International Journal of Scientific Engineering and Technology (ISSN : 2277-1581)
Volume No.3 Issue No.5, pp : 524-528 1 May 2014

Design and Implementation of GPIO Enumeration Library and Application for UEFI-BIOS

Srividya G Kedlaya[1], H R Bhagyalakshmi[2]

[1,2]Dept. of Electronics and Communication Engineering, B M S College of Engineering
Visvesvaraya Technological University, Bangalore, Karnataka, India

E-Mail: srigked@gmail.com, hrbbmsce@gmail.com

Abstract- General Purpose Input Output (GPIO) is a flexible software-controlled digital signal. Micro-processors heavily rely on GPIOs with only the BIOS firmware knowing how they're used. This paper aims at developing a GPIO library to enumerate all GPIOs' configuration status and to create a stand-alone user application, which helps a user to configure the GPIO pins without diving into BIOS firmware. UEFI specifications are followed and EDKII is used.

Keywords- GPIO, BIOS, UEFI, UEFI-Application, Microprocessor, EDKII

I. INTRODUCTION

The basic input output system (BIOS) also known as boot firmware, is built into the PC, and is the first code run by a PC when powered on. Industry has migrated from Legacy BIOS to a standard and modular pre-boot firmware infrastructure known as Extensible Firmware Interface (EFI) [i-iii]. EFI BIOS offers new & improved features and flexibility for code developers.

A General Purpose Input Output is an interface available on most modern microcontrollers to provide an ease of access to the devices internal properties. Generally there are multiple GPIO pins on a single microcontroller unit for the use of multiple interaction so simultaneous application [iv-v]. Each GPIO represents a bit connected to a particular pin. System-on-Chip (SOC) processors heavily rely on GPIOs to help with pin scarcity on SOCs. Most PC southbridges have a few dozen GPIO-capable pins with only the BIOS firmware knowing how they're used.

Now that the IA-32 processors are extended to 64-bit via Intel® 64, the industry is working on Unified EFI (UEFI) as the standard pre-boot firmware infrastructure going forward [vi-vii]. The UEFI specification defines an interface between an operating system and platform firmware. The interface consists of data tables that contain platform-related information, boot service calls, and runtime service calls that are available to the operating system and its loader [viii]. These provide a standard environment for booting an operating system and running pre-boot applications.

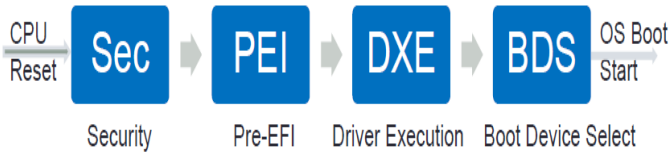
The EFI Development Kit (EDK) is the open-source component of the "Framework", Intel's implementation of the EFI Specification, which was developed under the project code named 'Tiano'. EDK II is a modern, feature-rich, cross-platform firmware development environment for the UEFI and Platform initialization (PI) specifications. The EDK II offers a better build and version tracking environment for UEFI and PI development [ix].

This paper is organized into five sections including this section. Section II presents an overview of related technologies. Implementation of the GPIO Library and Application are in section III while section IV discusses the results. Finally, future improvements and the conclusions are presented in section V.

II. RELATED TECHNOLOGY

A. *UEFI Boot Phases*

Different from the implement mechanism of legacy BIOS, UEFI BIOS [i-iii] has several phases to make the initialization process modularized. In a Framework firmware implementation, operating system initialization has four phases [i]: Security (SEC) phase, Pre-EFI Initialization (PEI) phase, Driver Execution Environment (DXE) phase and Boot Device Select (BDS) phase, as shown in Figure.1.



