DSP/BIOS GPIO Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Madhvapathi Sriram	September 11 2008	Created the document
0.2	Madhvapathi Sriram	October 15, 2008	Revised for Release 2.00.01.01
0.3	Imtiaz SMA	January 21, 2009	Updated the document for the IOM driver and IDriver contrast sections

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for GPIO peripheral using DSP/BIOS operating system running on DSP C674x

Note: The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

1.1 Terms and Abbreviations

Term	Description
API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System

1.1 Disclaimer

This is a design document for the GPIO driver for the DSP/BIOS operating system. Although the current design document explain the GPIO driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation as the BIOS 5.x driver is a direct port of BIOS 6.x driver. The BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IDriver model. The subsequent section explains how this document can be used to understand and modify the IOM drivers found in this product. Please note that all the flowcharts, structures and functions described here in this document are equally applicable to the GPIO driver 5.x.

1.2 IOM driver Vs IDriver

The following are the main difference between the BIOS 5.x and BIOS 6.x driver. Please refer to the reference documents for more details in the IOM driver model.

1. All the references to the stream module should be treated as references to a module that provides data streaming. In BIOS 5.x the equivalent modules are SIO and GIO.

2. This document refers to the IDriver model supported by the BIOS 6.x.All the references to the IDriver should be assumed to be equivalent to the IOM driver model.

3. The BIOS 6.x driver uses a module specifications file (*.xdc) for the declaration of the enumerations, structures and various constants required by the driver. The equivalent of this xdc file is the header file XXX.h and the XXXLocal.h.

   Note: The XXXLocal.h file contains all the declaration specified in the "internal" section of the corresponding xdc file.

4. In BIOS 6.x creation of static driver instances follow a different flow and cause functions in module script files to run during build time. In BIOS 5.x creation of driver (for both static and dynamic instances) result in the execution of the mdBindDev function at runtime. Therefore any references to module script files (*.xs files) can be ignored for IOM drivers.

5. The XXX_Module_startup function referenced in this document can be ignored for IOM drivers.

6. IOM drivers have an XXX_init function which needs to be called by the application once per driver. This XXX_init function initializes the driver data structures. This application needs to call this function in the application initialization functions which are usually supplied in the tci file.

7. The functionality and behavior of the functions is the same for both driver models. The mapping of IDriver functions to IOM driver functions are as follows:

IDriver	IOM driver
XXX_Instance_init	mdBindDev
XXX_Instance_finalize	mdUnbindDev
XXX_open	mdCreateChan
XXX_close	mdDeleteChan
XXX_control	mdControlChan
XXX_submit	mdSubmitChan

8. All the references to module wide config parameters in the IDriver model map to macro definitions and preprocessor directives (#define and #ifdef etc) for the IOM drivers. e.g.

   a. XXX_edmaEnable in IDriver maps to –D XXX_EDMA_ENABLE for IOM driver.

   b. XXX_FIFO_SIZE in IDriver maps to #define FIFO_SIZE in IOM driver header file

9. In BIOS 6.x a cfg file is used for configuring the BIOS and driver options whereas in the BIOS 5.x the “tcf” and “tci” files are used for configuring the options.

10. In BIOS 5.x driver support for multiple devices is implemented as follows. A chip specific compiler define is required by the driver source files (-DCHIP_OMAPL138). Based on the include a chip specific header file (e.g soc_OMAPL138) which contains chip specific defines is used by the driver. In order to support a new chip, a new soc_XXX header file is required and driver sources files need to be changed in places where the chip specific define is used.

1.3 Related Documents

1.	TBD	DSP/BIOS Driver Developer’s Guide
3.	SPRUFL8	GPIO Specs

1.4 Hardware

The GPIO device driver design is in the context of DSP/BIOS running on DSP C674x core

The GPIO module core used here has the following bocks:

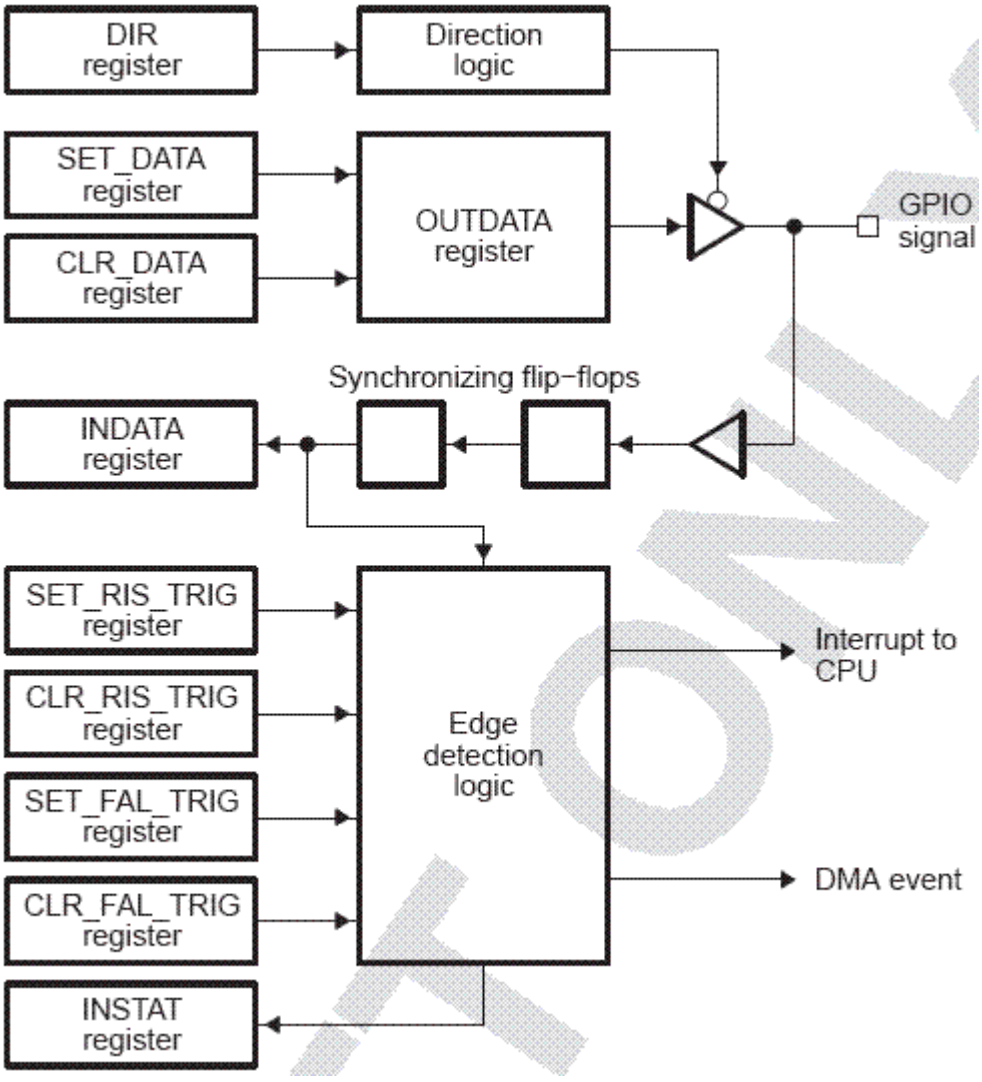

Figure 1 GPIO Block Diagram

1.5 Software

The GPIO driver discussed here is running DSP/BIOS on the C674x DSP.

1.5.1 Operating Environment and dependencies

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

1.5.2 System Architechture

The block diagram below shows the overall system architecture.

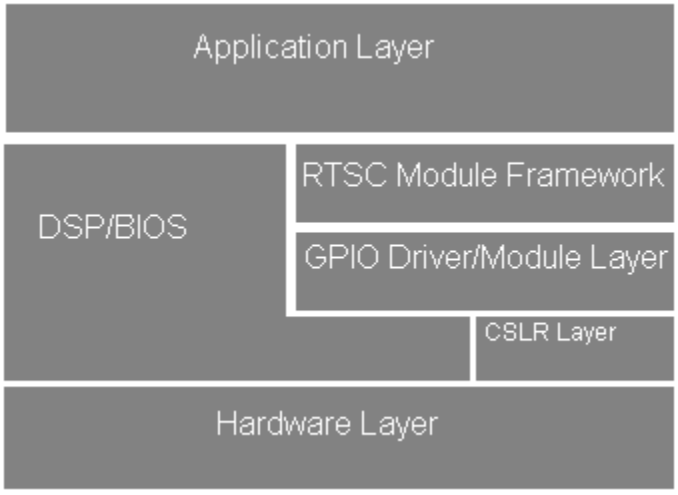

Figure 2 System Architecture

The GPIO driver is designed as a standalone mode, unlike those inheriting/using IDriver framework. The rationale behind this strategy is that the GPIO module exposes APIs which could be used by other driver modules also, apart from the application. In such a scenario, accessing the GPIO APIs incase of a IDriver inherited framework would be cumbersome, viz passing through IDriver tables etc. Hence, the design adopts the standalone module concept as used by other DSP/BIOS6 modules like Clock, Hwi etc, which expose functional APIs to access hardware.

Figure 2 shows the overall device driver architecture.

1.6 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. Refer to GPIO device driver API reference documentation for complete details on APIs.

1.6.1 Module Interface

The Module interface constitutes the Device Driver Manifest to Application, as a standalone component whose APIs can be directly called from an application or a driver.

The Module interface projects the GPIO driver as a standalone module. Hence, driver and module could be used interchangeably in the current context.

GPIO module specification file (GPIO.xdc)

The XDC file defines the following in its public section: the data structures, enums, constants, IOCTLS, error codes and module wide config variables that shall be exposed for the user.

These definitions would include

ENUMS: GPIO Direction, GPIO Availability etc

STRUCTURES: Bank parameters, Command Argument Structures etc

CONSTANTS: error ids and ioctls

Also this files specifies the list of configurable items which could be configured/specified by the application during instantiation (instance parameters)

In its private section it would contain the data structures, enums, constants and module wide config variables. The Instance object (the driver object) contains all the info related to that instance. This information might be irrelevant to the User. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application

The RTSC framework translates this into the driver header file (GPIO.h) and this header file shall be included by the applications, for referring to any of the driver data structures/components. Hence, XDC file contains everything that should be exposed to the application and also accessed by the driver.

