

---

菊风协议构架

# Zero Operating System

发表日期: 2006-5-22

访问 <http://www.juphoon.com> 了解更多产品信息

---

## 菊风 ZOS 服务规格手册

菊风系统软件有限公司

<http://www.juphoon.com>

电话: +86-574-87287820

传真: +86-574-87304379

Copyright © 2007, Juphoon System Software Corporation.

版权所有

## 目录

<b>1. 产品概述</b> .....	<b>6</b>
1.1 用途.....	6
1.2 本手册的阅读对象.....	6
1.3 内容简介.....	6
1.4 缩略词和缩写词.....	6
<b>2. 概念介绍</b> .....	<b>6</b>
2.1 ZOS上下文环境.....	6
2.2 ZOS模块总览.....	7
2.3 基本数据类型.....	8
2.4 平台类型.....	8
2.5 基本概念.....	9
2.5.1 模块ID.....	9
2.5.2 实例ID.....	9
2.5.3 任务ID.....	9
2.5.4 处理器ID.....	9
<b>3. 模块与任务</b> .....	<b>10</b>
3.1 任务概要.....	10
3.2 激活方式.....	11
3.3 任务初始化.....	11
3.4 任务销毁.....	12
3.5 任务消息处理.....	12
<b>4. 层间通信</b> .....	<b>12</b>
4.1 层间通信概要.....	12
4.2 模块服务点.....	13
4.3 消息发送.....	14
4.4 消息处理.....	15
<b>5. 计时器管理</b> .....	<b>15</b>
5.1 计时器概要.....	15
5.2 平台计时器.....	16
5.3 计时器接口.....	17
<b>6. 内存管理</b> .....	<b>18</b>
6.1 内存管理概要.....	18
6.2 BUCKET POOL.....	19
6.3 BUCKET POOL 接口.....	21
6.4 BUCKET POOL 示例.....	21
<b>7. 缓冲区管理</b> .....	<b>23</b>
7.1 缓冲区概要.....	23

7.2	数据缓冲区.....	24
7.3	数据缓冲区接口.....	25
7.4	数据缓冲区示例.....	26
<b>8.</b>	<b>配置说明.....</b>	<b>27</b>
8.1	库文件配置.....	27
8.1.1	综合配置.....	28
8.1.2	内存池配置.....	29
8.1.3	消息缓冲池配置.....	30
8.1.4	数据缓冲池配置.....	30
8.1.5	日志配置.....	31
8.1.6	模块配置.....	32
8.1.7	任务配置.....	33
8.1.8	计时器配置.....	33
<b>9.</b>	<b>创建应用程序.....</b>	<b>34</b>
9.1	模块ID和任务ID.....	34
9.2	头文件.....	36
9.3	对ZOS平台进行配置和初始化.....	36
9.4	设计应用程序任务.....	37
9.5	对FLAGS进行编译和链接.....	40
<b>10.</b>	<b>附录.....</b>	<b>41</b>
10.1	ZOS 编译选项.....	41
10.2	ZOS 基本宏常量.....	42
10.3	ZOS 基本宏操作.....	44
10.4	ZOS 基本函数指针.....	47
10.5	ZOS 错误码.....	49
10.6	ZOS 日志操作宏.....	54
10.7	ZOS 单链表操作宏.....	54
10.8	ZOS 双链表操作宏.....	58

## 表目录

表 1-1	缩略词和缩写词.....	6
表 2-1	ZOS 模块集.....	8
表 2-2	基本数据类型.....	8
表 2-3	平台类型.....	9
表 8-1	综合配置参数.....	29
表 8-2	消息缓冲池配置参数.....	30
表 8-3	数据缓冲池配置参数.....	31
表 8-4	日志配置参数.....	32
表 8-5	模块配置参数.....	33

---

表 8-6 任务配置参数.....	33
表 8-7 计时器配置参数.....	34
表 9-1 Compile & Link Flags.....	41
表 10-1 ZOS 编译选项表 .....	41

## 图目录

图 2-1 ZOS 的上下文 .....	7
图 2-2 ZOS模块集 .....	7
图 3-1 多个 ZOS 任务存在于一个系统任务中 .....	10
图 3-2 每个任务存在于一个独立的操作系统任务中 .....	10
图 4-1 ZOS 任务间消息通信 .....	13
图 5-1 ZOS 计时器任务驱动 .....	17
图 6-1 ZOS 内存管理机制 .....	19
图 7-1 数据缓冲区(Dbuf)结构 .....	24

## 1. 产品概述

ZOS(Zero Operating System)是菊风公司的操作系统服务平台，提供了支持多种操作系统环境下的统一抽象接口操作，使得软件产品能够独立于特定的处理机、编译器和操作系统等应用环境。此外，ZOS增强了系统服务功能，提供任务管理、消息队列、计时器管理、内存管理、数据缓冲区管理、日志管理，抽象了很多协议相关的服务功能接口，如ABNF、ASN.1编解码库。

### 1.1 用途

本手册的用途是帮助程序开发人员正确使用ZOS系统平台来开发他们自己的软件产品。

### 1.2 本手册的阅读对象

本手册的读者应具备操作系统基础知识。

### 1.3 内容简介

本手册详细介绍了ZOS服务规格及其使用方法；讲述了如何创建基于ZOS平台的应用程序。

### 1.4 缩略词和缩写词

下表罗列了在本手册中使用的缩略词和缩写词。

缩略/缩写词	完整描述
ANSI	American National Standards Institute
ZOS	Zero Operation System
ABNF	Augmented BNF
DBUF	DBUF
QTIMER	Queue Timer
RTIMER	Ring Timer

表 1-1 缩略词和缩写词

## 2. 概念介绍

### 2.1 ZOS上下文环境

ZOS(Zero Operating System)是菊风公司的的操作系统服务平台，提供了支持多种操作系统环境下的统一抽象接口操作，使得软件产品能够独立于特定的处理机、编译器和操作系统等应用环境。此外，ZOS增强了系统服务功能，提供任务管理、消息队列、计时器管理、内存管理、数据缓冲区管理、日志管理，抽象了很多协议相关的服务功能接口，如ABNF、ASN.1编解码库。

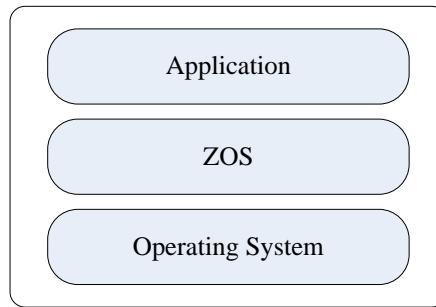


图 2-1 ZOS 的上下文

实际的Operating System可能是Windows、Linux、Solaris、VxWorks、pSOS、threadX等。

## 2.2 ZOS模块总览

下图列出了ZOS的各个模块：

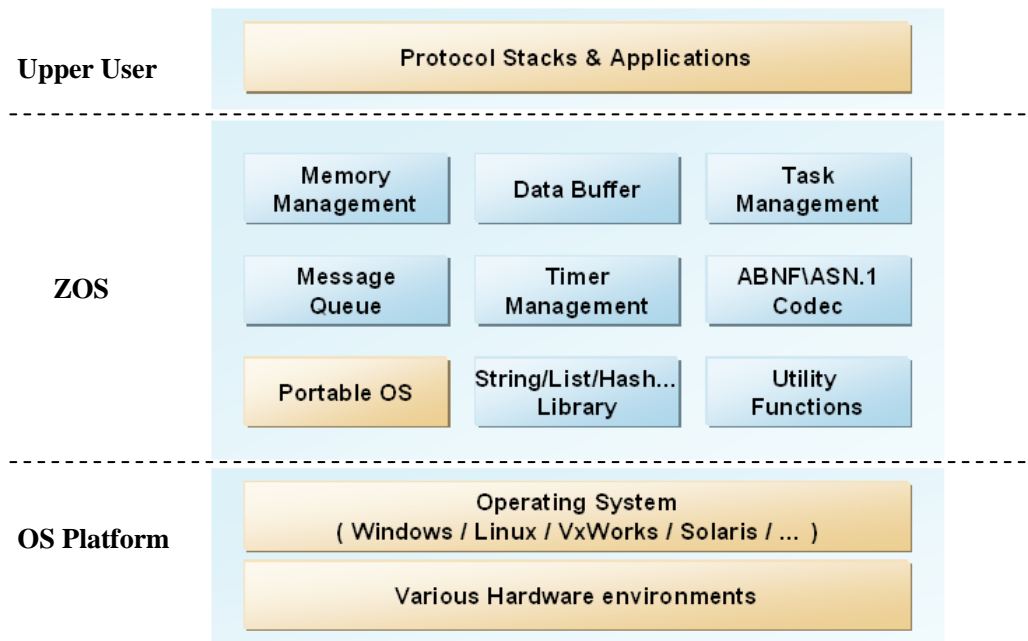


图 2-2 ZOS模块集

图中的各个模块的职责如下表描述：

模块名称	模块需求描述
内存管理	Memory Management模块，实现内存池资源管理，避免在上层应用运行过程中频繁的对于内存资源的申请和释放。 并提供面向对象的管理方式，用户可以定义若干个独立的内存管理对象。
数据缓冲区	Data Buffer模块，目的是实现尽量少的内存复制操作，同时又能够在协议开发过程中对于内存资源的灵活管理，通过提供规范的内存使用规则，避免内存泄漏、内存指针跑飞等问题。 由于协议开发需要大量用到缓冲区，缓冲区的设计必须提供高效的申请和释放机制。
任务管理	Task Management模块，提供统一的基于消息驱动事件触发的任务

	管理机制。
消息队列	Message Queue模块，提供消息队列机制。
计时器	Timer Management模块，面向不同应用需求，提供不同的精度、尺度的计时器机制。同时要提供在并发几十万个计时器的情况下的具有可用性的机制（包括计时器的精度、系统占用的资源等指标）。
ABNF\ASN.1编解码	ABNF\ASN.1 Codec模块，是相对独立的协议报文处理模块。要求提供高性能的编解码机制。 目前只实现ABNF Codec，ASN.1还未能提供。
基本算法模块	String/List/Hash等基本数据结构和算法模块，提供精巧高效的算法实现，在使用C语言的开发环境下，提供一系列丰富的算法实现。
实用工具	Utility Functions模块，提供一系列丰富的实用工具，包括：dump（调试堆）、log（日志）、gab（垃圾回收）、Fsm Map（状态机映射）等等。
OS可移植模块	Portable OS模块，将OS强依赖的模块进行集中封装，包括互斥信号量、Socket、中断、任务创建、Time

表 2-1 ZOS 模块集

## 2.3 基本数据类型

ZOS 平台提供了一些基本数据类型。表2-1列举了这些数据类型。

类型名称	类型
ZDOUBLE	double
ZFLOAT	float
ZLONG	long
ZINT	int
ZSHORT	short
ZCHAR	char
ZULONG	unsigned long
ZUINT	unsigned int
ZSIZE_T	unsigned int
ZUSHORT	unsigned short
ZUCHAR	unsigned char
ZBOOL	boolean
ZVOID	void

表 2-2 基本数据类型

## 2.4 平台类型

表2-2 列举了基本平台类型。

类型名称	类型
ZMUTEX	Mutex
ZSEM	Semaphore
ZTIME_T	Time
ZFUNCPTR	Function Pointer
ZVOIDFUNCPTR	Void Function Pointer
ZLOGID	Log ID
ZMODID	Module ID
ZINSTID	Instance ID
ZTASKID	Task ID
ZTIMERID	Timer ID
ZEVNTID	Event ID
ZPOOLID	Pool ID

