zilog®
Embedded in Life
An IXYS Company

eZ80® CPU

Zilog Real-Time Kernel

User Manual

UM007518-1211

This publication is subject to replacement by a later edition. To determine whether a later edition exists or to request copies of publications, visit www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

Revision History

Each instance in the Revision History table below reflects a change to this document from its previous version. For more details, click the appropriate links in the table.

Date	Revision Level	Description	Page
Dec 2011	18	Removed version reference.	vii
Aug 2010	17	Globally updated for the RZK v2.3.0 release.	All
Nov 2008	16	Updated the WLAN Configuration section and Table 5.	10
Sep 2008	15	Updated for the RZK v2.2.0 release; updated the RZK Board Support Package and In-Depth Questions About Using RZK sections, Figure 1 and Table 10. Added the WLAN Configuration and USB Configuration sections.	viii, 6, 10, 13, 14, 22
Jul 2007	14	Globally updated for style.	All
Jul 2007	13	Globally updated for the RZK v2.1.0 release.	All
Jun 2007	12	Updated for style. Removed RZK Characteristics Appendix. Updated Data Persistence Configuration and Table 10. Removed RZK Using the IAR Toolset description, Directory Structure for RZK, API Functions section, Creating and Running an RZK Project, Executing RZK Sample Programs sections.	All

Table of Contents

Introduction

This User Manual describes the Zilog Real-Time Kernel (RZK) software for application development on Zilog's eZ80 CPU-based microprocessors and microcontrollers. The current RZK release supports the eZ80 and eZ80Acclaim! product lines, which include the eZ80F91, eZ80F92 and eZ80F93 microcontrollers and the eZ80L92 microprocessor.

About This Manual

Zilog recommends that you read and understand the complete manual before using the product. This manual is designed to be used as an user guide for important data.

Intended Audience

This document is written for Zilog customers having exposure to RTOS and writing real-time application code and whom are also experienced at working with microprocessors/ microcontrollers and writing assembly code or compilers.

In addition to this manual, consider reading the documentation listed in Table 1.

Table 1. Related Documentation

Document Title	Document Number
Zilog Real-Time Kernel Product Brief	PB0155
Zilog Real-Time Kernel Quick Start Guide	QS0048
Zilog Real-Time Kernel Reference Manual	RM0006
eZ80 CPU User Manual	UM0077
eZ80Acclaim! Quick Start Guide	QS0020
eZ80F91 Product Specification	PS0192
eZ80F91 Development Kit User Manual	UM0142
eZ80F92/eZ80F93 Flash MCU Product Specification	PS0153
eZ80F92/eZ80F93 Ethernet Module Product Specification	PS0186
eZ80F92/eZ80F93 Module Product Specification	PS0189
eZ80F92/eZ80F93 Development Kit User Manual	UM0139
eZ80L92 Product Specification	PS0130
eZ80L92 Development Kit User Manual	UM0129
eZ80190 Product Specification	PS0066
Zilog Developer Studio II – eZ80Acclaim! User Manual	UM0144

Manual Organization

This document is organized into the following four chapters and an appendix:

RZK Overview

This chapter provides an overview of RZK, its features and objects.

Getting Started

This chapter provides the procedural steps for using RZK.

RZK Board Support Package

This chapter introduces the RZK board support package (BSP), which provides drivers for EMAC, UART, I^2C, SPI, USB, WLAN and RTC devices.

RZK Configuration

This chapter provides details about the RZK configurable parameters.

A Frequently Asked Questions

This Appendix provides the frequently asked questions and answers on RZK.

Abbreviations/Acronyms

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

Table 2. RZK-Related Abbreviations/Acronyms

ADC	Analog-to-Digital Converter
IJT	Interrupt Jump Table
IPC	Inter Process Communication
IVT	Interrupt Vector Table
BSP	Board Support Package
RZK	Zilog Real-Time Kernel
DDF	Device Driver Framework
ESD	Electro Static Discharge
ZTP	Zilog TCP/IP
ZDS	Zilog Developer Studio
EEPROM	Electrically Erasable Programmable Read Only Memory

Manual Conventions

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

Use of X.Y.Z and A.B.C

Throughout this document, `X.Y.Z` refers to the currently released version of RZK and A.B.C refers to the currently released version of ZDS II for eZ80Acclaim!.

Use of <tool>

Throughout this document, `<tool>` refers to ZDS II.

Courier New Typeface

Code lines and fragments, functions and various executable items are distinguished from general text by appearing in the Courier New typeface.

Safeguards

Always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD) to avoid permanent damage to the eZ80 Development Platform.

RZK Overview

The Zilog Real-Time Kernel (RZK) is a real-time, preemptive multitasking kernel designed for time-critical embedded applications. It is currently available with Zilog's eZ80 family of microprocessors and microcontrollers. The majority of the RZK code is written in ANSI C and supplied as a C library module. During compilation, you can link real-time applications with the RZK library. The resulting object is downloaded to the target platform or placed in ROM. RZK is designed to be used as a C library.

Features of RZK

The features of RZK include:

- Compact code
- Rapid context switching between threads
- Quick interrupt response
- Preemptive, priority-based and multitasking scheduler
- Timing support for delays, time-outs and periodic events
- Time-slicing option with adjustable time slices
- Priority inheritance facility
- Highly scalable and configurable options
- Minimal footprint
- Device Driver Framework (DDF) and Board Support Package (BSP)

RZK Objects

RZK modules are referred to as objects and are used for real-time application development as provided below:

- Threads
- Message queues
- Event groups
- Semaphores
- Timers
- Memory as Partitions/Regions

- Interrupts
- Device driver framework
- Board support package

Table 1 provides a brief description of the RZK objects.

