zilog

An IXYS Company

eZ80Acclaim*Plus!*™ Family of Microprocessors

Zilog Standard Library API

Reference Manual

RM003708-0910

www.zilog.com

Zilog Standard Library API
eZ80® Family of Microprocessors

zilog

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS 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

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

RM003708-0910

Revision History

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

Date	Revision Level	Description	Page No.
September 2010	08	Updated copyright date and logos.	All
December 2007	07	Updated the latest disclaimer page and implemented Style Guide. Also replaced eZ80Acclaim! to eZ80Acclaim*Plus!* and eZ80F91 MCU to eZ80F91.	All
May 2006	06	Added examples in setbaud_UARTx. Fixed broken cross-reference.	33
Mar 2006	05	Added registered trademark symbol ® to eZ80 and eZ80Acclaim!.	All
Oct 2004	04	Modified alternate function content and explanatory notes in open_UARTx. Added explanatory note in control_UARTx.	27 and 30
Oct 2004	03	Removed references to Fatbrain.com.	All

Table of Contents

Introduction

This Reference Manual describes the Zilog Standard Library (ZSL) and its associated application programming interface (API). ZSL is available as part of the Zilog Developer Studio Integrated Developer Environment (ZDS II-IDE) version 4.8.0 and later, for Zilog's eZ80Acclaim*Plus!*™ product line of microcontrollers and microprocessors.

ZSL is a set of library files that provides an interface between your application and the on-chip peripherals. Currently, Zilog's eZ80Acclaim!/ eZ80Acclaim*Plus!* product line includes the eZ80190 and eZ80L92 microprocessors, and the eZ80F91, eZ80F92, and eZ80F93 microcontrollers.

About This Manual

Zilog recommends that you read and understand everything in this manual before using the product. We have designed this manual to be used as a reference guide for the ZSL APIs.

Intended Audience

This document provides reference information for Zilog customers towards understanding the ZSL implementation. This reference manual guides you to interface the application with the on-chip peripherals of the eZ80Acclaim!/eZ80Acclaim*Plus!* family of devices.

Manual Organization

This reference manual is divided into following three chapters.

Zilog Standard Library Overview

This chapter provides an overview of Zilog Standard Library, directory structure, and its release and debug versions.

ZSL GPIO API Description

This chapter provides a detailed description of the GPIO APIs and how to interface your application with eZ80Acclaim!/eZ80Acclaim*Plus!* MPU/MCU GPIO peripheral(s).

ZSL UART API Description

This chapter provides a detailed description of the UART APIs and how to interface your application with eZ80Acclaim!/eZ80Acclaim*Plus!* MPU/MCU UART peripheral(s).

Related Documents

Table 1 lists the related documents for ZSL.

Table 1. Related Documentation

Zilog Developer Studio-eZ80Acclaim User Manual	UM0144
ZPAK II Product User Guide	PUG0015
eZ80Acclaim Development Kits Quick Start Guide	QS0020
eZ80® CPU User Manual	UM0077

Definition

Table 2 lists the definitions of some common terms used in this document.

Table 2. Definition of Common Terms

Terms	Definitions
ANSI C Standard	A standard for C language proposed by the American National Standards Institute.
Language tools	A suite of software applications that form a subset of the XTools development tools bundled in the ZDS II IDE. The language tools are oriented that allows you to create code and place it into the target platform. The language tools comprise of a compiler, an assembler, a linker, and a librarian.
IDE	Integrated Development Environment—a general term for software that provides a number of development tools in a unified package. Specifically, the Zilog Xtools development tools are bundled into the ZDS II IDE, which includes a GUI. The term IDE is often used internally to refer to the higher levels of the overall software package that accepts user commands and displays results of actions.

Abbreviations/Acronyms

Table 3 lists the abbreviations used in this document.

Table 3. Abbreviations/Acronyms

Abbreviations	Expansion
ALT2	Alternate Register 2
API	Application Program Interface
DDR	Data Direction Register
DR	Data Register
GPIO	General Purpose Input Output
ISR	Interrupt Service Routine
RTL	ANSI C Run-Time Library
SB	Send Break
SPR	Scratch Pad Register (of UART device)
UART	Universal Asynchronous Receiver Transmitter
ZDS II	Zilog Developer Studio II
ZSL	Zilog Standard Library

Manual Conventions

The following convention is adopted to provide clarity and ease of use:

Courier Typeface

Code lines and fragments, functions, and various executable items are distinguished from general text by appearing in the `Courier` typeface. This convention is used within tables.

For example, `zsldevinit.asm`.

Safeguards

When you use the Zilog Standard Library along with the ZDS II-IDE and any of the Zilog's development platforms, follow the precautions listed below to avoid permanent damage to the development platform.