Sec	PEI	DXE	BDS
-Provides uP boot strap vector -Initializes temporary RAM using CPU cache -Provides optional security features	-Init Proc -Update ucode -Initialize chipset -Detect corrupted flash and recover if corrupted -Find and initialize RAM	-Initialize uP, cache, -Initialize more c/s -Initialize SMM -Execute PCI enumeration -Initialize video, kbd, mouse -Initialize USB legacy: -Initialize drives -Create table interfaces (ACPI, SMBIOS, ...)	-Run down list of selected devices. -For each, try to find boot image -Load boot image into memory -Jump to boot image -If fail try next

Figure.1 EFI Boot Phases and services

The SEC phase is the first phase in the Framework architecture and is responsible for handling all platform restart events. The PEI phase is handed control from the SEC phase. PEI phase is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed [x]. The DXE phase is where most of the system initialization is performed. The DXE foundation produces the full complement of EFI Boot Services, EFI Runtime Services, and DXE Services. The BDS phase is responsible for implementing the platform boot policy, establishing the console devices and attempting the boot of operating systems [xi].

International Journal of Scientific Engineering and Technology (ISSN : 2277-1581)
Volume No.3 Issue No.5, pp : 524-528 1 May 2014

B. UEFI-BIOS Code Structure

UEFI-BIOS code is modular and is divided into different packages. Each EDK II Package is a container that includes a set of modules and their related definitions. Each Package is an EDK II distribution unit. It can be used to manage and release the big project to facilitate a user's distribution and reuse. The whole project sources can be split into different packages to reduce the release granularity [xii]. Some of the packages and their definitions are listed in Table 1.

TABLE I
PACKAGES IN UEFI-BIOS

Package	**Description**
Basetool	Provides build related tools for both EDK and EDK2. Also contains miscellaneous tools such as ECC, EOT
PlatformPkg	Provides the Platform specific code.
RefcodePkg	Provides Silicon specific reference code.
CryptoPkg	Several security features like Authenticated Variable Service, Driver Signing, etc., are introduced here.
EdkCompatibilityPkg	Provides header files and libraries that help to build the EDK module in UEFI 2.0 mode with EDK II Build.
FrameworkPkg	This package provides definitions and libraries that comply to Intel's UEFI & EFI Framework Specifications.
MdePkg	Provides all definitions (including functions, macros, structures and library classes) and libraries instances, which are defined in MDE Specification.
NetworkPkg	Provides IPv6 network stack drivers, IPsec driver, PXE driver, iSCSI driver and necessary shell applications for network conifguration.
ShellPkg	This package provides native EDKII implementation of a UEFI Shell 2.0.
UefiCpuPkg	This Package provides UEFI compatible CPU modules and libraries.

Each package has a unified directory structure that separate the different source files. The root directories in each package are: *Include*, *Library*, *Application* and *Drivers*. The *Include* directory contains all public header files exposed by this package and used by this package and other packages. The *Library* directory contains directories for each library instance module included. The *Application* directory contains directories for each UEFI applications module included. The *Driver* directory contains directories for each driver group and for each driver.

C. GPIO Library

GPIO Library which is developed in this paper is composed of the following files:

GpioLib.c file : It is a source file written in C language. It consists of the source code to perform different functions.
GpioLib.h file : It is a header file which supports the source file. It includes constant definitions and library files that are used by the source file. This header file is included in the source file.
GpioLib.inf file : It is a text file with .inf extension. It is called the information file. It follows the standard EDK II INF file format. This file is used by the build tool to understand the information about the library.

The source file *GpioLib.c* and the information file *GpioLib.inf* are placed in the *Library* section of the Platform package mentioned in Table 1. The header file *GpioLib.h* is placed under the *Include* section of the Platform package.

D. UEFI Shell

The UEFI Shell [xiii] is a simple, interactive pre-boot environment that allows UEFI device drivers to be loaded, UEFI applications to be launched, and operating systems to be booted. In addition, UEFI Shell also provides a set of basic commands used to manage files and the system environment variables etc. Figure.2 depicts the UEFI Shell interface.