表 2-3 平台类型

## 2.5 基本概念

本节将讲述四种普通ID结构。

### 2.5.1 模块ID

模块ID是一个十六比特的无符号整数。模块可以分为两大类，一类是ZOS基本模块，另一类是用户定义的模块。

### 2.5.2 实例ID

一个模块可能被多个实例共享，所以某个模块的一个实例也被赋予一个ID号，便于区分。和模块ID一样，实例ID也是一个十六比特的无符号整数。

### 2.5.3 任务ID

任务ID号是用来区分不同的任务。任务ID是一个三十二比特的无符号整数。其中左边的十六位存放的是一个模块ID,右边的十六位存放的是一个实例ID。

### 2.5.4 处理器ID

一个操作系统可能受多个处理器支持。每个处理器有一个处理器ID，当它们相互通讯时，操作系统可以通过处理器ID识别不同的处理器。目前尚不支持分布式操作系统。

### 3. 模块与任务

#### 3.1 任务概要

模块实例（任务）必须被作为操作系统中的一个可执行调度的任务放在底层操作系统中，以便模块中的实例（即ZOS任务）能够运行。而这种附属方式，可以有两种：多个 ZOS 任务存在一个操作系统任务中；每个任务存在于一个独立的操作系统任务中。

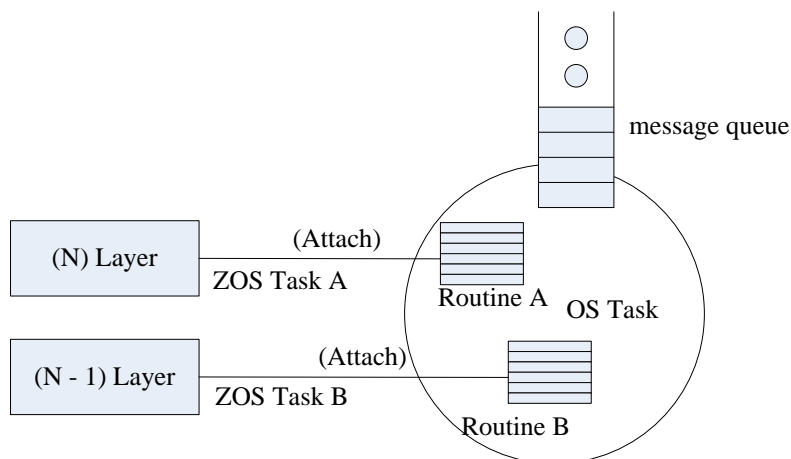


图 3-1 多个 ZOS 任务存在于一个系统任务中

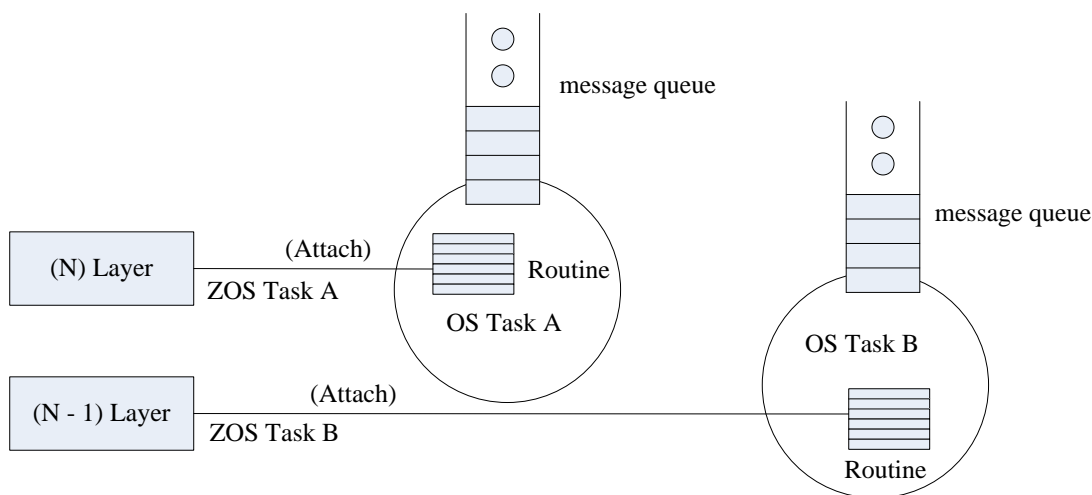


图 3-2 每个任务存在于一个独立的操作系统任务中

ZOS 任务一般需要完成以下基本功能：

- 任务初始化
  - 层中模块实例的初始化
- 任务消息处理
  - 处理来自其他模块实例的消息
- 任务计时器处理
  - 处理来自计时器任务的超时通知事件

### 3.2 激活方式

ZOS 任务有两种激活方式:

- 直接使用 ZOS 的 **Zos\_TaskSpawn** 来启用任务调度入口点, 它会调用操作系统的任务创建函数(如 Linux 下的线程创建函数 **pthread\_create**) 使得 ZOS 任务成为系统的一个调度单元。

```
/* task entry function */
typedef ZULONG (*PFN_ZTASKENTRY)(ZVOID *);

/* zos spawn task */
ZINT Zos_TaskSpawn(ZTASKID zTaskId, ZCHAR *pcName, ZINT iPriority,
                  ZULONG dwStackSize, ZULONG dwQueueSize, ZULONG dwTimerNum,
                  PFN_ZTASKENTRY pfnEntry, ZVOID *pParm);
```

- 使用模块注册功能来注册任务, 而任务的初始化函数与消息处理函数由模块管理器负责激活

```
/* zos instance initialize function */
typedef ZINT (*PFN_ZINSTINIT)(ZINSTID zInstId);

/* zos instance destroy function */
typedef ZVOID (*PFN_ZINSTDESTROY)(ZINSTID zInstId);

/* zos instance message process function */
typedef ZINT (*PFN_ZINSTMSGPROC)(ZVOID *pMsg);

/* zos register one module */
#define ZOS_MOD_REG(_taskid, _name, _queuesize, _timernum, \
                  _pfninit, _pfnDestroy) \
    Zos_ModReg(_taskid, _name, _queuesize, _timernum, \
              (PFN_ZINSTINIT)_pfninit, (PFN_ZINSTDESTROY)_pfnDestroy)

/* zos deregister one module */
#define ZOS_MOD_DEREG(_taskid) \
    Zos_ModDereg(_taskid)

/* zos start one module */
#define ZOS_MOD_START(_taskid, _priority, _stacksize, _pfnmsgproc) \
    Zos_ModStart(_taskid, _priority, _stacksize, (PFN_ZINSTMSGPROC)_pfnmsgproc)
```

### 3.3 任务初始化

**PFN\_ZINSTINIT** 是任务初始化函数类型, 在模块管理器初始化已注册任务的时候, 它会执行任务初始化函数, 同时传入实例ID, 使得模块能够初始化不同的实例。

任务初始化函数命名的格式是: **ZINT Xxx\_TaskInit(ZINSTID zInstId)** 如:

SIP 协议栈任务初始化函数名为:

```
/* sip task init */  
ZINT Sip_TaskInit(ZINSTID zInstId)
```

### 3.4 任务销毁

**PFN\_ZINSTDESTROY** 是任务销毁函数类型，它表示在任务结束前执行模块任务的退出工作，比如释放模块资源。

### 3.5 任务消息处理

**PFN\_ZINSTMSGPROC** 是任务消息处理函数，也是系统任务的调度入口点。在所有注册模块已经初始化之后，模块管理器就会依次调用 **Zos\_TaskSpawn** 来启用各个任务调度入口点。

任务消息处理函数命名的格式是：**ZINT Xxx\_TaskMsgProc(ST\_ZOS\_MSG \*pstMsg)** 如：

SIP 任务消息处理函数名为：

```
/* sip task message process */  
ZINT Sip_TaskMsgProc(ST_ZOS_MSG *pstMsg)
```

以上两种方式创建的任务都是可以支持消息处理和计时器功能的，并且可以设置任务名称、堆栈大小与调度优先级，而任务ID是由产品统一规划的。

## 4. 层间通信

### 4.1 层间通信概要

ZOS 任务间通信是通过消息驱动的方式来实现的，事件驱动程序不是由事件的顺序来控制，而是由事件的发生来控制，而这种事件的发生是随机的、不确定的，并没有预定的顺序，这样就允许程序的用户用各种合理的顺序来安排程序的流程。它是一种“被动”式程序设计方法，程序开始运行时，处于等待用户输入事件状态，然后取得事件并作出相应反应，处理完毕又返回并处于等待事件状态。

事件驱动围绕着消息的产生与处理展开。事件驱动是靠消息循环机制来实现的。消息是一种报告有关事件发生的通知，基本上可以分为以下几类：

- 系统消息，如计时器任务发送的超时事件消息
- 管理层管理消息
- 上层用户消息
- 下层用户消息

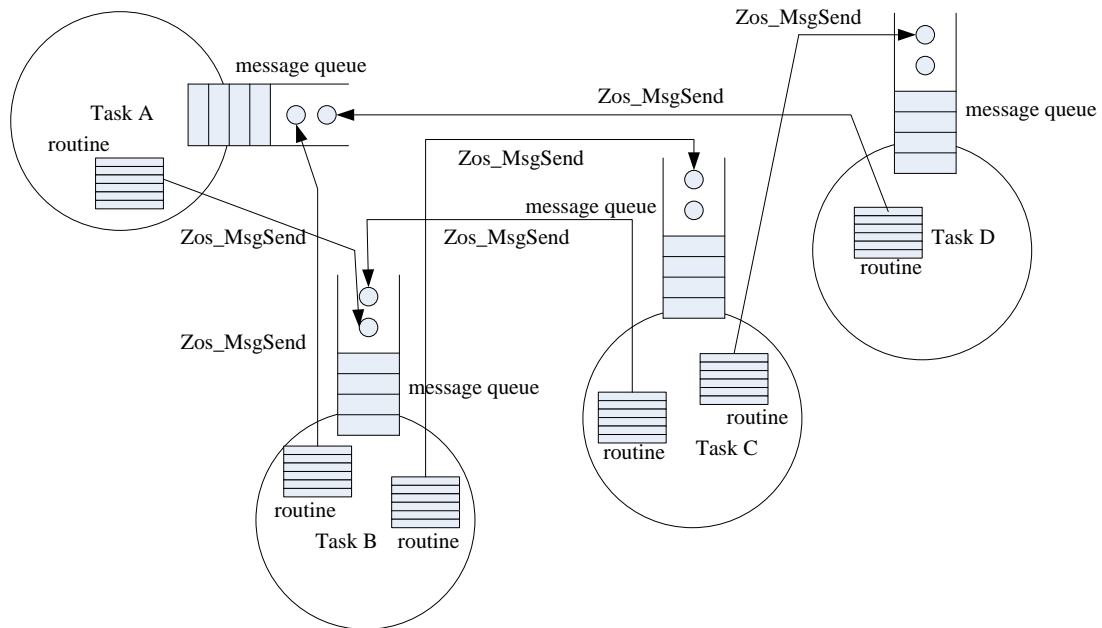


图 4-1 ZOS 任务间消息通信

## 4.2 模块服务点

为了保证在任务间互相传递消息，在任务创建或模块注册的时候，消息队列（一种FIFO对列）的值不能为零，否则任务就无法接受消息。此外，任务间消息传递需要提供模块服务投递信息，其数据结构如下：

```

/* zos module service post */
typedef struct tagZOS_MSP
{
    ZCPUID zSendCpuId;           /* sender cpu id */
    ZTASKID zSendTaskId;        /* sender task id */
    ZCPUID zRecvCpuId;          /* receiver cpu id */
    ZTASKID zRecvTaskId;        /* receiver task id */
    ZEVNTID zEvtId;             /* event id */
    ZPOOLID zPoolId;            /* memory pool id */
} ST_ZOS_MSP;

/* zos message */
typedef struct tagZOS_MSG
{
    ST_ZOS_MSP stMsp;           /* module service post */
    ZULONG dwMsgLen;            /* message length */
    ZULONG dwDataLen;           /* data length beside message header */
} ST_ZOS_MSG;