Note: *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 connector P3 on the development platform.

2. Power-down precautions:

   when powering down, follow the sequence below:
   (a) Quit the monitor program.
   (b) Remove power from the development platform.

Zilog Standard Library Overview

This chapter provides an overview of the Zilog Standard Library (ZSL), architecture and its debug and release versions. It also provides detailed information on how to build libraries using the batch script files. This chapter also explains the start-up routine and a summary of the ZSL APIs.

The ZSL is a collection of various libraries, each comprised of device driver APIs to program various on-chip peripherals to the eZ80190 and eZ80L92 microprocessors and the eZ80F91/F92/F93 microcontrollers. This library is a collection of various device drivers that allows you to communicate with on-chip peripherals or devices without having a knowledge of register details and how to program them.

Zilog's comprehensive set of standard library APIs is easy to use. You can use the source code files provided with the ZSL release to modify these libraries to suit your specific requirements.

Zilog Standard Library Architecture

Figure 1 displays a block diagram of the ZSL architecture.

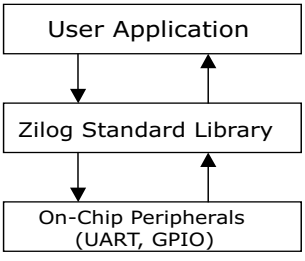

Figure 1. The ZSL Architecture

Zilog Standard Library Directory Structure

Figure 2 displays the directory structure of the ZSL. Table 4 on page 3 describes the contents of the ZSL sub-directory.

Figure 2. ZSL Directory Structure

Note: *In* Figure 2, `\{zdsroot}` *specifies the root directory of the ZDS II installation, for example,* `ZDS II_eZ80Acclaim!/eZ80AcclaimPlus!_4.8.0 and later`.

Table 4. ZSL Directory Structure Description*

Directory Path	Description
\Include	Contains subfolders that contain the include files
\Include\std	Header files relevant to the C Run Time Library (RTL)
\Include\zilog	Header files relevant to the ZSL device drivers
\Lib	Contains subfolders that contain the libraries files
\Lib\std	Library files relevant to the C Run Time Library (RTL)
\Lib\zilog	Library files relevant to the device drivers
\Src	Contains subfolder which contains source for each of the device
\Src\boot\common	Boot-related files common to all targets
\Src\boot\ez80F91	Boot-related files specific to ez80F91
\Src\boot\ez80F92	Boot-related files specific to ez80F92
\Src\boot\ez80F93	Boot-related files specific to ez80F93
\Src\boot\ez80190	Boot-related files specific to ez80190
\Src\boot\ez80L92	Boot-related files specific to ez80L92
\Src\<device>\common	Device-related files common to all targets
\Src\<device>\ez80F91	Device-related files specific to ez80F91
\Src\<device>\ez80F92	Device-related files specific to ez80F92
\Src\<device>\ez80F93	Device-related files specific to ez80F93
\Src\<device>\ez80190	Device-related files specific to ez80190
\Src\<device>\ez80L92	Device-related files specific to ez80L92

Note: <device> specifies the on-chip peripheral device; for example, GPIO or UART.

ZSL Debug and Release Version

ZSL has two versions—the debug version and the release version available for every eZ80Acclaim!/eZ80Acclaim*Plus!* on-chip peripheral device. The debug version of the library is built to contain debug information without any optimization, whereas the release version is built to contain no debug information but is optimized for speed. The debug version of the library is built by defining the macro `PARAMETER_CHECKING`, which is used by some of the APIs to check for the validity of the parameters passed. This macro is absent in the ZSL release version, which does not perform any check on the API parameters. As a result, there is a significant difference in the overall size of the generated library from the two versions. See individual APIs in this manual to check if an API uses the `PARAMETER_CHECKING` macro or not.

Building the Zilog Standard Libraries

You can develop applications using these APIs for specific peripherals, and make use of the ZSL to interface with eZ80Acclaim!/ eZ80Acclaim*Plus!* peripheral devices. However, if you must customize these library files by modifying the source code, this section describes how the modified library is built using batch files and ZDS II script files.

As a general rule, when batch files are executed, the libraries for each on-chip peripheral device are rebuilt and copied to the `..\lib\zilog` directory. When a device has a different set of features across different target processors, separate libraries are built for each target processor. In addition, you must make a selection to build either the debug version or the release version of the ZSL.

For the UART peripheral, the register definitions are not same for all the eZ80Acclaim!/eZ80Acclaim*Plus!* processors. Therefore, separate libraries are built for each processor.

The following table provides two examples:

Processor	UART Library
eZ80F91	`uartf91D.lib`
	`uartf91.lib`
eZ80F92	`uartf92D.lib`
	`uartf92.lib`

For GPIO peripherals, which features register addresses and definitions common to all of the target devices, a single set of debug (`gpioD.lib`) and release (`gpio.lib`) libraries are built.

