Lattice Propel 1.0 Application Programming Interface

Reference Guide

FPGA-AN-02027-1.0

May 2020

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS and with all faults, and all risk associated with such information is entirely with Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Acronyms in This Document 4
1. Introduction 5
1.1. Purpose 5
1.2. Audience 5
2. RISC-V CPU 6
2.1. Overview 6
2.2. CPU HAL 6
2.2.1. PIC 6
2.2.2. Timer 8
2.2.3. Register Access 11
3. General IPs 14
3.1. GPIO 14
3.2. UART 16
3.3. EFB 19
3.3.1. UFM Access 19
3.3.2. I^{2}C Slave 23
Reference 25
Technical Support Assistance 26
Revision History 27

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
BSP	Board Support Package, the layer of software containing hardware-specific drivers and libraries to function in a particular hardware environment.
EFB	Embedded Function Block, a hard block in Lattice FPGA device.
eSDK	Embedded System Design and Develop Kit, a set of software development tools that allows the creation of applications for software package on the Lattice embedded platform.
ESI	Embedded System Solutions.
GPIO	General Purpose Input Output.
HAL	Hardware Abstraction Layer, a software interface to hide the detail of the hardware design and provide general services to the upper layer.
I^2C	Inter Integrated Circuit.
ISR	Interrupt Service Routine that is called when the corresponding interrupt occurs.
PIC	Programmable Interrupt Controller, handling the interrupts from the peripheral devices.
RISC-V	A free and open instruction set architecture (ISA) enabling a new era of processor innovation through open standard collaboration.
SoC	System on Chip.
UART	Universal Asynchronous Receiver-Transmitter.
UFM	User Flash Memory.

1. Introduction

Lattice Propel 1.0 is a complete set of graphical and command-line tools to create, analyze, compile, and debug both FPGA-based hardware and software processor systems.

1.1. Purpose

Embedded System Solutions (ESI) take an important role in FPGA system design allowing you to develop software for a processor in an FPGA device. It provides flexibility for you to control various peripherals from a system bus.

To develop an embedded system on an FPGA, you need to design the SoC with an embedded processor and develop system software on the processor. Lattice Propel can help develop your system with a RISC-V processor, peripheral IP, and a set of tools.

The purpose of this document is to introduce the Application Programming Interface (API) for the IPs in Lattice Propel eSDK and to guide you to develop your own system software.

1.2. Audience

The intended audience for this document are embedded system designers and embedded software developers using Lattice MachXO3D FPGA devices. The technical guidelines assume readers have expertise in the embedded system area and FPGA technologies.

2. RISC-V CPU

2.1. Overview

The RISC-V Processor is a configurable CPU soft IP based on the open source Vex RISC-V core, which integrates JTAG debugger, PIC, and Timer. The RISC-V core supports RV32I instruction set and five-stage pipelines. JTAG debugger, PIC, and Timer could be enable or disable optionally based on the system requirement.

Lattice HAL provides a set of APIs to help integrate the CPU into the system and develop the software, which needs to make use of all the modules and services the CPU provides.

2.2. CPU HAL

2.2.1. PIC

The Programmable Interrupt Controller (PIC) aggregates up to eight external interrupt inputs (IRQs) into one interrupt output to CPU (meip). The interrupt status is a memory mapped register that can be used to read the values of IRQs through the bus interface. Individual IRQs can be configured by programming the corresponding enable and polarity register. For the design detail, refer to RISC-V MC CPU IP Core – Lattice Propel Builder (FPGA-IPUG-02114).

The APIs of the PIC provides the necessary interfaces to the user to handle the peripherals' interrupts, which makes use of the system interrupt handling framework.