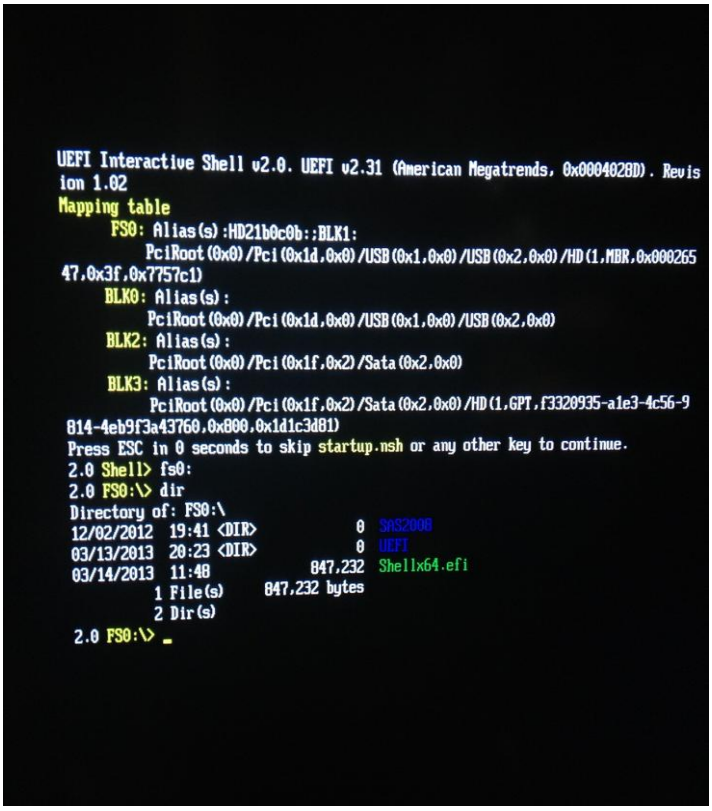


Figure.2 UEFI Shell interface

E. UEFI Application

UEFI application [xiii] is a stand-alone entity which can be launched from UEFI shell. It can be used to perform a specific task when the application is called and exit upon completing the task. In our case, a UEFI application is developed to get the GPIO information, and configure a GPIO pin.

The application also consists of three files. *GpioApp.c* source file, *GpioApp.h* header file and *GpioApp.inf* information file. These files are placed in the *Applications* section of the Platform package. After the BIOS source code is built, a '.efi'

International Journal of Scientific Engineering and Technology (ISSN : 2277-1581)
Volume No.3 Issue No.5, pp : 524-528 1 May 2014

file is generated, in this case, *GpioApp.efi*. This is the stand-alone application file which can be launched in UEFI shell.

III. IMPLEMENTATION OF GPIO LIBRARY AND APPLICATION

A. GPIO Library Functionalities

The GPIO Library majorly has two functions.

First: To read the GPIO configuration registers and interpret the status of every GPIO - There are various registers in particular to every GPIO pin. The library is capable of identifying those registers from memory, reading them and interpreting their values in form of verbal message strings. A function module *Table()* reads all the GPIO configuration registers in order, and gives out the information of all GPIO pins in lists of strings. Another function *PinInfo()* takes a particular GPIO pin number as input and reads only that particular pin related configuration register, and gives out the information of only that pin.

Second: To change the configuration status of a GPIO pin- In contrast to the first functionality, here the Library can take different GPIO attributes as input, and write to the GPIO registers. A function *GpioWrite()* takes the GPIO pin number, and various pin attributes as inputs, it decides what bits need to be written to, what values to be written, and selects the appropriate GPIO configuration register to write the selected values in selected bit positions. The remaining bit values are unaltered by suitable masking, so that the remaining attributes' configuration is kept as before. This is done so that, only the attributes selected by the user are altered and remaining are not touched.

The implementation flowchart of the GPIO Library is shown in Figure.3.

