs. All GPIO drivers can be set-up to be part of a multi-driver set-up.

1.3 Limitations

Some controllers cannot support some of the features described in this document. Please refer to the header in specific driver code for a description / list of these limitations. This is described near the top of the files: `???_gpio.h` and `???_gpio.c`.

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and μAbassi.

2 Target Set-up

All there is to do to configure and enable the use of the GPIO driver in an application based on Abassi is to include the following file in the build:

- `???_gpio.c` (For Abassi & standalone)
- `SAL.c` (For Standalone)
- `ISRhandler_???.s` (For Standalone)

and to set-up the include search directory order making sure the file `???_gpio.h` is found (and `SAL.h` for the standalone)

If interrupts are used, one or multiple GPIO interrupt handlers (`QSPIintHndl_`*`n`*`()`, Section 5.1.9) must be attached to the interrupt controller. In Abassi this is simply done using the `OSisrInstall()` component.

The GPIO driver may or may not, depending on the target platform, be independent from other include files.

2.1 Build Options

There are a few build options that allow the GPIO driver to be configured for the needs of the target application. The following table lists all of them there is an alternate token naming, refer to section 4) :

Table 2-1 Build Options

Token Name	Default	Description
`OS_PLATFORM`	`Target dependent`	Number indicating the target platform. Refer to `???_i2c.h` and `Platform.h` to see the list of supported platforms and the default one.
`GPIO_USE_ISR`	`1`	Boolean to enable/disable the handling of ISR from the GPIO controller
`GPIO_ARG_CHECK`	`1`	Boolean to enable/disable the check on the validity of the API function arguments
`GPIO_MULTIPLE_DRIVER`	`0`	To use drivers for different GPIO controllers at the same time.
`GPIO_FPGA_ADDR_`*`N`*	`Target dependent`	Base address of FPGA GPIO controllers. *N* can have the value 1 to 10

2.1.1 OS_PLATFORM

The build option `OS_PLATFORM` informs the GPIO driver about the platform it is operating on. It is needed to inform the GPIO driver implicitly knows the total number of GPIO banks and I/O pins on the target platform.

The information on the numbering used for `OS_PLATFORM` is available in the `Platform.txt` and `Platform.h` files also supplied as part of the distribution.

2.1.2 GPIO_USE_ISR

The build option `GPIO_USE_ISR` informs the driver if it supports GPIO interrupts or not. This is a Boole\an, when set to non-zero value, it includes the code to handle interrupts, and when set to a zero it does not include it. By default, `GPIO_USE_ISR` is set to a non-zero value meaning the code to handle GPIO interrupt is included.

2.1.3 GPIO_ARG_CHECK

The build options `GPIO_ARG_CHECK` controls if the driver checks the validity of the API function arguments or not. This build option is a Boolean; when set to a non-zero value, the driver checks the validity of the arguments and returns an error code when the arguments are invalid. When set to a zero value, it does not check the validity of the arguments. By default, `GPIO_ARG_CHECK` is set to a non-zero value meaning the code check the validity of the arguments.

2.1.4 GPIO_MULTIPLE_DRIVER

See section 4.

2.1.5 GPIO_FPGA_ADDR_N

When the target platform is a SoC FPGA, it is likely the GPIO driver has support for GPIO modules added in the FPGA fabric. The build options `GPIO_FPGA_ADDR_`*`N`* (*`N`* can have the value 1 to 10) specifies the base address in the memory space of the processor of up to ten (10) FPGA GPIO controllers. The numbering (*`N`* in the build option name) must start at 1 and be contiguous. By default, none of the ten (10) `GPIO_FPGA_ADDR_`*`N`* build options are defined.