Table 1. Description of RZK Objects

RZK Objects	Description
Thread	A thread is the basic object of RZK. RZK enables a particular thread to execute among all other threads, based on its priority and readiness to execute.
Message Queue	Two or more threads can use a message queue to communicate with each other asynchronously. The length of each message is a variable, provided the size is within the maximum message size given at the time of creation. Message contents are user-defined. By default, messages are added to the end of queue. An option is provided to insert the message at the head of the queue.
Event Group	Event is an optional object of RZK. Events can be grouped and operated upon logically. An event object allows a single thread to wait on one or more external event, using a single event API.
Semaphore	Semaphore is an optional object of RZK. Semaphore is the only object, which requires priority inheritance protocol and is tightly coupled with scheduling methods. As the semaphore uses a mutual-exclusion mechanism, it is not directed to a specific thread.
Timer	A Timer is an optional object of RZK. Timer objects invoke user-supplied functions that are to be processed at set periodic intervals.
Memory as Partitions/ Regions	Memory is an optional object of RZK. However, it is required for dynamic memory allocation by other objects (for example, message queues). Unused system memory is organized into two categories: • Partitions: fixed size memory blocks • Regions: variable size memory blocks
Interrupts	An interrupt is a signal from a device attached to a computer or from a program within the computer that causes the main program that operates the computer (the operating system) to stop and service the interrupt.
Device driver framework	The device driver framework is a common set of APIs to access any device that is present in the global device table.
Board support package	The board support package consists of drivers for the EMAC, UART, SPI, RTC and I^2C peripheral blocks of the eZ80 CPU.

Note: For more information about RZK objects, refer to the Zilog Real-Time Kernel Reference Manual (RM0006).

Limitations

The `RZK_X.Y.Z_Lib_<tool>` release has the following limitations:

- The nested interrupt handling for the same type of interrupt is not possible
- Kernel-aware debug facility is not available

Developing Software Components

You can build your applications to run on RZK, which must be configured according to the target platform; configuration details are provided in the RZK Configuration section on page 14.

The RZK release also contains sample applications that can be downloaded directly onto target platforms.

Note: Configuration changes may be required according to your choice of target platform.

Getting Started

RZK is automatically installed when the ZDSII_eZ80Acclaim!_A.B.C file is installed. You can find that RZK is installed under

```
<ZDSII installed directory>Program files\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib
```

For information about system requirements for the target and host computer, refer to the Zilog Real-Time Kernel Quick Start Guide (QS0048), which is available free for download from the zilog.com website. These requirements must be met before proceeding to write or build applications based on RZK.

Using RZK

RZK is designed to be used as a C library. The RZK objects referenced in the application software are extracted from the RZK library and combined with the application objects to produce a target-downloadable image. This image can be downloaded to the target system RAM or placed into ROM, EEPROM or Flash.

Observe the following procedure to use RZK in the ZDSII environment:

1. Include the following header files in the application in the sequence provided below:
   a. `ZSysgen.h.`
   b. `ZTypes.h.`
   c. Header files such as `ZThread.h`, `Zmemory.h` and other application header files related to the objects used in the application.

2. Add your application code (creating the resources/threads, etc.) between the `RZK_KernelInit()` and `RZK_KernelStart()` function calls in the `main()` function shown below. The `main()` function runs at the highest priority.

```
int main(int argc, void *argv[])
{
  RZK_KernelInit();
                 // Your application threads/resources creation
                 // code here//
  RZK_KernelStart();
}
```

Note: For information about the `main()`, `RZK_KernelInit()` and `RZK_KernelStart()` functions, refer to the Zilog Real-Time Kernel Reference Manual (RM0006).

3. Compile and/or assemble all of the application software with the default settings provided in the project settings.
4. Link all of the necessary files including object files to the RZK library and any other development tool libraries.
5. To run the application, download the complete application program image onto the target system.

Note: Details about the macros that must be defined for adding or removing files using RZK are described in the RZK Configuration chapter on page 14.

RZK Board Support Package

The RZK Board Support Package (BSP) provides drivers for EMAC, WLAN, UART, I^{2}C, SPI, USB and RTC devices. These drivers, with the exception of the USB driver, use the RZK Device Driver Framework (DDF) that provides a common interface for gaining access to various devices. The RZK BSP also provides drivers for different Flash devices that do not use the DDF. These drivers are simple APIs for read/write/erase operations in the Flash memory. The RZK BSP includes the following features:

- A DDF to provide a common interface to gain access to many different devices
- Reentrancy handled within the driver
- Drivers offer minimum interrupt latency
- Configurable

BSP Use Case Model

Figure 1 displays the RZK BSP use case model.

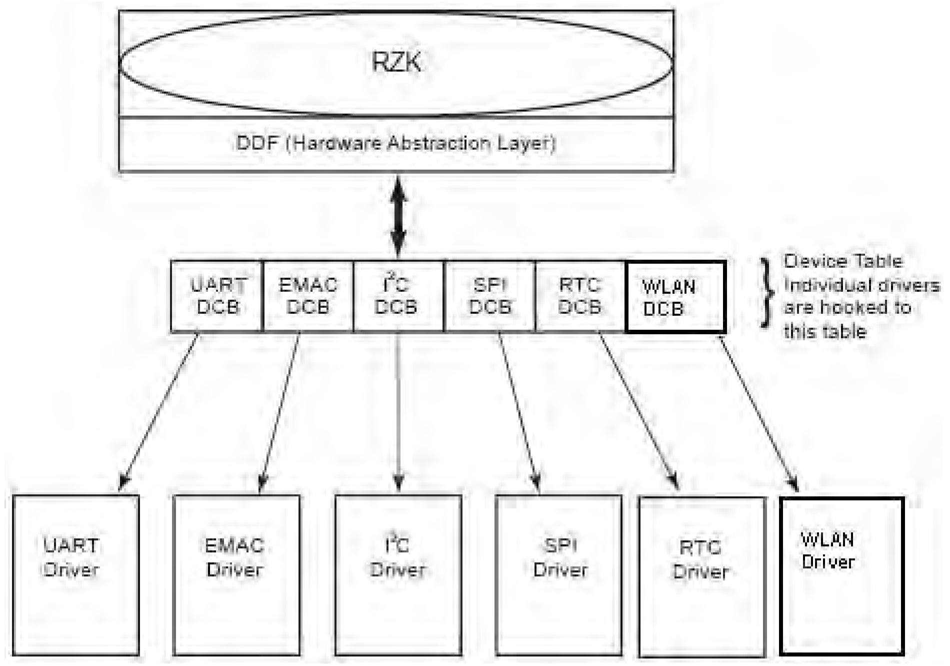

Figure 1. RZK BSP Use Case Model

RZK provides a very simple and generic device driver model that can be used to develop drivers for different types of devices; RZK BSP is based on this device driver model.