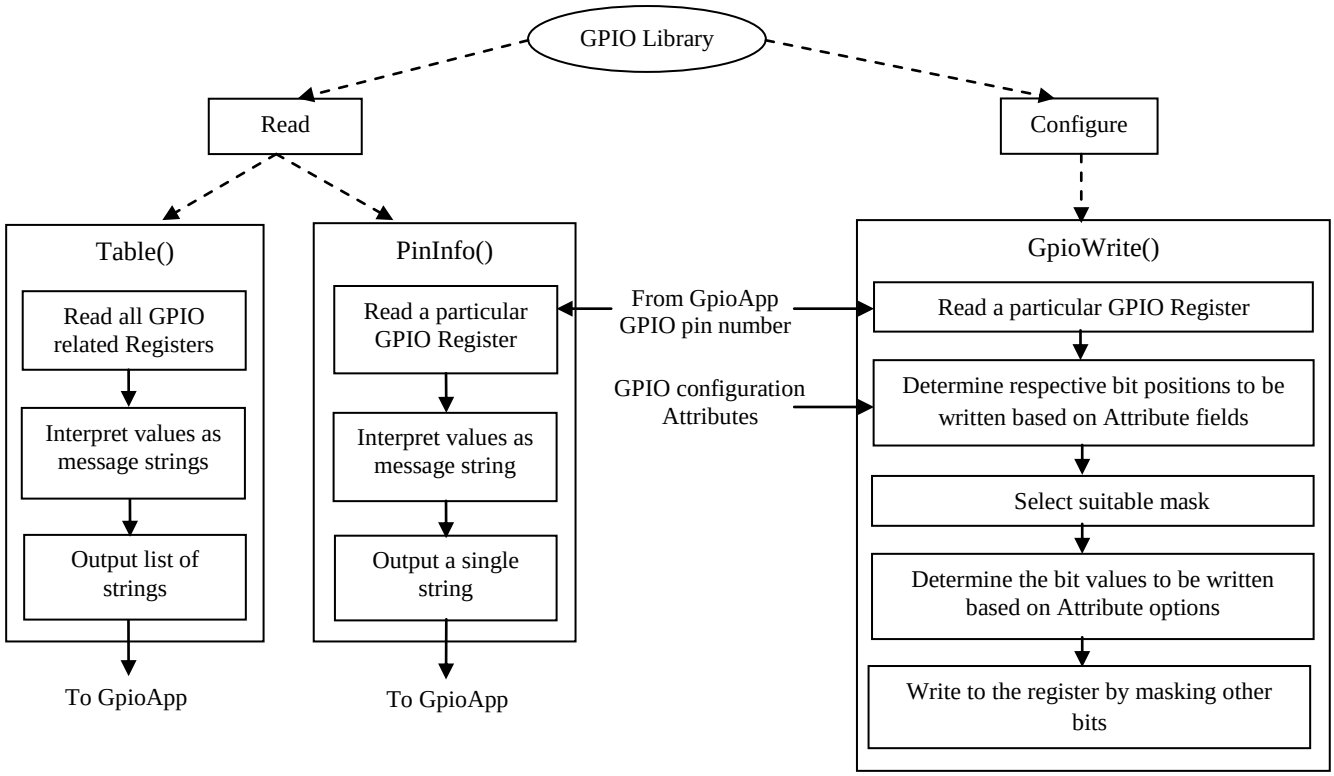


Figure.3 Implementation Flowchart of the GPIO Library

B. UEFI Application Functionalities

The UEFI Application is launched in the UEFI Shell Interface by typing the Application name *GpioApp* as the first argument. It has various functionalities.

First: Usage() – The user must know how to use the Application, what all tasks can be performed by it, what inputs can be given. This is done by the *Usage()* function. When the Application is launched with only one argument, which is its name *GpioApp*, *Usage()* displays the information about the capable tasks of, and valid inputs to the Application.

Second: To read GPIO Table – When Application is launched with two arguments <*GpioApp*> <*table*>, the application calls the *Table()* of the GPIO Library. A list of strings given out by the library is formatted and displayed in the form of a table.