The source directory for every on-chip peripheral device contains a single batch file that you must execute to build all the libraries for the target devices pertinent to a device. The source directory also contains individual batch files which you can execute to build the libraries specific to a target processor.

Each of these batch files calls other batch files that perform specific operations in the overall build process as described in the following sections. For example, the UART device on the eZ80F91, Figure 3 on page 6 displays the hierarchy of batch file calls.

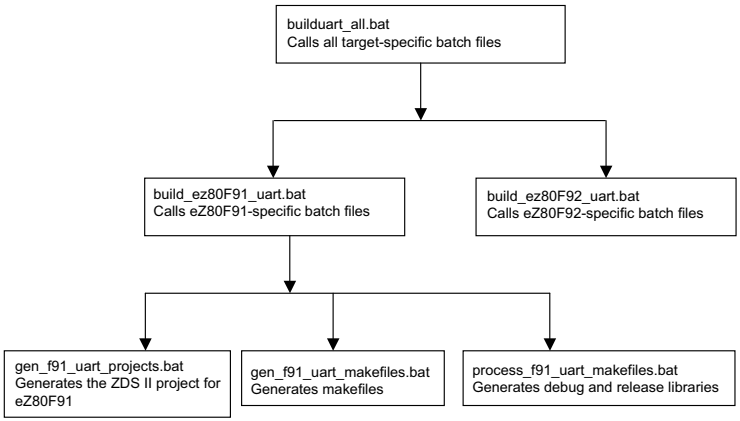

Figure 3. Flow of Batch File Calls to Build the UART–eZ80F91 Library

Follow the steps below to build a library:

1. **Generating a ZDS II Project File**—A ZDS II project is created for the specific target processor 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 file also calls another script file to add all of the source files associated with the library into the project. For example, in case of eZ80F91 UART, the `gen_f91_uart_projects.bat` batch file is used to generate the project file. It calls the `gen_f91_uart_projects.scr` script file to create the ZDS II project and invoke another script file, `add_f91_uart_projectfiles.scr`, to add all of the source files relevant to the library.
2. **Generating Make Files**—From the project generated in Step 1, Make files for both the debug and release versions are generated

using a batch file and a ZDS II script file. For the eZ80F91 UART, the `gen_f91_uart_projects.bat` batch file invokes a ZDS II script file, `gen_f91_uart_projects.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 generate the debug and release versions of the library. The libraries are automatically copied to the standard repository under `..\lib\zilog` directory. For example, in the case of the eZ80F91 UART, the `process_f91_uart_makefiles.bat` batch file generates the `uartf91D.lib` and `uartf91.lib` files.

Start-up Routine

The ZSL is integrated with ZDS II, which allows you to choose the device(s) required for your application, and also specify some of the device-dependent parameters. The option to choose the device and specify its parameters is available in ZDS II under the menu sequence **Project → Settings → ZSL.** For more information on using ZSL from within ZDS II, refer to *ZDS II–eZ80Acclaim User Manual (UM0144)*, available with the ZDS II tool package on www.zilog.com.

ZDS II copies the Zilog Standard Library device initialization file `zsldevinit.asm` into the project when ZSL is selected from within ZDS II. The initialization file contains the `_open_periphdevice()` function that calls the initialization routines for all of the 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.

Your application initializes the required device(s) to their default values without calling the `startup` routine. To do so, the 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 description on topics related to the Zilog Standard Library APIs for eZ80Acclaim!/eZ80Acclaim*Plus!* peripheral devices.

Standard Data Types

The ZSL employs user-defined data types in all the APIs. These user-defined data types are defined in the header file `defines.h`, which is located in the following directory:

```
..\include\zilog
```

API Definition Format

All ZSL APIs are described under following headings:

Prototype

This section contains the exact declaration of the API.

Description

This section provides a description of the API.

Argument(s)

This section describes all the parameters (if any) to the API. By default, all parameters are input parameters unless explicitly specified.

Return Value(s)

This section describes all return values (if any) of the API. This also includes the possible error values.

Table 5 lists the eZ80Acclaim!/eZ80Acclaim*Plus!* devices for which ZSL APIs are provided with the release of ZDS II–eZ80Acclaim!/ eZ80Acclaim*Plus!* v 4.8.0 and later.

Table 5. ZSL APIs for eZ80Acclaim*Plus!* On-Chip Devices

Device Name	Type of API	Description
UART	UART (generic) APIs	These APIs are the standard RTL I/O routines.
	UARTx APIs	These APIs are specific to a particular UART device, either UART0 or UART1. The *x* in the API name represents the selected UART device.
GPIO	GPIO APIs	These APIs are specific to the GPIO ports A, B, C, and D. The x in the API name represents the selected GPIO port.