The RZK device driver model contains a hardware abstraction layer called the DDF, which provides a common interface to gain access to various devices. DDF accesses these devices using the global device table. The driver functions are connected to this device table. Each device is identified by its *handle*, which is the pointer to the device entry in the device table `usrDevBlk`.

The RZK DDF provides APIs that perform operations on any of the drivers present in the BSP. These APIs are briefly described in Table 2.

Table 2. RZK Device Driver Framework APIs

DDF API	Description
RZKDevAttach	Attaches the device for communication
RZKDevDetach	Detaches the device from communication
RZKDevOpen	Opens the device for communication
RZKDevRead	Reads from the device
RZKDevWrite	Writes to the device
RZKDevIOCTL	Performs I/O control operations
RZKDevClose	Closes the device for communication

Note: For details about DDF APIs, refer to the Zilog Real-Time Kernel Reference Manual (RM0006).

Board Support Package Configuration

This section discusses the available RZK BSP configurations.

UART Configuration

The configurable parameters for the UART driver are located in the `uart_conf.c` source file that accompanies the RZK release in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any of the UART parameters, you must include this file in the project and set the appropriate values, as defined in Table 3.

Table 3. Configurable UART Drivers

Variable/Macro	Default Value	Description/Valid Values
UART0_THD_STACK_SIZEH	1024	UART0 Interrupt thread stack size.
UART0_TASK_PRIOH	6	UART0 Interrupt thread priority.
UART1_THD_STACK_SIZEH	2048	UART1 Interrupt thread stack size.
UART1_TASK_PRIOH	6	UART1 Interrupt thread priority.
serparams	—	Array of structure serialparam that contains values that the UART device is to be initialized with. The valid serparams values below describe the structure members.
serparams baud	2400/9600/19200/38400/ 57600/115200	Sets the baud rate to any of the default values.
serparams data bits	7/8	Sets the data bits to any of the default values.
serparams stop bits	1/2	Sets the stop bits to any of the default values.
serparams parity	PAREVEN/ PARODD/ PARNONE	Sets the parity to any of the default values.
serparams settings	Can contain combinational values with logical OR (|) operation, as described below:	
	SERSET_DTR_ON	This Flag directs the serial driver to assert the data terminal ready (DTR) signal when the corresponding serial device (UART) is open.
	SERSET_RTSCTS	This Flag directs the UART driver to use the ready to send (RTS)/clear to send (CTS) flow control over the serial link.
	SERSET_DTRDSR	This Flag is currently not used by the UART driver.
	SERSET_XONXOFF	This Flag is currently not used by the UART driver.
	SERSET_ONLCR	This Flag directs UART driver to convert each outgoing new-line character (for example, `\n`) to a new line + carriage return (for example, `\r\n`). This setting is required by some terminal emulators to ensure that the console output is displayed correctly. This Flag must not be used with PPP.

Table 3. Configurable UART Drivers (Continued)

Variable/Macro	Default Value	Description/Valid Values
serparams settings (cont'd)	SERSET_SYNC	This Flag directs UART driver to use a synchronous I/O routine to transfer data over the serial port. Synchronous I/O operations require UART driver to poll for the underlying UART hardware instead of using interrupts. This polling instance is not efficient and usually results in many lost characters. Zilog does not recommend the use of this Flag setting.
	SERSET_IGNHUP	If the serial driver detects the loss of a valid Carrier Detect signal, the driver assumes that the physical link is disconnected by the remote end of the serial connection. As a result, UART driver automatically closes the underlying serial device and terminates all of the PPP or serial communications. If this Flag is included in the serparams structure, then UART driver ignores the loss of the valid Carrier Detect signal.

Example

To configure UART for a 57600 bps baud rate, 8 data bits, 1 stop bit and no parity, and to specify the use of the `SERSET_ONLCR` and `SERSET_IGNHUP` flags, the following code line must be entered in the `serparams` array:

```
{57600, 8, 1, PARNONE, SERSET_ONLCR | SERSET_IGNHUP}
```

EMAC Configuration

The configurable parameters for the EMAC driver are located in the `emac_conf.c` source file that accompanies the RZK release in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any of the EMAC parameters, include this file in the project and set the appropriate values, as defined in Table 4.

Table 4. Configurable EMAC Drivers

Variable/Macro	Default Value	Description/Valid Values
f91_mac_addr	{ 0x00, 0x90, 0x23, 0x73, 0x50, 0x49 };	The default MAC address must be initialized in this array. You must change this value according to your application setup. If more than one system has the same MAC address and if DHCP is enabled in ZTP, the same IP address is assigned to all of the systems with the same MAC address.
F91_emac_config (valid for eZ80F91 module only)	{ 1568, F91_AUTO, BUF32 };	A structure that contains values with which the EMAC device is to be initialized. Below are valid values for different structure members: F91_emac_config txBufSize = 0-1568 F91_emac_config mode = F91_10_HD, F91_10_FD, F91_100_HD, F91_100_FD or F91_AUTO F91_emac_config bufSize = 32
EMAC_THD_STACK_SIZE H	4096	EMAC interrupt thread's stack size.
EMAC_TASK_PRIOH	6	EMAC interrupt thread's priority.*

Note: * Do not change this value without the knowledge of the whole system. The behavior of the system is indeterminate if the default value is changed.

PHY Initialization

The `F91PhyInit.c` file contains the `phyInit()` initialization routine for the PHY. This `phyInit()` routine currently configures the AMD Am89C874 PHY device featured on the eZ80F91 Module and the MICREL KS8721 PHY device featured on the eZ80F91 Mini-Module. You can modify the `phyInit()` routine to initialize other PHY devices.

WLAN Configuration

The configurable parameters for the WLAN driver are located in the `wlan_conf.c` file which is available in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

For a description of the available WLAN drivers, see Table 5. If these parameters are set to 0/NULL (as in the default case), the user must call the `scan` and `join` commands to connect to a particular AP. Otherwise, the values provided in these parameters are considered to connect to the specified AP without user interaction.