2.2.1.1. API Reference

pic_init	
void pic_init(unsigned int base)	
Parameter	**Description**
base	Base address of the PIC module, Propel SDK automatically parses the address map of the SoC system and passes the information to software.
Returns	**Description**
void	—
Description	
This function is supposed to be called when the platform is initializing. This function should be called before calling any PIC related functions.	

pic_int_enable	
void pic_int_enable(unsigned char src)	
Parameter	**Description**
src	The corresponding INT number of the device connecting on the PIC.
Returns	**Description**
void	—
Description	
This function is used to enable the interrupt for a specified source. The interrupt service routine is called for the source, only if it is enabled by this function. The interrupt can be disabled by calling 'pic_int_disable()' with the same interrupt source number as the parameter.	

pic_int_disable	
void pic_int_disable(unsigned char src);	
Parameter	**Description**
src	The corresponding INT number of the device connecting on the PIC.
Returns	**Description**
void	—
Description	
This function is used to disable the interrupt for a specified source. The interrupt service routine is not invoked, if the interrupt is disabled by this function. The interrupt can be enabled by calling 'pic_int_enable()' with the same interrupt source number as the parameter.	

pic_isr_register	
bool pic_isr_register(unsigned char src, void (*isr)(void *), void *context) ;	
Parameter	**Description**
src	The corresponding INT number of the device connecting on the PIC.
isr	The function pointer to the interrupt service routine of the corresponding device.
context	The context of the interrupt service routine for the device.
Returns	**Description**
bool	True: the interrupt registration succeeded. False: the interrupt registration failed.
Description	
This function is used to register an interrupt service routine for a specified device. After registration, the ISR is invoked automatically when the interrupt happens, if the interrupt is enabled by pic_int_enable().	

pic_int_polarity_set	
void pic_int_polarity_set(unsigned char src, unsigned char bit)	
Parameter	**Description**
src	The corresponding INT number of the device connecting on the PIC.
bit	The value of the polarity of the interrupt.
Returns	**Description**
void	—
Description	
This function is used to set the polarity of the specified device at runtime. The current polarity can be got via calling pic_int_polarity_get().	

pic_int_polarity_get	
unsigned char pic_int_polarity_get(unsigned char src)	
Parameter	**Description**
src	The value of the polarity of the specified interrupt.
Returns	**Description**
unsigned char	The polarity of the interrupt.
Description	
This function is used to get the polarity of the specified interrupt pin.	

2.2.1.2. API Usage Example

The code episode shows the typical usage of the APIs of PIC. After initialization of the PIC module, you can register a callback for a specified interrupt source. When interrupt happens from that source, the registered callback is invocated to handle the event.

```
#include "hal.h"

/*supported number of interrupt source*/
#define INT_NUM    6
int main()
{
        /* Initialize the PIC with base address and the number of interrupt source */
        pic_init(CPU0_INST_PICTIMER_START_ADDR, INT_NUM);

        /* Register an ISR callback, invocated when the corresponding interrupt happens */
        pic_isr_register(IRQ_NUM, pisr_callback, (void *)context);

        while (1)
        {
                ……
        }
        return 0;
}
```

2.2.2. Timer

The Timer module provides 64-bit real-time counter register (mtime) and time compare register (mtimecmp). An output interrupt signal that is connected to "mtip" of RISC-V core is asserted when the value of mtime is greater than or equal to mtimecmp. You can refer to RISC-V MC CPU IP Core – Lattice Propel Builder (FPGA-IPUG-02114) for detail.

The APIs of timer provide a set of interfaces for you to access the timer registers, to start or stop a registered timer service.