```

在模块服务投递数据结构中，消息发送方（源任务）应该把自身和目的任务的处理机ID和任务ID填写清楚，并且给事件的类型赋值。消息接收方（目的任务）可以根据发送方的处理机ID和任务ID辨别发

送方是哪个层实体，也可以通过比较 **ST\_ZOS\_MSP** 中接收方的处理机ID和任务ID来判断消息的合法性，然后根据事件类型处理各种事件。

**ST\_ZOS\_MSP** 一般都是存放在消息体或模块任务管理器中。若存放在消息体中则表明消息的发送方和接收方。若存放在模块任务管理器中则表明 ZOS 任务间的服务附属关系；服务双方可在初始化兼容模块信息的时候设置模块关联信息。

### 4.3 消息发送

举例来说，消息发送的普遍方式是类似于 SIP 传输层任务收到协议报文后向上层传递：

```

/* sip transport data indication */
ZINT Sip_TptDataInd(ST_SIP_MSG_EVT *pstMsgEvt)
{
    ST_ZOS_MSG *pstSysMsg;
    ST_SIP_MSG_EVT *pstEvt;
    ST_ZOS_MSP stMsp;

    /* set upper layer task info */
    stMsp.zSendCpuId = ZCPUID_LOCAL;
    stMsp.zSendTaskId = ZTASKID_SIP_TPT;
    stMsp.zRecvCpuId = ZCPUID_LOCAL;
    stMsp.zRecvTaskId = ZTASKID_SIP;
    stMsp.zEvtId = pstMsgEvt->ucEvtType;
    stMsp.zPoolId = SIP_POOLID;

    /* alloc system message */
    pstSysMsg = Zos_MsgAlloc(&stMsp, sizeof(ST_SIP_MSG_EVT));
    if (pstSysMsg == ZNULL)
        return ZFAILED;

    /* set message event */
    pstEvt = (ST_SIP_MSG_EVT *)ZOS_MSG_GET_DATA(pstSysMsg);
    Zos_MemCpy(pstEvt, pstMsgEvt, sizeof(ST_SIP_MSG_EVT));

    /* print buffer for debug */
    if (pstEvt->pstMsgBuf)
        Sip_LogDbuf(pstEvt->pstMsgBuf);

    /* send message */
    if (Zos_MsgSend(pstSysMsg) != ZOK)
    {
        Zos_MsgFree(pstSysMsg);
        return ZFAILED;
    }
}

```

```
    return ZOK;
}
```

## 4.4 消息处理

举例来说，消息处理的普遍方式是类似于 SIP 协议栈任务处理上下层和计时器的消息：

```
/* sip task message process */
ZINT Sip_TaskMsgProc(ST_ZOS_MSG *pstMsg)
{
    ...
/* process message from sender */
    switch (pstSysMsg->stMsp.zSendTaskId)
    {
        case ZTASKID_TIMER:
            /* process zos timer message */
            Sip_TmrMsgProc(...);
            break;
        case ZTASKID_SIP_TPT:
            /* process transport message */
            Sip_TptMsgProc(...);
            break;
        case ZTASKID_SUA:
            /* process sip user agent session message */
            Sip_CoreMsgProc(pstSessEvt);
            break;
        default:
            /* received unknown event from unknown task */
            break;
    }

    return ZOK;
}
```

## 5. 计时器管理

### 5.1 计时器概要

计时器在通信系统中有以下用途：

- 协议任务和系统任务需要周期性的完成某些功能，比如 MG（媒体网关）需要周期性的发送 MGCP 协议注册消息给 MGC（媒体网关控制器）
- 协议任务和系统任务可能需要使用超时时器，当指定时间一到就启动相应的处理工作。比如当一个任务用特定的请求与另一个任务联系时就可能使用这种计时器；如果第二个任务在规定的时

间内没有响应（超时了），第一个任务就会启动某种错误恢复动作。协议任务可能需要多个计时器。例如，一个任务需要每隔30秒发送一条请求消息，同时在最近一次收到响应消息后过去180秒后就要启动超时动作，表明与对方失去联系。

## 5.2 平台计时器

在系统平台中，有三类计时器：

- 循环计时器，数值表示为 **ZTIMER\_MODE\_CYCLE**
- 超时释放计时器，数值表示为 **ZTIMER\_MODE\_NOCYCLE**
- 100 毫秒计时器，数值表示为 **ZTIMER\_MODE\_100MS**

循环计时器是指在计时器超时时触发用户事件，然后立即重新启动计时工作。如果用户需要停止循环计时器，必须要调用停止或删除计时器的接口来终止计时器的运行。

非循环计时器是指在计时器超时时触发用户事件，然后就停止计时工作。如果用户需要启动新的计时工作，必须要调用启动计时器接口来使得计时器能运行起来。

100毫秒计时器是一种高精度的计时器，是属于ZOS的计时器管理空间。默认情况下它是一种非循环计时器，如果需要启动一个循环计时器，可以通过创建计时器时设置计时器模式为 **ZTIMER\_MODE\_CYCLE | ZTIMER\_MODE\_100MS** 。

系统平台有一个独立的计时器任务，负责管理所有的计时器对象，它的默认计时点滴是100毫秒，具体数值由 **g\_stZosSysCfg.stTimer.dwTaskDelayVal** 设置。在 ZOS 任务创建或模块注册的时候，如果设置了计时器数目，ZOS 任务本身也会创建一个计时器队列，由任务自己独立维护管理，然后向计时器任务注册一个循环计时器，默认循环计时点滴为500毫秒，这是考虑到 ZOS 任务在一般情况下不需要100毫秒级的计时精度，具体数值由 **g\_stZosSysCfg.stTimer.dwTimerInterval** 设置。这样每隔500毫秒，计时器任务就会发一个超时事件给 ZOS 任务，然后 ZOS 任务处理本身的计时器队列。综上所述，ZOS 任务也相当于一个系统计时器任务，只是精度没有系统计时器任务高。这样设计的好处是系统计时器任务的工作负荷被每个 ZOS 任务分担了，降低了它的开销了，提高了系统的稳定性。

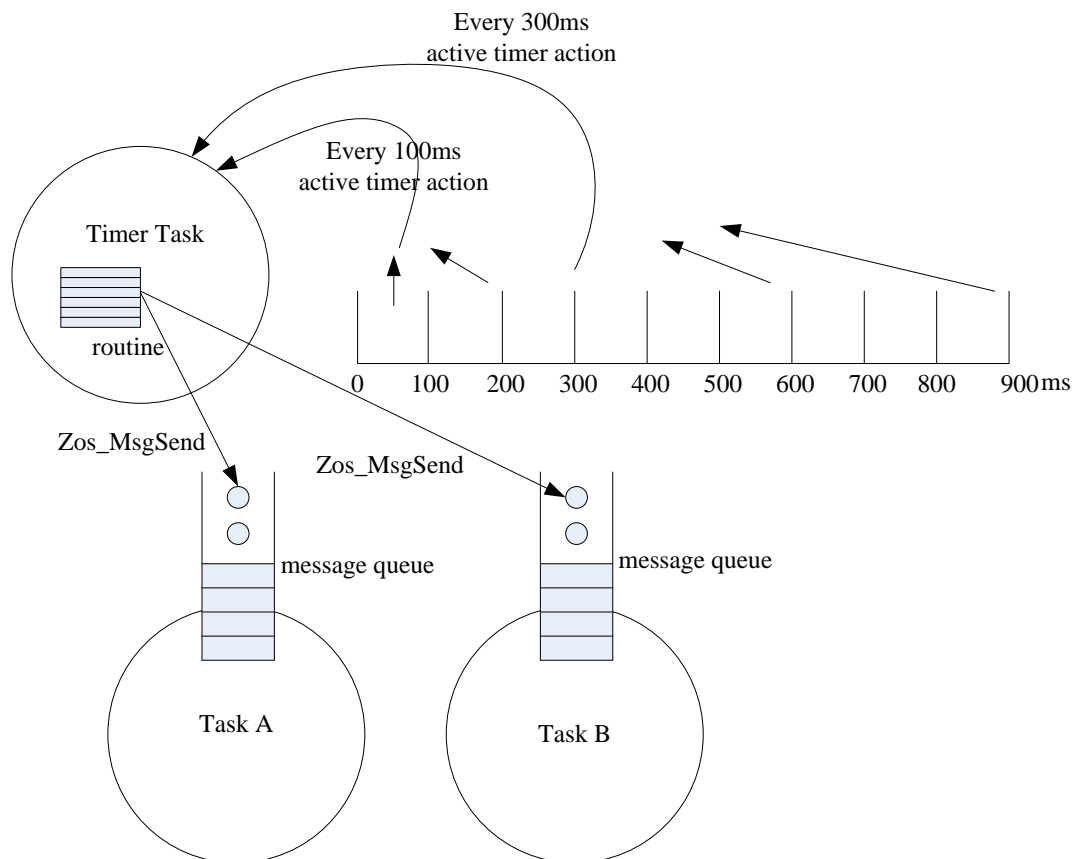


图 5-1 ZOS 计时器任务驱动

在上图中，系统计时器任务每隔一段计时点滴时间长度，就会发送相应事件给启用定时器服务的 ZOS 任务中的消息队列。

### 5.3 计时器接口

- 计时器到时激活动作 **PFN\_ZTIMERACTIVE** 的声明为：

```
/* zos timer active function for callback */
typedef ZVOID (*PFN_ZTIMERACTIVE)(ZTIMERID zTimerId, ZULONG dwTimerType,
                                   ZULONG dwParm);
```

在计时器任务超时后，它会把超时事件发送给相应的任务，在任务得到调度后，任务计时器处理函数就会执行激活动作。

通过调用函数 **Zos\_TimerCreate** 可以创建一个计时器。该函数声明的如下：

```
/* zos create a timer */
ZINT Zos_TimerCreate(ZTASKID zTaskId, ZUCHAR ucMode, ZTIMERID *pzTimerId);
```

创建计时器后，用户可以随时调用函数 **Zos\_TimerStart** 来启动计时器。该函数的声明如下：

```
/* zos start a timer */
ZINT Zos_TimerStart(ZTASKID zTaskId, ZTIMERID zTimerId, ZULONG dwTimerType,
                   ZULONG dwTimeLen, ZULONG dwParm, PFN_ZTIMERACTIVE pfnActive);
```

如果要停止正在运行的某个计时器，则需调用 **Zos\_TimerStop** 函数。该函数的声明如下：

```
/* zos stop a timer */
ZINT Zos_TimerStop(ZTASKID zTaskId, ZTIMERID zTimerId);
```