ZSL GPIO API Description

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

GPIO Port Initialization in the Startup Routine

The ZSL is integrated with ZDS II, thereby allowing you to select or deselect eZ80Acclaim!/eZ80Acclaim*Plus!* GPIO ports. For more information on start-up routines, see Start-up Routine on page 7. When a GPIO port is selected in ZDS II using the sequence Project → **Settings** → **ZSL**, ZDS II generates a compiler predefine, `ZSL_DEVICE_PORTx`, in which `x` can be one of the A, B, C, or D GPIO ports.

ZDS II also adds a device initialization file, `zsldevinit.asm`, to the project. The `zsldevinit.asm` file uses compiler predefines (macros) to initialize the ports to their default states. The `_open_periphdevice()` function in `zsldevinit.asm` calls `open_Portx()` API for each of the ports selected from within the ZDS II IDE.

GPIO APIs

Table 6 lists all GPIO APIs.

Table 6. ZSL GPIO APIs

open_Portx	setmode_Portx
control_Portx	close_Portx
getsettings_Portx	

open_Portx

Prototype

```
VOID open_Portx(PORT * pPort);
```

Description

The `open_Portx()` API opens the selected port by initializing the port registers with the values supplied in the `PORT` structure parameter. If `NULL` value is passed, the API configures the port as standard digital input pin (GPIO mode 2) by setting the port registers with appropriate values as defined in `gpio.h` file. The API then calls the `control_Portx()` API to set the port register values.

Argument(s)

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

Return Value(s)

None.

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 `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.

getsettings_Portx

Prototype

```
VOID getsettings_Portx(PORT * pPort);
```

Description

The `getsettings_Portx()` API retrieves values of the selected port registers. The values of DR, DDR, ALT1, and ALT2 registers are returned via the `PORT` structure.

Argument(s)

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

Return Value(s)

None.

setmode_Portx

Prototype

```
UCHAR setmode_Portx(UCHAR pins, GPIOMODES mode);
```

Description

This `setmode_Portx()` API is used to configure one or more pins of the selected GPIO port for any of the nine modes supported by the eZ80Acclaim!/eZ80Acclaim*Plus!* family of devices. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured.

For example, the operating mode for Port A Pin 7 (PA7) is set by the values contained in PA_DR[7], PA_DDR[7], PA_ALT1[7], and PA_ALT2[7] registers.

A combination of the GPIO Control register bits allows individual configuration of each port pin for nine modes. For example, to set Pin 1 of Port A (PA1) into output mode (mode 1), bit 1 of each of the PA_DDR, PA_ALT1, PA_ALT2 must be set to 0. To accomplish this setting, call the `setmode_PortA()` API by specifying the bit corresponding to the pin by using the definitions provided in the `gpio.h` file as illustrated below:

```
setmode_PortA(PORTPIN_ONE,GPIOMODE_OUTPUT);
```

More than one pin can be set to the same mode by applying the OR logical operator to the pins in the call to the API. For example, to set pin 5 and pin 7 of Port A into output mode, the API is used as illustrated below:

```
setmode_PortA(PORTPIN_FIVE|PORTPIN_SEVEN,GPIO_OUTPUT);
```

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

Argument(s)

`pins`	The bitwise ORed value indicates the pins of a port as defined as `gpio.h` file.
`mode`	The mode to which the pins are configured.

Return Value(s)

`GPIOERR_FAILURE`	The specified mode is not supported by this port. This return value implies that either the mode parameter passed is not one of the nine modes or the API is called to set Port A for alternate function mode, which is invalid.

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 (GPIO mode 2). All the registers, except the DR register, are set with reset values.

Argument(s)

None.

Return Value(s)

None.

ZSL GPIO Macros

Table 7 lists the ZSL GPIO macro definitions.

Table 7. ZSL GPIO Macro Definitions

Macro	Description
`SETDR_PORTB(x)(PB_DR = (x))`	Set Port B Data Register with the value specified by argument x
`SETDDR_PORTB(x)(PB_DDR = (x))`	Set Port B Data Direction Register with the value specified by argument x
`SETALT1_PORTB(x)(PB_ALT1 = (x))`	Set Port B Alternate Register-1 with the value specified by argument x
`SETALT2_PORTB(x)(PB_ALT2 = (x))`	Set Port B Alternate Register-2 with the value specified by argument x
`GETDR_PORTB(x)(x) = PB_DR)`	Read Port B Data Register into argument x
`GETDDR_PORTB(x)((x) = PB_DDR)`	Read Port B Data Direction Register into argument x
`GETALT1_PORTB(x)((x) = PB_ALT1)`	Read Port B Alternate Register-1 into argument x
`GETALT2_PORTB(x)((x) = PB_ALT2)`	Read Port B Alternate Register-2 into argument x
`SETDR_PORTC(x)(PC_DR = (x))`	Set Port C Data Register with the value specified by argument x
`SETDDR_PORTC(x)(PC_DDR = (x))`	Set Port C Data Direction Register with the value specified by argument x
`SETALT1_PORTC(x)(PC_ALT1 = (x))`	Set Port C Alternate Register-1 with the value specified by argument x
`SETALT2_PORTC(x)(PC_ALT2 = (x))`	Set Port C Alternate Register-2 with the value specified by argument x