2.2.2.1. API Reference

timer_init	
void timer_init(struct timer_ctx_s * this_timer, unsigned int base_addr, unsigned int cpu_freq);	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
base_addr	Base address of the timer device, Propel SDK automatically parses the address map of the SoC system and passes the information to the software.
cpu_freq	The CPU running frequency, which is used to configure the required timer slice.
Returns	**Description**
void	—
Description	
This function is supposed to be called when the platform is initializing. This function should be called before calling any timer related functions.	

timer_start	
unsigned char timer_start(struct timer_ctx_s * this_timer , void (*callback)(void *), void *userCtx, unsigned int periodic, unsigned int count)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
callback	The pointer to the callback function that is called when the timer interrupt happens.
userCtx	The pointer to the context that is passed to the user-callback function.
periodic	The flag to indicate whether or not the timer event is periodic. 1: periodic, timer is reloaded automatically when timer interrupt happens. 0: not periodic, timer event happens only once.
count	The time delay for the timer, 1 ms as the granularity.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to register a user-callback and start the timer with the required time delay. The timer can be stopped by calling timer_stop().	

timer_stop	
unsigned char timer_stop()	
Parameter	**Description**
void	—
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to stop an active timer. After calling this function, you need to re-start the timer by calling the timer_start() again.	

timer_get_mtime	
unsigned char timer_get_mtime(struct timer_ctx_s * this_timer, unsigned long int *value)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
value	The pointer to a 64-bit integer that storing the value of mtime.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to get the current mtime value of the timer.	

timer_set_mtime	
unsigned char timer_set_mtime(struct timer_ctx_s * this_timer, unsigned long long int value)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
value	The 64-bit integer value that is set to mtime.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to set the mtime value of the timer.	

timer_get_mtimecmp	
unsigned char timer_get_mtimecmp(struct timer_ctx_s * this_timer, unsigned long long int *value)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
value	The pointer to a 64-bit integer that stores the value of mtimecmp.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to get the value of mtimecmp.	

timer_set_mtimecmp	
unsigned char timer_set_mtimecmp(struct timer_ctx_s * this_timer, unsigned long long int value)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
value	The 64-bit integer value that set to mtimecmp.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to set the mtimecmp value of the timer.	

timer_reload	
unsigned char timer_reload(struct timer_ctx_s * this_timer, unsigned int delay)	
Parameter	**Description**
this_timer	The pointer to the instance of the current timer device.
delay	The delay time for the next timer interrupt happens. The granularity is 1 ms.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to reload the delay time for the timer. This function is supposed to be used when the timer is not periodic and need to trigger the timer interrupt one more time manually.	

2.2.2.2. API Usage Example

The typical usage of the timer can be divided into two cases, one is to get the real time counter of the timer, and the other is to register a callback function and make it be called once or periodic when the timer expires.

The following code episode shows an example of how to use a timer.

```
#include "hal.h"
int main()
{
        struct timer_inst timer;
        /* initialize the timer with base address and CPU frequency */
        timer_init(&timer, CPU0_INST_PICTIMER_START_ADDR,
                CPU_FREQUENCY);

        /* start the timer with a registered callback */
        timer_start(&timer, ptimer_callback, NULL, true, 10);
         ……
```

```
        /*stop the timer*/
        timer_stop();
        return 0;
}
```

2.2.3. Register Access

The Register Access provides a set of APIs to read, write or modify the memory mapped address of the peripheral devices registers, including 8-bit, 16-bit and 32-bit services.

2.2.3.1. API Reference

reg_32b_write	
unsigned char reg_32b_write(unsigned int reg_addr, unsigned int value)	
Parameter	**Description**
reg_addr	The address of register that user wants to write.
value	The value that is written to the register.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to write 32-bit data into a specified register.	

reg_32b_read	
unsigned char reg_32b_read(unsigned int reg_addr, unsinged int *reg_32b_value)	
Parameter	**Description**
reg_addr	The address of register that user wants to read.
reg_32b_value	The pointer to buffer to hold the data read back.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to return 32-bit data read from the peripheral register.	

reg_32b_modify	
unsigned char reg_32b_modify(unsigned int reg_addr, unsigned int bits_mask, unsigned int value)	
Parameter	**Description**
reg_addr	The address of register that user wants to modify the value.
bits_mask	Bits that is modified within the register.
value	The value that user wants to write to the register. Only masked bits are affected.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to modify the masked bits of the specified register value.	