Please note that by nature of the specification of the xdc file, all the variables (independent or part of structure) need to be initialized in xdc file itself.

GPIO module script file (Gpio.xs)

The script file is the place where static instantiations and references to module usage are handled. This script file is invoked when the application compiles and refers to the GPIO module/driver. The GPIO.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg for the Gpio module, the module use function in the Gpio.xs file is used to initialize the hardware instance specific details like base addresses, interrupt numbers, frequency etc. This data is stored for further use during instantiation. This gives the flexibility to design, to handle multiple SOC with single c-code base (as long as the IP does not deviate).

2. Handling static instantiation of the GPIO instance

   When the GPIO instance is instantiated statically in the application cfg file, (please note that dynamic instantiation is also possible from a Cfile) ,the instance static init function is called. Here the instance state is populated based on the instance number, like the default state of the driver, default values of interrupt numbers etc.

The GPIO module implements the following interfaces

S.No	IOM Interfaces	Description
1	Gpio_Instance_init()	Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in instance static init in the module script file.
2	Gpio_Instance_finalize()	Does the final clean up before the module instance is destroyed.
3	Gpio_setPinDir ()	Set a GPIO pin as input/output
4	Gpio_setPinVal ()	Set value at a GPIO pin (one/zero)
5	Gpio_getPinVal ()	Get the value present at a GPIO pin (one/zero)

5	Gpio_SetGroupVal ()	Set the value to a group of GPIO pins in a bank
6	Gpio_getGroupVal()	Get the value at a group of GPIO pins in a bank
7	Gpio_setRisingEdgeTrigger()	Set the rising edge trigger for interrupt from a pin
8	Gpio_setFallingEdgeTrigger()	Set the falling edge trigger for interrupt from a pin
9	Gpio_clearRisingEdgeTrigger()	Clear the rising edge trigger for interrupt from pin
10	Gpio_clearFallingEdgeTrigger()	Clear the fallin edge trigger for interrupt from pin
11	Gpio_bankInterruptEnable()	Enable interrupts from a bank
12	Gpio_bankInterruptDisable()	Disable interrupts from a bank
13	Gpio_regIntHandler()	Register an interrupt handler for a GPIO bank/pin interrupt
14	Gpio_unregIntHandler()	Unregister an existing interrupt handler for a GPIO bank/pin interrupt
15	Gpio_setPinUseStatus	Set the in use (availability) status for a pin
16	Gpio_getPinUseStatus	Get the in use (availability) status for a pin
17	Gpio_setBankinUseStatus	Set the in use (availability) status for a bank
18	Gpio_getBankInUseStatus	Get the in use (availability) status for a bank

1.6.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the GPIO module to configure GPIO registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure.

1.7 Design Philosophy

This device driver is written in conformance to the DSP/BIOS standalone model and handles access to and from the GPIO hardware.

1.7.1 The Module and Instance Concept

The RTSC framework provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation.

The module word usage (GPIO module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall be a module variable. Default parameters shall be a module wide variable since, it should be available during module start up or use call, to initialize all the channels to their default values.

This instance word usage (GPIO instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent the device directly by holding info like the individual GPIO bank info, device configuration settings, hardware configuration etc. Each hardware instance shall map to one GPIO instance. This is represented by the Instance_State in the GPIO module configuration file.

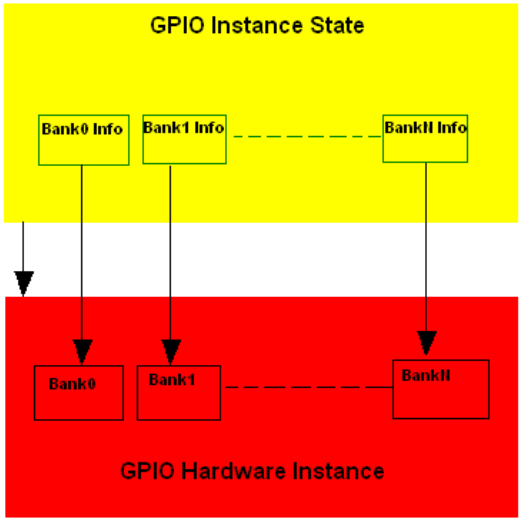

Figure 3 Instance Mapping

Hence every module shall only support the as many number of instantiations as the number of GPIO hardware instances on the SOC

1.7.2 Design Constraints

GPIO standalone driver module imposes the following constraint(s).

- GPIO driver shall only expose APIs for GPIO hardware access. It shall not support exclusive operations like a data transfer mechanism over the pin/group of pins in a bank. Any application/module requiring such features shall wrap the basic set/get pin/bank value APIs into its own transfer function APIs.

- The GPIO driver provided shall for now support only fixed number of banks (eight banks) due to constraint from the RTSC framework on setting the lengths for variable length arrays for BankParams.

2 GPIO Driver Software Architecture

This section details the data structures used in the GPIO module and the interface it presents to the application layer. A diagrammatic representation of the GPIO module functions is presented and then the usage scenario is discussed in some more detail.

Following this, we'll discuss the deployed driver or the dynamic view of the driver where the driver operational scenarios are presented.

2.1 Static View

2.1.1 Functional Decomposition

The driver is designed keeping a device, also called instance, in mind. It is designed to be a standalone module.

This driver uses an internal data structure, BankInfo, to maintain the information of each bank and its associated pins. This information is populated to default values during module instantiation and later preserves the changes as done by the user. This bank information is held inside the Instance State of the module. However, this instance state is translated to the Gpio_Object structure by the RTSC frame work. The data structures used to maintain this information are explained in greater detail in the following *Data Structures* sub-section. The following figure shows the static view of GPIO driver.

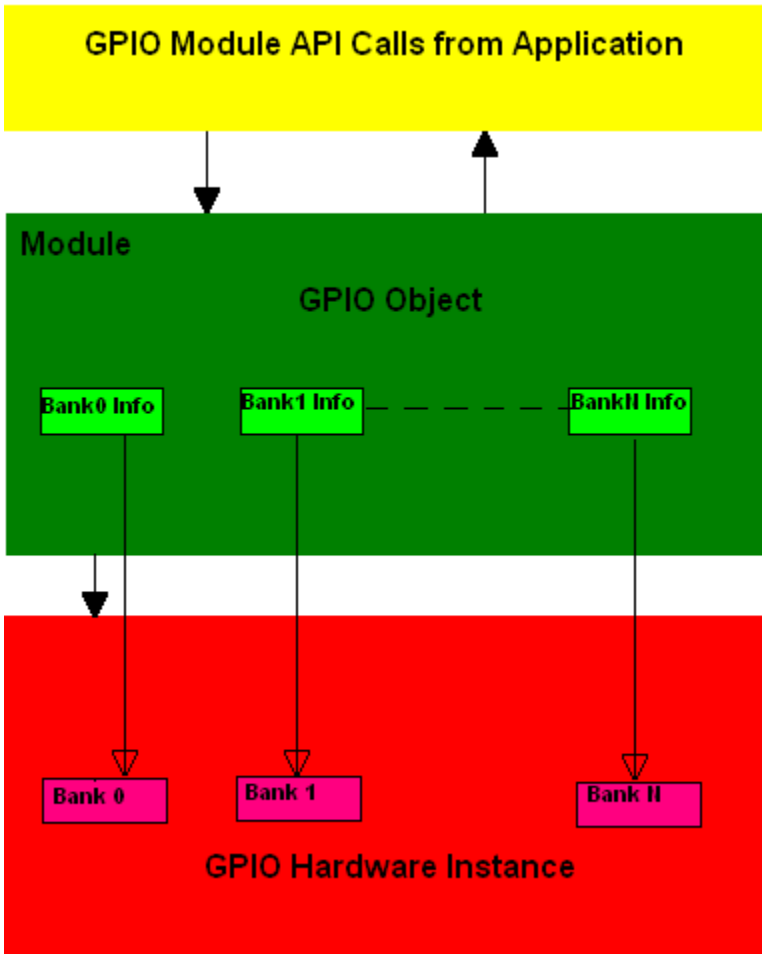

Figure 4 GPIO driver static view

2.1.2 Data Structures

The module employs the Instance State (Gpio_Object) and BankInfo structures to maintain information of the instance and banks respectively.

In addition, the driver has one more structure defined – bank parameters (BankParams). The BankParams structure is used to pass on data to initialize the driver during module start up or initialization. This params structure is used to specify if a GPIO bank/pin is available for use in the system, the hardware interrupt number assigned to this bank/pin event etc while instantiation.

The following sections provide major data structures maintained by GPIO module and the instance.

2.1.2.1 The Instance State (Gpio_Object)

The instance state comprises of all data structures and variables that logically represent the actual hardware instance on the hardware. It preserves the input and output channels for transmit and receive, parameters for the instance etc. The handle to this is sent out to the application for access when the module instantiation is done via create statically (application CFG file) or dynamically (C file during run time). The parameters that are to be passed for this call is described in the section Device Parameters

S.No	Structure Elements *(Gpio_Object)*	Description
1	*instNum*	Preserve the instance number of the GPIO hardware instance
2	*BankInfo*	Preserve the dynamic bank information
3	*BaseAddress*	The base hardware register address for this GPIO instance

2.1.2.2 The Bank Information Structure (Bank)

The information about each bank and its associated pins is necessary to be preserved in the driver to prevent invalid access, clash of (re)use of pin/bank, interrupt enable status, interrupt numbers etc.

S.No	Structure Elements (Gpio_Bank*)*	Description
1	*bankNum*	The number of this bank
2	*inUse*	Boolean to check/mark if this bank is available for use or

		already designated
3	*intEnabled*	Boolean to check/mark if the interrupt (if applicable) for this bank is enabled
4	*PinInfo[]*	Preserve the information of the pins associated with this bank (of Type Gpio_Pin)
5	*evtNum*	CPU interrupt event number (if applicable) assigned for this bank
6	*hwiNum*	The hardware interrupt number assigned by the user for this bank CPU event number
7	*isrHandler*	The interrupt handler registered to this bank CPU event

2.1.2.3 The Pin Information Structure (Pin)

The information about each pin in a bank is necessary to be preserved in the driver to prevent invalid access, clash of (re)use, interrupt enable status, interrupt numbers etc.