Table 7. ZSL GPIO Macro Definitions (Continued)

Macro	Description
`GETDR_PORTC(x)((x) = PC_DR)`	Read Port C Data Register into argument x
`GETDDR_PORTC(x)((x) = PC_DDR)`	Read Port C Data Direction Register into argument x
`GETALT1_PORTC(x)((x) = PC_ALT1)`	Read Port C Alternate Register-1 into argument x
`GETALT2_PORTC(x)((x) = PC_ALT2)`	Read Port C Alternate Register-2 into argument x
`SETDR_PORTD(x)(PD_DR = (x))`	Set Port D Data Register with the value specified by argument x
`SETDDR_PORTD(x)(PD_DDR = (x))`	Set Port D Data Direction Register with the value specified by argument x
`SETALT1_PORTD(x)(PD_ALT1 = (x))`	Set Port D Alternate Register-1 with the value specified by argument x
`SETALT2_PORTD(x)(PD_ALT2 = (x))`	Set Port D Alternate Register-2 with the value specified by argument x
`GETDR_PORTD(x)((x) = PD_DR)`	Read Port D Data Register into argument x
`GETDDR_PORTD(x)((x) = PD_DDR)`	Read Port D Data Direction Register into argument x
`GETALT1_PORTD(x)((x) = PD_ALT1)`	Read Port D Alternate Register-1 into argument x
`GETALT2_PORTD(x)((x) = PD_ALT2)`	Read Port D Alternate Register-2 into argument x

ZSL UART API Description

This chapter provides detailed description of the ZSL UART APIs. To use the ZSL UART APIs, the file `uart.h` must be included in the application program.

UART Initialization

The ZSL is integrated with ZDS II and allows you to select or deselect eZ80Acclaim!/eZ80Acclaim*Plus!* UARTs. For more information on start-up routines, see Start-up Routine on page 7. While initializing the UART devices in the Start-up routine, the following points must be considered:

1. When a UART device is selected, by using the sequence **Project** → **Settings** → **ZSL** in ZDS II, the ZDS II generates a compiler pre-define, `ZSL_DEVICE_UARTx`, for the `_open_periphdevice()` routine to use. The `_open_periphdevice()` routine uses the `open_UARTx()` function to initialize the UARTx device with default values when the `ZSL_DEVICE_UARTx` symbol is supplied. Therefore, your application can use 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 D is initialized; UART1 device requires Port C 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 the standard RTL I/O functions, `putch()`, `getch()`, `kbhit()`, and `peekc()`, are mapped to the default UART device—implying that the standard RTL I/O functions invoke the default UART device APIs. In the Zilog Standard Library distribution, UART0 is configured as the default device. To use UART1 as the default device,

in `uart.h` file, the macro `DEFAULT_UART0` is undefined, a macro `DEFAULT_UART1` is defined and the library is rebuilt.

4. You can modify the default values to suit the application specifications by making appropriate changes in the device-specific source code files. Default device parameters like optimization type, mode of operations, and blocking or non-blocking, can be changed to suit the requirements. Default values for one or both the UART devices may be modified. However, on modifying the default values, the library must be rebuilt for the changes to take effect.

Caution: *Exercise caution while making changes to default values else undesirable results can occur. By default, the UART devices are set to interrupt mode, with blocking calls. Certain calls, such as* `getch()`*, block indefinitely, and represent the standard defined behavior. If the* `getch()` *call is changed to a non-blocking type of call, ZSL cannot be used by the RTL function call,* `getch()`*.*

5. The UART driver operates in two modes—interrupt mode and poll mode. In each mode, the calls are always of the blocking type.The interrupt mode of operation uses a software buffer for both transmit and receive FIFO buffers. Therefore, each UART device features two FIFO buffers defined as global static arrays of fixed size (64 bytes) in the file `zsldevinit.asm`. You can change the default size to any value and rebuild the application for changes to take effect.

Generic UART APIs

Table 8 lists the generic UART APIs.