Table 5. Configurable WLAN Drivers

Variable/Macro	Default Value	Description/Valid Values
g_Ssid	{0,""}	The first parameter of g_Ssid is the length of the SSID and second is the SSID string. For example {4,"ABCD"}.
wepKey[WEP_KEYLEN]	0	The encryption key for WEP.
wepKeyLen	0	The encryption key length.
Note: The Zdots SBC WLAN solution supports NO-Encryption, WEP-128-bit and 64-bit encryptions. The WEP128-bit encryption needs a 13 byte encryption key and WEP 64-bit requires 5 byte encryption key.		

RTC Configuration

The configurable parameters for the RTC driver are located in the `Rtc_conf.c` source file that accompanies the RZK release in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any of the RTC parameters, include this file in the project and set the appropriate values, as defined in Table 6.

Table 6. Configurable RTC Drivers

Variable/Macro	Default Value	Description/Valid Values
RTC_TASK_PRIOH	10	Priority of the RTC interrupt thread.
RTC_THD_STACK_SIZEH	1024	Stack size of RTC interrupt thread.

SPI Configuration

The configurable parameters for the SPI driver are located in the `spi_conf.c` source file that accompanies the RZK release in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any of the SPI parameters, include this file in the project and set the appropriate values, as defined in Table 7.

Table 7. Configurable SPI Drivers

Variable/Macro	Default Value	Description/Valid Values
SPI_TASK_PRIOH	10	Priority of the SPI interrupt thread.
SPI_THD_STACK_SIZEH	1024	Stack size of SPI interrupt thread.

I^{2}C Configuration

The configurable parameters for the I^{2}C driver are located in the `i2c_conf.c` source file that accompanies the RZK release in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any of the I^{2}C parameters, include this file in the project and set the appropriate values, as defined in Table 8.

Table 8. Configurable I^{2}C Drivers

Variable/Macro	Default Value	Description/Valid Values
I2C_TASK_STACK_SIZEH	1024	Stack size of the I^{2}C interrupt thread.
I2C_RX_MAX_BUFF_SIZEH	100	Size of the Rx circular queue.
I2C_TASK_PRIORITYH	6	Priority of the I^{2}C Interrupt thread.
i2cConfigParams	{0xB0, 0x00,0x58, I2C_MASTER, RZK_FALSE, 0x00,2}	
i2cConfigParams currSlaveAddr		The slave address with which the master communicates, for example, 0xA0.
i2cConfigParams selfAddr		The address of eZ80 when acting as a slave.
i2cConfigParams speed		The speed of the I^{2}C bus, for example, 0x58.
i2cConfigParams mode		I2C_MASTER/I2C_SLAVE; determines whether eZ80 will act as a master or slave.
i2cConfigParams useSubAddr		RZK_TRUE/RZK_FALSE; determines whether to use subaddresses for the slave device.
i2cConfigParams subAddr		subaddress value for the slave; used only if the useSubAddr Flag is set to RZK_TRUE.
i2cConfigParams addrLen		Length of subaddress; the number of subaddress bytes sent to the slave are based on this value.

Flash Drivers

Flash drivers are implemented as simple APIs that do not operate within the constructs of the DDF. To access these functions directly, include the device header files in the application.