S.No	Structure Elements (Gpio_Pin*)*	Description
1	*inUse*	Boolean to check/mark if this bank is available for use or already designated
2	*intEnabled*	Boolean to check/mark if the interrupt (if applicable) for this pin is enabled
3	*evtNum*	CPU interrupt event number (if applicable) assigned for this pin
4	*hwiNum*	The hardware interrupt number assigned by the user for this pin CPU event number

5	*isrHandler*	The interrupt handler registered to this pin CPU event

2.1.2.4 The Bank Parameters

During module instantiation a set of parameters are required which shall be used to keep track of information like which GPIO bank/pin is available for general purpose use and hardware interrupt number associated for that pin/bank CPU interrupt event. This is passed via create call for the module instance. These parameters are passed via BankParams structure and preserved in the BankInfo structure. The BankParams are of type BankConfig The user only need to provide the info for banks/pins which need to updated as free and HWI numbers. By default the driver maintains that all the GPIO bank/pins are busy and no HWI interrupt is assigned to the pin/bank CPU interrupt events. Please note that there are two ways to create an instance (static-using CFG file and dynamic using application c file) and each of these methods have different way of passing devparams. For further information of these methods please refer xdc documentation

S.No	Structure Elements (Gpio_BankConfig)	Description
1	*PinConfInfo[]*	This member provides the pin config/parameters per bank
2	*hwiNum*	Hardware interrupt number for this bank interrupt event (if applicable)
3	*inUse*	Boolean to check/mark if this bank is available for use or already designated

2.1.2.5 The Pin Parameters

The pin parameters are passed as a member of Bank parameters shown above via PinConfInfo. This data is of type PinConfig and explained below

S.No	Structure Elements (Gpio_BankConfig)	Description
1	*free*	Boolean to check/mark if this pin is available for use or already designated
2	*hwiNum*	Hardware interrupt number for this pin interrupt event (if applicable)

2.1.2.6 The Bank and Pin event numbers information

The bank and pin event numbers information forms an important data throughout the driver module. This information is used to enable the events during interrupt handler registration, validating the interrupt handler registration itself etc. The bank and pin event numbers are static information, i.e., the information is known and fixed for a SoC. For example C6747 has only bank events and their numbers are know and fixed. This information is thus maintained in the driver while commissioning the driver itself by the owner of the driver module. The arrays used for storing this information are ***Gpio_BankEvtNum[NUM_BANKS]*** and ***Gpio_PinEvtnum[NUM_BANKS][NUM_PINS_PER_BANK]***, where the first dimension in PinEvtnum is the bank index the pin belongs to and the second dimension is the pin number in that bank.

2.2 Dynamic View

2.2.1 The Execution Threads

The GPIO module involves following execution threads:

Application thread: Instantiation of GPIO module instance, GPIO Bank/Pin operations, will be under application thread.

Interrupt context: Though the GPIO module provides the facility of (un)registering the interrupt handlers, the handlers are application supplied functions and hence are in the application thread context. Hence, the GPIO does not involve in any interrupt processing. This is because, the functionality used over a pin/bank is application specific and each pin/bank can be used in different user context and scenarios. So, it is best to be left to the user to process the interrupt arriving on a bank/pin.

2.2.2 Input / Output using GPIO driver

In GPIO, the application can perform IO operation using Gpio_[set/get] PinVal and Gpio_[set/get]GroupVal APIs. The Gpio_Object should be provided along with Pin or Bank command argument which contains the information like the pin or pin on which the operation is being performed and the value being assigned. These APIs shall aid only in getting and setting values at the specified pins. Any application requiring full featured data transfer operations shall have their own wrappers around these APIs and implement the functionality. This is because, the usage scenario for the general purpose pins/banks is entirely dependent on the application use case. Only the set/get values generalization is provided.

2.2.3 Functional Decomposition

2.2.3.1 module$use() in XS file

One caveat in the RTSC framework design of this package is that we have designed to have a soc.xs capsule file, instead of a soc.h file, which will have the SoC specific information, like interrupt/event numbers, base addresses, CPU/module frequency values etc. This move is adopted to keep the driver C code free from compiler switches and everything of this sort in the form of either configuration variables for the module or loading the platform specific data from the soc.xs capsule. This loading of data is done in the module use function.

The module shall have an array of device instance configuration structures (default DeviceInfo), which shall contain, base address, event number, frequency and such instance specific details. The length of this is defined by the array member of this instance in the soc.xs file.

The default device information is populated for all the instance numbers in this function since this information is needed to later prepare the instances in instance static init and the instance init functions.

2.2.3.2 instance$static$init() in XS file

This function context is the where the instance statically created is initialized.

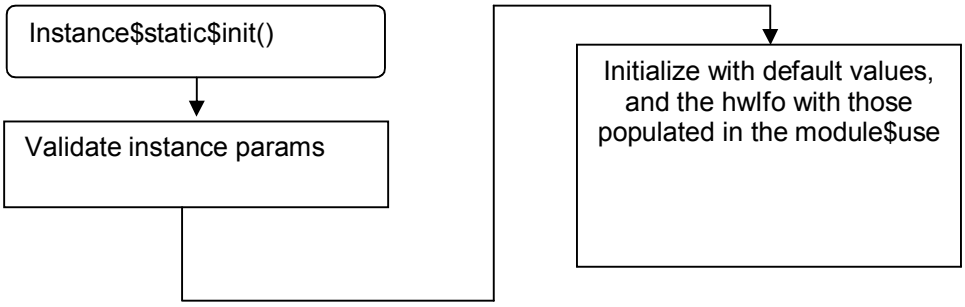

Figure 5 instance$static$init() flow diagram

2.2.3.3 Gpio_Instance_init()

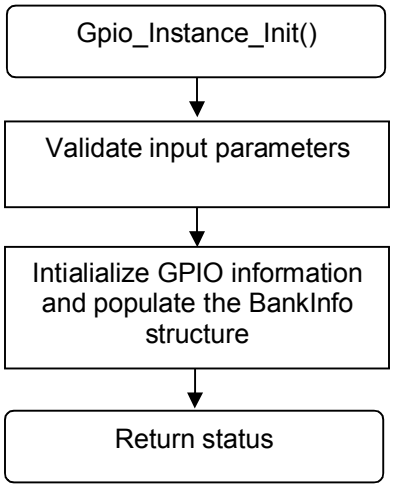

Figure 6 Gpio_Instance_Init() flow diagram

2.2.3.4 Gpio_Instance_finalize()

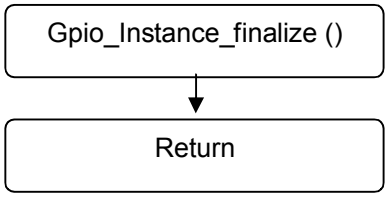

Figure 7 Gpio_Instance_finalize () flow diagram

The Gpio_Instance_init and Gpio_Instance_finalize functions are called by the RTSC/BIOS framework.

The instance init function does a start up initialization for the driver. This function is called when the module is instantiated dynamically by the Gpio_create call by the application. At this module instantiation, the user supplied information about the GPIO banks and pins in the parameters is used to update the GPIO BankInfo maintained by the driver. The user only needs to provide the GPIO bank or pin info details that need to be updated as free during instantiation. By default all are marked as busy and as no h/w interrupt numbers are assigned for the pin/bank events.

The instance finalize function does a final clean up before the driver could be relinquished of any use. However, in the present stage of the module, we do not allocate any resources. Hence the instance finalize does not do anything particularly and just returns.

2.2.3.5 Gpio_setPinDir ()

The GPIO module should provide the user a means for setting the direction of a GPIO pin as input or output. This API provides the same.

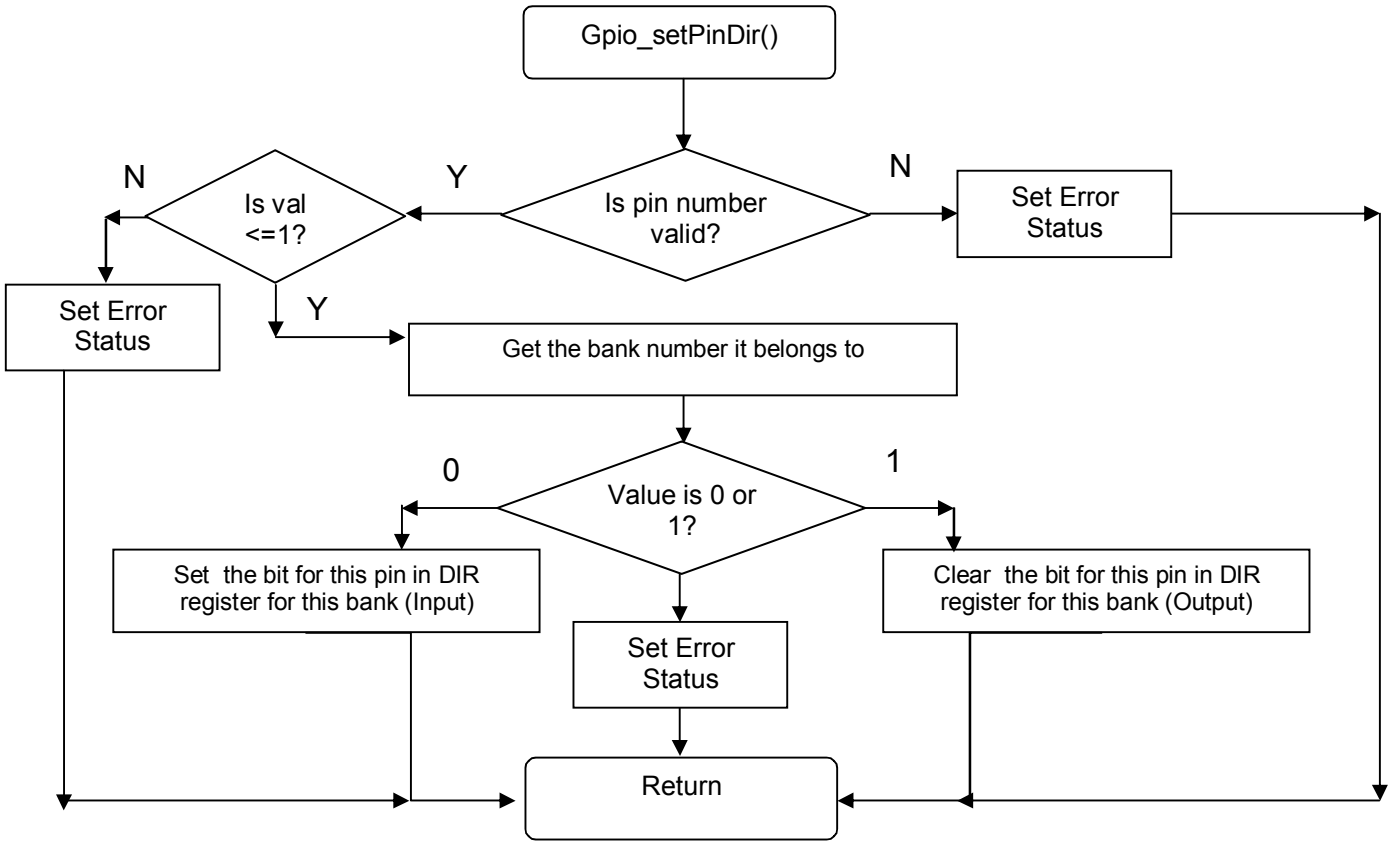

Figure 8 Gpio_setPinDir () flow diagram

2.2.3.6 Gpio_setPinVal()

This API shall allow the user to set a value of 0 or 1 at the pin specified. Hence any other value given by the user for this pin is invalid.