For example, if two GPIO modules are added in the FPGA fabric and the base addresses of the modules are respectively `0xFC300000` and `0xFC301000`, then these 2 build options must be defined with these values:

Table 2-2 FPGA GPIO module example

Build Option Name	Value	Description
`GPIO_FPGA_ADDR_1`	`0xFC300000`	FPGA GPIO module located at address `0xFC300000`. The I/O pin numbers for that module start at 1000
`GPIO_FPGA_ADDR_2`	`0xFC301000`	FPGA GPIO module located at address `0xFC301000`. The I/O pin numbers for that module start at 2000

3 Model

GPIO controllers with many IO pins are commonly broken down into 16 or 32 bit wide individual controllers or regions, called in here “banks”. To increase the portability and simplify the API, only I/O pin numbers, starting at 0 and incrementing contiguously, are used; bank numbers are never used in the driver interface. Refer to the header of the code itself for the mapping between the I/O pin numbers and pair of banks and I/O numbers.

When the build option(s) `GPIO_FPGA_ADDR_`*`N`* is (are) used, the individual FPGA controller I/O pin numbers start on exact multiple of 1000. For example, for the controller at address `GPIO_FPGA_ADDR_1`, the first IO pin number is 1000, next IO pin is 1001, etc. For the controller at address `GPIO_FPGA_ADDR_2`, the first IO pin number is 2000, etc up to 10000 for `GPIO_FPGA_ADDR_10`. The drivers have no provisions to check the validity of an I/O pin number for a FPGA module. For example, if a FPGA module with 32 I/O pins is declared with `GPIO_FPAG_ADDR_1`, then any I/O pin value between 1000 and 1999 specified in the API argument will access that module. Using an invalid I/O pin number will either do nothing or crash the application. The checking of the validitiy of the argument (enable with the build option `GPIO_ARG_CHECK`) only applied to non-FPGA I/O pins.

When interrupts are supported with the build option `GPIO_USE_ISR` set to a non-zero value, the application must provide an interrupt call back function (This function is `GPIOintHndl()`, refer to section ???). The call back function is application specific, as only the application knows what to do when an input pin has triggered an interrupt. The driver deals with clearing the interrupt request when possible, e.g. mainly with edge triggered interrupts. For level sensitive interrupts, the clearing of the interrupt request can only be performed in the call back function by changing the level at the source of the signal, or by disabling the associated interrupt. Depending on the target platform, more than one interrupt handlers may have to be attached. Refer to the header in the code itself as this information is provided (I/O pin range are associated with its `GPIOintHndl_`*`n`*`()`)

The configuration of the GPIO pin is done through the `gpio_dir()` (Section ???) and `gpio_cfg()` (Section ???), where the former sets the direction of the pins (input or output) and the later configures the behavior of each pins. `gpio_cfg()` supports these “set-up” through bit fields (multiple configuration can be ORed together). Refer to section ??? for a full description.

4 Multiple Drivers

It is possible to use 2 or more drivers for different GPIO controllers. Example of the need for this is a processor with on-chip GPIO(s) on a board with different type of GPIO(s), or a SocFPGA with custom GPIO(s) added in the FPGA fabrics that are of different type than the processor system GPIO(s). To use multiple drivers the build option `GPIO_MULTIPLE_DRIVER` must be defined and set to a non-zero value. This changes the API names of the driver by pre-pending the GPIO type to the function names. For example, if `alt_gpio.c` is used, the APIs are named as following:

Table 4-1 API remapping

Original	Multiple
`gpio_cfg()`	`alt_gpio_cfg()`
`gpio_dir()`	`alt_gpio_dir()`
`gpio_get()`	`alt_gpio_get()`
`gpio_bank_get()`	`alt_gpio_bank_get()`
`gpio_init()`	`alt_gpio_init()`
`gpio_set()`	`alt_gpio_set()`
`gpio_bank_set()`	`alt_gpio_bank_set()`
`GPIOintHndl()`	`alt_GPIOintHndl()`
`GPIOintHndl_#()`	`alt_GPIOintHndl_#()`

The prefix is always the prefix in the file name; e.g. `alt_gpio.c` prefix is "`alt`" and `xlx_gpio.c` is "`xlx`".