The format of this header file is `XXXXX_Driver.h`, in which `XXXXX` represents the MT28F008, AM29LV160, AT49BV162 or IntFlash drivers that are located in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Inc
```

USB Configuration

The configurable parameters for the USB driver are located in the `EZ80D12_conf.c` file. This source file is located in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

To change any USB device configuration parameters, include this file in the project and set the appropriate values, as defined in Table 9. For more information about USB Specification, refer to www.usb.org.

Table 9. Configurable USB Driver Parameters

Variable/Macro	Default Value	Description/Valid Values
EZ80D12_TASK_PRIO	16	Priority of USB thread.
EZ80D12_THD_STACK_SIZE	2048	Stack size of the USB interrupt thread.
g_remoteWakeUp	0x02	Remote wakeup is enabled and can wake up the host during suspend.
G_powerStat	0x01	Decides if the device is Self powered or Bus powered.
g_testMode	EZ80D12_SET_ZERO	Test mode allows the device to exhibit various conditions.
gst_DevDescrip	—	Standard Device Descriptor.
gst_CompositDescrip	—	Composite Descriptor, which consists of Configuration Descriptor + Interface Descriptor + End Point Descriptor (EP2in + EP2Out).

RZK Configuration

This chapter describes the different RZK configurations that can be applied by the real-time application and lists the dependent libraries and files that must be added to the project workspace after a new application project workspace is created. Table 10 lists the macros to be defined and the files that must be added to the project workspace, for these different RZK configurations.

Table 10. Macros and Files in Different RZK Configurations

Source Configuration File	Macro	Library File	RZK Configuration
RZK_Conf.c uart_conf.c (if UART settings are different from default settings) eZ80eval.c (for printf through UART0 port) eZ80Hw_Conf_ZDS.c get_heap.s	RAM_MAP	—	For use with RAM project workspaces*
	EVB_F91_MINI	—	Configured to work with eZ80F91 Mini Module*
	_EZ80XXX	RZKeZ80XXX.lib BSPeZ80XXX.lib	Library files across platforms
	RZKDBG, RZKPI	RZKDebugPI.lib	DEBUG-PI (Debug with Priority Inheritance support)
	RZKPI	RZKNDebugPI.lib	NO_DEBUG-PI (No Debug with Priority Inheritance support)
		RZKNDebugNPI.lib	NO_DEBUG-NO_PI (No Debug with No Priority Inheritance support)
	RZKDBG	RZKDebugNPI.lib	DEBUG-NO_PI (Debug with No Priority Inheritance support)

Note: *Used in addition to the RZK configuration macros, libraries or configuration files.

Table 10. Macros and Files in Different RZK Configurations (Continued)

Source Configuration File	Macro	Library File	RZK Configuration
RZK_Conf.c eZ80Hw_Conf_ZDS.c get_heap.c uart_conf.c (if UART settings were different from default settings) eZ80eval.c (for printfs through UART0 port)	RZKDBG, RZKPI	RZKDebugPI.lib RZKeZ80XXX.lib BSPeZ80XXX.lib	DEBUG-PI (Debug with Priority Inheritance support)
	RZKPI	RZKNDebugPI.lib RZKeZ80XXX.lib BSPeZ80XXX.lib	NO_DEBUG-PI (No Debug with Priority Inheritance support)
		RZKNDebugNPI.lib RZKeZ80XXX.lib BSPeZ80XXX.lib	NO_DEBUG-NO_PI (No Debug with No Priority Inheritance support)
	RZKDBG	RZKDebugNPI.lib RZKeZ80XXX.lib BSPeZ80XXX.lib	DEBUG-NO_PI (Debug with No Priority Inheritance support)
ZFS_Conf.c	RZKFS	RZKFS.lib	Zilog File System.*
		NOFS.lib	Zilog File System stub file; used when file system footprint is not required but some components in the system call Zilog File System APIs.*
eZ80Hw_Conf_ZDS.c	Modify `RZK_HW_Init()` for optimized and custom hardware initialization.	—	Optimized and custom hardware initialization; provides a `RZK_HW_Init()` function that is called from within `RZK_KernelInit()`, which initializes the hardware required by RZK. You can initialize custom devices with this function.
uart_conf.c	—	—	Different UART configurations other than the default configuration.*
rtc_conf.c	—	—	Different RTC configuration other than the default configuration.*

Note: *Used in addition to the RZK configuration macros, libraries or configuration files.

Table 10. Macros and Files in Different RZK Configurations (Continued)

Source Configuration File	Macro	Library File	RZK Configuration
emac_conf.c	—	—	EMAC configuration.*
	—	NOEMAC.obj	EMAC driver stub file: used when an EMAC footprint is not required. However, some components in the system will call EMAC driver APIs.*
wlan_conf.c	ZDOT_WLAN	BSPZDOTS.lib	To configure WLAN parameters other than default.
I2C_Conf.c	—	—	Different I^2C configuration other than default configuration.*
Spi_Conf.c	—	—	Different SPI configuration other than default configuration.*
DataPer_Conf.c	g_data_per_cfg (variable)	—	Data persistence configuration.*

Note: *Used in addition to the RZK configuration macros, libraries or configuration files.

Each of the source configuration files listed in Table 10 must be included in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

All macros must be defined in the Preprocessor Definitions field of each requisite source configuration file. If a library file is listed, it must be included in the project workspace.

Data Persistence Configuration

Data persistence is defined as the ability of memory to retain data when power is removed from a system. The type of memory to be used for this purpose is labeled *nonvolatile memory*. RZK provides data persistence functions with user-defined values. These functions include a number of required variables, including variables for the EMAC address, IP address, IP gateway, netmask and DHCP status (enabled or disabled) to effectively set and allow the restoration of settings made prior to power-down or reboot.

When considering environments where power failures or reboots can occur, RZK data persistence functions, such as the `SetDataPersistence` and `GetDataPersistence` APIs, are advantageous. The `DATA_PER_CFG_t` structure defines the nonvolatile memory device driver routines that are called by these two APIs. However, you must modify the structure `DATA_PER_CFG_t` present in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Conf
```

The DATA_PER_CFG_t structure is defined below.

```
typedef struct
{
  void *      p_start_addr ;
  /* start address of the location where data must be stored */
  UINT32      ul_size ;
  /* size of the erasable block */
  DEV_INIT          pfn_dev_init ;
  DEV_READ          pfn_dev_read ;
  DEV_WRITE         pfn_dev_write ;
  DEV_ERASE         pfn_dev_erase ;
  DEV_CLOSE         pfn_dev_close ;

} DATA_PER_CFG_t, *PDATA_PER_CFG_t ;
```

The location and type of nonvolatile memory in which values are stored can be configured whether this memory is present in internal or external Flash. Table 11 specifies the location in which these values are stored for the different platforms and configurations in the eZ80Acclaim! product line.

Table 11. Data Storage Location of Values Related to Data Persistence

Platform	eZ80F91	eZ80F92	eZ80F93	eZ80L92
RAM	External Flash at address 0x104000	External Flash at address 0x104000	External Flash at address 0x104000	External Flash at address 0x104000
FLASH	Internal Flash (information page) at address 0x0	Internal Flash (information page) at address 0x0	Internal Flash (information page) at address 0x0	External Flash at address 0x02000
COPY TO RAM	Internal Flash (information page) at address 0x0	Internal Flash (information page) at address 0x0	Internal Flash (information page) at address 0x0	External Flash at address 0x02000

In addition to any standard linker directives, RZK provides a number of additional linker directives settings for eZ80Acclaim! target processors, including settings that establish data persistence within memory. These settings can be made in ZDS II by navigating to Input Additional Directives within the Linker. Observe the following procedure to establish data persistence settings for project workspaces.

1. In the **Compiler** tab, under **Code Generation** for all targets, deselect the **One segment per module** checkbox.

2. In the **Linker** tab, navigate to **Input** ⟶ **Additional Directives** (for all targets except the eZ80190 device) and add the following linker directives:

```
CHANGE per_api_TEXT is RAM
CHANGE _driver_TEXT is RAM
CHANGE per_api_TEXT is DATA
CHANGE _driver_TEXT is DATA
```

3. If the chosen configuration is a `COPY_TO_RAM` configuration, add the following statement in the linker settings within the **Linker** ⟶ **Input** ⟶ **Additional Directives** navigation:

```
CHANGE TEXT is RAM
```

4. For the eZ80L92 target, if the chosen configuration is `FLASH` or `COPY_TO_RAM`, add the following address range in the linker settings within the panel within the **Linker** ⟶ **Address Spaces** ⟶ **RAM** navigation:

```
0–3FFF,6000–FFFFF
```

Table 12 describes the configurable parameters that are defined according to the kinds of libraries required:

Table 12. Default Debug Priority Inheritance Values for the RZK_Conf.c File

RZK Macro	Default Value	Description
MAX_THREADSH	30 (minimum value = 2)	Specifies the maximum threads that RZK can create at a time. This macro's value = 2 + the maximum number of system threads required to be created. One thread control block (TCB) is used for the kernel idle thread and one for the RZK timer.
RZK_TIME_SLICEH	20	Specifies the default time slice for round-robin mode at a time. This macro must contain a value greater than 0 (zero).
RZK_SYSTIMERSTACK_SIZEH	2048	This macro must be defined to provide stack size for RZK's system thread.
MAX_MESSAGEQSH	28	Specifies the maximum number of message queues that can be created at a time. If the system does not require message queues, then this macro is set to 0 (zero).
MAX_EVENTGROUPSH	20	Specifies the maximum number of event groups that can be created at a time. If the system does not require event groups, then this macro is set to 0 (zero).

