Z8 Encore!® Family of Microcontrollers

Zilog Standard Library API

Reference Manual

RM003805-0508

www.zilog.com

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPO NENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Revision History

Each instance in Revision History reflects a change to this document from its previous revision. For more details, refer to the corresponding pages and appropriate links in the table below.

Date	Revision Level	Description	Page No
May 2008	05	Updated UART Initialization in the Startup Routine, Table 12, replaced ZDS II v4.10.1 with ZDS II v4.11.0.	72, 74
April 2008	04	Updated Zilog logo, Zilog text, Disclaimer section and implemented Style Guide. Updated Building the Zilog Standard Libraries section.	All
October 2006	03	Updated for Z8 Encore! XP F1680 Series library files.	All

Table of Contents

Introduction

This reference manual describes Zilog Standard Library (ZSL) and ZSL application programming interfaces (APIs). ZSL is available as part of the Zilog Developer Studio II-Integrated Development Environment (ZDS II–IDE) v4.11.0 release for Zilog's Z8 Encore!® product line of microcontrollers.

ZSL is a set of library files that provides an interface between user application and on-chip peripherals of Z8 Encore! microcontrollers.
Z8 Encore! XP products include F64XX, F0822, F042A series of microcontrollers with 1 KB to 64 KB memory sizes.

About This Manual

Zilog recommends you to read and understand this manual completely before using the product. This manual is designed to be used as a reference guide for ZSL APIs.

Intended Audience

This document provides relevant information on ZSL implementation. This reference manual serves as a guide for interfacing the user application with on-chip peripherals of the Z8 Encore! microcontrollers.

Manual Organization

This manual is divided into three chapters as described briefly below:

ZSL Overview

This chapter provides an overview of ZSL, ZSL directory structure, and ZSL release and debug versions.

ZSL GPIO API Description

This chapter provides information on how to interface a user application with the Z8 Encore! microcontroller GPIO peripheral and details of the APIs provided to interface with it.

ZSL UART API Description

This chapter provides information on how to interface a user application with the Z8 Encore! microcontroller UART peripheral(s) and details of the APIs provided to interface with it.

Related Documents

In addition to this manual, you must be familiar with the documents listed in Table 1.

Table 1. Related Documents

Document Title	Document Number
Zilog Developer Studio II—Z8 Encore!® User Manual	UM0130
ZPAKII Debug Interface Tool Product User Guide	PUG0015
Z8 Encore! XP® F0822 Series Flash MCU Evaluation Kit Quick Start Guide	QS0025
Z8 Encore! XP® F64XX Series Development Kit Quick Start Guide	QS0028
eZ8 CPU User Manual	UM0128

Latest software and updated documents are available for download at www.zilog.com.

Abbreviations and Expansion

Table 2 lists the abbreviations/acronyms used in this document.

Table 2. Abbreviations and Expansion

Abbreviations/ Acronyms	Expansion
ADDR	Address Register
ANSI	American National Standards Institute
API	Application Program Interface
CTL	Control Register
DMA	Direct Memory Access
EOF	End of File (a macro defined in the `stdio.h` file)
GPIO	General-Purpose Input/Output
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
PA	GPIO Port A
PB	GPIO Port B
PC	GPIO Port C
PD	GPIO Port D
PE	GPIO Port E
PF	GPIO Port F
PG	GPIO Port G
PH	GPIO Port H
PRAM	Program RAM
RTL	ANSI C Run-Time Library
UART	Universal Asynchronous Receiver/Transmitter

Table 2. Abbreviations and Expansion (Continued)

Abbreviations/ Acronyms	Expansion
ZDS II	Zilog Developer Studio II
ZSL	Zilog Standard Library

Conventions

The following assumptions and conventions are adopted to provide clarity and ease of use:

Courier Typeface

Commands, code lines and fragments, bits, equations, hexadecimal addresses, and various executable items are distinguished from general text by the use of the `Courier` typeface.

Hexadecimal Values

Hexadecimal values are designated by a lowercase h and appear in the `Courier` typeface.

- Example: STAT is set to `F8h`.

Asterisks

An asterisk preceding a parameter denotes the parameter as a pointer.

Safeguards

It is important that you understand the following safety terms, which are defined here.

Caution: *This symbol means a procedure or file can become corrupted if you do not follow directions.*

In general, when using ZSL in conjunction with the ZDS II-IDE and any one of Zilog's development platforms, follow the precautions listed below to avoid permanent damage to the platform.

Caution: *Always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD).*

1. Power-up precautions
   (a) Apply power to the PC and ensure that it is running properly.
   (b) Start the terminal emulator program on the PC.
   (c) Apply power through the appropriate connector on the development platform.
2. Power-down precautions
   (a) Quit the monitor program.
   (b) Remove power from the development platform.

Online Information

Zilog website provides valuable product information, documentation, and downloads of the latest production-released version of the ZDS II development tool. The following documents are available for download at www.zilog.com:

- Product Specifications
- User Manuals
- Application Notes
- Reference Manuals
- Product Briefs

ZSL Overview

This chapter provides an overview of Zilog Standard Library (ZSL), ZSL architecture, debug and release versions of ZSL, and how to build libraries using the batch (script) files. The startup routine and a summary of ZSL APIs are also included in this chapter.

ZSL for Z8 Encore!® is a set of library files, which contains device driver APIs to program various on-chip peripherals of Z8 Encore! microcontroller. Each library contains device drivers which allow you to communicate with on-chip peripherals or devices without much knowledge of their register and programming details.

ZSL APIs are easy to use, refer to the source code files provided with ZSL release to modify the libraries to suit specific requirements.

Zilog Standard Library Architecture

Figure 1 displays a block diagram of ZSL architecture.

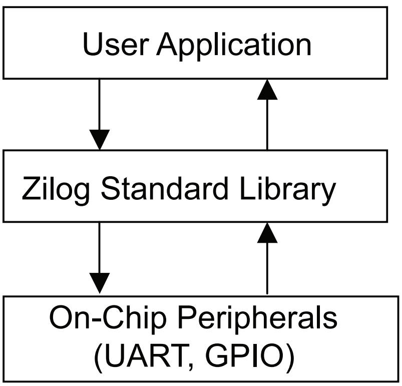

Figure 1. Block Diagram of ZSL Architecture

ZSL for Z8 Encore!® consists of various libraries, each of which is used for a specific memory model and configuration. Table 3 describes each of these libraries.

Table 3. Z8 Encore! ZSL Constituent Libraries

Library Name	Description
`zslSY.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—no debug information, speed optimization
`zslSYD.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—with **D**ebug information, no optimization.
`zslST.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—no debug information, speed optimization
`zslSTD.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—with **D**ebug information, no optimization
`zslLY.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—no debug information, speed optimization
`zslLYD.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—with **D**ebug information, no optimization
`zslLT.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—no debug information, speed optimization
`zslLTD.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—with **D**ebug information, no optimization

ZSL also provides various libraries for Z8 Encore! XP® F1680 Series, each of which is used for a specific memory model and configuration. Table 4 describes each of the library files for Z8 Encore! XP F1680 Series.

Table 4. ZSL Library Files for Z8 Encore! XP F1680 Series

Library Name	Description
`zslF1680SY.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680SYD.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—with **D**ebug information, no optimization.
`zslF1680ST.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680STD.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—with **D**ebug information, no optimization.
`zslF1680LY.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680LYD.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—with **D**ebug information, no optimization
`zslF1680LT.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680LTD.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—with **D**ebug information, no optimization

Z8 Encore! XP® F1680 Series implements a new feature of user-controlled Program RAM (PRAM) area to store Interrupt Service Routines (ISRs) of high-frequency interrupts. The PRAM mechanism ensures low-average current and quick response for high frequency interrupts. To avail this feature, the ISRs in ZSL UART must be provided with the option of being placed in the PRAM segment. To enable this, the ZDS II IDE provides a check box in ZSL tab named **Place ISR into PRAM**. When you select this check box, ZDS II addresses the library `zslF1680U0XXX.lib` or `zslF1680U1XXX.lib` or both to place ISRs for UART0 and UART1 in PRAM.

Note: ***Place ISR into PRAM*** *feature is effective only when the UART is set in interrupt mode. To set the UART in interrupt mode, edit the header file* `include\zilog\uartcontrol.h` *by defining the symbol* `UART0_MODE/UART1_MODE` *as* `MODE_INTERRUPT`*, and rebuild the libraries. For more information on rebuilding ZSL, see* Building the Zilog Standard Libraries *on page 9.*

For Z8 Encore! XP F1680 Series the default ZSL libraries are in `zslF1680XXX.lib` files. The following functions are placed in PRAM segment within each libraries:

`zslF1680U0XXX.lib`:

- `void isr_UART0_RX( void )`
- `void isr_UART0_TX( void )`

`zslF1680U1XXX.lib`:

- `void isr_UART1_RX( void )`
- `void isr_UART1_TX( void )`

Table 5 lists library files to place ISRs for UART0 in PRAM.