All build options, if not prefixed, apply to all the drivers. To set build options on a per-driver basic all there is to do is used the build option that has been pre-fixed with the same prefix used in the API but in uppercase. For example to set each of the multiple drivers to include the interrupt handling code, the build option `GPIO_USE_ISR` should defined and set to a non-zero value. When a driver specific build option is defined then the driver for which the build option applies ignores the equivalent global build.

A custom wrapper must be provided. The following code shows such a driver for the `alt_gpio` and a fictitious driver `xyz_gpio`. The example maps I/O pin numbers from 0 to 299 to the `alt_gpio` driver, the I/O pin numbers 300 to 999 to the `xyz_gpio` driver and all I/O pin number greater or equal to 1000 to the `alt_gpio`. The I/O pin numbers 1000 and up are for one or more FPGA IP modules.

The wrapper has all the standard GPIO API functions and in these functions, depending on the I/O pin specified in the argument, either the `alt_gpio` driver is used or the `xyz_gpio` driver is used. The I/O pin numbers used are such they don't need to be remapped when the `alt_gpio` driver is called but they have to be remapped for the `xyz_gpio` driver. i.e. when the `xyz_gpio` is called the I/O pin numbers between 300 and 999 (as used with the wrapper), they are remapped to the I/O pin numbers between 0 to 699.

The wrapper also remaps the interrupt handler. In the case of the `alt_gpio` driver, it needs its interrupt handlers #0, #1, #2, #1000 and #2000; these are mapped as is. The `xyz_gpio` needs its interrupt handlers #0 and #1 and these two are remapped by the wrapper to interrupt handlers #3 and #4.

Table 4-2 Multiple GPIO wrapper example (`gpio.c`)

```
#include "gpio.h"

/* ---------------------------------------------------------------------------------- */

int gpio_cfg(int IOpin, int Cfg)
{
int RetVal;

   ReVal = ((IOpin < 300) || (IOpin >= 1000))
         ? alt_gpio_cfg(IOpin, Cfg)
         : xyz_gpio_cfg(IOpin-300, Cfg);

   return(RetVal);
}

/* ---------------------------------------------------------------------------------- */

int gpio_dir(int IOpin, int Dir)
{
int RetVal;

   ReVal = ((IOpin > 300) || (IOpin >= 1000))
         ? alt_gpio_dir(IOpin, Cfg)
         ? xyz_gpio_dir(IOpin-300, Cfg);

   return(RetVal);
}

/* ---------------------------------------------------------------------------------- */

int gpio_get(int IOpin)
{
int RetVal;

   ReVal = ((IOpin < 300) || (IOpin >= 1000))
         ? alt_gpio_get(IOpin)
         ? xyz_gpio_get(IOpin-300);

   return(RetVal);
}

/* ---------------------------------------------------------------------------------- */

int32_t gpio_bank_get(int IOpin, int *Err)
{
int32_t RetVal;

   ReVal = ((IOpin < 300) || (IOpin >= 1000))
         ? alt_gpio_bank_get(IOpin, Err)
         ? xyz_gpio_bank_get(IOpin-300, Err);

   return(RetVal);
}

/* ---------------------------------------------------------------------------------- */

int gpio_init(int Forced)
{
int RetVal;

   RetVal  = alt_gpio_init(Forced);
   RetVal |= xyz_gpio_init(Forced);
   return(RetVal);
}

/* ------------------------------------------------------------------------------------ */

int gpio_set(int IOpin, int Value)
{
int RetVal;

   ReVal = ((IOpin < 300) || (IOpin >= 1000))
         ? alt_gpio_set(IOpin, Value)
         ? xyz_gpio_set(IOpin-300, Value);

   return(RetVal);
}

/* ------------------------------------------------------------------------------------ */

int gpio_bank_set(int IOpin, int32_t Value)
{
int RetVal;

   ReVal = ((IOpin < 300) || (IOpin >= 1000))
         ? alt_gpio_bank_set(IOpin, Value)
         ? xyz_gpio_bank_set(IOpin-300, Value);

   return(RetVal);
}

/* ------------------------------------------------------------------------------------ */

void GPIOintHndl_0(void)    { alt_GPIOintHndl_0();    }
void GPIOintHndl_1(void)    { alt_GPIOintHndl_1();    }
void GPIOintHndl_2(void)    { alt_GPIOintHndl_2();    }
void GPIOintHndl_3(void)    { xyz_GPIOintHndl_0();    }
void GPIOintHndl_4(void)    { xyz_GPIOintHndl_1();    }
void GPIOintHndl_1000(void) { alt_GPIOintHndl_1000(); }
void GPIOintHndl_2000(void) { alt_GPIOintHndl_2000(); }

/* EOF */
```