Note: *Refer to the Zilog Real-Time Kernel Reference Manual (RM0006) and its appendix about Interrupt Handling to configure the RZK timer.

Table 12. Default Debug Priority Inheritance Values for the RZK_Conf.c File (Continued)

RZK Macro	Default Value	Description
MAX_SEMAPHORESH	20	Specifies the maximum number of semaphores that can be created at a time. If the system does not require semaphores, then this macro is set to 0 (zero).
MAX_TIMERSH	20	Specifies the maximum number of software timers that can be created at a time. If the system does not require software timers, then this macro is set to 0 (zero).
MAX_PARTITIONSH	20	Specifies the maximum number of memory partitions that can be created at a time. If the system does not require partitions, then this macro is set to 0 (zero).
MAX_REGIONSH	20	Specifies the maximum number of regions that can be created at a time. If the system does not require regions, then this macro is set to 0 (zero).
MAX_REGIONS_TABH	250	Specifies the maximum number of allocations that can be made within the regions at any given time.
RZK_MAX_DCBH	25	Specifies the maximum number of entries that can be made in the device control block.
RZK_DEVICE_DRIVERH	1	Defined if RZK drivers are used.
RZK_STACK_SIZEH	2048	Specifies the size of stack required for kernel's idle thread.
RZK_SYSTIMERSTACK_SIZEH	2048	Specifies the size of stack required for the RZK timer thread.
RZK_SYSTICKS_INSECH	100	Specifies the number of ticks per second for the system.*
RZK_DEVTICKS_PERSYSTICKH	1	Specifies the number of device ticks per system tick.*
SYSTEM_CLOCKH	48000000L	Specifies the processor's clock speed.*
HWTIMER_TO_USE	0	Specifies the hardware timer to be used as the source for the RZK system; valid values range from 0–3.

Note: *Refer to the Zilog Real-Time Kernel Reference Manual (RM0006) and its appendix about Interrupt Handling to configure the RZK timer.

Table 12. Default Debug Priority Inheritance Values for the RZK_Conf.c File (Continued)

RZK Macro	Default Value	Description
RZK_MAX_CWD_PATH_LEN	128	Specifies the maximum length of the current working directory path in bytes; valid only if RZKFS is defined.
RZK_CWD_PATH	"EXTF:/"	Specifies the current working directory for the threads that are created in the `main()` function; valid only if `RZKFS` is defined.
Note: *Refer to the Zilog Real-Time Kernel Reference Manual (RM0006) and its appendix about Interrupt Handling to configure the RZK timer.		

Table 13 lists the four different configurations of RZK libraries.

Table 13. Four Configurations of RZK Libraries

RZK Library	Description
RZKDebugPI	Debug and Priority Inheritance protocol enabled.
RZKDebugNPI	Debug enabled but Priority Inheritance protocol disabled.
RZKNDebugNPI	Debug and Priority Inheritance protocol disabled.
RZKNDebugPI	Debug disabled but Priority Inheritance protocol enabled.

The real-time application can therefore be used first with the RZK `DebugPI` configuration to remove any errors during testing of the system. If the application is error-free, it is preferable to use the RZK `NDebugNPI` or RZK `NDebugPI` configuration, based on your application requirements. By using RZK `NDebugNPI`, you can save memory.

Appendix A. Frequently Asked Questions

This chapter presents answers to many commonly-asked questions about the Zilog Real-Time Kernel.

General Questions about RZK

Q: What is RZK?

A: RZK is Zilog's real-time, priority-based, preemptive and multitasking kernel designed for time-critical embedded applications. It can currently be used with Zilog's eZ80 family of microprocessors, for which it is specifically optimized. The response time of RZK is quick when compared with other commercially-available operating systems.

Q: Why should I use RZK?

A: You can use RZK as your kernel, especially if you already have eZ80 processors. RZK is a reliable kernel and has many protocol stacks implemented on it. Next, the modular design concepts of RZK allows you to tailor it to meet your product requirements. Moreover, RZK provides all of the standard benefits of an RTOS with very low memory and processing overhead. Therefore, it is quite affordable to use with 8-bit processors as they relate to product and development costs and resource overhead for most present-day implementations.

The modular structure of RZK makes it convenient to tailor it to customer needs. How is RZK released and what are the benefits, if any, of the way it is packaged? What are the benefits of using the RZK libraries?

The modular design of RZK breaks up the software into a number of objects. RZK is released as set of libraries, with each RZK object forming a library. When you develop your application using RZK, you include all of the RZK libraries; however, the APIs that you use determines which libraries are linked to your application. This mechanism reduces your memory footprint. Moreover, the application can be accommodated where memory requirements are lower.

Q: What are the features of RZK?

A: RZK provides extensive coverage of easy-to-use APIs. It also provides low interrupt latency, allows nested interrupts and supports interrupt threads. Thread switching time is kept to a minimum by exploiting the Zilog microprocessor and microcontroller architecture.

Q: What are the sizes of the RZK components?

A: The sizes of RZK components are very small compared to other RTOS'.

Q: How many target platforms are supported by RZK?

A: Currently, RZK supports the eZ80Acclaim! series of microcontrollers, which includes the eZ80F91, eZ80F92 and eZ80F93 devices and the eZ80 product line, which includes the eZ80L92 device.

RZK also supports all variants of Zilog's eZ80 family of microprocessors and microcontrollers, with the exception of the eZ80190 processor.