Table 5. Library Files to place ISRs for F1680 Series UART0 in PRAM

Library Name	Description
`zslF1680U0SY.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680U0SYD.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—with **D**ebug information, no optimization.
`zslF1680U0ST.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680U0STD.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—with **D**ebug information, no optimization.
`zslF1680U0LY.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680U0LYD.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—with **D**ebug information, no optimization
`zslF1680U0LT.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680U0LTD.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—with **D**ebug information, no optimization

Table 6 lists the library files to place ISRs for UART1 in PRAM.

Table 6. Library Files to place ISRs for F1680 Series UART1 in PRAM

Library Name	Description
`zslF1680U1SY.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680U1SYD.lib`	Drivers for applications with a **S**mall memory model and using d**Y**namic frames—with **D**ebug information, no optimization.
`zslF1680U1ST.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680U1STD.lib`	Drivers for applications with a **S**mall memory model and using s**T**atic frames—with **D**ebug information, no optimization.
`zslF1680U1LY.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—no debug information, speed optimization
`zslF1680U1LYD.lib`	Drivers for applications with a **L**arge memory model and using d**Y**namic frames—with **D**ebug information, no optimization
`zslF1680U1LT.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—no debug information, speed optimization
`zslF1680U1LTD.lib`	Drivers for applications with a **L**arge memory model and using s**T**atic frames—with **D**ebug information, no optimization

Zilog Standard Library Directory Structure

Figure 2 displays the directory structure of ZSL. Table 7 lists the files contained in each sub-directory.

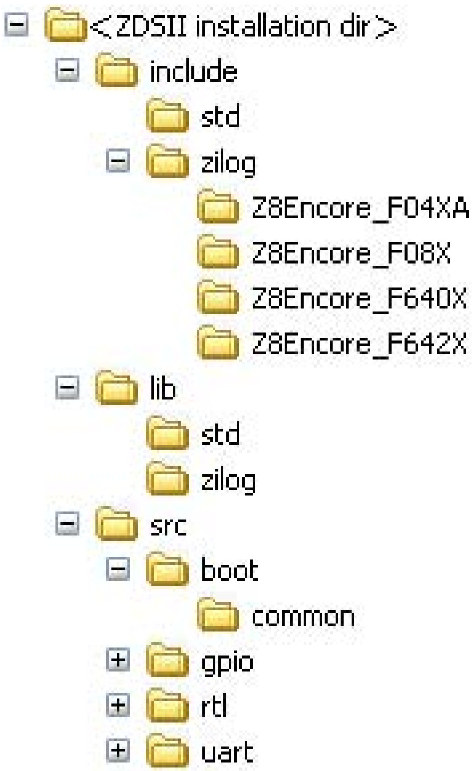

Figure 2. ZSL Directory Structure

Note: *In Figure 2,* `<ZDS installation dir>` *specifies the root directory of ZDS II installation—for example, ZDSII_Z8 Encore!_4.11.0.*

Table 7. ZSL Directory Structure Description

Path\Folder	Description
\include	Contains subfolders that contain the include files
\include\std	Contains all the header files relevant to the C Run Time Library (RTL)
\include\zilog	Contains header files relevant to ZSL device drivers
\include\zilog\<series>	Contains boot-related files specific to each Z8 Encore!® series
\lib	Contains subfolders that contain the libraries files
\lib\std	Contains all the library files relevant to the C Run Time Library (RTL)
\lib\zilog	Contains all the library files relevant to the device drivers
\src	Contains subfolder which contains source for each of the device
\src\boot\common	Contains boot-related files common to all targets
\src\<device>\common	Contains device-related files common to all targets

Note: `<series>` denotes the Z8 Encore! series.
`<device>` denotes the on-chip peripheral device; for example, GPIO or UART.

ZSL Debug and Release Version

There are two ZSL versions—the debug and release version available for each Z8 Encore!® on-chip peripheral or device. The debug version of the library is built to contain debug information without any optimizations, whereas the release version is built to contain no debug information and is optimized for speed. The debug version of the library is built with the macro `DEVICE_PARAMETER_CHECKING` defined (where `DEVICE` is any device such as UART or GPIO), which is used by some of the APIs to check for the validity of the parameters passed. This macro is absent in ZSL release version, which does not perform any check on the API parameters. Thus, there is a significant difference in overall size of the generated library from the two versions. See individual APIs in this manual to check whether an API uses the `DEVICE_PARAMETER_CHECKING` macro or not.

Building the Zilog Standard Libraries

You can develop applications using the APIs provided for specific peripherals and make use of the Zilog Standard Library to interface with the peripherals on the Z8 Encore! microcontrollers. However, for those who require to customize the library files by modifying the source code, this section describes how the modified library is built using the batch files and ZDS II script files.

As a general rule, when the batch files are executed, the libraries for each on-chip peripheral or device are rebuilt and copied into the `<ZDS installation dir>\lib\zilog` folder. The source directory contains one single batch file to build all the libraries of all the devices. Follow the steps below to build the library:

1. **Generating ZDS II project file:** In this step, a ZDS II project is created for the specific target microcontroller using a ZDS II script file. The script file used for this purpose has the same name as the calling batch file with a `.scr` extension. The script file creates a ZDS II project and configures the project settings for both the debug

and release versions of the library. The script then calls other script files to add all source files of different devices that make the library. So the batch file, `gen_zsl_project.bat`, generates the project file. It calls `gen_zsl_project.scr` script file to create ZDS II project and invoke other script files, `add_gpio_projectfiles.scr` and `add_uart_projectfiles.scr`, to add all the source files relevant to the library.

2. **Generating Make files:** From the project generated in step 1, *make* files for both debug and release versions are generated using a batch file and a ZDS II script file. The batch file, `gen_zsl_project.bat`, invokes a ZDS II script file, `gen_zsl_makefiles.scr`, to create both the debug and release versions of the make files.

3. **Generating libraries:** The *make* files generated in step 2 are used along with ZDS II to finally generate the debug and release versions of the library. The libraries are automatically copied to the repository under the `<ZDS installation dir>\lib\zilog` directory. The batch file, `process_zsl_makefiles.bat`, generates all the libraries as listed in Table 3 on page 2.

Notes: 1. *The batch file* `buildallzsl.bat` *allows you to build all libraries for ZSL.*

2. *'The ZSL fast call libraries ('register' parameter passing) are named with an extension 'F' in the name. For example, the fast call lib for* `zslLY.lib` *is* `zslLYF.lib` *and* `zslLYD.lib` *is* `zslLYFD.lib`*'. Also note that these libraries will be included automatically when 'register' parameter passing is selected.*

Startup Routine

The ZSL is integrated with ZDS II, which allows you to choose the device(s) required for the user application, and also specifies some of the device-dependent parameters. Select **Project** → **Settings** → **ZSL** in ZDS II interface to choose the device and to specify the device parameters. For information on using ZSL from within ZDS II, refer to the *Zilog Developer Studio II—Z8 Encore!® User Manual (UM0130)* available with the ZDS II tool package or on www.zilog.com.

ZDS II copies the Zilog Standard Library device initialization file `zsldevinit.asm` into the user project when ZSL is selected from within ZDS II. The initialization file contains the `_open_periphdevice()` function that calls the initialization routines for all devices used in the user-application. The `_open_periphdevice()` routine is invoked from the `startup` routine before the `main()` function is called. Depending on the device selected, ZDS II defines specific macros for each device. For details on initialization of the specific devices, see chapters on the API descriptions of the specific devices.

The user application initializes the required device(s) to their default values without calling the `startup` routine. To do so, the user application must call the `_open_periphdevice()` function before making any specific calls to the device(s).

Zilog Standard Library API Overview

This section provides a brief overview on topics related to the APIs provided by ZSL to write applications which use the peripheral devices on Z8 Encore!® microcontrollers.

Standard Data Types

ZSL makes use of the user-defined data types in all APIs. These user-defined data types are defined in the header file `defines.h`, located in the following directory:

```
<ZDS installation dir>\include\zilog
```

API Definition Format

Descriptions for each ZSL API follows a standard format. In this document, header file names are listed at the top of each page, followed by the API description. A brief discussion of the format for each API description follows.

Prototype

This section contains the declaration of the API call.

Description

This section describes the API.

Argument(s)

This section describes the arguments (if any) to the API.

Return Value(s)

This section describes the return value of the API, if any.

Example(s)

This section provides examples of how the API function is called.

Table 8 lists the Z8 Encore!® devices for which ZSL APIs are provided with the current release of ZDS II—Z8 Encore! v4.11.0.

Table 8. List of ZSL APIs for Z8 Encore! On-Chip Devices

Device Name	Type of APIs	Description
UART	UART (Generic) APIs	These APIs are the standard RTL I/O routines.
	UART*x* APIs	These APIs are specific for a particular UART device, either UART0 or UART1. The *x* in the API name represents the selected UART device.
GPIO	GPIO*x* APIs	These APIs are specific for the GPIO Ports A, B, C, D, E, F, G, and H. The *x* in the API name represents the selected GPIO Port.

ZSL GPIO API Description

This chapter provides detailed descriptions of the Zilog Standard Library (ZSL) general-purpose input/output (GPIO) APIs.

To use ZSL GPIO APIs, the file `gpio.h` must be included in the application program.