函数 **Zos\_TimerDelete** 的功能是删除一个计时器，它的函数声明如下：

```
/* delete one timer, if the timer is active, it will stop the timer */
ZINT Zos_TimerDelete(ZTASKID zTaskId, ZTIMERID zTimerId);
```

函数 **Zos\_TimerIsRun** 是用来判断某个计时器是否正在运行，它的函数声明如下：

```
/* is the timer in running state */
ZBOOL Zos_TimerIsRun(ZTASKID zTaskId, ZTIMERID zTimerId);
```

## 6. 内存管理

### 6.1 内存管理概要

尽管操作系统提供了内存管理功能，但为了保证可移植性与可管理性，而且由于不同的产品容量有不同的限制和不同的性能要求，系统平台也提供了内存管理功能。

内存管理的机制是建立一个内存池(pool)，内存池被分成多个大小不同的 bucket 组，比如32字节、64字节的 bucket 组。当用户申请内存的时候 ZOS 将从最适合大小的 bucket 组中把空闲的 bucket 分配给用户。比如当用户申请大小为50字节的内存时，ZOS 首先从64字节大小的 bucket 组中查找空闲 bucket，如果没有可用的空闲 bucket，就到比较大的 bucket 组中继续查找，直到找到可用的空闲 bucket。若找不到可用的空闲 bucket，则返回失败信息。

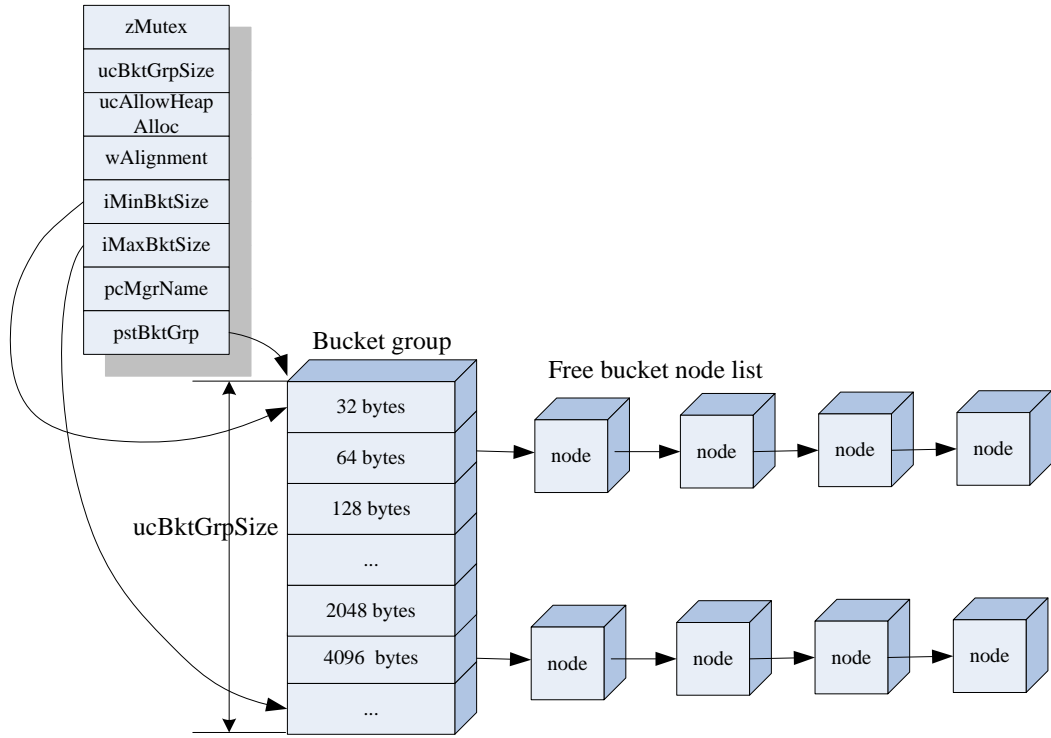


图 6-1 ZOS 内存管理机制

如上图所示，每一个 bucket 节点表明一块内存空间；所有的 bucket 都由 bucket 组管理。事实上，内存池就是从系统内存堆分配的一块足够大的内存块。当 bucket 组中的 bucket 用完的时候，ZOS 可以从系统内存堆中分配更多的 bucket，前提是 bucket 可增加数目是非零。

Bucket 管理机制是一种通用而高效的内存管理方法。在 ZOS 平台中，内存管理、消息管理、数据缓冲区管理都是采用 bucket 管理机制实现的。

## 6.2 Bucket Pool

以下是定制 Bucket Pool 的相关数据结构：

```

/* zos bucket config info */
typedef struct tagZOS_BKT_INFO
{
    ZULONG dwBktSize;           /* bucket size */
    ZULONG dwMaxCount;         /* bucket maixmum count */
    ZULONG dwIncCount;         /* bucket inc count per time */
} ST_ZOS_BKT_INFO;

/*****
zos pool config
the data structure should be init by user

+-----+
| *pcName      |
| ucIsNeedMutex | indicate the num of bkt info
| ucInfoGrpSize -----+
| aucSpare[2]   |           |
+-----+           V
| *pstInfoGrp -----> +-----+
|           |           | bkt info 1 |
+-----+           | bkt info 2 |
| pfnHeapAlloc |           | ...      |
| pfnHeapFree  |           | bkt info n |
+-----+           +-----+

*****/
typedef struct tagZOS_POOL_CFG
{
    ZCHAR *pcName;           /* bucket manager name */
    ZUCHAR ucIsNeedMutex;   /* is need mutex */
    ZUCHAR ucInfoGrpSize;   /* info group size */
    ZUCHAR aucSpare[2];     /* for 32 bit alignment */
    ST_ZOS_BKT_INFO *pstInfoGrp; /* info group */
    PFN_ZHEAPMALLOC pfnHeapAlloc; /* heap memory alloc function */
    PFN_ZHEAPFREE pfnHeapFree; /* heap memory free function */
} ST_ZOS_POOL_CFG;

```

创建和删除Bucket Pool的接口函数 **Zos\_PoolCreate** 和 **Zos\_PoolDelete**:

```
/* zos create memory pool */
ZPOOLID Zos_PoolCreate(ST_ZOS_POOL_CFG *pstPoolCfg);

/* zos delete memory pool */
ZVOID Zos_PoolDelete(ZPOOLID zPoolId);
```

在默认情况下，Bucket Pool 申请内存资源接口是由 **ST\_ZOS\_POOL\_CFG** 中的 内存申请指针 **PFN\_ZHEAPMALLOC** 决定的。当指针为空的时候，系统将会调用 `malloc` 函数；同样，释放内存对应的就是 `free` 函数。

### 6.3 Bucket Pool 接口

从 Bucket Pool 中申请释放资源的主要接口函数有：

```
/* zos memory pool alloc memory block with specific size */
ZVOID * Zos_PoolAlloc(ZPOOLID zPoolId, ZUINT iSize);

/* zos memory pool alloc memory block with specific size and zero block */
ZVOID * Zos_PoolAllocClr(ZPOOLID zPoolId, ZUINT iSize);

/* zos memory pool free memory block */
ZVOID Zos_PoolFree(ZPOOLID zPoolId, ZVOID *pMem);

/* zos memory pool get size by the memory address */
ZINT Zos_PoolGetSize(ZPOOLID zPoolId, ZVOID *pMem, ZUINT *piSize);

/* zos memory pool check valid adress of memory block */
ZBOOL Zos_PoolIsValid(ZPOOLID zPoolId, ZVOID *pData);
```

### 6.4 Bucket Pool 示例

Bucket Pool 管理技术的使用方法是首先定义一个 bucket 组的相关信息 **ST\_ZOS\_BKT\_INFO**，然后调用 **Zos\_PoolCreate** 创建 bucket 管理器，调用 **Zos\_PoolAlloc** 从内存池中申请内存块，而调用 **Zos\_PoolFree** 可以释放内存块。以下代码说明了 ZOS 平台内存模块的实现过程。

```
/* zos default memory bucket config info group */
static ST_ZOS_BKT_INFO m_astZosCfgMemDftBktInfoGrp[] =
{
    /* size,          maximum count,      increment count */
    {32,             0,                   10},
    {64,             0,                   10},
    {128,            0,                   10},
    {256,            0,                   10},
    {512,            0,                   10},
    {1024,           0,                   10},
    {2048,           0,                   10},
    {4096,           0,                   10},
    {8192,           0,                   10}
};

/* zos memory pool id for memory management */
static ZPOOLID m_zZosMemPoolId = ZNULL;

/* zos memory initialization */
ZINT Zos_MemInit()
{
    /* create memory pool */
    m_zZosMemPoolId = Zos_PoolCreate(&g_stZosSysCfg.stMem);
    if (m_zZosMemPoolId == ZNULL)
    {
        return ZFAILED;
    }

    return ZOK;
}

/* zos memory malloc */
ZVOID * Zos_Malloc(ZSIZE_T zSize)
{
    if (zSize >= ZMAXINT || zSize == 0)
        return ZNULL;

    return Zos_PoolAlloc(m_zZosMemPoolId, zSize);
}

/* zos memory free */
ZVOID Zos_Free(ZVOID *pMem)
{
    /* free data into the memory pool */
}
```

```
Zos_PoolFree(m_zZosMemPoolId, pMem);  
  
return;  
}
```

## 7. 缓冲区管理

### 7.1 缓冲区概要

缓冲区主要用于 ZOS 层间任务之间的数据交换，数据可以是协议报文、层任务控制消息等。使用缓冲区来交换数据可以减少数据复制的次数。如果系统需要花大量的CPU时间和内存带宽用于缓冲区之间的数据复制，则系统性能就会严重下降。例如传输层收到消息后会先把协议报文放到一个缓冲区中，然后把缓冲区发送给协议模块。

缓冲区也是一种高效而稳定的内存控制方式，尤其当模块需要使用大量的动态内存块时。如果这些内存是从内存池中申请，那么就必须在合适的地方释放这些内存块，稍有不慎就会产生内存泄露，从而影响系统的稳定性，而且大量的分配释放操作也影响了系统性能。但如果这些内存块是从缓冲区中申请的，那么只要该缓冲区被释放了，所有的内存块都被安全的释放了，不会存在内存泄露问题，且系统的稳定性也得到了提高。比如在协议编解码中，需要产生大量的协议消息控制块，这些控制块都是从一个数据缓冲区中申请的。当用户使用完这些协议消息控制块后，只要释放这一个数据缓冲区就可以了，因此编解码的性能和稳定性都得到了很大的提高了。

缓冲区管理提供了一种在系统内对缓冲区进行分配、操作和释放的通用机制。分配就是从全局缓冲区池获得缓冲区，包括将数据复制到缓冲区，从缓冲区中读出数据，在缓冲区的头部和尾部添加或删除数据，将两个缓冲区的数据拼接起来及复制缓冲区副本等。释放就是将缓冲区返还给全局缓冲区，以便其他任务或模块能对其进行分配。

有两种缓冲区管理方式：全局缓冲区管理和局部缓冲区管理。在全局管理方式下所有 ZOS 任务都从一个缓冲区池中分配缓冲区；而在局部管理方式下 ZOS 任务可以独自创建缓冲区池，且缓冲区的分配不会占用其他ZOS 任务的缓冲区资源。