Table 4-3 Multiple GPIO wrapper example (`gpio.h`)

```
#ifndef __GPIO_H__
#define __GPIO_H__       1

#include "alt_gpio.h"                          /* alt GPIO driver                 */
#include "xyz_gpio.h"                          /* xyz GPIO driver                 */

#ifndef GPIO_MULTI_DRIVER                      /* It must be defined and set to !=0  */
  #define GPIO_MULTI_DRIVER      0             /* Set 0 to trigger the error message */
#endif

#if ((GPIO_MULTI_DRIVER) == 0)
  #error "GPIO_MULTI_DRIVER must be defined and set to a non-zero value"
#endif


/* ---------------------------------------------------------------------------------- */

int gpio_cfg      (int IOpin, int Cfg);
int gpio_dir      (int IOpin, int Dir);
int gpio_get      (int IOpin);
int gpio_bank_get (int IOpin, int *Err);
int gpio_int      (int Forced);
int gpio_set      (int IOpin, int Value);
int gpio_bank_set (int IOpin, int32_t Value);

/* ---------------------------------------------------------------------------------- */

extern void GPIOintHndl_0(void);
extern void GPIOintHndl_1(void);
extern void GPIOintHndl_1000(void);
extern void GPIOintHndl_2000(void);

#endif

/* EOF */
```

5 API

In this section, the API of all common GPIO driver functions is provided.

5.1.1 gpio_cfg

Synopsis

```
#include “???_gpio.h”

int gpio_cfg(int IOpin, int Cfg);
```

Description

`gpio_cfg()` is the component used to configure the operation of a single input I/O pin. The I/O pin number is indicated by the argument `IOpin` and the configuration to apply is indicated by the argument `Cfg`.

Arguments

`IOpin`	I/O pin number (Number starting at 0)
`Cfg`	Configuration bit field, see Options

Returns

`int`	`== 0`, success
	`!= 0`, error

Component type

Function

Options

The argument `Cfg` is an Oring of any of the following tokens:

`GPIO_CFG_ISR_LEVEL_0`:	enable and set the I/O pin interrupt type to level sensitive, with a low input level (0).
`GPIO_CFG_ISR_LEVEL_1`:	enable and set the I/O pin interrupt type to level sensitive, with a low input level (1).
`GPIO_CFG_ISR_EDGE_0_1`:	enable and set the I/O pin interrupt type to edge sensitive, with a transition on the input from low (0) to high (1).
`GPIO_CFG_ISR_EDGE_1_0`:	enable and set the I/O pin interrupt type to edge sensitive, with a transition on the input from high (1) to low (0).
`GPIO_CFG_ISR_EDGE_ANY`:	enable and set the I/O pin interrupt type to edge sensitive, with any transition on the input: from low (0) to high (1) or high (1) to low(0).
`GPIO_CFG_ISR_OFF`:	disable the I/O pin interrupt.
`GPIO_CFG_DEB_OFF`:	disable input debouncing on the I/O pin.
`GPIO_CFG_DEB_ON`:	enable input debouncing on the I/O pin.