Q: How much does RZK cost?

A: Feel free to contact us via our Customer Service page to obtain RZK pricing details. The price depends upon packaging options.

Q: What kind of applications can I build with RZK?

A: You can use RZK to build a variety of applications, especially those that require extreme levels of reliability and serviceability. RZK can be used in all types of embedded applications because of its quick response to external events.

Some of the applications that can use RZK are wireless protocols, such as IrDA and the telecommunications protocols, such as ISDN and TCP/IP.

In-Depth Questions About Using RZK

Q: Can I use RZK for designing real-time systems?

A: Yes. RZK is designed for real-time systems that require less interrupt latency and fast context switching time with less memory requirements. Also, timings for each of the APIs are available for designing real-time applications.

Q: What is the development process for RZK?

A: The ZDS II IDE that works with the eZ80 processor is used to create your project. Your project uses the RZK libraries and generates a `.hex` image or a `.lod` file according to your specific requirement. The `.hex` image is ready to be burned to EPROM or Flash memory; the `.lod` file is ready to be downloaded to RAM. When you decide to debug the program, it is downloaded to the target platform using the ZPAK II tools or USB smart cables and the ZDS II IDE. This debugger operates similarly to its operation for local debugging. RZK has built in debug features when in debug mode. After the final application is debugged and ready to be burned to ROM, the debug option in RZK can be undefined.

Q: What's the learning curve? How long will it take me to become productive with RZK?

A: You require only a few minutes. If you know how to use an IDE (such as Microsoft Visual Studio) and are familiar with real-time operating system concepts and components, all you have got to do is open the example projects shipped with RZK, compile and link them. Then, download to the target platform to see the application running. Refer to the Zilog Real-Time Kernel Quick Start Guide (QS0048), which is available free for download on zilog.com and also provided in the documentation set for RZK in the following path:

```
<ZDSII installed directory>\Program Files\Zilog\
ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Docs
```

Q: I want to develop ROMable and self-booting programs for an embedded system using an eZ80 device and RZK. What software do I need?

A: An IDE tool such as ZDS II, RZK and a Flash Loader application (ZDS II features an integrated Flash Loader utility) are required to boot from Flash and not from RAM. Generate a `.hex` image of the application (you can use ready-made projects that are included with the RZK release) and send it to the Flash Loader. The Flash Loader takes care of writing your code to the Flash memory.

Q: Are the stacks for each thread protected from overrun or underrun?

A: No, RZK does not provide stack protection. It is your responsibility to protect the stack or allocate sufficient stack for the functioning of the threads.

Q: When I create a thread (using the API `RZKCreateThread()`) or a timer (using the API `RZKCreateTimer()`) or any other object, it always returns a NULL with an error code of 4, which is `RZKERR_CB_UNAVAILABLE`. What is the cause for this problem and how can I get rid of it?

A: This type of problem occurs when it is not possible to allocate a control block for the particular object you are trying to create. This issue occurs when you try to create more objects than the maximum specified objects allowed in the `RZK_Conf.c` file. Increase the appropriate object's count to create the object that you require. For a more detailed explanation, see the RZK Configuration chapter on page 14.

Q: When I post a message to a higher-priority thread from an ISR, the context switch does not occur immediately. Why is that?

A: RZK follows the priorities below, in descending order, for executing components:

`ISR Interrupt Thread User Priority Threads`

Execution is always transferred to the ISR, because it holds the highest priority. Thus, context switching does not occur even when messages are posted to higher-priority threads from an ISR that runs from the context of a lower-priority thread. Context switching occurs only when all of the current pending interrupts are serviced.

Q: What happens when I specify zero wait time in calls such as `RZKAcquireSemaphore` or `RZKReceiveFromQueue`?

A: When zero wait time is specified in these calls, they become non-blocking calls and return immediately.

Q: Is `RZKCreateSemaphore` with zero initial count value supported by RZK?

A: Yes, this feature is supported by RZK.

Q: When an RZK thread is deleted does it also release all of the resources created during its execution?

A: No. RZK does not release any of the resources created when that thread is deleted. But you can achieve the same functionality by deleting the unwanted resources in the cleanup functions, which will be executed during the thread deletion.

Q: Does RZK support the Priority Inheritance protocol?

A: Yes. RZK supports Priority Inheritance protocol. For more information about how to choose the option, refer to the Zilog Real-Time Kernel Reference Manual (RM0006), which is available free for download from zilog.com and also located in the following path:

```
<ZDSII installed directory>\Program
Files\Zilog\ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Docs
```

Q: Does the Priority Inheritance protocol take care of both unbounded and bounded priority inversion?

A: Yes, it does. RZK supports priority inheritance for both unbounded and bounded priority inversion.

Q: How much overhead is consumed by the Priority Inheritance protocol?

A: The Priority Inheritance protocol impacts the system only minimally. Typically, system overhead involves changing a thread's priority and scheduling.

Q: Can I write an application to be run from Flash and debug it through ZDS II?

A: Yes. You can write an application that runs from Flash. Observe the following procedure to write the application:

1. Create a Flash-based project and add the application and required files to it.
2. Modify the settings as specified in this manual for Flash memory and generate a `.hex` file to be downloaded to Flash memory by using the Flash Loader integrated within ZDS II.
3. Select **Projects** → **Settings** → **Linker** and define the **Category** as **Output**. Select **IEEE 695** as the **executable format** and click **OK.** This setting is required because the IEEE 695 format contains the debug information.
4. Build the application. Make sure that the **Chip Select** settings are mirrored as described in this manual.
5. Connect to the target by selecting **Build** → **Debug** → **Connect To Target**.
6. Reset the eZ80 Development Platform by selecting **Build** → **Debug** → **Reset**. Now you can debug your programs as usual.

Q: How do I work with interrupts in RZK?