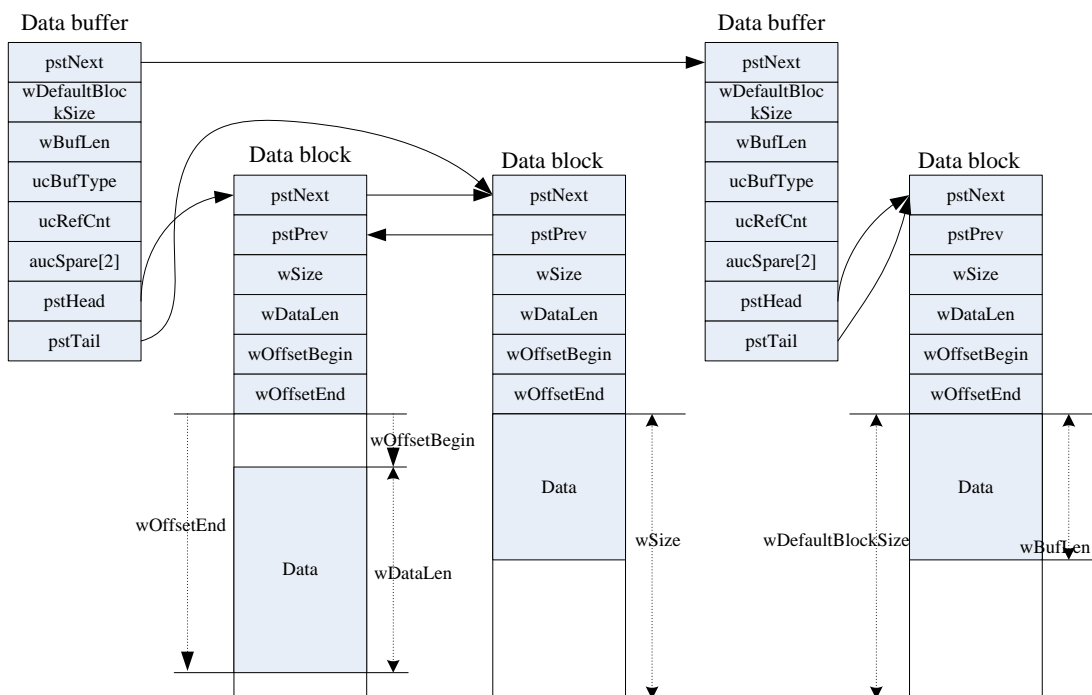


图 7-1 数据缓冲区(Dbuf)结构

## 7.2 数据缓冲区

ZOS 的缓冲区管理称为数据缓冲区(Data Buffer)管理，如上图所示，数据缓冲区控制块中有所有缓冲区的数据长度，链接了所有的数据块的链表。控制块中的默认数据块的大小有独特的用法，当请求的数据块小于默认大小的时候，ZOS 会从内存池中分配如默认大小的内存块。但如果所请求的数据块大于默认数值时，缓冲区会首先尝试着从内存池中分配所需数据块的大小的内存块，如果能分配到，则放置于缓冲区中的数据块链表中，否则分配失败。

数据缓冲区有两种类型：字节对齐和结构对齐类型。字节对齐类型表明在分配数据内存块的时候，内存块所占用的地址空间是字节连续的，其内存块首地址不需要是4的倍数；而结构对齐类型则表明每次分配的数据内存块大小都是4的倍数，如请求2字节数据内存块意味着分配4字节的数据内存块。

此外数据缓冲区可以被多次引用，当引用计数为0时，也就意味数据缓冲区被释放了。

除了可以从内存池中动态的申请缓冲区，也可以通过静态内存区来创建缓冲区。

数据缓冲区的主要数据结构由2部分组成，即缓冲区头部和数据块，如下所示：

```
/* zos data buffer data block */
typedef struct tagZOS_DBUF_DATA
{
    struct tagZOS_DBUF_DATA *pstNext; /* next data block */
    struct tagZOS_DBUF_DATA *pstPrev; /* previous data block */
    ZULONG dwSize;                    /* block size */
    ZULONG dwDataLen;                 /* data length in block */
    ZULONG dwOffsetBegin;             /* data begin offset in block */
    ZULONG dwOffsetEnd;               /* data end offset in block */
    /*lint -save -e* */
    ZCHAR acData[0];                  /* data memory */
    /*lint -restore */
} ST_ZOS_DBUF_DATA;

/* zos data buffer block */
typedef struct tagZOS_DBUF
{
    struct tagZOS_DBUF *pstNext;      /* next data buffer */
    ZPOOLID zPoolId;                  /* memory pool id for dbuf alloc */
    ZULONG dwBufLen;                  /* buffer used length */
    ZULONG dwDftBlkSize;              /* default data block size in buffer */
    ZUCHAR ucBufType;                 /* buffer mode ZDBUF_TYPE_BYTE... */
    ZUCHAR ucRefCnt;                 /* buffer reference count */
#ifdef ZOS_SUPT_DUMP
    ZDUMPID zDumpId;                  /* stack dump */
#endif
    ST_ZOS_DBUF_DATA *pstHead;        /* the first data block in buffer */
    ST_ZOS_DBUF_DATA *pstTail;       /* the last data block in buffer */
} ST_ZOS_DBUF;
```

### 7.3 数据缓冲区接口

数据缓冲区的主要用户接口有：

```
/* create a data buffer */
ST_ZOS_DBUF * Zos_DbufCreate(ZPOOLID zPoolId, ZUCHAR ucType,
                             ZULONG dwDftBlkSize);

/* delete dbuf and all data block */
ZVOID Zos_DbufDelete(ST_ZOS_DBUF *pstBuf);

/* zos only free all data block */
ZVOID Zos_DbufFree(ST_ZOS_DBUF *pstBuf);

/* alloc data memory from dbuf */
ZVOID * Zos_DbufAlloc(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);

/* alloc data memory from dbuf and clear data to 0 */
ZVOID * Zos_DbufAllocClr(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);

/* length (used) of all data block in dbuf */
ZULONG Zos_DbufLen(ST_ZOS_DBUF *pstBuf);

/* size of all data block in dbuf */
ZULONG Zos_DbufSize(ST_ZOS_DBUF *pstBuf);
```

《ZOS服务接口手册》包含了有关数据缓冲区接口的进一步说明。

## 7.4 数据缓冲区示例

数据缓冲区的使用示例如下：

```
/* sdp test decode message */
TSdp_DecodeMsg(ZCHAR *pcMemMsg)
{
    ST_ZOS_DBUF *pstMemBuf;
    ST_SDP_ANCMT *pstSdpMsg;

    /* create the dbuf */
    pstMemBuf = Zos_DbufCreate(ZNULL, ZDBUF_TYPE_STRUCT, 1024);
    if (pstMemBuf == ZNULL)
        return ZFAILED;
```

```
/* allocate memory for sdp message */
pstSdpMsg = Zos_DbufAlloc(pstMemBuf, sizeof(ST_SDP_ANCMT));
if (pstSdpMsg == ZNULL)
{
    /* delete memory buffer */
    Zos_DbufDelete(pstMemBuf);
    return ZFAILED;
}

/* initialize the error info */
Abnf_ErrInit(&stErrInfo);

/* decode message */
if (Sdp_DecodeMsg(&stDataStr, pstMemBuf, &stErrInfo, pstSdpMsg) != ZOK)
{
    /* delete memory buffer */
    Zos_DbufDelete(pstMemBuf);

    /* destroy error info */
    Abnf_ErrDestroy(&stErrInfo);
    return ZFAILED;
}

/* delete memory buffer */
Zos_DbufDelete(pstMemBuf);

/* destroy error info */
Abnf_ErrDestroy(&stErrInfo);

return ZOK;
}
```

## 8. 配置说明

本章主要告诉用户如何根据自身需要对ZOS平台进行配置，帮助用户成功创建基于ZOS的应用程序。

### 8.1 库文件配置

通常情况下，用户使用是预先配置的默认参数，但用户可以根据实际情况自定义配置 ZOS 的平台参数，比如优化内存配置、设置支持的模块数目、设置平台日志选项等。本章将详述所有ZOS配置参数。

对 ZOS 平台的配置是通过修改一个全局结构体变量 `g_stZosSysCfg` 来实现的。结构体变量 `g_stZosSysCfg` 的声明如下：

```

/* zos system config */
typedef struct tagZOS_SYS_CFG
{
    ST_ZOS_GEN_CFG stGen;           /* generic config */
    ST_ZOS_POOL_CFG stMem;         /* memory config */
    ST_ZOS_POOL_CFG stMsg;        /* message config */
    ST_ZOS_POOL_CFG stDbuf;       /* dbuf config */
    ST_ZOS_LOG_CFG stLog;         /* log config */
    ST_ZOS_MOD_CFG stMod;         /* module config */
    ST_ZOS_TASK_CFG stTask;       /* task config */
    ST_ZOS_TIMER_CFG stTimer;     /* timer config */
} ST_ZOS_SYS_CFG;

```

ZOS参数共有八类，如下所示：

- General Config
- Memory Pool Config
- Message Pool Config
- Data Buffer Pool Config
- Log Config
- Module Config
- Task Config
- Timer Config

本文将在接下来的八节中对这些参数进行介绍。

### 8.1.1 综合配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stGen` 是一个结构体变量，它包含了进行综合配置所需的所有参数。结构体 `stGen` 的声明如下：

```

/* zos general config parameter */
typedef struct tagZOS_GEN_CFG
{
    ZUCHAR ucIsSuptAssert;        /* is support ASSERT */
    ZUCHAR aucSpare[3];          /* for 32 bit alignment */
    PFN_ZPRINTDISP pfnPrintfDisp; /* printf display function */
    PFN_ZPRINTDISP pfnLogStrDisp; /* log string display function */
    PFN_ZHEAPMALLOC pfnHeapMalloc; /* heap memory alloc */
    PFN_ZHEAPFREE pfnHeapFree;    /* heap memory free */
} ST_ZOS_GEN_CFG;

```

下表列举了 `stGen` 的所有成员变量。

结构体成员	默认值	成员描述
uclsSuptAssert	ZFALSE	该变量是一个标记，用来表示 ZOS 平台是否支持宏 ZOS_ASSERT。如果将该变量的值设置成 ZTRUE, 则宏 ZOS_ASSERT 会被替换成函数 Zos_Assert。
pfnPrintfDisp	ZNULL	Zos_Printf 的重定向，如果值为 ZNULL，显示重定向函数为 printf。
pfnLogStrDisp	ZNULL	日志输出的重定向，如果值为 ZNULL，显示重定向函数为 printf。
pfnHeapMalloc	ZNULL	ZOS 使用该函数为 heap system 分配内存。默认值 ZNULL 表示 ZOS 将使用系统函数 malloc() 来分配内存。请注意，分配内存函数和释放内存函数必须成对出现，如 malloc() & free()。
pfnHeapFree	ZNULL	ZOS 使用该函数释放 heap system 的内存。默认值 ZNULL 表示 ZOS 将使用系统函数 free()来释放内存。请注意，分配内存函数和释放内存函数必须成对出现，如 malloc() & free()。

表 8-1 综合配置参数

### 8.1.2 内存池配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stMem` 包含了与内存池配置相关的所有参数。结构体 `stMem` 的声明如下：

```

/* zos pool config */
typedef struct tagZOS_POOL_CFG
{
    ZCHAR *pcName;           /* bucket manager name */
    ZUCHAR ucIsNeedMutex;   /* is need mutex */
    ZUCHAR ucInfoGrpSize;   /* info group size */
    ZUCHAR aucSpare[2];     /* for 32 bit alignment */
    ST_ZOS_BKT_INFO *pstInfoGrp; /* info group */
    PFN_ZHEAPMALLOC pfnHeapAlloc; /* heap memory alloc function */
    PFN_ZHEAPFREE pfnHeapFree; /* heap memory free function */
} ST_ZOS_POOL_CFG;