GPIO Port Initialization in the Startup Routine

ZSL is integrated with ZDS II, allowing you to select or deselect Z8 Encore!® MCU GPIO ports (see Startup Routine on page 10). When a GPIO port is selected in ZDS II interface using **Project→Settings → ZSL**, ZDS II generates a compiler pre-define, `_ZSL_DEVICE_PORTx`, where x is any one of the A, B, C, D, E, F, G, or H GPIO ports.

ZDS II also adds a device initialization file, `zsldevinit.asm`, into the user project. The `zsldevinit.asm` file uses compiler pre-defines (macros) to initialize the ports to their default state. The function `_open_periphdevice()` in `zsldevinit.asm` calls the ZSL GPIO API `open_Portx()` function for each of the ports selected from within the ZDS II interface.

GPIO APIs

Z8 Encore!® family of microcontrollers support eight different ports named Port A through Port H. However, not all the ports are available on all devices in the Z8 Encore! family. A given port can have different features on different devices. So there are two kinds of GPIO APIs:

- **Common APIs**—For features that are common across all devices in the Z8 Encore! family. Table 9 lists common APIs with hyperlinks to their descriptions.
- **Target Specific APIs**—For features that are present only on some variants of Z8 Encore! family. Table 10 on page 15 lists specific APIs with applicable target devices and ports.

In addition to APIs, ZSL defines a number of GPIO-related macros. For more information, see ZSL GPIO Macros on page 70.

Table 9. ZSL Common GPIO APIs

API Name	Description
open_Portx()	Opens a specified GPIO Port
control_Portx()	Configures a specified GPIO Port
setmodeInput_Portx()	Sets Port bits for Input mode
setmodeOutput_Portx()	Sets Port bits for Output mode
setmodeOpendrain_Portx()	Sets Port bits for Open Drain mode
setmodeHighDrive_Portx()	Sets Port bits to a High Drive enable mode
setmodeStopRecovery_Portx()	Sets Port bits to a Stop Recovery mode
close_Portx()	Closes a specified GPIO Port

Table 10. ZSL Target Specific GPIO APIs

API Name	Description and Valid Z8 Encore! Devices/Ports
setmodePullUp_Portx()	Set port bits to PullUp mode. F08 Series: Ports A, B, and C XP Series: Ports A, B, C, and D 4 K Series: Ports A, B, and C
setmodeAltFunc_Portx()	Set port bits to Alternate function mode. F04 Series: Port A F64XX Series: ports A, B, C, D, and H 4K Series: Port A All other Z8 Encore! devices: Ports A, B, and C
setmodeInterrupt_PortC()	Set Port C bits (0 to 3 bits) to Interrupt mode All Z8 Encore! devices: Port C
setmodeInterrupt_PortA_XP()	Set Port A bits to Interrupt mode
setmodeInterrupt_PortA_8Pn()	Set Port A bits to Interrupt mode
setmodeInterrupt_PortA_4K()	Set Port A bits to Interrupt mode
setmodeInterrupt_PortA_F08()	Set Port A bits to Interrupt mode 8 K series: Port A.
setmodeInterrupt_Portx_F64()	Set Port bits to Interrupt mode 64 K series: ports A and D
setmodeInterrupt_Portx_F1680()	Set Port bits to Interrupt mode F1860 series: ports A and D
setmodeAltFuncSet1_Portx()	Set Port bits to Alternate Function Set-1 mode XP series: ports B and C

Table 10. ZSL Target Specific GPIO APIs (Continued)

API Name	Description and Valid Z8 Encore! Devices/Ports
setmodeAltFuncSet2_Portx()	Set Port bits to Alternate Function Set-2 mode XP series: ports B and C 4 K series: ports B and C 8 Pin Devices: Port A
setmodeAltFuncSet3_PortA()	Set Port A bits to Alternate Function Set-3 mode 8 Pin Devices: Port A
setmodeAltFuncSet4_PortA()	Set Port A bits to Alternate Function Set-4 mode 8 Pin Devices: Port A
setmodeLEDDrive_PortC()	Set Port C bits to LED Drive mode XP series: Port C

open_Portx()

Prototype

```
void open_Portx();
```

Description

The `open_Portx()` API opens the selected port by initializing the port registers to input mode. The appropriate port register values are defined in the `gpio.h` file.

Argument(s)

None.

Return Value(s)

None.

Example

```
#include <ez8.h>

void init_ports( void )
{
        /*! open Port A in default (input) mode */
        open_PortA() ;

        /*! open Port B in default (input) mode */
        open_PortB() ;
}
void get_ports( void )
{
        /*! Read Port A pins */
        data1 = PAIN ;

        /*! Read Port B pins */
        data2 = PBIN ;
}
```

control_Portx()

Prototype

```
void control_Portx(PORT * pPort);
```

Description

The `control_Portx()` API sets the values of the selected port registers by using the values in the `PORT` structure parameter. This API is used to set all the registers of the port at one time. To set individual registers, the predefined macros defined in the `gpio.h` file are used.

Argument(s)

`*pPort`	A pointer to the structure of type `PORT` defined in the `gpio.h` file

Return Value(s)

None.

setmodeInput_Portx()

Prototype

```
char setmodeInput_Portx( uchar pins )
```

Description

The `setmodeInput_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the input mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the input mode for Port A Pin 7 (PA7) is set by the values in the registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into input mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeInput_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to input mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into input mode, the API is used as given below:

```
setmodeInput_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the input mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pins for input mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInput_PortA( PORTPIN_ALL ) )
      {
      return -1 ;
      }

      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for input mode */
      if( GPIOERR_INVALIDPINS ==
             setmodeInput_PortB( PORTPIN_ALL ) )
      {
      return -1 ;
      }
}
void get_ports( void )
```

```
{
        /*! Read Port A pins */
        data1 = PAIN ;

        /*! Read Port B pins */
        data2 = PBIN ;
}
```

setmodeOutput_Portx()

Prototype

```
char setmodeOutput_Portx( uchar pins )
```

Description

The `setmodeOutput_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the output mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the output mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into output mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeOutput_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to output mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into output mode, the API is used as given below:

```
setmodeOutput_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the output mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pins for output mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeOutput_PortA( PORTPIN_ALL ) )
      {
      return -1 ;
      }

      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for output mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeOutput_PortB( PORTPIN_ALL ) )
      {
      return -1 ;
      }
}
```

```
void write_ports( void )
{
      /*! Write to Port A pins */
      PAOUT = data1 ;

      /*! Write to Port B pins */
      PBOUT = data2 ;
}
```

setmodeOpendrain_Portx()

Prototype

```
char setmodeOpendrain_Portx( uchar pins )
```

Description

The `setmodeOpendrain_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the open drain mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the open drain mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into open drain mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeOpendrain_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to input mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into input mode, the API is used as given below:

```
setmodeOpendrain_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the open-drain mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pins for open drain mode */
      if( GPIOERR_INVALIDPINS ==
setmodeOpenDrain_PortA
      ( PORTPIN_ALL ))
      {
      return -1 ;
      }

      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for open drain mode */
      if( GPIOERR_INVALIDPINS ==
setmodeOpenDrain_PortB(
            PORTPIN_ALL ))
      {
      return -1 ;
      }
```

```
}
void write_ports( void )
{
   /*! Write to Port A pins (pull-ups are connected to
            these pins)*/
      PAOUT = data1 ;

   /*! Write to Port B pins (pull-ups are connected to
            these pins)*/
      PBOUT = data2 ;
}
```

setmodeHighDrive_Portx()

Prototype

```
char setmodeHighDrive_Portx( uchar pins )
```

Description

The `setmodeHighDrive_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to high drive mode (open-source mode). The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the high drive mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into high drive mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeHighDrive_PortA( PORTPIN_ONE ) ;
```

Similarly, more than one pin is set to input mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into input mode, the API is used as given below:

```
setmodeHighDrive_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the high drive mode successfully.

Example

```
#include <ez8.h>
char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;
      /*! configure Port A pins for high-drive mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeHighDrive_PortA( PORTPIN_ALL ))
      {
      return -1 ;
      }

      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for high-drive mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeHighDrive_PortB( PORTPIN_ALL ))
      {
      return -1 ;
      }
}
void write_ports( void )
{
```

```
        /*!
         * Write to Port A pins (pull-downs are
         * connected to these pins)
         */
        PAOUT = data1 ;

        /*!
         * Write to Port B pins (pull-downs are
         * connected to these pins)
         */
        PBOUT = data2 ;
}
```

setmodeStopRecovery_Portx()

Prototype

```
char setmodeStopRecovery_Portx( uchar pins )
```

Description

The `setmodeStopRecovery_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the stop recovery mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the stop recovery mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into stop recovery mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeStopRecovery_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to stop recovery mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into stop recovery mode, the API is used as given below:

```
setmodeStopRecovery_PortA( PORTPIN_FIVE|PORTPIN_SEVEN
) ;
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins` The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the stop recovery mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pins for stop recovery
       *  source mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeStopRecovery_PortA(PORTPIN_ALL))
      {
      return -1 ;
      }
      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for stop recovery
       *  source mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeStopRecovery_PortB(PORTPIN_ALL))
      {
      return -1 ;
      }
}
```

setmodePullUp_Port*x*()