reg_16b_write	
unsinged char reg_16b_write(unsigned int reg_addr, unsigned short value)	
Parameter	**Description**
reg_addr	The address of register that user wants to write.
value	The value that is written to the register.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to write 16-bit data into a specified register.	

reg_16b_read	
unsigned char reg_16b_read(unsigned int reg_addr, unsigned short *reg_16b_value)	
Parameter	**Description**
reg_addr	The address of register that user wants to read.
reg_16b_value	The pointer to the buffer to hold the data read back.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to return a 16-bit data read from the peripheral register.	

reg_16b_modify	
unsigned char reg_16b_modify(unsigned int reg_addr, unsigned short bits_mask, unsigned short value)	
Parameter	**Description**
reg_addr	The address of register that user wants to modify the value.
bits_mask	Bits that are modified within the register.
value	The value that user wants to write to the register. Only masked bits are affected.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to modify the masked bits of the specified register value.	

reg_8b_write	
unsigned char reg_8b_write(unsigned int reg_addr, unsigned char value)	
Parameter	**Description**
reg_addr	The address of register that user wants to write.
value	The value that is written to the register.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to write 8-bit data into a specified register.	

reg_8b_read	
unsigned char reg_8b_read(unsigned int reg_addr, unsigned char *reg_8b_value)	
Parameter	**Description**
reg_addr	The address of register that user wants to read.
reg_8b_value	Pointer to the buffer to hold the data read back from the address.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to return 8-bit data read from the peripheral register.	

reg_8b_modify	
unsigned char reg_8b_modify(unsigned int reg_addr, unsigned char bits_mask, unsigned char value)	
Parameter	**Description**
reg_addr	The address of register of which user wants to modify the value.
bits_mask	Bits that are modified within the register.
value	The value that user wants to write to the register. Only masked bits are affected.
Returns	**Description**
unsigned char	Return 0 if no error.
Description	
This function is used to modify the masked bits of the specified register value.	

2.2.3.2. API Usage Example

The code episode below shows the typical usage of the APIs to access the 32-bit memory mapped register of the peripherals. You can read or write the peripheral with the memory mapped address as the parameter. You can also modify the specified bits of the register with *1* as the masks.

```
#include "hal.h"
#define  BITS_MASK      0x0F
#define  BITS_SET       0x03
int main()
{
        unsigned int reg_value = 0;

        /* Read the value of the specified memory mapped address */
        reg_value = reg_32b_read(reg_address);

        /* Write the new value to the specified address */
        reg_value |= BITS_SET;
        reg_32b_write(reg_address, reg_value);

        /* Modify the masked bits of the register */
        reg_32b_modify(reg_address, BITS_MASK, BITS_SET);

        return 0;
}
```

3. General IPs

Lattice Propel provides a set of general IPs for you to build SoC system. Combined with the IP package, BSP provides to help develop the system software.

3.1. GPIO

Lattice GPIO peripheral soft IP provides dedicated interface to configure each GPIO as either an input or an output pin. When configured as an input, the GPIO module can detect the state of a GPIO by reading the state of the associated register. When configured as an output, it takes the value written into the associated register and control the state of the controlled GPIO.

The APIs of the GPIO module provides a set of interfaces for you to control the GPIOs easily.

3.1.1.1. API Reference

gpio_init	
unsigned char gpio_init(struct gpio_instance *this_gpio, unsigned int base_addr, unsigned int gpio_num, unsigned int gpio_dirs)	
Parameter	**Description**
this_gpio	The pointer to the instance of the current GPIO device.
base_addr	Base address of the GPIO module, Propel SDK =automatically parses the address map of the SoC system and passes the information to the software.
gpio_num	The number of the GPIOs the module supports. The number should be between 1 to 32.
gpio_dirs	The direction of the GPIOs that user wants to set. Each bit specifies the direction of the corresponding GPIO. 0 : GPIO for input. 1 : GPIO for output.
Returns	**Description**
unsigned char	0 : Succeeded in initializing the GPIO module. 1 : Failed to initialize the GPIO module.
Description	
This function is supposed to be called when the platform is initialized. This function should be called before calling any GPIO related functions.	