```

下表列举了 `stMem` 的所有成员变量。

结构体成员	默认值	成员描述
pcName	"zos memory"	内存池的 bucket manager 名称。
uclsNeedMutex	ZTRUE	该变量用来表示是否需要使用互斥锁来保护内存池。

uclInfoGrpSize	ZOS_GET_TABLE_SIZE(m_astZosCfgMemDftBktInfoGrp)	表示内存池中 bucket information group 的大小。注意，该成员变量表示的是 pstInfoGrp 数组的大小，用户需通过使用宏 ZOS_GET_TABLE_SIZE() 与 pstInfoGrp 保持一致。
pstInfoGrp	m_astZosCfgMemDftBktInfoGrp	内存池所使用的 bucket information group。
pfnHeapAlloc	ZNULL	ZOS 通过调用该函数为内存池分配 system heap 内存。默认值 <b>ZNULL</b> 表示 ZOS 将调用 <b>stGen.pfnHeapMalloc</b> 来分配内存。
pfnHeapFree	ZNULL	ZOS 通过调用该函数释放为内存池分配的 system heap 内存。默认值 <b>ZNULL</b> 表示 ZOS 将调用 <b>stGen.pfnHeapFree</b> 来释放内存。

表 8-1 内存池配置参数

### 8.1.3 消息缓冲池配置

结构体变量 *g\_stZosSysCfg* 的成员之一 *stMsg* 包含了与消息缓冲池配置相关的所有参数。结构体 *stMsg* 的声明和 *stMem* 的完全相同。*stMsg* 的成员变量如下表所列：

结构体成员	默认值	成员描述
pcName	"zos message"	消息缓冲池的 bucket manager 名称。
uclsNeedMutex	ZTRUE	该变量用来表示是否需要使用互斥锁来保护消息缓冲池。
uclInfoGrpSize	ZOS_GET_TABLE_SIZE(m_astZosCfgMsgDftBktInfoGrp)	表示消息缓冲池中 bucket information group 的大小。注意，该成员变量表示的是 pstInfoGrp 数组的大小，用户需通过使用宏 ZOS_GET_TABLE_SIZE() 与 pstInfoGrp 保持一致。
pstInfoGrp	m_astZosCfgMsgDftBktInfoGrp	消息缓冲池所使用的 bucket information group。
pfnHeapAlloc	Zos_Malloc	ZOS 通过调用该函数为消息缓冲池分配 system heap 内存。
pfnHeapFree	Zos_Free	ZOS 通过调用该函数释放分配给消息缓冲池的 system heap 内存。

表 8-2 消息缓冲池配置参数

### 8.1.4 数据缓冲池配置

结构体变量 *g\_stZosSysCfg* 的成员之一 *stDbuf* 包含了与消息缓冲池配置相关的所有参数。结构体 *stDbuf* 的声明和 *stMem* 的完全相同。*stDbuf* 的成员变量如下表所列：

结构体成员	默认值	成员描述
pcName	"zos dbuf"	数据缓冲池的 bucket manager 名称。

uclsNeedMutex	ZTRUE	该变量用来表示是否需要使用互斥锁来保护数据缓冲池。
uclInfoGrpSize	ZOS_GET_TABLE_SIZE(m_astZosCfgDbufDftBktInfoGrp)	表示数据缓冲池中 bucket information group 的大小。注意，该成员变量表示的是 pstInfoGrp 数组的大小，用户需通过使用宏 ZOS_GET_TABLE_SIZE() 与 pstInfoGrp 保持一致。
pstInfoGrp	m_astZosCfgDbufDftBktInfoGrp	数据缓冲池所使用的 bucket information group。
pfnHeapAlloc	Zos_Malloc	ZOS 通过调用该函数为数据缓冲池分配 system heap 内存。
pfnHeapFree	Zos_Free	ZOS 通过调用该函数释放分配给数据缓冲池的 system heap 内存。

表 8-3 数据缓冲池配置参数

### 8.1.5 日志配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stLog` 包含了日志相关参数。结构体 `stLog` 的声明如下：

```

/* zos log config */
typedef struct tagZOS_LOG_CFG
{
    ZBOOL bTaskSupt;           /* log task support flag */
    ZULONG dwTaskStackSize;   /* log task stack size */
    ZINT iTaskPriority;        /* log task priority */
    ZULONG dwTaskDelayTime;   /* log task delay time(milliseconds) */
    ZULONG dwLineSize;        /* log line size */
    ZCHAR *pcZosLogFileName; /* zos log file name */
    ZULONG dwZosLogLevel;     /* zos log level */
    ZULONG dwZosLogBufSize;   /* zos buffer size */
    ZULONG dwZosLogFileSize; /* zos log file size */
    ZBOOL bZosLogPrint;       /* zos log print while start to run */
} ST_ZOS_LOG_CFG;

```

下表列举了 `stLog` 的所有成员变量。

结构体成员	默认值	成员描述
bTaskSupt	ZFALSE	当此变量被设置为 <b>ZTRUE</b> ，就会有另一个日志任务产生来负责日志工作。
dwTaskStackSize	40960	当需要生成一个日志任务的时候就需要使用任务堆栈大小参数。
iTaskPriority	ZTASK_PRIORITY_MIN	当需要生成一个日志任务的时候就需要使

		用任务优先级参数。 该参数的取值范围如下： ZTASK_PRIORITY_MAX ZTASK_PRIORITY_NORMAL ZTASK_PRIORITY_MIN ZTASK_PRIORITY_INCREMENT
dwTaskDelayTime	60000	当日志任务开始处理新一轮来自其他 ZOS 任务的日志请求时，日志任务需要先延迟一段时间。该参数表明了日志任务将收集到的日志请求写入日志文件的频率。时间单位为毫秒。
dwLineSize	1024	每个日志中每行的长度。
pcZosLogFileName	"zos.log"	ZOS 日志文件名。
dwZosLogLevel	ZLOG_LEVEL_FATAL   ZLOG_LEVEL_ERROR   ZLOG_LEVEL_INFO	ZOS 日志级别： ZLOG_LEVEL_NULL, ZLOG_LEVEL_ALL, 或者时 ZLOG_LEVEL_FATAL, ZLOG_LEVEL_ERROR, ZLOG_LEVEL_WARNING, ZLOG_LEVEL_INFO, ZLOG_LEVEL_DBG 的组合。
dwZosLogBufSize	120 * 50	日志缓冲区大小。若将此变量设的太小，则会引发磁盘 disk I/O 操作。
dwZosLogFileSize	2000000	日志文件缓冲区大小。如果日志文件大小超出这个限制，则新的日志覆盖旧的日志。
bZosLogPrint	ZTRUE	该变量表示在将字符串写入日志文件的同时是否还要将其写到标准输出流 <b>stdout</b> 。

表 8-4 日志配置参数

### 8.1.6 模块配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stMod` 包含了软件模块相关参数。结构体 `stMod` 的声明如下：

```

/* zos module config */
typedef struct tagZOS_MOD_CFG
{
    ZUSHORT wInstNumPerMod;          /* instance number per module */
    ZUSHORT wModCount;              /* module count */
} ST_ZOS_MOD_CFG;

```

下表列举了 `stMod` 的所有成员变量。

结构体成员	默认值	成员描述
wInstNumPerMod	5	每个模块的实例编号。模块的一个实例也是一个存在于 ZOS 中的任务。
wModCount	25	ZOS 所允许的模块最大编号。

表 8-5 模块配置参数

### 8.1.7 任务配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stTask` 包含了任务相关参数。结构体 `stTask` 的声明如下：

```

/* zos task config
 * time-slice in ticks or 0 to disable round-robin
 */
typedef struct tagZOS_TASK_CFG
{
    ZULONG dwDftTimeSlice;          /* default time slice */
    ZULONG dwDftStackSize;         /* default stack size */
    ZULONG dwDftQueueSize;         /* default queue size */
} ST_ZOS_TASK_CFG;

```

下表列举了 `stMod` 的所有成员变量。

结构体成员	默认值	成员描述
dwDftTimeSlice	0	操作系统在调度任务时所使用的时间片值。目前该参数是在 VxWorks 系统中有效。建议用户不要去修改它。
dwDftStackSize	0x10000	若用户没有确定该参数的值，ZOS 就会使用默认栈大小值去产生一个任务。用户只需将其置为 0，ZOS 就会使用默认值。
dwDftQueueSize	100	ZOS 在产生一个任务时所使用的默认任务队列大小值。任务队列是用来传送内部任务消息的。若该变量的值设置的太小会导致任务消息丢失。

表 8-6 任务配置参数

### 8.1.8 计时器配置

结构体变量 `g_stZosSysCfg` 的成员之一 `stTimer` 包含了计时器相关参数。结构体 `stTimer` 的声明如下：

```

/* zos timer config */
typedef struct tagZOS_TIMER_CFG
{
    ZULONG dwTimerNum;           /* timer number */
    ZULONG dwTimerInterval;     /* timer interval */
    ZULONG dwTaskDelayVal;      /* timer task delay value */
    ZULONG dwTaskStackSize;     /* timer task stack size */
    ZINT iTaskPriority;          /* timer task priority */
} ST_ZOS_TIMER_CFG;

```

下表列举了 *stTimer* 的所有成员变量。

结构体成员	默认值	成员描述
dwTimerNum	100	计时器任务使用的计时器数量。
dwTimerInterval	500	该参数表明过多长时间计时器任务会给每个 ZOS 任务发送超时消息。通常，一个 ZOS 任务是被来自其他任务的消息所触发(包括 ZOS 计时器任务)。计时器任务不间断的发送超时消息使其他 ZOS 任务能有机会检查所使用的计时器中是否有超时的。计时器时间单位为毫秒。请注意，该参数值不可小于 100。
dwTaskDelayVal	100	该参数表明当计时器任务给每个 ZOS 任务发送计时器消息后需要延迟多长时间。
dwTaskStackSize	32768	产生计时器任务时所使用的任务栈大小值。
iTaskPriority	ZTASK_PRIORITY_NORMAL	所产生的计时器任务的优先级。

表 8-7 计时器配置参数

## 9. 创建应用程序

用户可以很容易的创建基于 ZOS 的应用程序。创建步骤如下：

- 定义模块 ID 和任务 ID。
- 包含正确的头文件。
- 对 ZOS 平台进行配置和初始化。
- 设计用户应用程序的任务
- 使用正确的 flag 对应用程序进行编译和链接。

本章将告诉用户如何通过对他们自己的源文件进行修改来创建基于 ZOS 的应用程序。

### 9.1 模块ID和任务ID

名词“模块”和“任务”在基于 ZOS 的环境中被使用。通常在大多数操作系统中（如 Win32、Linux 及 Solaris），一个 ZOS 任务实例就是一个线程。一个 ZOS 模块就是一组相关的 ZOS 任务实例。