Prototype

```
char setmodePullUp_Portx( uchar pins )
```

Description

The `setmodePullUp_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the pull up mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the pull up mode for Port A Pin 7 (PA7) is set by the values in the registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into pull up mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodePullUp_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to pull up mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into pull up mode, the API is used as given below:

```
setmodePullUp_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Notes: 1. *This API does not alter states of other pins.*

2. *Pull up mode is supported only in Ports A, B and C of the Z8 Encore!® F08 Series and in Ports A, B, C, and D of the Z8 Encore! XP® Series.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

GPIOERR_INVALIDPINS	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
GPIOERR_SUCCESS	Indicates that the port was configured to the pull up mode successfully.

Example

```
#include <ez8.h>
char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

/*! configure Port A pins for weak
       *  pull-up mode */
      if( GPIOERR_INVALIDPINS ==
            setmodePullUp_PortA( PORTPIN_ALL ))
      {
            return -1 ;
      }
      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for weak
       *  pull-up mode */
      if( GPIOERR_INVALIDPINS ==
            setmodePullUp_PortB( PORTPIN_ALL ))
      {     return -1 ;
}
}
void write_ports( void )
{
      /*! Write to Port A pins */
      PAOUT = data1 ;
      /*! Write to Port B pins */
      PBOUT = data2 ;
}
```

setmodeAltFunc_Portx()

Prototype

```
char setmodeAltFunc_Portx( uchar pins )
```

Description

The `setmodeAltFunc_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore!® microcontroller to the alternate function mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the alternate function mode for Port A Pin 7 (PA7) is set by the values in the registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into alternate function mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeAltFunc_PortA( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to alternate function mode by ORing the pins in the call to the API. For example, to set Pin 5 and Pin 7 of Port A into alternate function mode, the API is used as given below:

```
setmodeAltFunc_PortA( PORTPIN_FIVE|PORTPIN_SEVEN ) ;
```

Note: *This API does not alter states of other pins. The alternate function mode is supported in Port A of the Z8F04 Series, Ports A, B, C, D, and H, of the Z8F64XX Series, and in Ports A, B, and C of all other targets.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

GPIOERR_INVALIDPINS	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
GPIOERR_SUCCESS	Indicates that the port was configured to the alternate function mode successfully.

Example

```
#include <ez8.h>
char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;
      /*! configure Port A pins for
            alternate function mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeAltFunc_PortA( PORTPIN_ALL ))
      {
      return -1 ;
      }
/*! Port A pins are now available for alternate
function mode */
      /*! open Port B in default (input) mode */
      open_PortB() ;

      /*! configure Port B pins for
            alternate function mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeAltFunc_PortB( PORTPIN_ALL ))
      {
      return -1 ;
      }
/*! Port B pins are now available for alternate
function mode */
}
```

setmodeInterrupt_PortC()

Prototype

```
char setmodeInterrupt_PortC( uchar pins, uchar priority )
```

Description

The `setmodeInterrupt_PortC()` API is used to configure one or more pins of the GPIO Port C of Z8 Encore!® microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, to set Pin 1 of Port C into interrupt mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortC(PORTPIN_FIVE,
      INTPRIORITY_NOMINAL ) ;
```

Similarly more than one pin is set to alternate function mode by ORing the pins the API call. For example, to set Pin 5 and Pin 7 of Port C into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortC( PORTPIN_FIVE|PORTPIN_SEVEN,
      INTPRIORITY_HIGH) ;
```

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In debug mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt
void isr_PC1( void )

{
      //! Handle PC1 interrupt here.
}

char init_ports( void )
{
      /*! open Port C in default mode */
      open_PortC() ;

      /*! set the interrupt vector for
            Port C bit one */
      SETVECTOR( PC1_IVECT, isr_PC1 ) ;

      /*! configure Port C pin 1 for interrupt mode */
```

```
        if( GPIOERR_INVALIDPINS ==
              setmodeInterrupt_PortC( PORTPIN_ONE,
              INTPRIORITY_HIGH ))
        {
        return -1 ;
        }
}
```

setmodeInterrupt_PortA_XP()

Prototype

```
char setmodeInterrupt_PortA_XP( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_PortA_XP()` API is used to configure one or more pins of GPIO Port A on the Z8 Encore!® Z8F04 XP Series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set Pin 1 of Port A to falling edge with a low priority call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_XP( PORTPIN_ONE,
      EDGE_FALLING, INTPRIORITY_LOW)
```

Similarly more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_XP( PORTPIN_FIVE|PORTPIN_SEVEN,
      EDGE_RISING, INTPRIORITY_LOW)
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>

#pragma interrupt

void isr_PA1( void )
{
//! Handle PA1 interrupt here.
}

char init_ports( void )
{
 /*! open Port A in default mode */
 open_PortA() ;
```

```
/*! set the interrupt vector for Port A bit one */
 SETVECTOR( PA1_IVECT, isr_PA1 ) ;

 /*! configure Port A pin 1 for interrupt mode */
 if( GPIOERR_INVALIDPINS == setmodeInterrupt_PortA_XP(
PORTPIN_ONE,EDGE_FALLING, INTPRIORITY_HIGH ))
 {
  return -1 ;
 }

}
```

setmodeInterrupt_PortA_8Pn()

Prototype

```
char setmodeInterrupt_PortA_8Pn( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_PortA_8Pn()` API is used to configure one or more pins of GPIO Port A on the Z8 Encore!® Z8F04 8-pin series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set pin 1 of Port A to falling edge with a low priority call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_8Pn( PORTPIN_ONE,
      EDGE_FALLING, INTPRIORITY_LOW)
```

Similarly more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_8Pn
    ( PORTPIN_FIVE|PORTPIN_SEVEN,
      EDGE_RISING, INTPRIORITY_LOW)
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt

void isr_PA1( void )
{
//! Handle PA1 interrupt here.
}


char init_ports( void )
{
 /*! open Port A in default mode */
 open_PortA() ;
```

```
/*! set the interrupt vector for Port A bit one */
 SETVECTOR( PA1_IVECT, isr_PA1 ) ;

/*! configure Port A pin 1 for interrupt mode */
 if( GPIOERR_INVALIDPINS == setmodeInterrupt_PortA_8Pn
   ( PORTPIN_ONE,EDGE_FALLING, INTPRIORITY_HIGH ))

{
  return -1 ;
 }
}
```

setmodeInterrupt_PortA_4K()

Prototype

```
char setmodeInterrupt_PortA_4K( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_PortA_4K()` API is used to configure one or more pins of GPIO Port A on the Z8 Encore!® Z8F04 4K series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set Pin 1 of Port A to falling edge with a low priority call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_4K( PORTPIN_ONE,
      EDGE_FALLING, INTPRIORITY_LOW)
```

Similarly more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_4K( PORTPIN_FIVE|PORTPIN_SEVEN,
      EDGE_RISING, INTPRIORITY_LOW)
```

Note: *This API does not alter states of other pins.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt

void isr_PA1( void )
{
//! Handle PA1 interrupt here.
}

char init_ports( void )
{
 /*! open Port A in default mode */
 open_PortA() ;
```

```
/*! set the interrupt vector for Port A bit one */
 SETVECTOR( PA1_IVECT, isr_PA1 ) ;

/*! configure Port A pin 1 for interrupt mode */
 if( GPIOERR_INVALIDPINS == setmodeInterrupt_PortA_4K(
PORTPIN_ONE,EDGE_FALLING, INTPRIORITY_HIGH ))

{
  return -1 ;
 }
}
```

setmodeInterrupt_PortA_F08()

Prototype

```
char setmodeInterrupt_PortA_F08( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_PortA_F08()` API is used to configure one or more pins of the selected GPIO ports Z8 Encore!® Z8F08 series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set Pin 1 of Port A to falling edge with a low priority, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_F08(PORTPIN_ONE,
      EDGE_FALLING,INTPRIORITY_LOW)
```

Similarly, more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_F08(PORTPIN_FIVE |
      PORTPIN_SEVEN, EDGE_RISING, INTPRIORITY_LOW)
```

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In debug mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt

void isr_PA1( void )
{
      //! Handle PA1 interrupt here.
}

char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! set the interrupt vector for
            Port A bit one */
```

```
SETVECTOR( PA1_IVECT, isr_PA1 ) ;
      /*! configure Port A pin 1 for interrupt mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInterrupt_PortA_F08( PORTPIN_ONE,
            EDGE_FALLING, INTPRIORITY_HIGH ))

      {
      return -1 ;
      }
}
```

setmodeInterrupt_Portx_F64()

Prototype

```
char setmodeInterrupt_Portx_F64( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_Portx_F64()` API is used to configure one or more pins of the selected GPIO ports Z8 Encore!® Z8F64XX series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set Pin 1 of Port A to falling edge with a low priority, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_F64(PORTPIN_ONE,
      EDGE_FALLING, INTPRIORITY_LOW)
```

Similarly more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_F64 ( PORTPIN_FIVE |
      PORTPIN_SEVEN, EDGE_RISING, INTPRIORITY_LOW)
```

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt

void isr_PA1( void )
{
      //! Handle PA1 interrupt here.
}

#pragma interrupt
void isr_PD2( void )
{
      //! Handle PD2 interrupt here.
}
```

```
char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! set the interrupt vector for
            Port A bit one */
      SETVECTOR( PA1_IVECT, isr_PA1 ) ;

      /*! configure Port A pin 1 for interrupt mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInterrupt_PortA_F64( PORTPIN_ONE,
            EDGE_FALLING, INTPRIORITY_HIGH ))
      {
      return -1 ;
      }
      /*! open port D in default mode */
      open_PortD() ;

      /*! set the interrupt vector for
            port D bit two */
      SETVECTOR( PD2_IVECT, isr_PD2 ) ;

      /*! configure port D pin 2 for interrupt mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInterrupt_PortD_F64 PORTPIN_TWO,
            EDGE_RISING, INTPRIORITY_NOMINAL ))
      {
      return -1 ;
      }

}
```

setmodeInterrupt_Portx_F1680()

Prototype

```
char setmodeInterrupt_Portx_F1680( uchar pins,
      uchar edge, uchar priority )
```

Description

The `setmodeInterrupt_Portx_F1680()` API is used to configure one or more pins of the selected GPIO ports Z8 Encore!® Z8F1680 series of microcontroller to the interrupt mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. The API also provides an option to set the type of triggering to either falling or rising edge along with the priority for the interrupt.