gpio_set_direction	
unsigned char gpio_set_direction(struct gpio_instance *this_gpio, unsigned int index, unsigned int gpio_dir)	
Parameter	**Description**
this_gpio	The pointer to the instance of the current GPIO device.
index	The value of the lines index of the GPIO.
gpio_dir	The direction of the GPIO that user wants to set. 0 : the GPIO for input use. 1 : the GPIO for output use.
Returns	**Description**
unsigned char	0 : Succeeded in setting the direction for a specified GPIO pin. 1 : Failed to set the direction.
Description	
This function is used to set the GPIO direction for the specified GPIO pin.	

gpio_output_write	
unsigned char gpio_output_write(struct gpio_instance *this_gpio, unsigned int index, unsigned int value)	
Parameter	**Description**
this_gpio	The pointer to the instance of the current GPIO device.
index	The value to specify the GPIO pins to output.
value	The value of the output of the GPIO.
Returns	**Description**
unsigned char	0 : Succeeded in outputting the power level through the GPIO pin. 1 : Failed to output the power level through the GPIO pin.
Description	
This function is used to write the output data to the specified GPIO pin.	

gpio_input_get	
unsigned char gpio_input_get(struct gpio_instance *this_gpio, unsigned int index, unsigned int *data)	
Parameter	**Description**
this_gpio	The pointer to the instance of the current GPIO device.
index	The number of the GPIO pin that user wants to read the value.
data	The pointer to the data buffer to hold the GPIO input status.
Returns	**Description**
unsigned char	0 : Succeeded in getting the input value of the specified GPIO pin. 1 : Failed to get the input value of the specified GPIO pin.
Description	
This function is used to read the input data of the specified GPIO pin.	

3.1.1.2. API Usage Example

The typical usage of the GPIO is to output different power levels through the corresponding pin, or to get the input level on the GPIO when the direction is set as input. The example code below shows the control to two GPIO pins: GPIO0 is used for output, while GPIO1 is used as input.

```
#include “gpio.h”
#define   NUM_GPIO        0x08
#define   GPIO0           0x01
#define   GPIO1           0x02
int main()
{
        struct gpio_instance gpio_inst;
        unsigned int input_val = 0;

        /* Initialize the GPIO instance with base address and supported number of pins, default output*/
        gpio_init(&gpio_inst, GPIO0_INST_BASE_ADDR, NUM_GPIO, 0xff);

         /* Set GPIO1 as input */
        gpio_set_direction(&gpio_inst, GPIO1, GPIO_INPUT);
```

```
    /* Output low(0) on GPIO0*/
    gpio_output_write(&gpio_inst, GPIO0, 0);

    /* Get the input value from GPIO1 */
    gpio_input_get(&gpio_inst, GPIO1, &input_val);

    return 0;
}
```

3.2. UART

Lattice UART is a universal asynchronous receiver-transmitter used to interface to RS232 serial devices. The APIs provide a set of interface for you to configure the UART device or communicate via the RS232 interface easily.

3.2.1.1. API Reference

uart_init	
unsigned char uart_init(struct uart_instance *this_uart, unsigned int base_addr, unsigned int sys_clk, unsigned int baud_rate, unsigned char stop_bits, unsigned char data_width)	
Parameter	**Description**
this_uart	The pointer to the instance of the current UART device.
base_addr	Base address of the UART module. Propel SDK automatically parses the address map of the SoC system and passes the information to software.
sys_clk	The frequency of the system clock.
baud_rate	The value of the baud rate of the UART.
stop_bits	The value of the stop-bit of the UART. Stop-bit is the end flag of one data frame. The value can be set as 1 or 2.
data_width	The value of the data width of the UART. Data width indicates the valid data bits in one data frame. The value can be set as 5, 6, 7 or 8. Normally, 8 is the general data width.
Returns	**Description**
unsigned char	0 : Succeeded in initializing the UART module. 1 : Failed to initialize the UART module.
Description	
This function is used to Initializes UART instance. This function is supposed to be called when the platform is initializing. This function should be called before calling any UART related functions.	