Third: To read a selected GPIO pin's information – Application when launched with three arguments <*GpioApp*> <*GpioPin#*> <*read*> it calls *PinInfo()* of GPIO Library and formats the output string into a legible display.

Fourth: To configure a selected GPIO pin – A GPIO pin has many electrical attributes like pullups, current source, open drain mode, etc., and software attributes like interrupt, wake, high impedance etc., these attributes can be selectively configured by giving <*GpioApp*> <*GpioPin#*> <*configure*> <*attribute1*> <*attribute2*> … <*attribute-n*>

International Journal of Scientific Engineering and Technology (ISSN : 2277-1581)
Volume No.3 Issue No.5, pp : 524-528 1 May 2014

Fifth: To display suitable error messages for Invalid inputs – The Application discards invalid or inappropriate user inputs and displays suitable error messages.

The implementation flowchart of the GPIO Application is shown in Figure.4.

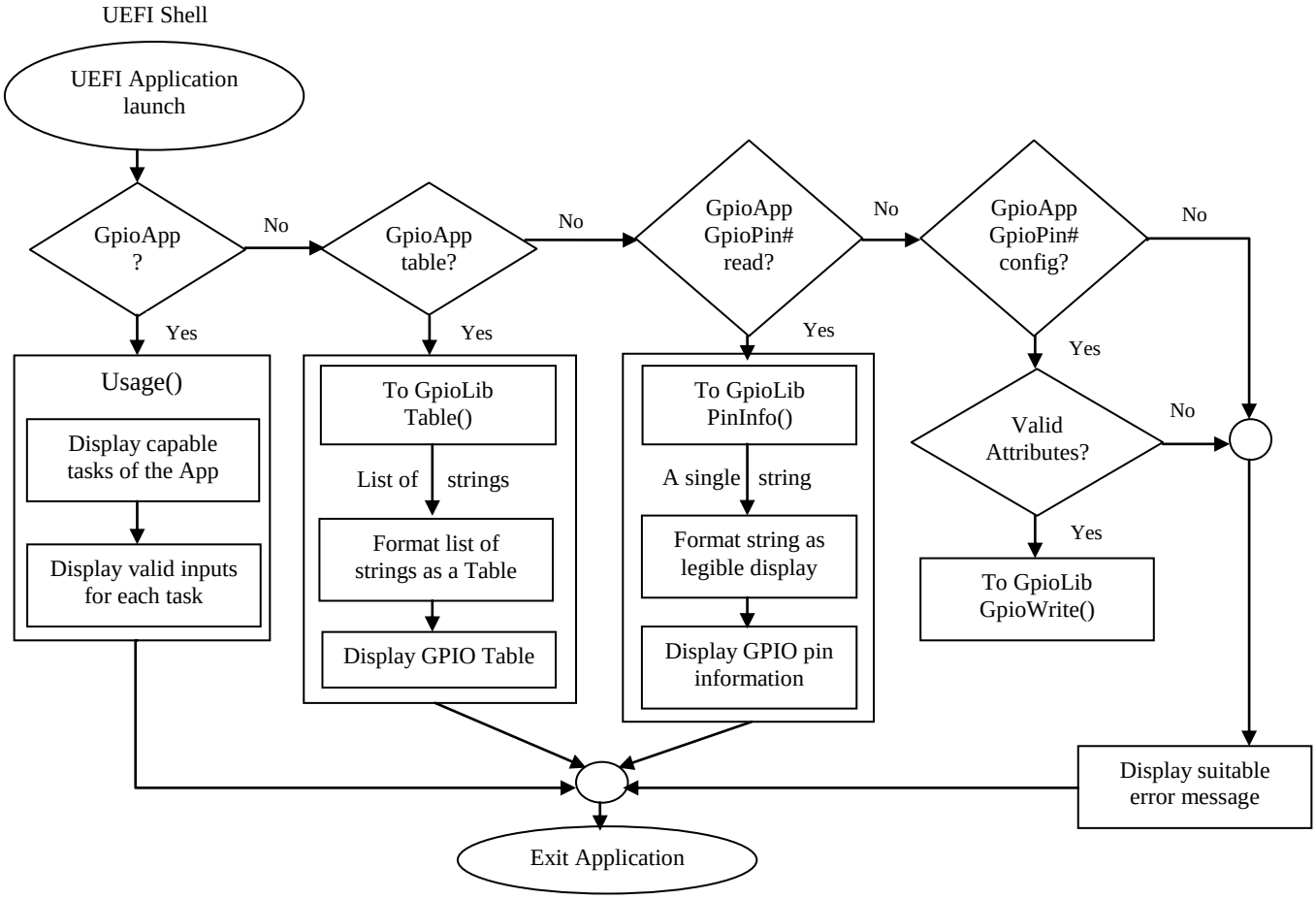


Figure.4 Implementation Flowchart of the UEFI Shell Application

IV. RESULTS AND DISCUSSION

Currently, only BIOS firmware can configure GPIOs and a library such as the GPIO library does not exist. Any non-BIOS engineer who wants to configure a GPIO pin must invariably depend on a BIOS engineer to get the required configuration. This consumes lot of time for both of them and work gets slower. Also debugging and validating the configuration status of all GPIO pins is very time consuming and confusing at times. These issues are the concern of this paper where in, the developed GPIO