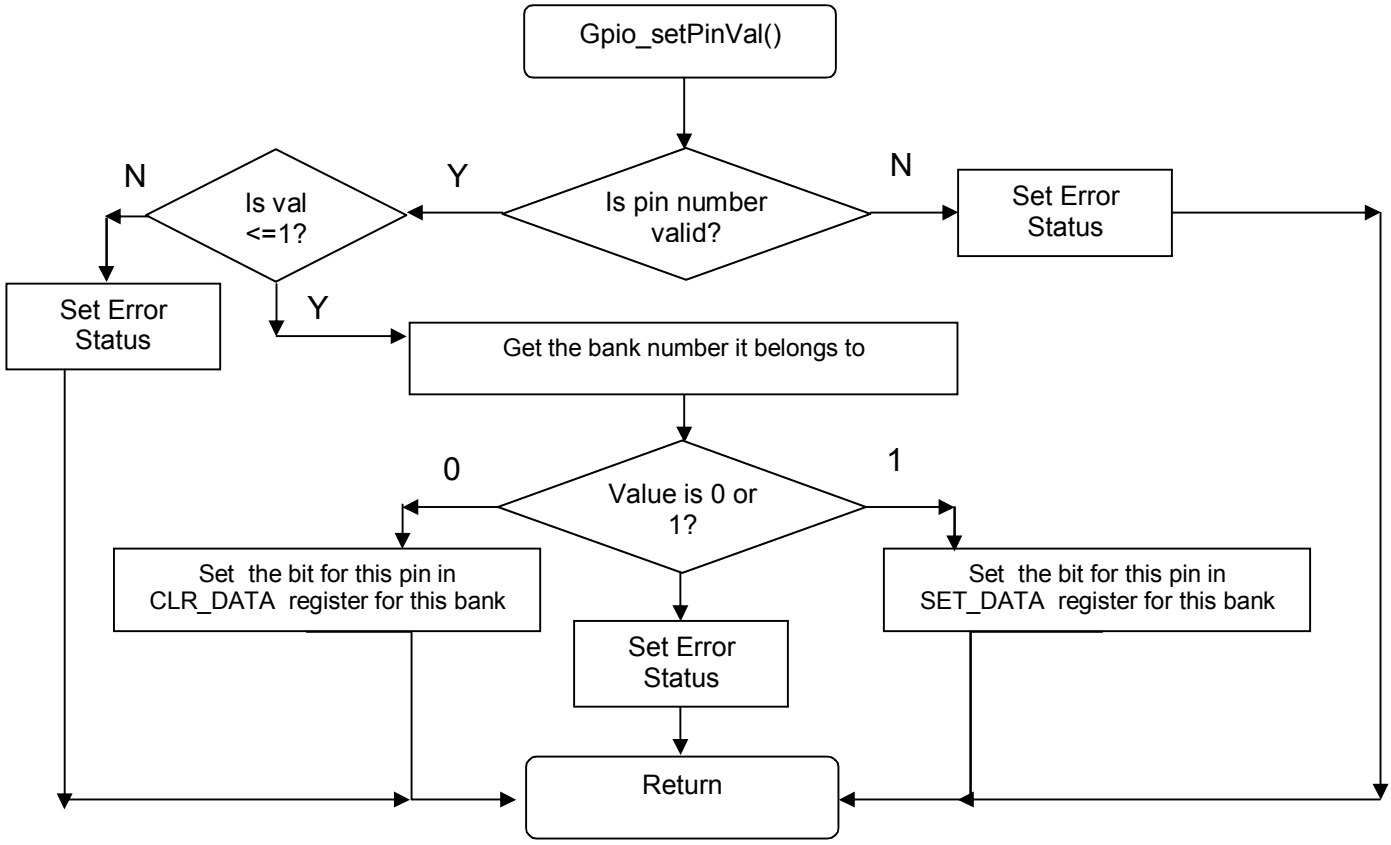

Figure 9 Gpio_setPinVal() flow diagram

2.2.3.7 Gpio_getPinVal()

This API shall allow the user to get the current value at a GPIO pin. The pin number should be specified in the pin command argument

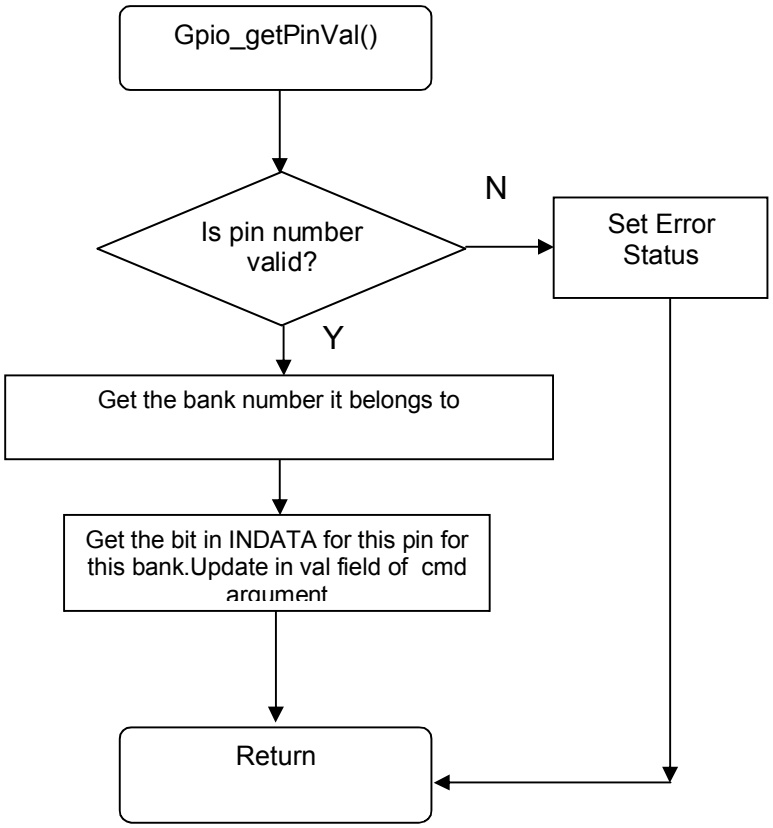

Figure 10 Gpio_getPinVal() flow diagram

2.2.3.8 Gpio_setGroupVal()

This API shall allow the user to set given value to a group of pins in a bank. The pins must belong to same bank. This API shall aid the user in designing application specific multi-bit data transfer functions over GPIO pins. Here the mask for pins that are being set should be given and the value being set should be given.

The mask prevents writes to irrelevant pins in this context and the value is the value ot be written. When the SET_DATA register is written with this masked value, the pins corresponding to the 1's in the value get set. However, for the 0's in the value to reflect on the corresponding pins, we need to write 1 to that bit in the CLR_DATA register. This is done by 1's complement of the value given and writing it to the CLR_DATA register. Note that writing zero's to SET/CLR_DATA register does not have any effect on the pin values.

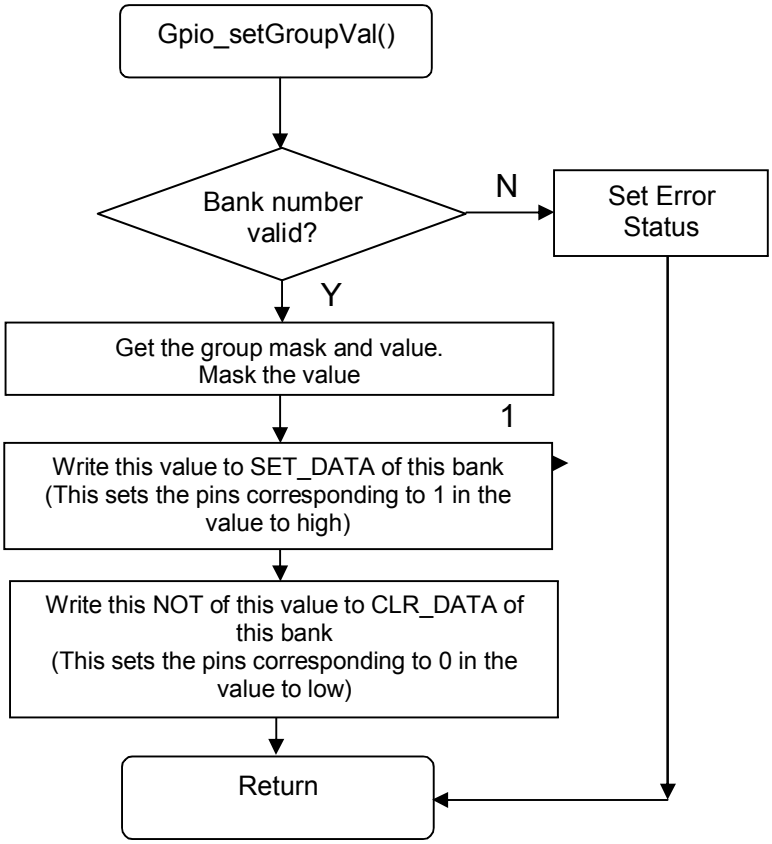

Figure 11 Gpio_setGroupVal() flow diagram

2.2.3.9 Gpio_getGroupVal()

This API shall allow the user to get values currently at a group of pins in a bank. The pins must belong to same bank. This API shall aid the user in designing application specific multi-bit data transfer functions over GPIO pins. Here no mask is required. The user shall get the data of all the pins in the bank and he may mask as required in the application. The group value isobtained by reading the IN_DATA register which contains the actual data present on the pins.