一个模块可以拥有的实例的最大数量的默认值是5，这个值可以经重新配置 ZOS 而改变。ZOS 给每个模块预先分配了 ID 号。

```
/* zos basic module configuration */
#define ZMODID_ROOT      0      /* root module id */
#define ZMODID_SYS      1      /* system module id */
#define ZMODID_UTAL     2      /* utal module id */
#define ZMODID_SIP      3      /* protocol sip module id */
#define ZMODID_RTP      4      /* protocol rtp module id */
#define ZMODID_H323     5      /* protocol h323 module id */
#define ZMODID_RTSP     6      /* protocol rtsp module id */
#define ZMODID_TEST     7      /* test module id */
#define ZMODID_SUA      8      /* sip user agent module id */
#define ZMODID_DNS      9      /* dns resolver module id */
```

每个任务也有一个ID号。

```
/* zos basic module task identifier */
#define ZTASKID_ROOT      ZTASKID_MAKE(ZMODID_ROOT, 0)
#define ZTASKID_TIMER    ZTASKID_MAKE(ZMODID_SYS, 0)
#define ZTASKID_LOG      ZTASKID_MAKE(ZMODID_SYS, 1)
#define ZTASKID_UTAL     ZTASKID_MAKE(ZMODID_UTAL, 0)
#define ZTASKID_SIP      ZTASKID_MAKE(ZMODID_SIP, 0)
#define ZTASKID_SIP_TPT  ZTASKID_MAKE(ZMODID_SIP, 1)
#define ZTASKID_RTP      ZTASKID_MAKE(ZMODID_RTP, 0)
#define ZTASKID_RTP_TPT  ZTASKID_MAKE(ZMODID_RTP, 1)
#define ZTASKID_H323     ZTASKID_MAKE(ZMODID_H323, 0)
#define ZTASKID_H323_TPT ZTASKID_MAKE(ZMODID_H323, 1)
#define ZTASKID_RTSP     ZTASKID_MAKE(ZMODID_RTSP, 0)
#define ZTASKID_RTSP_UA  ZTASKID_MAKE(ZMODID_RTSP, 1)
#define ZTASKID_TEST     ZTASKID_MAKE(ZMODID_TEST, 0)
#define ZTASKID_TEST_0   ZTASKID_MAKE(ZMODID_TEST, 1)
#define ZTASKID_TEST_1   ZTASKID_MAKE(ZMODID_TEST, 2)
#define ZTASKID_TEST_2   ZTASKID_MAKE(ZMODID_TEST, 3)
#define ZTASKID_SUA      ZTASKID_MAKE(ZMODID_SUA, 0)
#define ZTASKID_DNS      ZTASKID_MAKE(ZMODID_DNS, 1)
```

用户所创建的模块和任务必须由用户自己来定义它们的 ID 号。例如，用户写了一个 `exm_zos` 应用程序，它由两个 ZOS 任务组成，则用户需对这个应用程序作如下定义：

```
/* exm module ID */
#define ZMODID_EXM          20

/* exm task 0 ID */
#define ZTASKID_EXM_0      ZTASKID_MAKE(ZMODID_EXM, 0)

/* exm task 1 ID */
#define ZTASKID_EXM_1      ZTASKID_MAKE(ZMODID_EXM, 1)
```

注意：由于模块ID号0到9是为菊风产品所保留的，用户定义 ID 号的时候必须从8开始，最大数值为 0xFFFF（但不能超过 `g_stZosSysCfg.stMod.wModCount`）。

## 9.2 头文件

在使用 ZOS 平台提供的函数之前，用户必须将以下头文件包括到源文件中：

```
#include "zos.h"                /* zos system environment */
```

请参看《ZOS Release Notes》了解更多有关该头文件的信息。

## 9.3 对ZOS平台进行配置和初始化

通常，基于 ZOS 的应用程序应该对系统初始化参数进行配置，在主程序入口处对系统进行初始化。另外，用户也可以修改 ZOS 参数的值。因此一个自定义初始化函数是很有必要的。相关代码可能如下：

```
ZINT main()
{
    /* generic config */
    Zos_SysCfgInit();

    /* system init */
    if (Zos_SysInit() != ZOK)
    {
        return -1;
    }

    /* configure module */
    Exam_CfgInit();

    /* example start */
    Exam_Start();

    /* system destroy */
    Zos_SysDestroy();

    return 0;
}
```

## 9.4 设计应用程序任务

在 ZOS 环境下，用户可以有两种模式来设计应用程序任务。在第一种模式下，用户需要将任务登记到 ZOS，然后再启动任务。使用第一种模式需要两个宏：**ZOS\_MOD\_REG** 和 **ZOS\_MOD\_START**。在 ZOS 产生一个新的任务之前，或者销毁一个任务之后，或者收到一条发送给某个任务的 ZOS 消息时需要调用三个函数。在例子中我们附上了三个函数：**Exm\_TaskInit()**、**Exm\_TaskDestroy()** 和 **Exm\_TaskMsgProc()**。

```
/* example task 0 start */
ZINT Exm0_Start()
{
/* reg EXM_0 */
if (ZOS_MOD_REG(ZTASKID_EXM_0, "EXM_0", 50, 10, Exm_TaskInit,
                Exm_TaskDestroy) != ZOK)
    return ZFAILED;

Zos_Printf("reg EXM_0 ok.\r\n");

/* start EXM_0 */
if (ZOS_MOD_START(ZTASKID_EXM_0, ZTASK_PRIORITY_NORMAL, 0,
                  Exm_TaskMsgProc) != ZOK)
    return ZFAILED;

    return ZOK;
}
```

**Exm\_SendMsg** 是用来向某个ZOS任务发送消息的。

```
/* send message to EXM_0 */
ZINT Exm_SendMsg()
{
    ST_EXM_MSG *pstExmMsg;
    ST_ZOS_MSG *pstSysMsg;
    ST_ZOS_MSP stMsp;

    /* init upper layer msp */
    stMsp.zSendCpuId = ZCPUID_LOCAL;
    stMsp.zSendTaskId = 0; /* in fact from main task */
    stMsp.zRecvCpuId = ZCPUID_LOCAL;
    stMsp.zRecvTaskId = ZTASKID_EXM_0;
    stMsp.zPoolId = NULL;

    /* alloc system message */
    pstSysMsg = Zos_MsgAlloc(&stMsp, sizeof(ST_EXM_MSG));
    if (pstSysMsg == ZNULL)
    {
        Zos_Printf("alloc ZOS message failed!\r\n");
        return ZFAILED;
    }

    /* set information */
    pstExmMsg = (ST_EXM_MSG *)ZOS_MSG_GET_DATA(pstSysMsg);
    Zos_StrNCpy(pstExmMsg->acData, "Hello ZOS!", 15);

    /* send ZOS message */
    if (Zos_MsgSend(pstSysMsg) != ZOK)
    {
        /* free system message */
        Zos_MsgFree(pstSysMsg);
        Zos_Printf("send ZOS message failed\r\n");
        return ZFAILED;
    }

    return ZOK;
}
```

另一种模式是使用 *Zos\_TaskSpawn* 函数来产生新的任务，但它无法从其他 ZOS 任务那收到消息，也无法使用计时器机制。当需要产生的任务的运行模式不是基于事件的模式时（如从运输层接发消息的任务或者负责从音频卡采样数据的任务），可以用此模式产生新的任务。

```

/* example task 1 start */
ZINT Exm1_Start()
{
/* spawn EXM_1 */
if (Zos_TaskSpawn(ZTASKID_EXM_1, "EXM_1", ZTASK_PRIORITY_NORMAL,
                 0, 0, 100, Exm_Perm, 0) != ZOK)
    return ZFAILED;

    return ZOK;
}

```

## 9.5 对Flags进行编译和链接

如今 ZOS 已成功移植到Windows、VxWorks、Linux和Solaris；并且计划支持Win32 GNU环境，如Cygwin和MingW，还有Unix。

要对基于 ZOS 的应用程序源文件进行成功编译，需要对用户的编译环境作适当的修改。需要作修改的有 Visual Studio Project 或者 win32中的 makefile 或者是 Unix 中的 makefile。如何建立这些编译环境本文将不作论述。本节将详述如何对编译环境作修改并在附录中附上一个makefile文件以作示例。

下表列举了需要进行编译和链接的 Flags。

Flags	Description
Include Dir	Directories including ZOS header files must be included in compile phase. Get those headers from released product set and add words in compile environment like this: <code>/I "..\..\include/zos"</code> in Win32 environment or <code>-I"..\..\include/zos"</code>
Platform Macro	A macro named <b>ZPLATFORM</b> must be defined to indicate the platform on which SIP applications will be compiled and run. Available values for <b>ZPLATFORM</b> are: <b>ZPLATFORM_VXWORKS</b> <b>ZPLATFORM_WIN32</b> <b>ZPLATFORM_LINUX</b> <b>ZPLATFORM_SOLARIS</b> The words may be like this: <b>/D ZPLATFORM=ZPLATFORM_WIN32</b> <b>or -DZPLATFORM=ZPLATFORM_LINUX</b>
Link Dir	Library directories including all necessary Juphoon product libraries should be indicated in compile environment. The words may like be this: <b>/libpath:"..\lib" or -L"..\lib"</b>
Link Library	ZOS lib must be linked at the link phase. The words may me like this: <b>zos.lib or -lzos</b> Other system libraries may be also linked. For example, the winsock library ws2_32.lib

	in Win32 environment and pthread library in POSIX environment.
--	--

表 9-1 Compile &amp; Link Flags

## 10. 附录

### 10.1 ZOS 编译选项

下表是 ZOS 平台中的主要编译选项，分为内部编译选项和外部编译选项。

编译选项	编译描述
ZPLATFORM	宏常量，需要在编译时指定，其取值范围为： <b>ZPLATFORM_WIN32</b> <b>ZPLATFORM_WINCE</b> <b>ZPLATFORM_VXWORKS</b> <b>ZPLATFORM_THREADX</b> <b>ZPLATFORM_LINUX</b> <b>ZPLATFORM_FREEBSD</b> <b>ZPLATFORM_SOLARIS</b> 如在 Win32 环境中，可设置编译选项： <b>-DZPLATFORM=ZPLATFORM_WIN32</b>
ZOS_SUPT_64BIT	表明是否需要支持 64 位对齐操作。在 ZOS 平台库中，根据该选项提供对应的库。而用户如需要 ZOS 支持 64 位对齐操作，则必须打开编译选项 如在 Win32 环境中，可设置编译选项： <b>-DZOS_SUPT_64BIT</b>
ZCPU_SPARC	指定 CPU 类型，如果不是 SPARC CPU 则不需要关心此编译选项
ZCPU_68K	指定 CPU 类型，如果不是 68K CPU 则不需要关心此编译选项
ZOS_SUPT_MEM_DBG	属于 ZOS 平台内部编译选项，表明是否支持内存调试模式，打开此选项，在内存申请释放等
ZOS_SUPT_DBG	属于 ZOS 平台内部编译选项，表明是否支持模块的调试功能接口
ZOS_SUPT_DUMP	属于 ZOS 平台内部编译选项，表明是否支持一般资源的 dump 调试功能接口。该选项会影响数据缓冲区、ABNF 错误跟踪
ZOS_SUPT_DUMP_TASK	属于 ZOS 平台内部编译选项，表明是否在一般资源的 dump 调试中记录任务相关信息
ZOS_SUPT_POOL_DUMP	属于 ZOS 平台内部编译选项，表明是否支持内存池中的数据块的 dump 调试
ZOS_NOTUSE_STDLIB	属于 ZOS 平台内部编译选项，表明是否使用标准字符库函数
ZOS_NOTUSE_STDIO	属于 ZOS 平台内部编译选项，表明是否使用标准输入输出库函数