library gives out all configuration details of GPIO pins in form of a table, which can be just handy at times of validating and debugging. There is also an option of reading a selected pin's information using this library. The two ways of display is incorporated to enable flexibility of the library to use it as desired by individual users. Also, to solve the problem of non-BIOS engineers, the library is also made capable of configuring the GPIO pins. To ensure that these functionalities are used by all users in an easier and BIOS independent fashion, a UEFI Shell Application is developed which uses the GPIO Library and performs the desired tasks in a user-friendly fashion.

The display for individual GPIO pin is shown in Figure.5 and the GPIO table display in Figure.6.

```
Sl No.  Attributes      : Configuration
 1.     N/G             : GPIO
 2.     Mode            : M1
 3.     Gp_Cfg          : GPI
 4.     TxStat          : NA
 5.     INT_Sel         : Line_10
 6.     INT_Type        : INT_Disable
 7.     Pull_Type       : P_5K_Low
 8.     OpenDrainEn     : Disabled
 9.     CurntSrcEn      : Disabled
 10.    GlitchFiltrCfg  : En_Rx_Data
 11.    Invert_RX_TX    : No_Inversion
 12.    INT_Mask        : Unmasked
 13.    GPE             : NA
 14.    SMI_En          : NA
 15.    SCI_En          : NA
 16.    Wake_Mask       : Wake_Unmasked
 17.    PadCfg_Lock     : Unlocked
```

Figure.5 Information of a selected GPIO pin

International Journal of Scientific Engineering and Technology (ISSN : 2277-1581)
Volume No.3 Issue No.5, pp : 524-528 1 May 2014