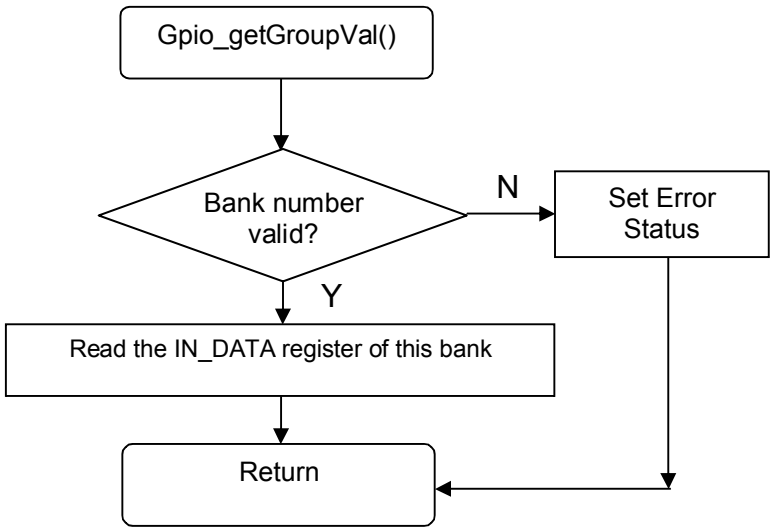

Figure 12 Gpio_getGroupVal() flow diagram

2.2.3.10 Gpio_setRisingEdgeTrigger()

This API shall be used by the user to set the rising edge trigger for the interrupt generation from a GPIO pin. When the rising edge trigger is set, the interrupt is generated at the rising edge state change at this pin. This is done by writing 1 to the SETRISE register of the bank.

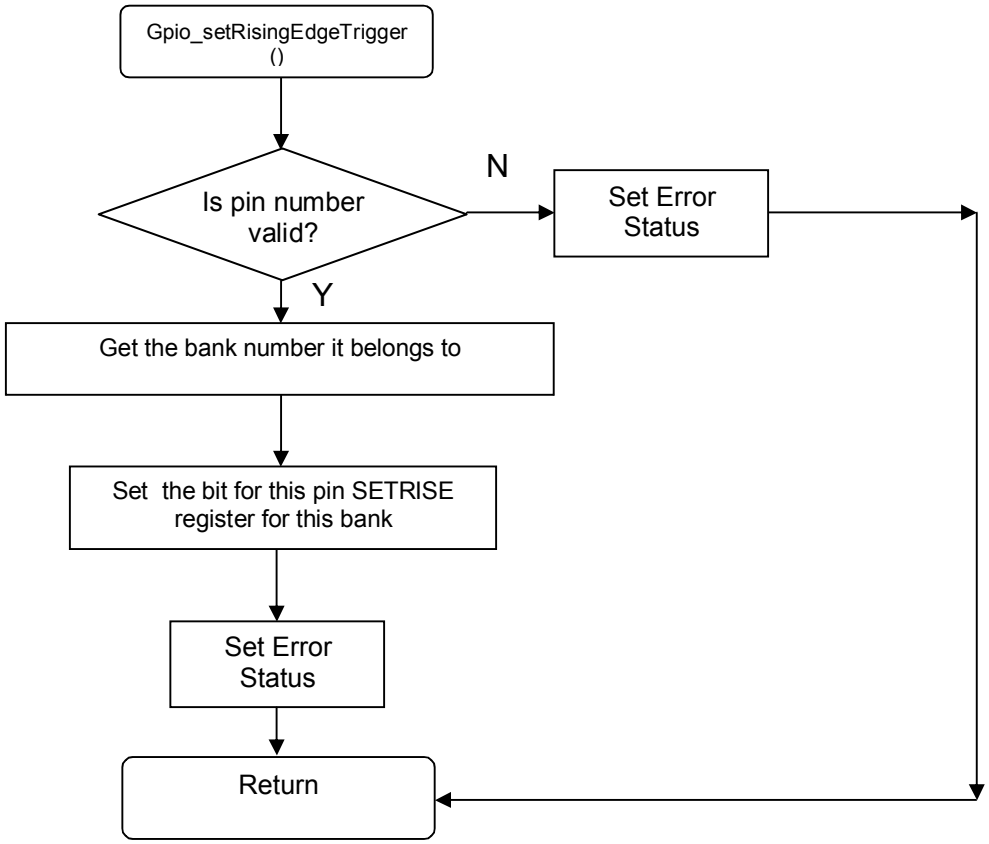

Figure 13 Gpio_setRisingEdgeTrigger() flow diagram

2.2.3.11 Gpio_clearRisingEdgeTrigger()

This API shall be used by the user to clear the rising edge trigger for the interrupt generation from a GPIO pin. When the rising edge trigger is clear, the interrupt is not generated at the rising edge state change at this pin. This is done by writing 1 to the CLRRISE register of the bank.

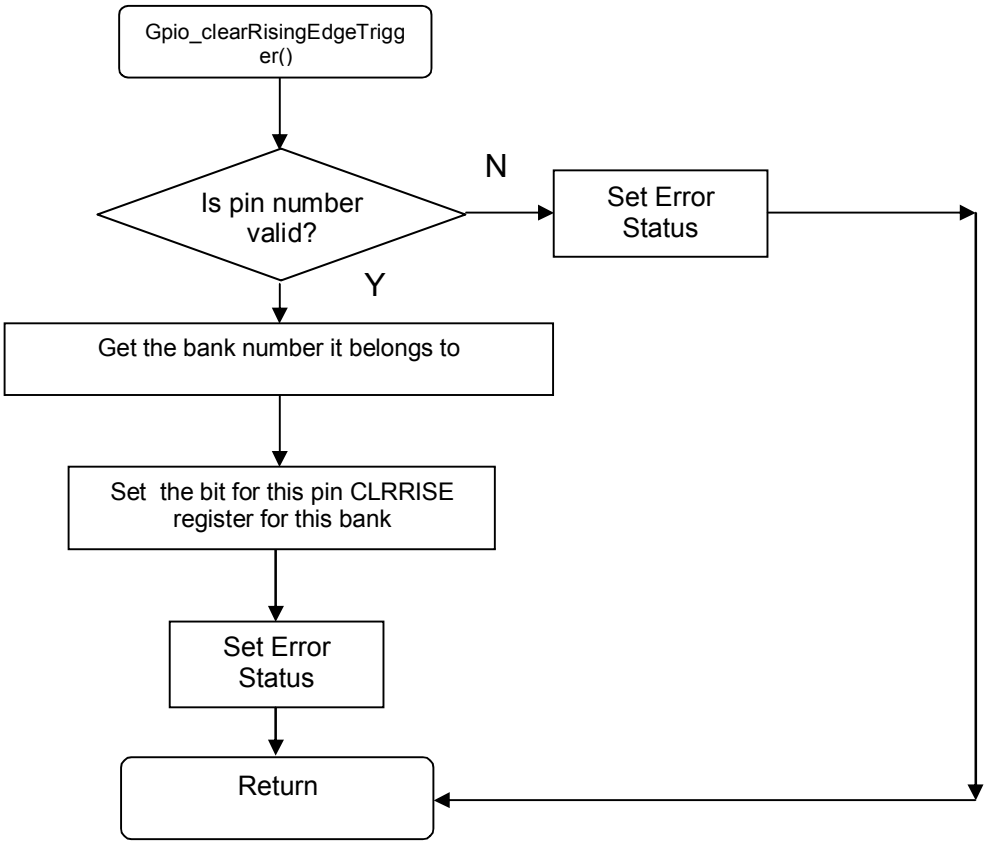

Figure 14 Gpio_clearRisingEdgeTrigger() flow diagram

2.2.3.12 Gpio_setFallingEdgeTrigger()

This API shall be used by the user to set the falling edge trigger for the interrupt generation from a GPIO pin. When the falling edge trigger is set, the interrupt is generated at the falling edge state change at this pin. This is done by writing 1 to the SETFALL register of the bank.

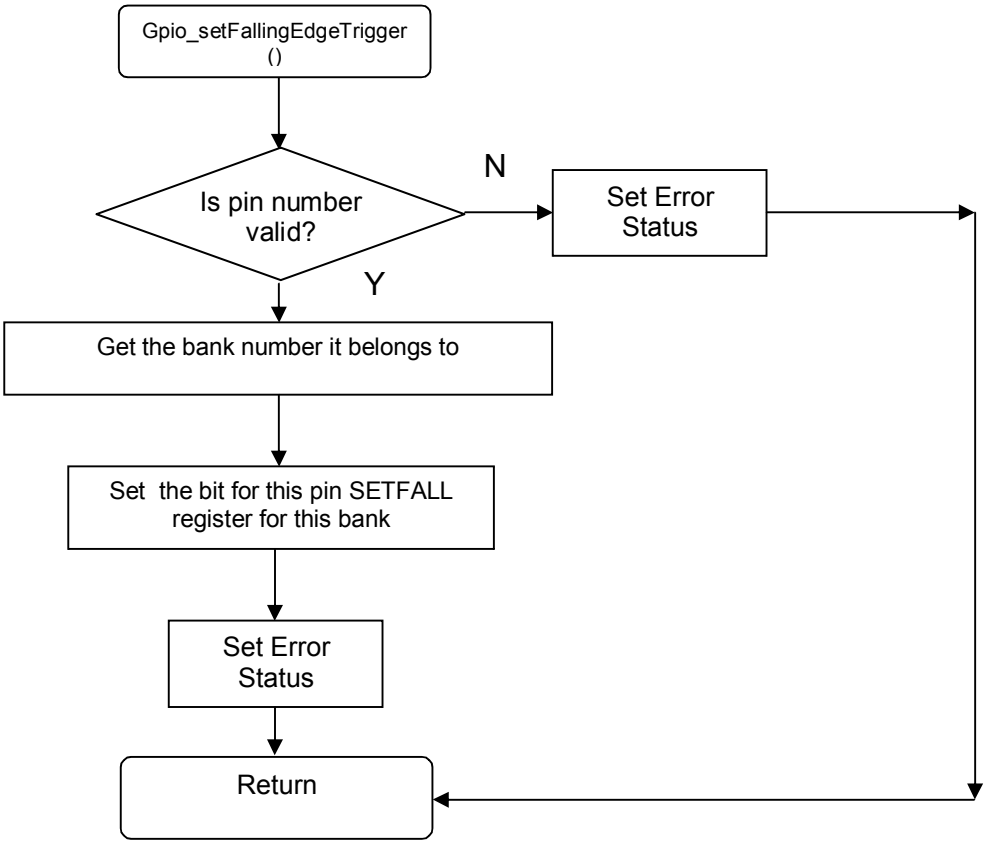

Figure 15 Gpio_setFallingEdgeTrigger() flow diagram

2.2.3.13 Gpio_clearFallingEdgeTrigger()

This API shall be used by the user to clear the falling edge trigger for the interrupt generation from a GPIO pin. When the falling edge trigger is clear, the interrupt is not generated at the falling edge state change at this pin. This is done by writing 1 to the CLRFALL register of the bank