For example, to set pin 1 of Port A to falling edge with a low priority, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
char setmodeInterrupt_PortA_F1680(PORTPIN_ONE,
      EDGE_FALLING, INTPRIORITY_LOW)
```

Similarly more than one pin is set to interrupt mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into interrupt mode, the API is used as given below:

```
setmodeInterrupt_PortA_F1680 ( PORTPIN_FIVE |
      PORTPIN_SEVEN, EDGE_RISING, INTPRIORITY_LOW)
```

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`edge`	The type of edge triggering for the interrupts. Valid values are: EDGE_FALLING EDGE_RISING
`priority`	The priority of the interrupt. The valid values are: INTPRIORITY_LOW INTPRIORITY_NOMINAL INTPRIORITY_HIGH

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the interrupt mode successfully.

Example

```
#include <ez8.h>
#pragma interrupt

void isr_PA1( void )
{
      //! Handle PA1 interrupt here.
}

#pragma interrupt
void isr_PD2( void )
{
      //! Handle PD2 interrupt here.
}
```

```
char init_ports( void )
{
      /*! open Port A in default mode */
      open_PortA() ;
      /*! set the interrupt vector for
            Port A bit one */
      SETVECTOR( PA1_IVECT, isr_PA1 ) ;
      /*! configure Port A pin 1 for interrupt mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInterrupt_PortA_F1680( PORTPIN_ONE,
            EDGE_FALLING, INTPRIORITY_HIGH ))
      {
      return -1 ;
      }
      /*! open port D in default mode */
      open_PortD() ;
      /*! set the interrupt vector for
            port D bit two */
      SETVECTOR( PD2_IVECT, isr_PD2 ) ;
      /*! configure port D pin 2 for interrupt mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeInterrupt_PortD_F1680 PORTPIN_TWO,
            EDGE_RISING, INTPRIORITY_NOMINAL ))
      {
      return -1 ;
      }
```

setmodeAltFuncSet1_Portx()

Prototype

```
char setmodeAltFuncSet1_Portx( uchar pins )
```

Description

The `setmodeAltFuncSet1_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the alternate function set 1 mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the alternate function set 1 mode for Port B Pin 7 (PB7) is set by the values in the registers PB_ADDR and PB_CTL[7]. To set pin 1 of Port B into alternate function set 1 mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeAltFuncSet1_PortB ( PORTPIN_ONE ) ;
```

Similarly more than one pin is set to alternate function set 1 mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port B into alternate function mode, the API is used as given below:

```
setmodeAltFuncSet1_PortB(PORTPIN_FIVE|PORTPIN_SEVEN) ;
```

Note: *Alternate function mode set 1 is supported only in Ports B and C of the Z8F04 XP Series.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the alternate function mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
      /*! open Port C in default mode */
      open_PortC() ;

      /*! configure Port C pin 3 for alternate
            function set-1 mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeAltFuncSet1_PortC( PORTPIN_THREE))

      {
      return -1 ;
      }

/*!
 * Port C pin 3 is now available for alternate function
 * set-1 mode. Namely, COUT for pin 3.
 */

      /*! open Port B in default (input) mode */
      open_PortB() ;
```

```
        /*! configure Port B pin 3 for alternate
              function set-1 mode */
        if( GPIOERR_INVALIDPINS ==
              setmodeAltFuncSet1_PortB( PORTPIN_THREE))
        {
        return -1 ;
        }
/*!
 * Port B pin 3 is now available for alternate function
 * set-1 mode. Namely, CLKIN for pin 3.
 */
}
```

setmodeAltFuncSet2_Portx()

Prototype

```
char setmodeAltFuncSet2_Portx( uchar pins )
```

Description

The `setmodeAltFuncSet2_Portx()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the alternate function set 2 mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the alternate function set 2 mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into alternate function set 2 mode call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeAltFuncSet2_PortA (PORTPIN_ONE);
```

Similarly more than one pin is set to alternate function set 1 mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into alternate function mode, the API is used as given below:

```
setmodeAltFuncSet2_PortA(PORTPIN_FIVE|PORTPIN_SEVEN) ;
```

Note: *This API does not alter states of other pins.*

Alternate function mode set 2 is supported only in Port A of Z8F04 8-pin devices and Ports B and C of Z8F04 XP and 4K Series.

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In debug mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the alternate function mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )
{
/*! open Port C in default mode */
 open_PortC() ;

/*! configure Port C pins 0 and 1 for alternate
function set-2 mode */
 if( GPIOERR_INVALIDPINS == setmodeAltFuncSet2_PortC(
PORTPIN_ZERO|PORTPIN_ONE))

{
  return -1 ;
}

/*!
  * Port C pins 0 and 1 are now available for alternate
function
  * set-2 mode. Namely, ANA4/CINP/LED and ANA5/CINN/
LED. However, the CINP/LED,
  * and CINN/LED alternate functions are available on
XP series only.
  */
```

```
/*! open Port B in default (input) mode */
 open_PortB() ;

/*! configure Port B pins for alternate function set-2
mode */
 if( GPIOERR_INVALIDPINS == setmodeAltFuncSet2_PortB(
PORTPIN_ALL))
{
  return -1 ;
}
/*!
  * Port B pins are now available for alternate
function
  * set-2 mode. Namely, ANA0, ANA1, ANA2, ANA3, etc.
  */

}
```

setmodeAltFuncSet3_PortA()

Prototype

```
char setmodeAltFuncSet3_PortA( uchar pins )
```

Description

The `setmodeAltFuncSet3_PortA()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the alternate function set 3 mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the alternate function set 3 mode for Port A Pin 7 (PA7) is set by the values contained in registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into alternate function set 3 mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeAltFuncSet3_PortA (PORTPIN_ONE);
```

Similarly more than one pin is set to alternate function set 1 mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into alternate function mode, the API is used as given below:

```
setmodeAltFuncSet3_PortA(PORTPIN_FIVE|PORTPIN_SEVEN);
```

Note: *This API does not alter states of other pins.*

Alternate function mode set 3 is supported only in Port A of the Z8F04 8-pin devices.

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In debug mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the alternate function mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )

{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pin 3 for alternate
function set-3 mode */
      if( GPIOERR_INVALIDPINS ==
setmodeAltFuncSet3_PortA( PORTPIN_THREE))

      {
            return -1 ;
      }

      /*!
       * Port A pin 3 is now available for alternate
function
       * set-3 mode. Namely, T1IN for pin 3.
       */

}
```

setmodeAltFuncSet4_PortA()

Prototype

```
char setmodeAltFuncSet4_PortA( uchar pins )
```

Description

The `setmodeAltFuncSet4_PortA()` API is used to configure one or more pins of the selected GPIO port of Z8 Encore! microcontroller to the alternate function set 4 mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the alternate function set 4 mode for Port A Pin 7 (PA7) is set by the values in the registers PA_ADDR and PA_CTL[7]. To set Pin 1 of Port A into alternate function set 4 mode, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeAltFuncSet4_PortA (PORTPIN_ONE);
```

Similarly more than one pin is set to alternate function set 1 mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7 of Port A into alternate function mode, the API is used as given below:

```
setmodeAltFuncSet4_PortA(PORTPIN_FIVE|PORTPIN_SEVEN) ;
```

Note: *This API does not alter states of other pins.*

Alternate function mode set 4 is supported only in Port A of the Z8F04 8-pin devices.

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the alternate function mode successfully.