A: Observe the following procedure for interrupt handling:

1. Install an interrupt handler for the appropriate interrupt using the `RZKInstall-InterruptHandler()` routine and provide the interrupt handle and the interrupt number.
2. Write a program such that the interrupt prolog first saves all registers and calls the C ISR routine (if present) and then restores all registers in the order it is saved. (You can accomplish this task by *pushing* the registers onto the stack and subsequently *popping* them from the stack in the order they were pushed.) For more details, refer to the Zilog Developer Studio II – eZ80Acclaim! User Manual (UM0144).
3. If RZK APIs are called in the C ISR routine then you are required to call `RZKISR-Prolog()` and `RZKISREpilog()` functions before and after the ISR routine respectively.

Q: I have two devices, TIMER1 and UART1. I don't want to miss the Interrupts of either device and want to handle both of them. How can I do it using RZK? Is it possible to get nested interrupts in RZK? If so, what precautions do I need to take so that nothing goes wrong?

A: First, install the interrupt handler for both the devices and then initialize the devices. RZK is designed to support nested interrupts for different devices. To include nested interrupts, make sure you have enabled the interrupts after the `RZKIsrProlog()` API is called in the ISR. Take the precaution to include enough time to service the interrupt and to execute the thread code. If an interrupt routine needs to be nested, it is usually because there are bursts of interrupts that have to be handled. A large buffer for the incoming data and a larger stack size is also required.

Q: I regularly get an `Uninitialized Interrupt` message. What does this message mean and why do I get it?

A: The message `Uninitialized Interrupt` specifies that one of your interrupt handlers has not been initialized using `RZKInstallInterruptHandler()` API. By default, RZK installs a handler that prints the message `Uninitialized Interrupt` and executes a `HALT` instruction.

Q: Why does the control not go to the highest priority thread, when I create it within the `main()` API?

A: The `main()` function executes at the highest priority. It acts as a launch pad for the application. The control is transferred to your application only after the `main()` function is executed completely. Although, you have assigned the highest priority to the thread that you have created, make sure to select the `RZK_THREAD_AUTOSTART` option for the operation mode of the thread. Otherwise, you must explicitly resume the thread by calling `RZKResumeThread()` API.

Q: Do the binary semaphores, used for mutual exclusion, allow multiple locking by a thread?

A: No. Binary semaphores are for mutual exclusion only and multiple locking by the same thread is not allowed in RZK.

Q: Does RZK support callback functions, such as callouts in pSOS and hookups in VxWorks?

A: Yes. However, the support is limited to the cleanup functions that are called when a thread is deleted.

Q: Can I change the blocking queue type from FIFO to priority or vice versa for messagequeues, semaphores or other RZK objects?

A: No. You cannot change the blocking queue type of a created object.

Q: What is the difference between the macros `RZK_SYSTICKS_INSECH` and `RZK_DEVTICKS_PERSYSTICKH`? The granularity of the system is dependent on which of these macros?

A: The `RZK_SYSTICKS_INSECH` represents system ticks per second, whereas `RZK_DEVTICKS_PERSYSTICKH` represents the number of device ticks that are present in one system tick. The granularity of the system depends upon the `RZK_SYSTICKS_INSECH` value.

Q: What are Debug Priority Inheritance, Debug Non Priority Inheritance, Non Debug Priority Inheritance and Non DebugNon Priority Inheritance? Why are they required?

A: These four sets of RZK libraries (four for each target platform) are categorized according to the implementation of priority inheritance protocol. Debug Priority Inheritance indicates that debugging and Priority Inheritance are enabled. The *N* in front of either Debug or Priority Inheritance indicates that they are disabled.

For example, The real-time application may therefore be used first with the RZK `DebugPI` configuration to remove any errors during testing of the system. If the application is error-free, it is preferable to use the RZK `NDebugNPI` or RZK `NDebugPI` configuration, based on the application requirements.

Q: I changed the RZK configuration from Debug Priority Inheritance to Non Debug Non Priority Inheritance by including the Non Debug Non Priority Inheritance libraries in the project and removing the Debug Priority Inheritance libraries from the project. Do I need to change anything else?

A: No.

Q: What is the include order for RZK-related header files?

A: The include order for RZK-related header files is:

```
#include "ZTypes.h"
#include "ZSysgen.h"
"other required RZK header files"
```

`ZSysgen.h` and `ZTypes.h` can be in any order, but all other RZK header files must be included after these two header files because all of the RZK header files use the macros and definitions of the `ZSysgen.h` and `ZTypes.h` files.

For details, refer to the Zilog Real-Time Kernel Reference Manual (RM0006), which is available free for download from zilog.com and also located in the following path:

```
<ZDSII installed directory>\Program
Files\Zilog\ZDSII_eZ80Acclaim!_A.B.C\ZTP\ZTPX.Y.Z_Lib\RZK\Docs
```

Q: I have a different memory map on my custom hardware as compared to the development board. What changes am I required to make to modify the RZK demo project to suit my custom hardware in the ZDSII environment?

A: RZK uses the bootup files provided by ZDS II. This means that there are no separate hardware configuration files required to configure the chip selects and the memory map. So if your custom hardware features a different memory map as compared to the one provided in the demo programs for the development platform, then the following changes must suit your custom hardware:

Modify the ROM and RAM address space to suit your memory map under **Project** ⟶ **Settings** ⟶ **Linker** ⟶ **Address Space** ⟶ **ROM** and **RAM** address spaces.

Incorporate the corresponding changes to the **Chip Selects**, the **Program Counter** and the **Stack Pointer** under **Project** ⟶ **Settings** ⟶ **Debugger** ⟶ **Setup**, **Program Counter**, **Stack Pointer Long** and the **Chip Select** Registers. Take special care while enabling/disabling the internal Flash and internal SRAM.

The only change that may be required in the configuration files is for the wait states of the internal flash. The following code string can be changed, depending on your requirements:

```
FLASH_CTRL = (FLASH_CTRL & ~FLASH_CTRL_7_WS) | FLASH_CTRL_3_WS;
```

This code string is available in the `RZK_HW_Init()` function, which is located in the `eZ80Hw_Conf_ZDS.c` file.

Q: This FAQ contains many questions and answers, but I still have questions for which I find no answers in this document. How can I get answers to my questions?

A: Zilog has a reputation for excellent customer service. Feel free to contact us via our Customer Service page.

Customer Support

To share comments, get your technical questions answered or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.