Table 8. Generic UART APIs

getch	kbhit
putch	peekc

getch

Prototype

```
INT24 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 gets blocked till the data becomes available. When the FIFO buffer is enabled, this `getch()` API returns the data byte at the top of the FIFO buffer.

This API calls the `read_UARTx()` API. If there is any error in the received data byte, an error code is set in the `UARTx_SPR` register. The application determines the error by updating the `UARTx_SPR` register with a known value, before calling the API and read the `UARTx_SPR` register again to determine whether that value has changed. For possible list of errors, see read_UARTx on page 42.

Argument(s)

None.

Return Value(s)

Returns the character received.

putch

Prototype

```
VOID putch(INT24 ich);
```

Description

The `putch()` API writes a data byte into the default UART transmit buffer. In case of the `INTERRUPT` mode where the FIFO buffer is enabled, the data byte is written into the end of the FIFO 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)

Character to write to transmit buffer.

Return Value(s)

None.

kbhit

Prototype

```
UCHAR kbhit(VOID);
```

Description

The `kbhit()` API detects for any keystrokes on the default UART device. If a keystroke is detected the `kbhit()` API returns `1` else `0` is returned. 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 returns the status.*

Argument(s)

None.

Return Value(s)

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

peekc

Prototype

```
INT24 peekc(VOID);
```

Description

The `peekc()` API is supported only when a UART device is configured for the `INTERRUPT` mode. This API checks for any data in the receive FIFO buffer and returns the data, if present. However, the character checked in the receive FIFO buffer is not cleared from the buffer. This `peekc()` function returns `-1` if invoked in `POLL` mode, or if the FIFO buffer contains an error.

If there is an error in the received data byte, an error code is set in the `UARTx_SPR` register. Your application determines the error by updating the `UARTx_SPR` register with a known value, before calling the API and reading the `UARTx_SPR` register again to determine whether that value has changed. For possible errors, see read_UARTx on page 42.

Argument(s)

None.

Return Value(s)

Returns the character on top of the FIFO buffer without clearing the character from the FIFO buffer.

Returns -1 in case of an error or when the API is invoked in the POLL mode. In this case, `UARTx_SPR` register contains the exact error code.

UARTx APIs

The UARTx APIs listed in this section are used for either of the UART devices on the eZ80Acclaim!/eZ80Acclaim*Plus!* family of microcontrollers. The x in UARTx signifies 0 or 1 for the UART0 or UART1 devices, respectively.

Table 9 lists the UARTx APIs.

Table 9. UART*x* APIs

open_UARTx	clearbreak_UARTx
control_UARTx	flush_UARTx
setbaud_UARTx	write_UARTx
setparity_UARTx	read_UARTx
setstopbits_UARTx	enableparity_UARTx
setdatabits_UARTx	disableparity_UARTx
settriggerlevel_UARTx	close_UARTx
sendbreak_UARTx	

open_UARTx

Prototype

```
UCHAR open_UARTx(UART * pUART);
```

Description

The `open_UARTx()` API opens the UARTx device by initializing the UARTx Control registers with the values supplied in the `UART` structure parameter. If `NULL` value is passed, then the API sets the UARTx Control registers with the default values.

In either case, this API calls the `control_UARTx()` API to set the UARTx Control register values, which configures the Port D bits 0 and 1 for alternate functions.

In either case, this API configures the required port pins for alternate functions and calls the `control_UARTx()` API to set the UARTx control registers.

Argument(s)

pUART	Pointer to a structure of the type UART as defined in `uart.h` file. If `NULL` value is passed, the following default values are set.		
	`uartmode`	→	interrupt or poll.
	`baudrate`	→	Default value: 57600 Other valid values are 9600, 19200, 38400, 115200.
	`databits`	→	Default value: 8 Other valid values are 8, 7, 6, 5.
	`stopbits`	→	Default value: 2 Other valid values are 1, 2.
	`parity`	→	Default value: disable Other valid values are `PAR_NOPARITY`, `PAR_ODPARITY`, `PAR_EVPARITY`.
	`fifotriggerlevel`	→	Default value: 1 Other valid values are1, 4, 8, 14.
	`hwflowcontrol`	→	Default value: disable
	`swflowcontrol`	→	Default value: disable

Return Value(s)

`UART_ERR_NONE`	On successful execution of the API.
`UART_ERR_INVBAUDRATE`	Error due to invalid baudrate value passed.
`UART_ERR_INVDATABITS`	Error due to invalid databits value passed.
`UART_ERR_INVSTOPBITS`	Error due to invalid stopbits value passed.

UART_ERR_INVPARITY	Error due to invalid parity value passed.
UART_ERR_INVTRIGGERLEVEL	Error due to invalid trigger level value passed.

Notes:
1. *The hardware and software flow controls are not yet supported in the Zilog Standard Library. Therefore,* `hwflowcontrol` *and* `swflowcontrol` *must always be disabled.*
2. *When the UART channels are used in* INTERRUPT *mode,* `set_vector()` *must be called by passing the appropriate vector and handler.*
3. *While using the eZ80F91 Mini Module, and in addition to calling* `open_UART0()`*, the* `ENABLE_UART0()` *macro must be explicitly called in the application prior to any UART related APIs.*

control_UARTx

Prototype

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

Description

The `control_UARTx()` API is used to configure the UARTx 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 appropriate UARTx device Control registers. The API checks the validity of the parameters passed when the debug version of ZSL is used. The `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, the API configures the UARTx with the value passed in the `pUART` parameter.

Argument(s)

`pUART` Pointer to a structure of the type UART as defined in `uart.h` file

`uartmode`	→	Interrupt
`baudrate`	→	Default value: 57600 Other valid values are 9600, 19200, 38400, 115200.
`databits`	→	Default value: 8 Other valid values are 8, 7, 6, 5.
`stopbits`	→	Default value: 2 Other valid values are 1, 2.
`parity`	→	Default value: disable Other valid values are `PAR_NOPARITY`, `PAR_ODPARITY`, `PAR_EVPARITY`.
`fifotriggerlevel`	→	Default value: 1 Other valid values are 1, 4, 8, 14.
`hwflowcontrol`	→	Default value: disable
`swflowcontrol`	→	Default value: disable

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVBAUDRATE`	Error due to invalid baudrate value passed.
`UART_ERR_INVDATABITS`	Error due to invalid databits value passed.
`UART_ERR_INVSTOPBITS`	Error due to invalid stopbits value passed.
`UART_ERR_INVPARITY`	Error due to invalid parity value passed.
`UART_ERR_INVTRIGGERLEVEL`	Error due to invalid trigger level value passed.

Note: *The hardware and software flow controls are not yet supported in the Zilog Standard Library. Therefore,* `hwflowcontrol` *and* `swflowcontrol` *must always be disabled.*

setbaud_UARTx

Prototype