uart_putc	
unsigned char uart_putc(struct uart_instance * this_uart, unsigned char ucChar)	
Parameter	**Description**
this_uart	The pointer to the instance of the current UART device.
ucChar	The value of the character that user wants to send out over the UART interface.
Returns	**Description**
unsigned char	0 : Succeeded in outputting a character through the UART interface. 1 : Failed to output a character through the UART interface.
Description	
This function is used to send a character over the UART.	

uart_getc	
unsigned char uart_getc(struct uart_instance * this_uart, unsigned char *pucChar)	
Parameter	**Description**
this_uart	The pointer to the instance of the current UART device.
pucChar	The pointer to the character received from the UART.
Returns	**Description**
unsigned char	Return value: 0 : Succeeded in getting a character from the UART interface. 1 : Failed to receive a character from the UART interface.
Description	
This function is used to retrieve a character from the UART.	

uart_set_rate	
unsigned char uart_set_rate(struct uart_instance * this_uart, unsigned int baudrate)	
Parameter	**Description**
this_uart	The pointer to the instance of the current UART device.
baudrate	The value of the baud rate of the UART.
Returns	**Description**
unsigned char	0 : Succeeded in setting the baud rate for the UART device. 1 : Failed to set the baud rate for the UART device.
Description	
This function is used to change the baud rate of UART device.	

uart_config	
unsigned char uart_config(struct uart_instance * this_uart, unsigned int dwidth, unsigned char parity_en, unsigned char even_odd, unsigned int stopbits)	
Parameter	**Description**
this_uart	The pointer to the instance of the current UART device.
dwidth	The value of the data width of the UART. Data width indicates the valid data bits in one data frame. The value can be set as 5, 6, 7 or 8. 8 is the most commonly-used data width.
parity_en	The value of the parity of the UART. 0 : No parity. 1 : Parity enabled.
even_odd	The value of the even_odd of the UART. 1 => even, 0 => odd
stopbits	The value of the stop-bit of the UART. Stop-bit is the end flag of one data frame. The value can be set as 1 or 2.
Returns	**Description**
unsigned char	0 : Succeeded in configuring the UART device with new settings. 1 : Failed to configure the UART device with new settings.
Description	
This function is used to configure the data width, parity, even odd check and stop-bit for the UART device at runtime.	

3.2.1.2. API Usage Example

The typical usage of the UART is to output or input a character. For input, *uart_getc()* needs to be called. For output, a redirected printf() is provided, which can be directly used to output the strings or integer via the UART interface.

```
#include “uart.h”
#include <stdio.h>
#define BAUD_RATE_SET   9600
int main()
{
        struct uart_instance uart_inst;
        unsigned char in_char = 0;

        /* Initialize the UART instance */
        uart_init(&uart_inst, UART0_INST_BASE_ADDR, CPU_FREQUENCY,
                        UART0_INST_BAUD_RATE, UART0_INST_LCR_STOP_BITS, UART0_INST_LCR_DATA_BITS);
        /* Modify the baud rate, or default is configured during IP generation */
        uart_set_rate(&uart_inst, BAUD_RATE_SET);

        /* Get a character form the UART interface */
        if ( uart_getc(&uart_inst, &in_char) == 0 )
        {
                /* Output the received character via UART by calling printf()*/
                printf(“The input character is %c.\n”, in_char);
        }
        return 0;
}
```

3.3. EFB

The Embedded Function Block (EFB) is a hard architectural block in Lattice FPGA device. The EFB driver provides a set of APIs for User Flash Memory (UFM) access and I²C slave function.

efb_init	
Unsigned char efb_init(struct efb_instance *this_efb, unsigned int base_addr)	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
base_addr	Base address of the EFB module. Propel SDK automatically parses the address map of the SoC system and passes the information to software.
Returns	**Description**
unsigned char	0 : Succeeded in initializing the EFB module. 1 : Failed to initialize the EFB module.
Description	
This function is used to initialize the EFB device. The function is supposed to be called when the platform is initializing. This function should be called before calling any EFB related functions.	