Example

```
#include <ez8.h>

char init_ports( void )

{
      /*! open Port A in default mode */
      open_PortA() ;

      /*! configure Port A pin 3 for alternate
function set-4 mode */
      if( GPIOERR_INVALIDPINS ==
setmodeAltFuncSet4_PortA( PORTPIN_THREE))

      {
            return -1 ;
      }

      /*!
       * Port A pin 3 is now available for alternate
function
       * set-4 mode.Namely,Analog Functions for pin 3.
       */
}
```

setmodeLEDDrive_PortC()

Prototype

```
char setmodeLEDDrive_PortC (uchar pins, byte drivelev-
els)
```

Description

The `setmodeLEDDrive_PortC()` API is used to configure one or more pins of the GPIO Port C of Z8 Encore!® microcontroller to LED drive mode. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. This API also gives an option to set the current drive levels for each pin configured. For example, to set pin 7 into LED drive mode with 3 milliamperes, call this API by specifying the bit corresponding to the pin by using the definitions in `gpio.h`, as given below:

```
setmodeLEDDrive_PortC (PORTPIN_SEVEN, DRIVELEVEL_3MA);
```

Similarly, more than one pin is set to LED drive mode by ORing the pins in the API call. For example, to set Pin 5 and Pin 7, the API is used as given below:

```
setmodeLEDDrive_PortC (PORTPIN_FIVE|PORTPIN_SEVEN,
DRIVELEVEL_3MA);
```

Note: *This API does not alter states of other pins. LED drive mode is supported only in Port C of the Z8F04 XP Series.*

Argument(s)

`pins`	The bitwise ORed value indicating the pins of a port as defined in the `gpio.h` file.
`drivelevels`	The current drive level in milliamperes for each pin being configured. The valid values are: DRIVELEVEL_3MA DRIVELEVEL_7MA DRIVELEVEL_13MA DRIVELEVEL_20MA

Return Value(s)

`GPIOERR_INVALIDPINS`	In DEBUG mode on some of the ports, this value indicates that one or more pins specified are not supported for that target.
`GPIOERR_SUCCESS`	Indicates that the port was configured to the LED drive mode successfully.

Example

```
#include <ez8.h>
char init_ports( void )
{
      /*! open Port C in default (input) mode */
      open_PortC() ;
      /*! configure Port C pins for LED Drive mode */
      if( GPIOERR_INVALIDPINS ==
            setmodeLEDDrive_PortC( PORTPIN_ALL,
            DRIVELEVEL_13MA ) )

      {
            return -1 ;
      }
}
void write_LEDs( void )
{
      /*! Write to Port C pins */
      PCOUT = data1 ;
}
```

close_Portx()

Prototype

```
void close_Portx(void);
```

Description

The `close_Portx()` API resets all the selected Port registers and configures the port as a standard digital input pin. However, this API does not reset the GPIO interrupt settings, if already configured.

Argument(s)

None.

Return Value(s)

None.

ZSL GPIO Macros

The ZSL GPIO macro definitions are listed in Table 11.

Table 11. ZSL GPIO Macro Definitions

#define	Description
`#define RESETBIT( x, y ) ((x)&=(BYTE)(0xFF^(y)))`	Resets all those bits in *x* as specified by the bit pattern in *y*.
`#define SETBIT( x, y ) ( (x) |= ((BYTE)(y)) )`	Sets all those bits in *x* as specified by the bit pattern in *y*.
`#define SETBITPA( x ) SETBIT( PAOUT, x )`	Sets all those Port A pins as specified by the bit pattern in *x*.
`#define RESETBITPA( x ) RESETBIT( PAOUT, x )`	Resets all those Port A pins as specified by the bit pattern in *x*.
`#define SETBITPB( x ) SETBIT( PBOUT, x )`	Sets all those Port B pins as specified by the bit pattern in *x*.

Table 11. ZSL GPIO Macro Definitions (Continued)

#define	Description
`#define RESETBITPB( x ) RESETBIT( PBOUT, x )`	Resets all those Port B pins as specified by the bit pattern in *x*.
`#define SETBITPC( x ) SETBIT( PCOUT, x )`	Sets all those Port C pins as specified by the bit pattern in *x*.
`#define RESETBITPC( x ) RESETBIT( PCOUT, x )`	Resets all those Port C pins as specified by the bit pattern in *x*.
`#define SETBITPD( x ) SETBIT( PDOUT, x )`	Sets all those Port D pins as specified by the bit pattern in *x*.
`#define RESETBITPD( x ) RESETBIT( PDOUT, x )`	Resets all those Port D pins as specified by the bit pattern in *x*.
`#define SETBITPE( x ) SETBIT( PEOUT, x )`	Sets all those Port E pins as specified by the bit pattern in *x*.
`#define RESETBITPE( x ) RESETBIT( PEOUT, x )`	Resets all those Port E pins as specified by the bit pattern in *x*.
`#define SETBITPF( x ) SETBIT( PFOUT, x )`	Sets all those Port F pins as specified by the bit pattern in *x*.
`#define RESETBITPF( x ) RESETBIT( PFOUT, x )`	Resets all those Port F pins as specified by the bit pattern in *x*.
`#define SETBITPG( x ) SETBIT( PGOUT, x )`	Sets all those Port G pins as specified by the bit pattern in *x*.
`#define RESETBITPG( x ) RESETBIT( PGOUT, x )`	Resets all those Port G pins as specified by the bit pattern in *x*.
`#define SETBITPH( x ) SETBIT( PHOUT, x )`	Sets all those Port H pins as specified by the bit pattern in *x*.
`#define RESETBITPH( x ) RESETBIT( PHOUT, x )`	Resets all those Port H pins as specified by the bit pattern in *x*.

ZSL UART API Description

This chapter provides detailed descriptions of Zilog Standard Library (ZSL) UART APIs.

To use ZSL UART APIs, the header file `ez8.h` must be included in the application program. The `ez8.h` file is placed in the `\include\zilog` folder under the root installation directory as displayed in Figure 2 on page 7. The application must also include the `uart.h` file.

UART Initialization in the Startup Routine

ZSL is integrated with ZDS II, allowing you to select or deselect Z8 Encore!® MCU UARTs (see Startup Routine on page 10). When initializing the UART devices in the Startup routine, the following points must be considered:

1. When a UART device is selected in ZDS II interface using **Project→Settings→ZSL**, the ZDS II generates a compiler pre-define–`_ZSL_DEVICE_UARTx`—for `_open_periphdevice()` routine. The `_open_periphdevice()` routine uses `open_UARTx()` function to initialize the UARTx device with default values when the `_ZSL_DEVICE_UARTx` symbol is supplied. Therefore, the user-application program uses the APIs directly to drive any UART device without making a specific call to the `init_UARTx()` routine.

2. To use the UART0 device, GPIO Port A is required to be initialized; UART1 device requires Port D to be initialized. These GPIO ports must be initialized before initializing the UART. ZDS II defines the macro, `_ZSL_DEVICE_PORTD` when UART0 is selected, and `_ZSL_DEVICE_PORTC`, when UART1 is selected. These ports are initialized to mode 2 in the `zsldevinit.asm` file.

3. All standard RTL I/O functions, `putch()`, `getch()`, and `kbhit()` are mapped to the default UART device—implying that the standard RTL I/O functions invoke the default UART device APIs. In ZSL dis-

tribution, UART0 is configured as the default device. To use UART1 as the default device, in `uartcontrol.h` file, change the value of macro `DEFAULT_UART` from `UART0` to `UART1` and rebuild the library.

4. The UART driver operates in two modes—SYNCHRONOUS mode and ASYNCHRONOUS mode. In SYNCHRONOUS mode the data is transmitted and received by polling on the UARTx transmit and receive registers. Thus in polling mode the `read_UARTx` and `write_UARTx` API are blocking in nature.

   Asynchronous communication is interrupt driven. In ASYNCHRONOUS mode the data transfer (both transmit and receive) happens in the interrupt service routines of the UART*x* devices, at the same time when the application is running. So the `read_UARTx` and `write_UARTx` APIs are non-blocking in nature, they return immediately to allow the application to run. However, some Z8 Encore!® Series such as EZ8F64XX, data transfer (transmission only) also happens through Direct Memory Access (DMA). The EZ8F64XX series MCUs have dedicated DMA1 for data transmission which is used by UART*x* devices. ZSL provides a compile time control to enable or disable DMA for data transmission. For more information, see write_UARTx() API on page 88.

5. Modify the default values to suit user-application specifications by making appropriate changes in the device-specific source code files. All the compile time configurations are listed in `uartcontrol.h` file. Table 12 on page 74 summarizes the compile time options available. If any of these parameters are modified, the library must be rebuilt.

Table 12. ZSL UART API Compile Time Options

Parameter	Description	Default value
`UARTx_MODE`	Selects the UART*x* mode of operation, either asynchronous (interrupt) or synchronous (poll). For more details see read_UARTx() and write_UARTx() APIs. Valid values are: `MODE_INTERRUPT`—Interrupt mode `MODE_POLL`—Polling mode	`MODE_POLL`
`DMA1_CTL`	Enables DMA1 to be used with the specified UART during data transmission by `write_UARTx()`.This option is only used in interrupt mode. Valid values are: `DMA_UART0`—Use UART0 with DMA1 `DMA_UART1`—Use UART1 with DMA1 `DMA_DISABLED`—Do not use DMA **Note:** *DMA is available only on EZ8F64XX series MCUs.*	`DMA_DISABLED`
`UARTx_BAUDRATE`	Baud rate to be used for UART*x* communication. The valid values are listed in the `uart.h` file.	`BAUD_38600`
`UARTx_STOPBITS`	Number of stop bits to be used for UART*x* communication. Valid values are listed in the `uart.h` file.	1 stop bit
`UARTx_PARITY`	Parity to be used in UARTx transmission. The valid values are listed in the `uart.h` file.	`PAR_NOPARITY`
`UARTx_ERRORCHECKING`	Selects whether `read_UARTx()` must check for any error in the incoming data.	Error checking disabled

Table 12. ZSL UART API Compile Time Options (Continued)

Parameter	Description	Default value
`UARTx_HWFLOW_CTL`	Selects whether hardware flow control is enabled for the transmitter.	HW flow control disabled
`UARTx_RX_INT_PRIORITY` `UARTx_TX_INT_PRIORITY`	Selects the interrupt priority for UART*x* interrupts. Valid values are defined in `defines.h`.	`INTPRIORITY_NOMINAL`

Generic UART APIs

Table 13 lists the generic UART APIs with hyperlinks to their description.