Most GPIO controllers don’t support all these type pof configurations. Refer to the header in the code itself for a detailed description of each configuration, to know if it is supported, ignored or report an error.

Notes

`gpio_cfg()` does not control the direction of an I/O pin, it configures the operation of input pins. To control the direction of an I/O pin, refer to `gpio_dir()` (section 5.1.2).

See Also

`gpio_dir()` (Section 5.1.2)
`gpio_get()` (Section 5.1.3)

5.1.2 gpio_dir

Synopsis

```
#include “???_gpio.h”

int gpio_fir(int IOpin, int Dir);
```

Description

`gpio_dir()` is the component used to set an I/O pin as an input or an output pin. The I/O pin number to set up is indicated by the argument `IOpin` and the direction is specified by the argument `Dir`.

Arguments

`IOpin`	I/O pin number (Number starting at 0)	
`Dir`	Direction of the I/O pin:	
	`== 0` or `GPIO_DIR_OUT`:	output direction
	`!= 0` or `GPIO_DIR_IN`:	input direction

Returns

`int`	`== 0`, success
	`!= 0`, error

Component type

Function

Options

Some FPGA IP module can be configured to have I/O pin set to a fixed direction. Refer to the header in the code itself for more details

Notes

`gpio_dir()` does not configure the operation of input pins. To configure theoperation of an input I/O pin, refer to `gpio_cfg()` (section).

See Also

`gpio_cfg()` (Section 5.1.1)
`gpio_get()` (Section 5.1.3)
`gpio_set()` (Section 5.1.6)

5.1.3 gpio_get

Synopsis

```
#include "???_gpio.h"

int gpio_get(int IOpin);
```

Description

`gpio_get()` is the component used to read the value of an input I/O pin. The I/O pin number to read is indicated by the argument `IOpin` and. If the I/O pin is an output pin, the value returned could either be the value held in the output pin register, or the value at the output pin itself, or it could be invalid. The result is controller dependent

Arguments

`IOpin`	I/O pin number (Number starting at 0)

Returns

`int`	== `0`, the input at the I/O pin is a low level (0)
	== `1`, the input at the I/O pin is a high level (1)
	< `0`, error (invalid I/O pin number)

Component type

Function

Options

Notes

See Also

`gpio_dir()` (Section 5.1.2)
`gpio_bank_get()` (Section 5.1.4)
`gpio_set()` (Section 5.1.6)
`gpio_bank_set()` (Section 5.1.7)

5.1.4 gpio_bank_get

Synopsis

```
#include "???_gpio.h"

int32_t gpio_bank_get(int IOpin, int *Err);
```

Description

`gpio_bank_get()` is the component used to read the value of a GPIO bank (the group of I/O pins in the same GPIO controller). The bank number to read is indicated by the argument `IOpin`: `IOpin` can have the value of any of the I/O pins located in desired bank to read. The argument `Err` indicates if there is a read error, which can be either an invalid `IOpin` value or a write only bank.

Arguments

`IOpin`	Any I/O pin number (Number starting at 0) in the bank to read
`Err`	pointer to an `int` reporting success or error `== 0` : read successful `!= 0` : error `Err` can be `NULL`, which will not be reported

Returns

`int32_t`	When successful `(*Err == 0)` value read Undetermined upon error (`*Err != 0`)

Component type

Function

Options

Notes

See Also

`gpio_dir()` (Section 5.1.2)
`gpio_get()` (Section 5.1.3)
`gpio_set()` (Section 5.1.6)
`gpio_bank_set()` (Section 5.1.7)

5.1.5 gpio_init

Synopsis

```
#include "???_gpio.h"

int gpio_init(int Forced);
```

Description

`gpio_init()` is the component used to initialize a GPIO module. A single argument is used, Forced, and it indicates if the GPIO has to be re-initialized if it has already been initialized.