$$$$$ GPIO Pad Configuration Information Table $$$$$

GpioNo	USE	I/O	o/p-LVL	i/p-LVL	I/P-INV	GPI_LxEB	I/p-sensing	Internal_Weak_Pull	GPIO_OWN	GP_I_ROUT	Blink_Enable	Serial_Blink_Enable	NMI_Enable	SMI_Enable
GPIO_0	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_1	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_2	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_3	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_4	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_5	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_6	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_7	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_8	G	I	-	H	I	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_9	G	I	-	L	NI	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_10	G	I	-	H	I	Edge	Enable	None	ACPI Driver	SCI	-	-	-	-
GPIO_11	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_12	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_13	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_14	G	I	-	L	I	Level	Enable	PullDown	GPIO Driver	SCI	-	-	-	-
GPIO_15	G	O	L	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_16	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_17	G	I	-	H	I	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_18	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_19	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_20	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_21	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_22	G	O	L	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_23	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_24	G	I	-	L	NI	Edge	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_25	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_26	G	I	-	L	NI	Edge	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_27	G	I	-	H	I	Level	Enable	None	ACPI Driver	SCI	-	-	-	-
GPIO_28	G	O	L	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_29	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_30	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_31	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_32	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_33	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_34	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_35	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_36	G	I	-	L	I	Level	Enable	None	GPIO Driver	SCI	-	-	NMI Disabled	SMI Disabled
GPIO_37	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_38	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_39	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_40	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_41	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_42	N	-	-	-	-	-	-	-	-	-	-	-	-	-
GPIO_43	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_44	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_45	G	I	-	H	I	Level	Enable	None	GPIO Driver	SCI	-	-	NMI Disabled	SMI Disabled
GPIO_46	G	O	L	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_47	G	I	-	H	I	Edge	Enable	None	GPIO Driver	SCI	-	-	NMI Disabled	SMI Disabled
GPIO_48	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_49	G	O	H	-	-	-	Enable	None	GPIO Driver	-	-	-	-	-
GPIO_50	G	I	-	L	NI	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_51	G	I	-	H	NI	Edge	Enable	PullUp	GPIO Driver	SCI	-	-	-	-
GPIO_52	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-
GPIO_53	G	I	-	H	NI	Edge	Enable	PullUp	GPIO Driver	SCI	-	-	-	-
GPIO_54	G	I	-	H	NI	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_55	G	I	-	H	NI	Level	Enable	None	GPIO Driver	SCI	-	-	-	-
GPIO_56	G	O	H	-	-	-	Disable	None	GPIO Driver	-	-	-	-	-

Figure.6 GPIO Information Table

V. CONCLUSION

GPIO or General Purpose Input Output is the easiest way to interact with basic peripheral components. Non-BIOS engineers find difficulty in configuring GPIOs and have to rely upon BIOS engineers. As a solution to this problem, this paper proposes a GPIO Library capable of enumerating all GPIOs' configuration status and also to configure a pin with desired attributes. A UEFI shell application is developed to perform these tasks in a user friendly fashion. It also comes in handy at times of debugging and validation. As a future scope, the application can be made more interactive for users and also can be extended to develop runtime applications.

ACKNOWLEDGMENT

The authors wish to thank ECE department of BMS college of Engineering, Bangalore, Karnataka, India for supporting this work.

REFERENCES

i. *Yin Hu, Haoyong Lv, "Design of Trusted BIOS in UEFI Base on USBKEY", International Conference on Intelligence Science and Information Engineering, 2011*

ii. *PENG Shuanghe, HAN Zhen, "Design and Implementation of Portable TPM Device Driver based on Extensible Firmware Interface", International Conference on Multimedia Information Networking and Security, 2009*

iii. *TANG Weimin, PENG Shuanghe, HAN Zhen, "Design and Implementation of UsbKey Device Driver based on Extensible Firmware Interface", ICSP2008 Proceedings, 2008*

iv. *Sasang Balachandran, "General Purpose Input/Output (GPIO)", ECE 480 Design team 3, 11/08/2009*

v. *EMF® 32 General Purpose Input Output, AN0012 Application Note*

vi. *Vincent Zimmer, Michael Rothman, Suresh Marisetty, "Beyond Bios- Developing with the Unified Extensible Firmware Interface", 2nd Edition, Intel Press, Nov. 2010*

vii. *Michael A. Rothman, "UEFI Overview", May 16th 2007.*

viii. *Unified Extensible Firmware Interface Specification Version 2.3.1, April 6, 2011.*

ix. *Driver Writer's Guide for UEFI 2.3.1, Version 1.01, August 2012*

x. *Intel Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification (PEI CIS) Version 0.9. September 16, 2003*

xi. *Intel Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification (DXE CIS), Version 0.9, September 16th, 2003*

xii. *EDK II Module Writer's Guide, Revision 0.01, April, 2009*

xiii. *http://sourceforge.net/apps/mediawiki/tianocore/*