Table 13. Generic UART APIs

APIs	Descriptions
getch()	Reads data byte from the UART device
putch()	Writes data byte into the UART transmit buffer
kbhit()	Detects keystrokes on the UART device

getch()

Prototype

```
int getch(void);
```

Description

The `getch()` API reads a data byte from the default UART device. If there is no data in the UART device, the API blocks till the data becomes available.

The API calls the underlying `read_UARTx()` API. If there is any error in the received data byte, an error code is set in the `g_recverr0` global variable. The application determines the error by updating the `g_recverr0` global variable with a known value before calling the API, and then reading the `g_recverr0` global variable again to determine whether that value changed. For a list of possible errors, see read_UARTx() API on page 92.

Argument(s)

None.

Return Value(s)

Returns the character received.

Example

```
#include <ez8.h>

void get_input(void)
{
        int ch;
        printf("Type a character\n");
        ch = getch();
        if( ch == '\n');
              printf("A new line is entered\n");
}
```

putch()

Prototype

```
int putch(int ich);
```

Description

The `putch()` API writes a data byte into the default UART transmit buffer. If the data byte written is a newline character, then the `putch()` API writes an additional carriage return character into the UART transmit buffer.

Argument(s)

`ich`	Character to be written in the transmit buffer

Return Value(s)

`ich`	Indicates success.
`EOF`	Indicates failure.

Example

```
#include <ez8.h>

void get_input(void)
{
        int ch;
        printf("Type a character\n");
        ch = getch();
        if( ch == '\n');
              printf("A new line is entered\n");
        else
        {
              printf("You entered:");
              putch(ch);
        }
}
```

kbhit()

Prototype

```
uchar kbhit(void);
```

Description

The `kbhit()` API checks for any keystrokes on the default UART device. If a keystroke is detected the `kbhit()` function returns 1, otherwise it returns 0. The API returns immediately without blocking when the UART is configured to work either in `POLL` mode or in the interrupt mode.

Note: *The API does not read the data but only returns the status. The application then calls* `getch()` *to get the keystroke.*

Argument(s)

None.

Return Value(s)

1	Indicates that a key was hit.
0	Indicates that no keystrokes were detected.

Example

```
#include <ez8.h>
void get_input(void)
{
        printf("Type any character to display menu\n");
        while(!kbhit());
        display_menu();
}
```

UARTx APIs

The UARTx APIs listed in this section are used for either UART0 or UART1 devices on the Z8 Encore!® microcontrollers. The x in the UARTx signifies 0 or 1 for the UART0 or UART1 device, respectively.

Table 14 provides the hyperlinks to the description of UARTx APIs.

Table 14. UART*x* APIs

APIs	Descriptions
open_UARTx()	Initializes the UARTx device
control_UARTx()	Configures the UARTx device
setbaud_UARTx()	Sets baud rate for the UARTx device
setparity_UARTx()	Sets parity bit option for the UARTx device
setstopbits_UARTx()	Sets stop bits for the UARTx device
write_UARTx()	Writes data bytes to the UARTx device
get_txstatus_UARTx()	Gets the status of an asynchronous transmission
read_UARTx()	Reads data bytes from the UARTx device
get_rxstatus_UARTx()	Gets the status of an asynchronous receive
close_UARTx()	Closes the UARTx device

The detailed descriptions of each of the UART*x* APIs begin on the next page.

open_UARTx()

Prototype

```
void open_UARTx();
```

Description

The `open_UARTx()` API opens the UART*x* device by initializing the UART*x* Control registers with default values. This API configures the appropriate port registers for alternate functions.

The following default values are set.

- UARTx mode—interrupt mode
- Baud rate—38400
- Data bits—8
- Stop bits—1
- Parity—disabled
- Hardware flow control—disabled

Argument(s)

None.

Return Value(s)

None.

Example

```
#include <ez8.h>
void init_devices(void)
{
      /* initialize uart0 with default values */
      open_UART0();
      /* Print welcome message */
      printf("Welcome to Zilog\n");
      close_UART0(); /* close the uart */
}
```

control_UARTx()

Prototype

```
uchar control_UARTx(UART * pUART);
```

Description

The `control_UARTx()` API is used to configure the UART*x* device with the values specified by the pointer to the `UART` structure passed as the parameter. The values in the structure are used to write into the appropriate UART*x* device Control registers.

If the debug version of ZSL is used, the API checks the validity of the parameters passed. Otherwise, the API configures the UART*x* with the value passed in the `pUART` parameter. For more information, see ZSL Debug and Release Version on page 9.

Argument(s)

`*pUART`	Pointer to a structure of type UART as defined in `uart.h` file		
	baudrate	→	57600 (valid values = 9600, 19200, 38400, 57600, 115200)
	stop bits	→	2 (valid values = 1, 2)
	parity	→	disable (valid values = PAR_NOPARITY, PAR_ODPARITY, PAR_EVPARITY)

Return Value(s)

`UART_ERROR_NONE`	No error
`UART_ERR_INVBAUDRATE`	Error due to invalid baud rate value passed.
`UART_ERR_INVSTOPBITS`	Error due to invalid stop bits value passed.
`UART_ERR_INVPARITY`	Error due to invalid parity value passed.

Example 1

```
#include <ez8.h>
void init_devices(void)
{
      UART uart ;
      char stat = UART_ERR_NONE ;

      /* configure UART0 with 9600 baud, 1 stop bits
            and no parity */
      uart.baudRate = BAUD_9600 ;
      uart.stopBits = STOPBITS_1 ;
      uart.parity = PAR_NOPARITY ;
      /*! Configure the UART */
      stat = control_UART0( &uart ) ;
      if( UART_ERR_NONE != stat )
```

```
        {
              return ntestcase ;
        }
        close_UART0();
  }
```

Example 2

```
  #include <ez8.h>
  void init_devices(void)
  {
        UART uart ;
        char stat = UART_ERR_NONE ;
        /* configure UART0 with 1200 (invalid) baud,
        1 stop bit and no parity */
        uart.baudRate = 1200 ;
        uart.stopBits = STOPBITS_1 ;
        uart.parity = PAR_NOPARITY ;
        /*! Configure the UART */
        stat = control_UART0( &uart ) ;
        if( UART_ERR_NONE != stat )
        {
              if( stat == UART_ERR_INVBAUDRATE )
                    global_err = TRUE;
        }
        close_UART0();
  }
```

setbaud_UARTx()

Prototype

```
uchar setbaud_UARTx(int32 baud);
```

Description

The `setbaud_UARTx()` API configures the baud rate for the UART*x* device with the specified value.

If the debug version of ZSL is used, the API checks the validity of the parameters passed. Otherwise, the API configures the UART*x* with the value passed in the `baud` parameter. For more information, see ZSL Debug and Release Version on page 9.

Argument(s)

`baud`	Specifies the new baudrate to be set. This value, along with the target clock frequency value set in the `zsldevinit.asm` file, is used to calculate the value for Baudrate Generator registers. baud rate → valid values = 9600, 19200, 38400, 57600, 115200

Note: *Not all devices in the Z8 Encore!® family work with all the abov baud rates. Check the appropriate device documentation for working values.*

Return Value(s)

`UART_ERROR_NONE`	No error
`UART_ERR_INVBAUDRATE`	Error due to invalid baud rate value passed

Example

```
#include <ez8.h>
void init_devices(void)
{
        UART uart ;
```

```
        char stat = UART_ERR_NONE ;
        /* configure UART0 with 9600 baud, 1 stop bits
              and no parity */
        uart.baudRate = BAUD_9600 ;
        uart.stopBits = STOPBITS_1 ;
        uart.parity = PAR_NOPARITY ;
        /*! Configure the UART */
        stat = control_UART0( &uart ) ;
        if( UART_ERR_NONE != stat )
        {
              return;
        }
        /* Change the baud rate to 115200 */
        stat = setbaud_UART0( BAUD_115200 ) ;
        if( UART_ERR_NONE != stat )
        {
              return;
        }
        close_UART0();
}
```

setparity_UARTx()

Prototype

```
uchar setparity_UARTx(uchar parity);
```

Description

The `setparity_UARTx()` API configures the parity for the UART*x* device.

If the debug version of ZSL is used, the API checks the validity of the parameters passed. Otherwise, the API configures the UART*x* with the value passed in the `parity` parameter. For more information, see ZSL Debug and Release Version on page 9.