3.3.1. UFM Access

The UFM is a general purpose Flash Memory, which is typically used to store system-level data, Embedded Block RAM initialization data, or executable code for microprocessors. The UFM is a flash sector that is organized in pages, each of which has 128 bits (16 bytes).

3.3.1.1. API Reference

ufm_page_read	
unsigned char ufm_page_read (struct efb_instance *this_efb, unsigned int pageno, unsigned int ufm, unsigned char *data, unsigned char *checksum)	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
pageno	The page number you want to read.
ufm	The UFM number you want to read. To MachXO3D devices, four UFMs are supported: • UFM0 • UFM1 • UFM2 • UFM3
data	The pointer to the buffer that stores the data read from EFM page.
checksum	The pointer to the buffer to store the check sum of the data read from the page.
Returns	**Description**
unsigned char	0 : Succeeded in reading data from the specified UFM page. 1 : Failed to read data from the specified UFM page.
Description	
This function is used to read the whole data from the specified page of UFM. See ufm_page_write() for how to write data into UFM page.	

ufm_page_write	
unsigned char ufm_page_write (struct efb_instance *this_efb, unsigned int pageno, unsigned int ufm, unsigned char *data, unsigned char *checksum);	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
pageno	The page number you want to write to.
ufm	The UFM number you want to read. To MachXO3D devices, four UFMs are supported: • UFM0 • UFM1 • UFM2 • UFM3
data	The pointer to the buffer that stores the data written to EFM page.
checksum	The pointer to the buffer to store the check sum of the data write to the page.
Returns	**Description**
unsigned char	0 : Succeed in writing data into the specified UFM page. 1 : Failed to write data into the specified UFM page.
Description	
This function is used to write the data into the specified pages of UFM. See ufm_page_read() for how to read data from UFM page.	

ufm_page_erase	
unsigned char ufm_erase(struct efb_instance *this_efb, unsigned int ufm);	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
ufm	The UFM number you want to read. To MachXO3D devices, four UFMs are supported: UFM0 UFM1 UFM2 UFM3
Returns	**Description**
unsigned char	0 : Succeeded in erasing all the data from the specified UFM. 1 : Failed to erase all the data from the specified UFM.
Description	
This function is used to erase the specified UFM. Note that before writing data to the UFM, erase should be performed to make sure the data can be written successfully.	

ufm_byte_write	
unsigned char ufm_byte_write(struct efb_instance *this_efb, unsigned int pageno, unsigned char byteno, unsigned int ufm, unsigned char data)	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
pageno	The page number you want to write to.
byteno	The byte number of the page you want to write to.
ufm	The UFM number you want to read. To MachXO3D devices, four UFMs are supported: • UFM0 • UFM1 • UFM2 • UFM3
data	The data that written to the page.
Returns	**Description**
unsigned char	0 : Succeeded in writing a byte to the specified UFM page. 1 : Failed to write a byte to the specified UFM page.
Description	
This function is used to update a single byte in the UFM page. Do not change other data in the page.	

ufm_byte_read	
unsigned char ufm_byte_read(struct efb_instance *this_efb, unsigned int pageno, unsigned char byteno, unsigned int ufm, unsigned char *data)	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
pageno	The page number you want to write to.
byteno	The byte number of the page you want to write to.
ufm	The UFM number you want to read. To MachXO3D devices, four UFMs are supported: • UFM0 • UFM1 • UFM2 • UFM3
data	The pointer to the buffer that holds the data read back.
Returns	**Description**
unsigned char	0 : Succeed in reading a byte to the specified UFM page. 1 : Failed to read a byte to the specified UFM page.
Description	
This function is used to read the specified byte of the page in the UFM.	