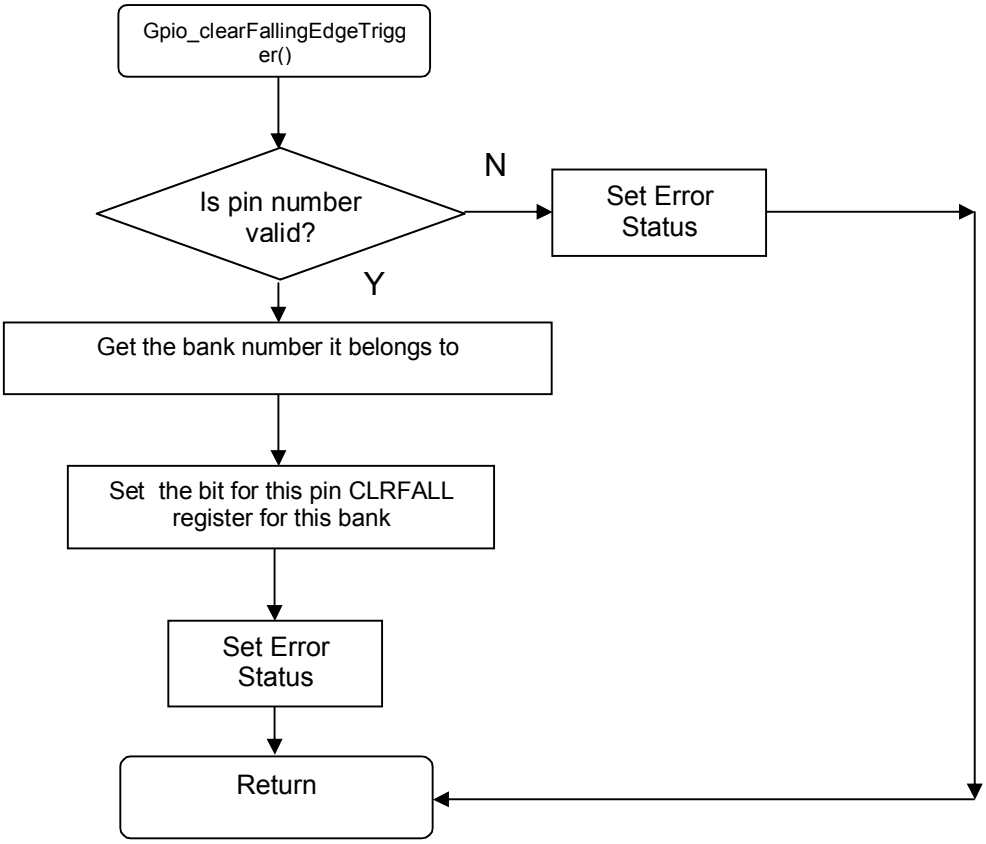

Figure 16 Gpio_clearFallingEdgeTrigger() flow diagram

2.2.3.14 Gpio_bankInterruptEnable()

This API shall be used by the user to enable interrupts from a GPIO bank. The bank number should be provided in the arguments. The interrupt from the banks are enabled a 1, to BANKINTEN register bit corresponding to that bank.

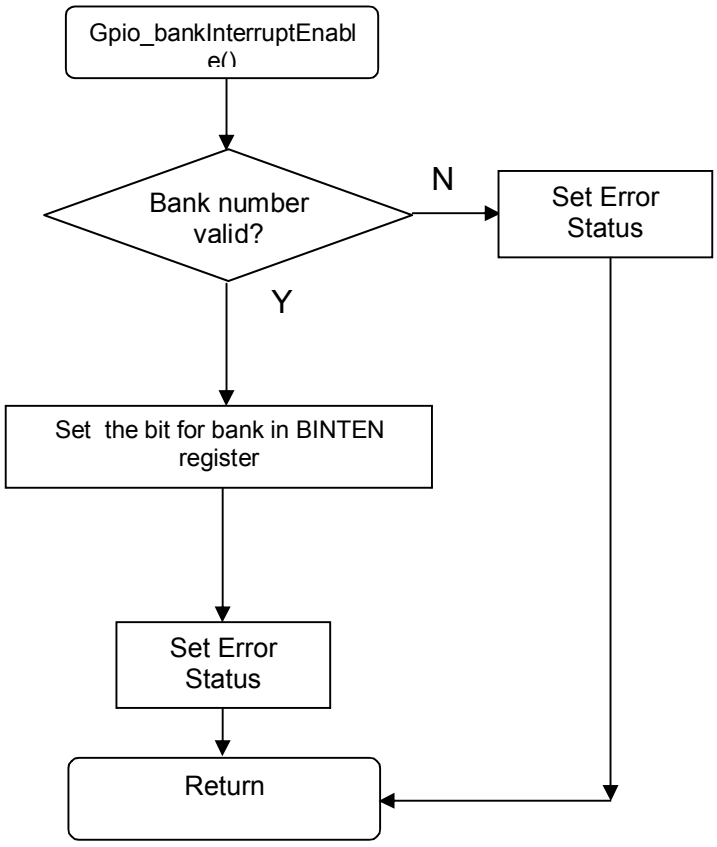

Figure 17 Gpio_bankInterruptEnable() flow diagram

2.2.3.15 Gpio_bankInterruptDisable()

This API shall be used by the user to enable interrupts from a GPIO bank. The bank number should be provided in the arguments. The interrupt from the banks are enabled a 1, to BANKINTEN register bit corresponding to that bank.

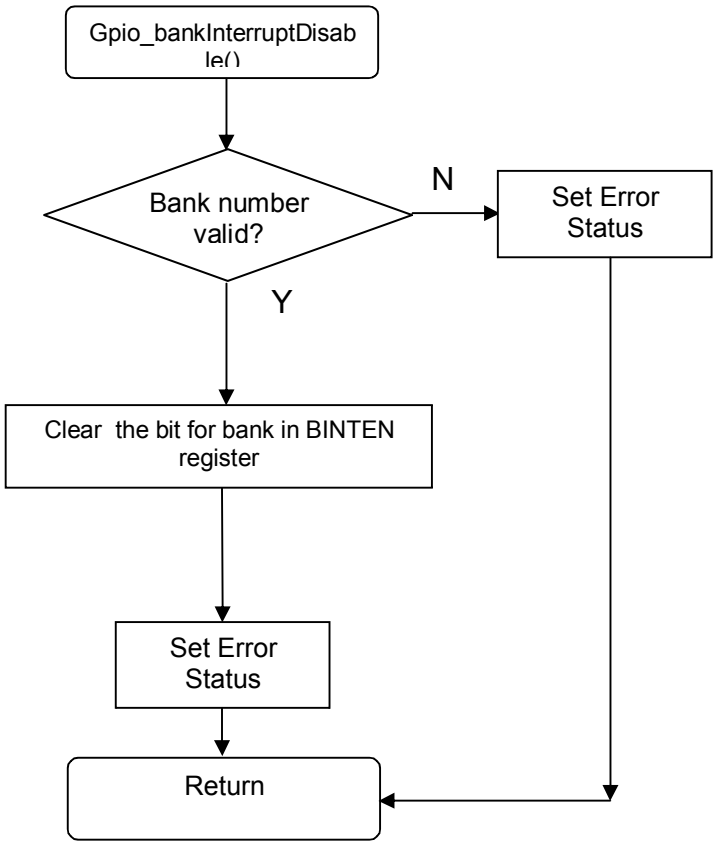

Figure 18 Gpio_bankInterruptDisable() flow diagram

2.2.3.16 Gpio_setPinInUseStatus()

This API shall be used by the user to set the pin inUse (availability as a GPIO) status. The pin number and the in use status should be provided in the arguments. The user shall mark the pin as inUse if, the pin is used for its functional purpose (if multiplexed) or if the pin is designated for use for specific purpose in the system. Then the status marking on this pin as inUse shall make GPIO operations on this pin invalid and return error

.