Argument(s)

`parity`	Specifies the new parity value	
	`parity` →	disable (valid values = PAR_NOPARITY, PAR_ODPARITY, PAR_EVPARITY)

Return Value(s)

`UART_ERROR_NONE`	No error
`UART_ERR_INVPARITY`	Error due to invalid parity values

setstopbits_UARTx()

Prototype

```
uchar setstopbits_UARTx(uchar stopbits);
```

Description

The `setstopbits_UARTx()` API sets the stop bits for the UART*x* device.

If the debug version of ZSL is used, the API checks the validity of the parameters passed. Otherwise, the API configures the UART*x* with the value passed in the `stopbits` parameter. For more information, see ZSL Debug and Release Version on page 9.

Argument(s)

`stopbits`	Number of valid stop bits set `stopbits` → 2 (valid values = 1, 2)

Return Value(s)

`UART_ERROR_NONE`	No error
`UART_ERR_INVSTOPBITS`	Error due to invalid stop bits

write_UARTx()

Prototype

```
uchar write_UARTx( char *pData, uint16 nbytes ) ;
```

Description

The `write_UARTx()` API writes data bytes into the UARTx device. The API accepts a pointer to the buffer containing data to be transmitted and the number of bytes to be transmitted. The API behaves differently depending on the mode in which the UART*x* device is configured as follows:

Writing in Poll mode—In POLL mode the data transmission is synchronous in nature. In POLL mode the `write_UARTx()` API transmits the data bytes by polling on the UART*x* transmit register. The API does not return until all the bytes are transmitted.

Writing in Interrupt mode—The data transmission in INTERRUPT mode is asynchronous in nature. In INTERRUPT mode `write_UARTx()` API uses the UARTx transmit interrupt to transmit the data bytes. The `write_UARTx()` API simply enables the transmit interrupt and returns immediately. The data transfer then takes place in the interrupt service routine of the UARTx device.

The caller of the API determines the status of the write operation by using `get_txstatus_UARTx()`, which returns either `UART_IO_PENDING` or `UART_IO_COMPLETE`, depending on the transmission status.

If the API is compiled by enabling Direct Memory Access (DMA) for data transmission then the `write_UARTx()` API uses DMA1 for data transfer. Now, the `write_UARTx()` API sets up the DMA registers for transmission of data and returns immediately. The completion of data transfer is indicated by a DMA interrupt to the caller in the form of `UART_IO_COMPLETE` when a call to `get_txstatus_UARTx()` is made.

Argument(s)

`*pData`	Pointer to a buffer containing the data to transmit
`nbytes`	Number of bytes to transmit

Return Value(s)

`UART_ERR_NONE`	The data is transmitted successfully.
`UART_ERR_BUSY`	Transmission is already in progress. The UARTx device is still servicing a previous `write_UARTx()` call at the time when this `write_UARTx()` call is made.

Example

```
#include <ez8.h>
int compute_sum(int, int);
char msg[] = "Welcome to the world of Encore!
microcontrollers from ZiLOG"
void init(int val1, val2)
{
      UART uart ;
      char stat = UART_ERR_NONE ;
      /* configure UART0 with 9600 baud, 1 stop bits
            and no parity */
      uart.baudRate = BAUD_9600 ;
      uart.stopBits = STOPBITS_1 ;
      uart.parity = PAR_NOPARITY ;
      /*! Configure the UART */
      stat = control_UART0( &uart ) ;
      if( UART_ERR_NONE != stat )
      {
            return;
      }

      if( write_UART0( msg, strlen(msg) )  ==
            UART_ERROR_NONE )
      {
            if( compute_sum(val1, val2) > 10 )
```

```
                        /* Update global variable */
                        global_threshold = 10;
                  /* Now check whether the transmission is
                        complete */
                  while(UART_IO_PENDING ==
                        get_txstatus_UART0() ) ;
            }

            close_UART0();
      }
```

get_txstatus_UARTx()

Prototype

```
uchar get_txstatus_UARTx( void )
```

Description

The `get_txstatus_UARTx()` API is used to get the status of asynchronous data transmission in the UARTx device. This API must be called by the application to know the status of the data transmission during INTERRUPT mode transfers. During INTERRUPT mode data transmission, `write_UARTx()` API returns immediately, allowing the calling application to perform other tasks while the data transmission is in progress. The calling application then knows the status of the transmission by calling the `get_txstatus_UARTx()` API.

Argument(s)

None.

Return Value(s)

`UART_IO_PENDING`	Indicates that data transmission in the UARTx device is still in progress.
`UART_IO_COMPLETE`	Indicates that data transmission in the UARTx device is complete.

Example

For more information, see the example for write_UARTx() on page 88.

read_UARTx()

Prototype

```
uchar read_UARTx(char *pData, uint16 *nbytes);
```

Description

The `read_UARTx()` API reads data bytes from the UART*x* device. This API accepts a pointer to a buffer for storing data bytes received and the number of bytes to be read. The API behaves differently depending on the mode in which the UART*x* device is configured, as follows:

Reading in Poll mode—In poll mode the data reception is synchronous in nature. In the poll mode the `read_UARTx()` API receives the data bytes by polling the UART*x* receive register. The API does not return until all the bytes are received. If the API is compiled using the `UARTx_ERRORHANDLING` macro, any error in the communication is reported as a return value. For more information on return values, see Return Value(s) on page 93.

Reading in Interrupt mode—The data reception in INTERRUPT mode is asynchronous in nature. In the INTERRUPT mode, `read_UARTx()` API uses the UART*x* receive interrupt to read data bytes. The `read_UARTx()` API enables the receive interrupt of the UART*x* device and returns immediately. The data reading then happens in the interrupt service routine of the UARTx device.

The caller of the API determines the status of the read operation by using the API `get_rxstatus_UARTx()`, which returns `ART_IO_COMPLETE`, indicating the completion of the read operation, or `UART_IO_PENDING`, indicating that reading is still in progress.

If the API is compiled using `UARTx_ERRORHANDLING` macro, any error in the received data byte is reported when `get_rxstatus_UARTx()` is called.

Argument(s)

`*pData`	Pointer to a buffer to receive data.
`*nbytes`	Pointer to an integer which indicates the number of bytes to read. When the API returns, this variable contains the actual number of bytes read. However, in INTERRUPT mode this value is valid only if `get_rxstatus_UARTx()` returns `UART_IO_COMPLETE` or any error return value listed on page 95.

Return Value(s)

`UART_ERROR_NONE`	Indicates that the read was successful.
`UART_ERR_FRAMINGERR`	Indicates that a framing error occurred in the byte received.
`UART_ERR_PARITYERR`	Indicates that a parity error occurred in the byte received.
`UART_ERR_OVERRUNERR`	Indicates an overrun error occurred in the byte received.
`UART_ERR_BREAKINDICATION`	Indicates that a break condition is set.

Example

```
#include <ez8.h>
int compute_sum(int, int);

void read_data(int val1, val2)
{
      UART uart ;
      char stat = UART_ERR_NONE ;

      /* configure UART0 with 9600 baud, 1 stop bits
            and no parity */
      uart.baudRate = BAUD_9600 ;
      uart.stopBits = STOPBITS_1 ;
```

```
        uart.parity = PAR_NOPARITY ;

        /*! Configure the UART */
        stat = control_UART0( &uart ) ;
        if( UART_ERR_NONE != stat )
        {
              return;
        }

        stat = read_UART0( readdata, &len ) ;
        if( UART_ERR_NONE != stat )
        {
              close_UART0();
              return;
        }

        /*! block here while receiver is busy */
        while(UART_IO_PENDING == get_rxstatus_UART0());
        close_UART0();
}
```

get_rxstatus_UARTx()

Prototype

```
uchar get_rxstatus_UARTx( void )
```

Description

The `get_rxstatus_UARTx()` API is used to get the status of the asynchronous read operation in the UART*x* device. This API must be called by the application to know the status of the read operation during INTERRUPT mode. During INTERRUPT mode data transmission the `read_UARTx()` API returns immediately, allowing the calling application to perform other tasks when data reception is in progress. The calling application then knows the status of the transmission by calling the `get_rxstatus_UARTx()` API.

Argument(s)

None.

Return Value(s)

`UART_IO_PENDING`	Indicates that data transmission in the UARTx device is still in progress.
`UART_IO_COMPLETE`	Indicates that data transmission in the UARTx device is complete.
`UART_ERR_FRAMINGERR`	Indicates that a framing error occurred in the byte received.
`UART_ERR_PARITYERR`	Indicates that a parity error occurred in the byte received.
`UART_ERR_OVERRUNERR`	Indicates an overrun error occurred in the byte received.
`UART_ERR_BREAKINDICATION`	Indicates that a break condition is set.

close_UARTx()

Prototype

```
void close_UARTx(void);
```

Description

The `close_UARTx()` API is used to close the UARTx device. Calling this API disables the interrupts related to the default UART device, and clears all the control registers to render the UART device non-functional after the call. The user-application uses the UART again only after making a call to the `open_UARTx()` API.

Argument(s)

None.

Return Value(s)

None.

Index

E

F

G

H

I

K

L

M

O

P

R

S

T

U

W

Z

Customer Support

For answers to technical questions about the product, documentation, or any other issues with Zilog's offerings, please visit Zilog's Knowledge Base at http://www.zilog.com/kb.

For any comments, detail technical questions, or reporting problems, please visit Zilog's Technical Support at http://support.zilog.com.