3.3.1.2. API Usage Example

The code episode in this section shows how to read, erase, and write to the UMF. Note that the UMF is implemented with flash memory. You should erase the UFM before writing data into it. Or, there comes unexpected behaviors.

```
#include "efb.h"
int main()
{
        struct efb_instance efb_inst;
        unsigned char check_sum = 0;
        unsinged char data_buffer[16];

        /* Initialize the EFB module before access the UFM */
        efb_init(&efb_inst, EFB0_INST_BASE_ADDR);

        /* Read the page0 of UFM1 into data buffer */
        ufm_page_read (&efb_inst, 0, UFM1, data_buffer, &checksum);

        /* Erase the UFM1 before write */
        ufm_erase(&efb_inst, unsigned int ufm);

        /* Write data into page1 of UFM1 */
        ufm_page_write (&efb_inst, 1, UFM1, data_buffer, &check_sum);

        return 0;
}
```

3.3.2. I^2C Slave

The EFB I^2C module can work as both master mode and slave mode. In this Propel 1.0, only slave mode is supported.

When working in the slave mode, interrupt service routine needs to be implemented and registered into the system interrupt handling framework. When the I^2C master device accesses to the slave device, an interrupt occurs, calling the interrupt handling routine to provide a proper response.

i2cslave_isr	
void (*i2cslave_isr)(void *ctx)	
Parameter	**Description**
ctx	The pointer to the context that the interrupt service routine runs in.
Returns	**Description**
void	—
Description	
The interrupt service routine you need to implement to handle the interrupts. This function should be registered via efb_i2c2_isr_register().	

efb_i2c2_isr_register	
unsigned char efb_i2c2_isr_register(struct efb_instance *this_efb, struct i2c_desc *i2c)	
Parameter	**Description**
this_efb	The pointer to the instance of the current EFB device.
i2c	The pointer to the I^2C descriptor holding the data buffer and callback function you provided.
Returns	**Description**
unsigned char	0 : Succeeded in registering a callback for the I^2C device. 1 : Failed to register a callback for the I^2C device.
Description	
This function is used to register a customer implemented interrupt service routine for EFB I^2C2.	

3.3.2.1. API Usage Example

Normally, the EFB I²C driver provides the default interrupt service routine for the slave function. While in some cases, customer needs to implement his/her own interrupt service routine to handle the data transactions on the bus. Following code gives an example on how to register your own ISR for the EFB I²C slave.

```
include "efb.h"

/* the i2c slave interrupt service routine user implemented based on requirement */
void cstm_i2c_isr(void *ctx);

int main()
{
        struct efb_instance efb_inst;
        char buffer[64];
        struct i2c_desc cstm_i2c_des = {buffer, cstm_i2c_isr};

        /* Initialize the PIC with base address and the number of interrupt source */
        pic_init(CPU0_INST_PICTIMER_START_ADDR, INT_NUM);

        /* Initialize the EFB module before access the i2c slave device */
        efb_init(&efb_inst, EFB0_INST_BASE_ADDR);

        // register I2C slave callback, this assumes no other interrupts sources coming from EFB
        efb_i2c2_isr_register(&efb_inst, &cstm_i2c_desc);

        while (1)
        {
                /*customer can add the i2c data handling code here based on its own isr*/
                ……
        }
        return 0;
}
```

Reference

- GPIO IP Core (FPGA-IPUG-02076)
- UART IP Core – Lattice Propel Builder (FPGA-IPUG-02105)
- RISC-V MC CPU IP Core – Lattice Propel Builder (FPGA-IPUG-02114)

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

Revision History

Revision 1.0, May 2020

Section	Change Summary
All	Initial release.

www.latticesemi.com