```
UCHAR setbaud_UARTx(INT24 baud);
```

Description

The `setbaud_UARTx()` API configures the baudrate for the UARTx device with the specified value. The API checks the validity of the parameters passed when the debug version of ZSL is used. The `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, the API configures UARTx with the value passed in the `baud` parameter.

Example:

```
setbaud_UART0(115200);
```

Argument(s)

`baud` Specifies the new baudrate to be set. This value along with the target clock frequency value set during the opening of UART, are used to calculate the value for Baudrate Generator registers.

`baudrate` → Default value: 57600
Other valid values are 9600, 19200, 38400, 115200.

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVBAUDRATE`	Error due to invalid baudrate value passed.

setparity_UARTx

Prototype

```
UCHAR setparity_UARTx(UCHAR parity);
```

Description

The `setparity_UARTx()` API configures the parity for the UARTx device. This API checks the validity of the parameters passed when the debug version of ZSL is used. The `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, the API configures the UARTx with the value passed in the `parity` parameter.

Argument(s)

`parity` Specifies the new parity value.

`parity`	→	Default value: disable Other valid values are `PAR_NOPARITY`, `PAR_ODPARITY`, `PAR_EVPARITY`.

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVPARITY`	Error due to invalid parity values.

setstopbits_UARTx

Prototype

```
UCHAR setstopbits_UARTx(UCHAR stopbits);
```

Description

The `setstopbits_UARTx()` API sets the stopbits for the UARTx device. The API checks the validity of the parameters passed when the debug version of ZSL is used. The `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, the API configures the UARTx with the value passed in the `stopbits` parameter.

Argument(s)

`stopbits`	Number of valid stopbits set.	
	`stopbits` →	Default value: 2 Other valid values are1, 2.

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVSTOPBITS`	Error due to invalid stopbits.

setdatabits_UARTx

Prototype

```
UCHAR setdatabits_UARTx(UCHAR databits);
```

Description

The `setdatabits_UARTx()` API configures the data bits for the UARTx device. The API checks the validity of the parameters passed when the debug version of ZSL is used; the `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, the API configures the UARTx with the value passed in the `databits` parameter.

Argument(s)

`databits`	Value of databits to be set.		
	`databits`	→	Default value: 8 Other valid values are 8, 7, 6, 5.

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVDATABITS`	Error due to invalid databits.

settriggerlevel_UARTx

Prototype