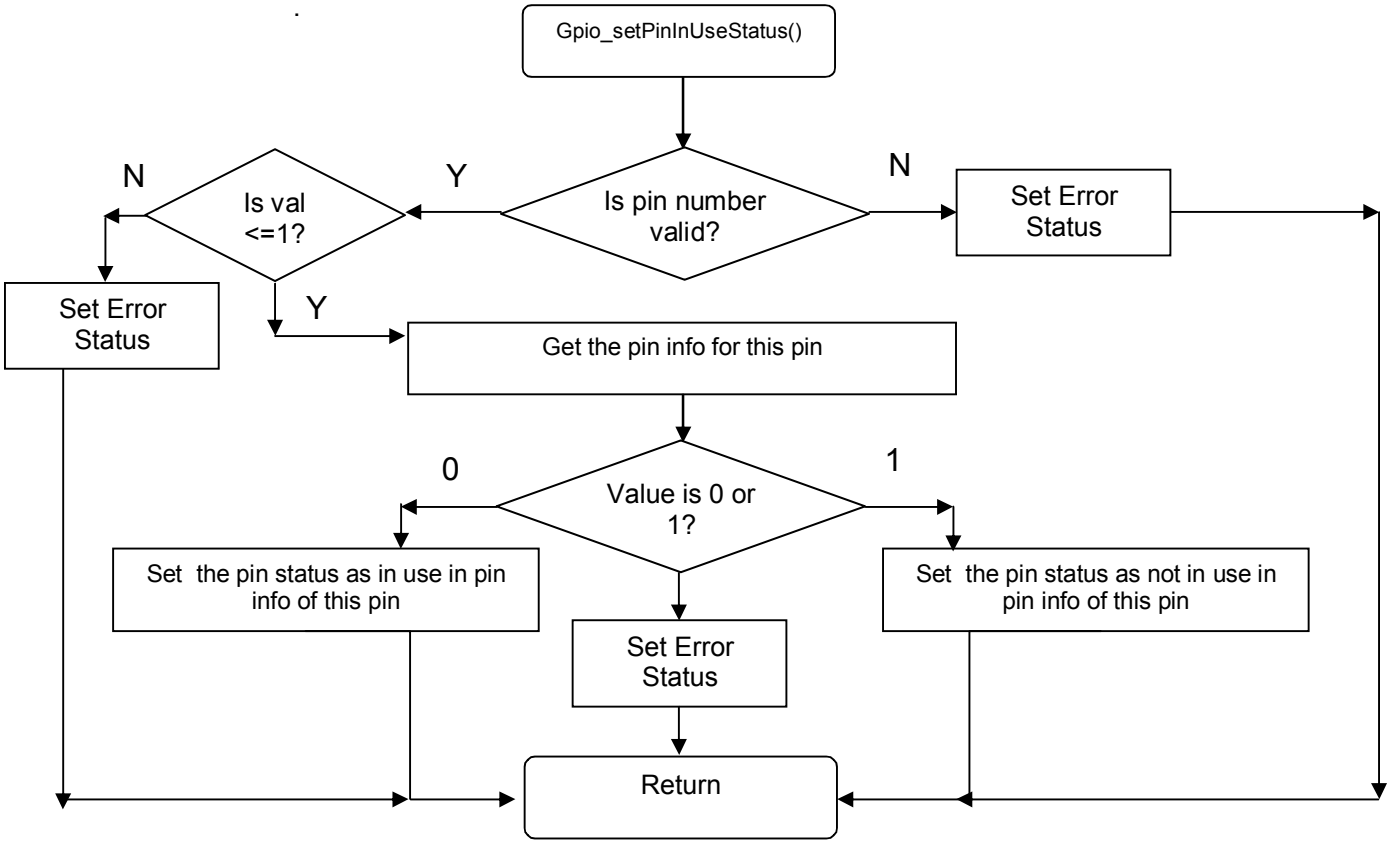

Figure 19 Gpio_setPinInUseStatus() flow diagram

2.2.3.17 Gpio_getPinInUseStatus()

This API shall be used by the user to get the pin inUse (availability as a GPIO) status. The pin number should be provided in the arguments. If the status is says that the pin is in use, then the pin is not available for GPIO operations and is used as a functional pin (multiplexed) or the pin is already designated for use.

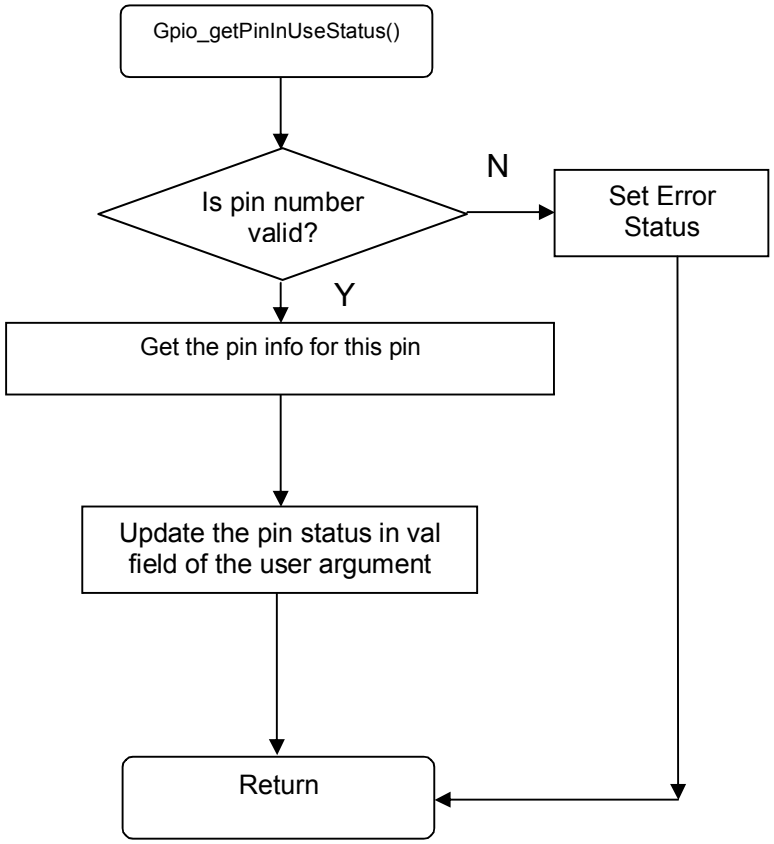

Figure 20 Gpio_getPinInUseStatus() flow diagram

2.2.3.18 Gpio_setBankInUseStatus()

This API shall be used by the user to set the bank inUse (availability as a GPIO) status. The bank number and the in use status should be provided in the arguments. The user shall mark the bank as inUse if, the bank is used for its functional purpose (if multiplexed) or if the pins in this bank are designated for use for specific purpose in the system. Then the status marking on this bank as inUse shall make GPIO operations on this bank/pins of this bank invalid and return error

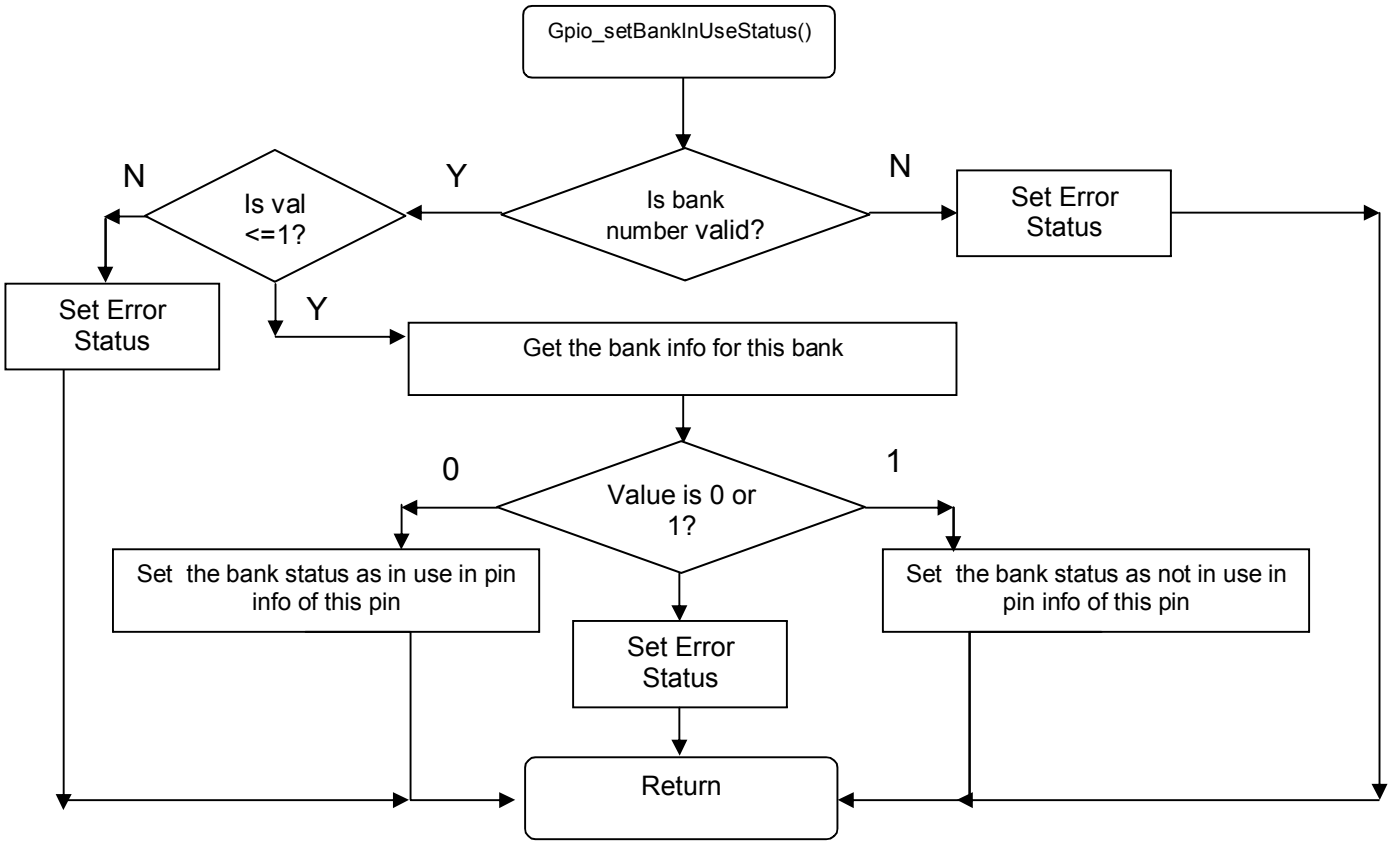

Figure 21 Gpio_setBanknUseStatus() flow diagram

2.2.3.19 Gpio_getBankInUseStatus()

This API shall be used by the user to get the bank inUse (availability as a GPIO) status. The bank number should be provided in the arguments. If the status is says that the bank is in use, then the bank is not available for GPIO operations and is already designated for use.