Arguments

`Forced`	`!= 0`, forces an initialization, even of already initialized
	`== 0`, initializes the module only if has never been initialized

Returns

`int`	`== 0`, success
	`!= 0`, error

Component type

Function

Options

Notes

See Also

5.1.6 gpio_set

Synopsis

```
#include "???_gpio.h"

int gpio_set(int IOpin, int Value);
```

Description

`gpio_init()` is the component used to set the output value of an output I/O pin. The I/O pin number to set is indicated by the argument `IOpin` and the value ot set is specified with tah argument `Value`. If the I/O pin is an input pin, the result is controller dependent but most likely will do nothing.

Arguments

`IOpin`	I/O pin number (Number starting at 0)
`Value`	value to set on the output pin:
	`== 0`, the output pin is set to a low level (0)
	`!= 0`, the output pin is set to a high level (1)

Returns

`int`	`== 0`, success
	`!= 0`, error

Component type

Function

Options

Notes

See Also

`gpio_dir()` (Section 5.1.2)
`gpio_get()` (Section 5.1.3)
`gpio_bank_get()` (Section 5.1.4)
`gpio_bank_set()` (Section 5.1.7)

5.1.7 gpio_bank_set

Synopsis

```
#include "???_gpio.h"

int gpio_bank_set(int IOpin, int32_t Value);
```

Description

`gpio_bank_set()` is the component used to write a value to a GPIO bank (the group of I/O pins in the same GPIO controller). The bank number to write to is indicated by the argument `IOpin`: `IOpin` can have the value of any of the I/O pins located in desired bank to write to. The argument `Value` specifies the value to write.

Arguments

`IOpin`	Any I/O pin number (Number starting at 0) in the bank to read
`Value`	value to set on the output pins

Returns

`int`	`== 0`, success
	`!= 0`, error

Component type

Function

Options

Notes

See Also

`gpio_dir()` (Section 5.1.2)
`gpio_get()` (Section 5.1.3)
`gpio_bank_get()` (Section 5.1.4)
`gpio_set()` (Section 5.1.6)

5.1.8 GPIOintHndl

Synopsis

```
#include "???_GPIO.h"

void GPIOintHndl_n(int IOpin);
```

Description

`GPIOintHndl()` is the call back function that must be provided when interrupts are used with a GPIO driver (refer to section 2.1.2). The sole argument is the I/O pin number that has triggered the interrupt.
Important: it is not to be confused with the GPIO interrupt handlers themselves, which have an alike name but with a number: `GPIOintHndl_`*`n`*.

Arguments

`IOpin`	I/O pin number (Number starting at 0) that has trigger an interrupt

Returns

`void`

Component type

Function

Options

Notes

`GPIOintHndl()` is called within an interrupt context, therefore care must be taken to only use services available in an interrupt.
When an I/O pin is configured to trigger an interrupt on edge, the driver deals with the removal of the interrupt request. When an I/O pin is configured to trigger an interrupt on a level, the driver cannot deal with the removal of the interrupt request so the call back function must perform the necessary to remove the interrupt request. This can be done by either removing the cause of the triggering level or by disabling the interrupt on the I/O pin.

See Also

`GPIOintHandl_n()` (Section 5.1.9)

5.1.9 GPIOintHndl_*n*

Synopsis

```
#include "???_gpio.h"

void GPIOintHndl_n(void);
```

Description

`GPIOintHndl_n()` is (are) the interrupt handler(s) used by the GPIO driver. Some target platforms only need a single interrupt handler (i.e. `GPIOintHndl_0()`) when other requires multiple interrupt handlers. Refer to the header in the code for a detailed description of the requirements.
All there is to do with these functions is to attach them to the associated interrupt from the interrupt controller.
Important: it is not to be confused with the GPIO interrupt call back function, which have an alike name but without a number: `GPIOintHndl`.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

The GPIO interrupt call-back function (`GPIOintHdnl()`) must be provided when interrupts are used with the GPIO driver.

See Also

`GPIOintHandl()` (Section 5.1.8)

6 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com