```
UCHAR settriggerlevel_UARTx(UCHAR trglevel);
```

Description

The `settriggerlevel_UARTx()` API configures the receive FIFO trigger level for the UARTx device by setting the `trglevel` value to the UARTx FIFO Control Register. When there are specified number of bytes in the FIFO buffer, receive-data interrupt is generated. This API checks the validity of the parameters passed when the debug version of ZSL is used. The `PARAMETER_CHECKING` macro is defined in the debug version. In the release version of ZSL, the macro `PARAMETER_CHECKING` is undefined and therefore this check is not performed. In the release version, this API configures the UARTx with the value passed in the `trglevel` parameter

Argument(s)

`trglevel`	Receive FIFO trigger level.		
	`fifotriggerlevel`	→	Default value: 1 Other valid values are 1, 4, 8, 14.

Return Value(s)

`UART_ERROR_NONE`	On successful execution of the API.
`UART_ERR_INVTRIGGERLEVEL`	Error due to invalid trigger level.

sendbreak_UARTx

Prototype

```
VOID sendbreak_UARTx(void);
```

Description

The `sendbreak_UARTx()` API sets the `Send Break` bit in the `UARTx_LCTL`, forcing the UARTx device to send continuous zeros.

Argument(s)

None.

Return Value(s)

None.

clearbreak_UARTx

Prototype

```
VOID clearbreak_UARTx(void);
```

Description

The `clearbreak_UARTx()` API clears the `Send Break` bit in the `UARTx_LCTL`, forcing the UARTx device to stop sending the break signals.

Argument(s)

None.

Return Value(s)

None.

flush_UARTx

Prototype

```
UCHAR flush_UARTx(UCHAR fifo);
```

Description

The `flush_UARTx` ( ) API flushes both hardware and software FIFO buffers. You can flush either the transmit FIFO buffer or the receive FIFO buffer or both by passing appropriate values in the `fifo` parameter. The contents of both the hardware and software FIFO buffers are cleared.

Argument(s)

`fifo` Specifies the FIFO buffers to be flushed.

`FLUSHFIFO_TX`	→	flushing Tx FIFO buffer.
`FLUSHFIFO_RX`	→	flushing Rx FIFO buffer.
`FLUSH_ALL`	→	flushing both Tx and Rx FIFO buffers.

Return Value(s)

None.

write_UARTx

Prototype

```
UCHAR write_UARTx(CHAR *pData, UINT24 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. Depending on the mode, the `write_UARTx()` API either polls or uses an interrupt to transmit the data bytes. When the GPIO port is configured for the `INTERRUPT` mode, this API puts the data in the transmit FIFO buffer. If the value of `nbytes` is greater than the size of the transmit FIFO buffer, and if the transmit FIFO buffer is full, the API blocks until the data bytes are transmitted.

Argument(s)

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

Return Value(s)

`UART_ERR_NONE`	Successful execution of the API.

read_UART*x*

Prototype

```
UCHAR read_UARTx(CHAR *pData, UINT24 *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. Depending on the mode, it either polls or uses the interrupts to receive data bytes.

If the specified number of bytes are not available, the API gets blocked indefinitely until the data becomes available. If the data byte received in the receive FIFO buffer contains errors, the API returns with an error code. The value in the `nbytes` parameter indicates the number of data bytes actually read and the last byte in the receive buffer is the data byte containing the error.

Note: *When an error occurs, ensure that the application flushes the receive FIFO buffer (if operating in the* `INTERRUPT` *mode), using the* `flush_UARTx()` *API before calling the next* `read_UARTx()` *API.*

Argument(s)

`pData`	Pointer to a buffer to receive data.
`nbytes`	Number of bytes to read. In case of an error, the API fills this variable with the actual number of bytes read. The application then determines the data byte that is with an error.

Return Value(s)

`UART_ERR_FRAMINGERR`	Indicates that a framing error occurred in the character received.
`UART_ERR_PARITYERR`	Indicates that a parity error occurred in the character received.
`UART_ERR_OVERRUNERR`	Indicates an overrun error occurred in the receive buffer register.
`UART_ERR_BREAKINDICATIONERR`	Indicates that a Break condition is set.

enableparity_UARTx

Prototype

```
VOID enableparity_UARTx(void);
```

Description

The `enableparity_UARTx()` API enables parity checking (even parity or odd parity) on the incoming data received by the UARTx device. When a parity error is detected in the incoming data, the UARTx device logs this error in the Line Status register. This error is indicated to the application in the read_UARTx on page 42 API.

Argument(s)

None.

Return Value(s)

None.

disableparity_UARTx

Prototype

```
VOID disableparity_UARTx(void);
```

Description

The `disableparity_UARTx()` API disables the parity checking (even parity or odd parity) on the incoming data received by the UARTx device.

Argument(s)

None.

Return Value(s)

None.

close_UARTx

Prototype

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

Description

The `close_UARTx()` API is used to close the UARTx device. When `close_UARTx()` API is called, interrupts related to the default UART device are disabled. It also clears the Control registers so as to render the UART device non-functional. The application uses the UART device again only after calling the `open_UARTx()` API.

Argument(s)

None.

Return Value(s)

None.

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.