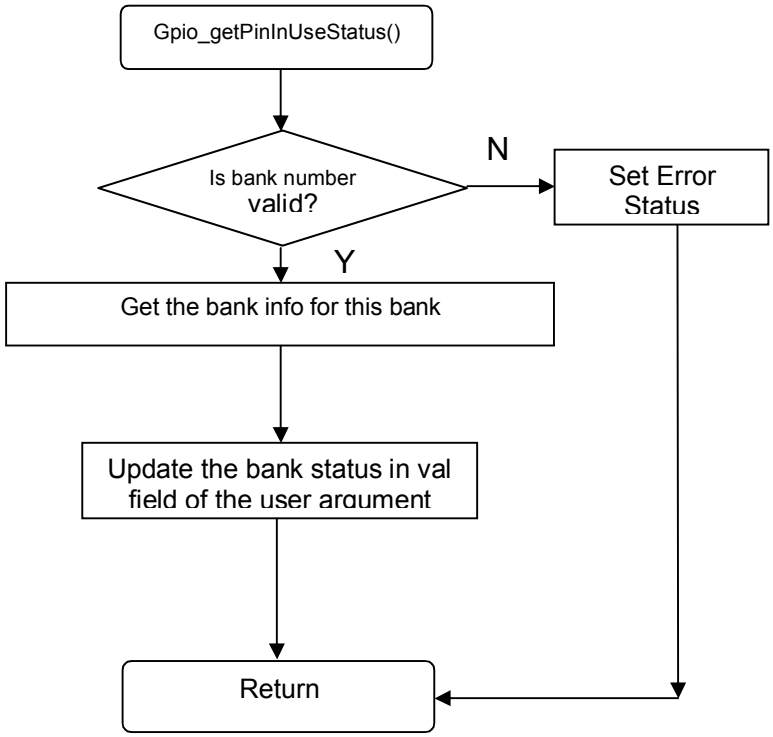

Figure 22 Gpio_getBankInUseStatus() flow diagram

2.2.3.20 Gpio_regIntHandler ()

This API shall be used by the application to register an interrupt handler for a GPIO pin/bank event. This API shall only facilitate the registration of the handler function provided by the application and does not actually service the interrupts. Hence, the handling of the associated interrupt is in the application context and not in the driver context. The GPIO shall not have any interrupt context. This is because the driver does not have the knowledge of the purpose of the interrupt, since the use case of general purpose pin/bank event in entirely decided by the application. The pin/bank number, associated handler need to be provided as arguments, to this API.

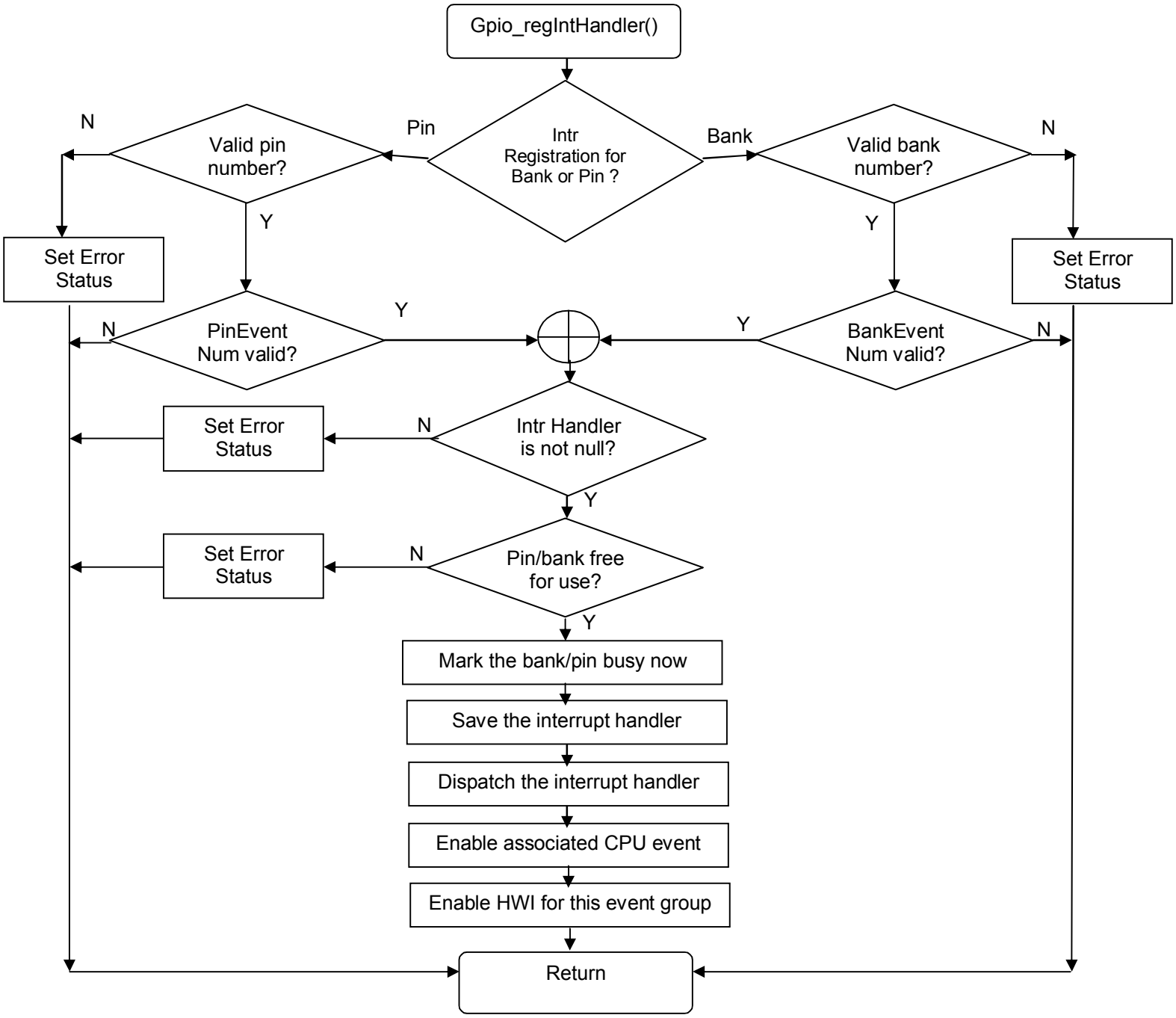

Figure 23 Gpio_RegIntHandler() flow diagram

2.2.3.21 Gpio_unregIntHandler ()

This API shall be used by the application to un-register an interrupt handler for a GPIO pin/bank event, which has been previously register. This may be helpful in re-registering a new handler or discarding the use of the interrupt from this bank/pin event. Here again, event number is checked for validity to prevent unregister call on an irrelevant pin.

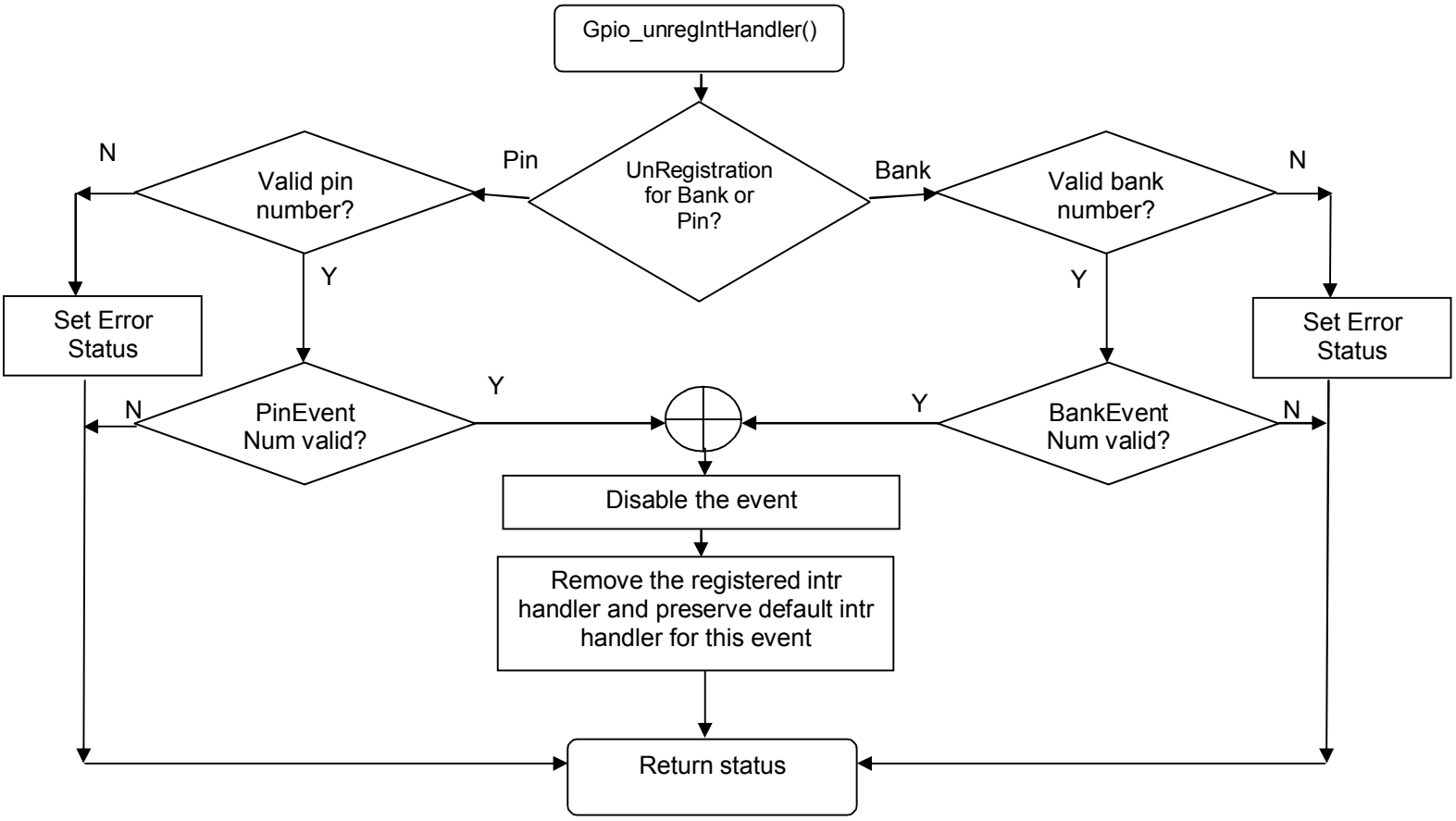

Figure 24 Gpio_unregIntHandler() flow diagram