表 10-1 ZOS 编译选项表

## 10.2 ZOS 基本宏常量

```
/* zos specific type maximum value */
#define ZMAXCPUID ZMAXULONG          /* maximum cpu id value */
#define ZMAXMODID ZMAXUSHORT         /* maximum module id value */
#define ZMAXINSTID ZMAXUSHORT        /* maximum instance id value */
#define ZMAXTASKID ZMAXULONG         /* maximum task id value */
#define ZMAXTIMERID ZMAXULONG        /* maximum timer id value */
#define ZMAXEVTID ZMAXULONG          /* maximum event id value */
#define ZMAXSAPID ZMAXULONG          /* maximum sap id value */
#define ZMAXENDPID ZMAXULONG         /* maximum endpoint id value */
#define ZMAXREASONID ZMAXULONG       /* maximum reason id value */

/* operating system type */
#define ZPLATFORM_WIN32 1             /* windows */
#define ZPLATFORM_WINCE 2            /* windows */
#define ZPLATFORM_VXWORKS 3          /* vxworks */
#define ZPLATFORM_THREADX 4          /* threadx */
#define ZPLATFORM_LINUX 5            /* linux */
#define ZPLATFORM_FREEBSD 6          /* freebsd */
#define ZPLATFORM_SOLARIS 7          /* solaris -- unix */

/* zos protocol type */
#define ZPROTOCOL_UNKNOWN 0          /* unknown protocol */
#define ZPROTOCOL_SDP 1              /* sdp protocol */
#define ZPROTOCOL_MGCP 2             /* mgcp text protocol */
#define ZPROTOCOL_MGCO_TXT 3         /* megaco text protocol */
#define ZPROTOCOL_MGCO_BIN 4         /* megaco binary protocol */
#define ZPROTOCOL_SIP 5              /* sip protocol */
#define ZPROTOCOL_RTSP 6             /* rtsp protocol */
#define ZPROTOCOL_H323 7             /* h.323 protocol */

/* zos data buffer type */
#define ZDBUF_TYPE_NULL 0            /* buffer type undefined */
#define ZDBUF_TYPE_BYTE 1           /* all data blocks are byte alignment */
#define ZDBUF_TYPE_STRUCT 2         /* all data blocks are 4 bytes alignment */

/* zos dlist maximum infinite size */
#define ZDLIST_INFINITE_SIZE ZMAXULONG

/* fsm next state needn't change, fsm keep state */
#define ZFSM_STA_NOCHANGE 0

/* fsm next state is invalid while input invalid event, fsm keep state */
#define ZFSM_STA_INVALID -1
```

```
/* fsm next state is error while input error event, fsm keep state */
#define ZFSM_STA_ERROR      -2

/* fsm decode function return failed */
#define ZFSM_LOCATE_FAIL    -1

#define ZFSM_OK             0 /* fsm run ok */
#define ZFSM_FAIL          -1 /* fsm run fail */
#define ZFSM_ERR_UNKNOWN_STA -2 /* fsm unknown state error */
#define ZFSM_ERR_UNKNOWN_EVNT -3 /* fsm unknown event error */
#define ZFSM_ERR_INVALID_EVNT -4 /* fsm invalid event error */
#define ZFSM_ERR_ERROR_EVNT -5 /* fsm error event error */
#define ZFSM_ERR_UNEXPECTED_EVNT -6 /* fsm unexpected event error */
#define ZFSM_ERR_DATA_ERROR -7 /* fsm object or event data error */

/* zos fsm dump stack size */
#define ZFSM_DUMP_STACK_SIZE 10

/* zos log option */
#define ZLOG_OPT_NULL      0x00000000 /* no option */
#define ZLOG_OPT_MUTEX    0x00000001 /* asynchronism option */
#define ZLOG_OPT_PRINT    0x00000002 /* print option */

/* zos log level */
#define ZLOG_LEVEL_NULL    0x00000000 /* null to be logged */
#define ZLOG_LEVEL_FATAL  0x00010000 /* fatal message to be logged */
#define ZLOG_LEVEL_ERROR  0x00020000 /* error message to be logged */
#define ZLOG_LEVEL_WARNING 0x00040000 /* warning message to be logged */
#define ZLOG_LEVEL_INFO   0x00080000 /* info message to be logged */
#define ZLOG_LEVEL_DBG    0x00100000 /* debug message to be logged */
#define ZLOG_LEVEL_ALL    0xFFFF0000 /* all message to be logged */
```

```
/* semaphore wait timeout macros */
#define ZSEMA_WAIT_FOREVER ZMAXULONG /* semaphore wait forever */
#define ZSEMA_NO_WAIT 0 /* semaphore no wait */

/* zos slist maximum infinite size */
#define ZSLIST_INFINITE_SIZE ZMAXULONG

/* zos string default max length of string */
#define ZSTRUL_MAXLEN 32
#define ZSTRUS_MAXLEN 16
#define ZSTRUC_MAXLEN 8

/* task options */
#define ZTASK_PREEMPT 0x00 /* enable task rescheduling */
#define ZTASK_NO_PREEMPT 0x01 /* disable task rescheduling */
#define ZTASK_TIMESLICE 0x02 /* enable round-robin selection */
#define ZTASK_NO_TIMESLICE 0x04 /* disable task rescheduling */

/* local cpu id */
#define ZCPUID_LOCAL 0

/* timer mode */
#define ZTIMER_MODE_CYCLE 0x01 /* cycle timer */
#define ZTIMER_MODE_NOCYCLE 0x02 /* not cycle timer */
#define ZTIMER_MODE_100MS 0x04 /* 100 ms timer */
```

### 10.3 ZOS 基本宏操作

```
/* get low/high byte of a word */
ZOS_GET_LOW_BYTE
ZOS_GET_HIGH_BYTE
/* get low/high word of a unsigned long */
ZOS_GET_LOW_WORD
ZOS_GET_HIGH_WORD
/* set low/high byte of a word */
ZOS_PUT_LOW_BYTE
ZOS_PUT_HIGH_BYTE
/* set low/high word of a word */
ZOS_PUT_LOW_WORD
ZOS_PUT_HIGH_WORD
ZOS_MAKE_WORD
ZOS_MAKE_LONG
/* check specific character property */
```

```
ZOS_ISALPHA
ZOS_ISUPPER
ZOS_ISLOWER
ZOS_ISDIGIT
ZOS_ISXDIGI
ZOS_ISSPACE
ZOS_ISPUNCT
ZOS_ISALNUM
ZOS_ISPRINT
ZOS_ISGRAPH
ZOS_ISCNTRL
ZOS_ISASCII
/* space or tab character test */
ZOS_ISBLANK
/* is whitespace */
ZOS_ISWS
ZOS_ISLWS
/* to upper/lower */
ZOS_TOUPPER
ZOS_TOLOWER
/* to ascii */
ZOS_TOASCII
/* is odd/even */
ZOS_ISODD
ZOS_ISEVEN
/* to byte type */
ZOS_TOZCHAR
ZOS_TOBYTE
/* to absolute value */
ZOS_TOABS
/* get member offset from structure */
ZOS_OFFSETOF

/* ZOS_ASSERT macro */
ZOS_ASSERT

/* zos calculate the hash key from case sensitive */
ZOS_HASH_GET_KEY_FROM_STR

/* zos calculate the hash key from non-case sensitive */
ZOS_HASH_GET_KEY_FROM_STR_NOCASE

/* zos register one module */
ZOS_MOD_REG
```

```
/* zos deregister one module */
ZOS_MOD_DEREG
/* zos start one module */
ZOS_MOD_START

/* zos message macors */
ZOS_MSG_GET_EVNT_TYPE
ZOS_MSG_GET_LEN
ZOS_MSG_GET_DATA
ZOS_MSG_GET_DATA_LEN
ZOS_MSG_GET_USED_LEN

/* zos size alignment */
ZOS_ALIGN

/* zos time to high resolution time */
ZOS_TIME_TO_HRTIME

/* zos string buffer print macros */
ZOS_PRINT_OUT_START
ZOS_PRINT_OUT_END
ZOS_PRINT_OUT_CHECK
ZOS_PRINT_PUT_STR
ZOS_PRINT_PUT_NSTR
ZOS_PRINT_PUT_SSTR
ZOS_PRINT_PUT_FMT1
ZOS_PRINT_PUT_FMT2
ZOS_PRINT_PUT_FMT3
ZOS_PRINT_PUT_FMT4

/* get maximum, minimum va
ZOS_MAX
ZOS_MIN

/* set operation */
ZOS_MASK
ZOS_SET
ZOS_CLR
ZOS_ISSET
ZOS_ZERO

/* counting and rounding *
ZOS_ROUNDDOWN
ZOS_ROUNDUP
```

```
ZOS_ROUNDUP2  
ZOS_ISPOWEROF2  
  
/* get table size */  
ZOS_GET_TABLE_SIZE
```

## 10.4 ZOS 基本函数指针

```
/* zos brick match function type for find_if */
typedef ZINT (*PFN_ZBKMATCH)(ZBKDATA zBkData, ZVOID *pCond);

/* zos brick action function type for for_each */
typedef ZVOID (*PFN_ZBKACTION)(ZBKDATA zBkData, ZVOID *pParm);

/* zos fsm state action about event */
typedef ZINT (*PFN_ZFSMACTION)(ZVOID *, ZVOID *);

/* zos fsm state table locate index from event */
typedef ZINT (*PFN_ZFSMLOCATE)(ZVOID *pFsmObj, ZINT iMajorEvt, ZINT iMinorEvt);

/* zos hash key and compare function pointer macros */
typedef ZINT (*PFN_ZHASHKEY)(ZULONG dwType, ZULONG dwParm1, \
                             ZULONG dwParm2, ZULONG *pdwHashKey);

typedef ZINT (*PFN_ZHASHCMP)(ZULONG dwEntry, ZULONG dwType, \
                             ZULONG dwParm1, ZULONG dwParm2);

/* zos instance initialize function */
typedef ZINT (*PFN_ZINSTINIT)(ZINSTID zInstId);

/* zos instance destroy function */
typedef ZVOID (*PFN_ZINSTDESTROY)(ZINSTID zInstId);

/* zos instance message process function */
typedef ZINT (*PFN_ZINSTMSGPROC)(ZVOID *pMsg);

/* task entry function */
typedef ZULONG (*PFN_ZTASKENTRY)(ZVOID *);

/* zos ring timer active function for callback */
typedef ZVOID (*PFN_ZRTIMERACTION)(ZTIMERID zTimerId,
                                   ZULONG dwTimerType, ZULONG dwParm);

/* zos print display functions. */
typedef ZINT (*PFN_ZPRINTDISP)(const ZCHAR *pcFormat, ...);

/* zos heap memory allocate function from specific os */
typedef ZVOID * (*PFN_ZHEAPMALLOC)(ZSIZE_T zSize);

/* zos heap memory free function from specific os */
typedef ZVOID (*PFN_ZHEAPFREE)(ZVOID *pMem);
```

```
/* zos timer active function for callback */
typedef ZVOID (*PFN_ZTIMERACTIVE)(ZTIMERID zTimerId, ZULONG dwTimerType,
                                   ZULONG dwParm);
```

## 10.5 ZOS 错误码

```
/* no error */
ZERR_NO

/* error code of task module */
ZERR_TASK_INIT
ZERR_TASK_NULLP
ZERR_TASK_INV
ZERR_TASK_TIMERCREATE
ZERR_TASK_TIMERDELETE
ZERR_TASK_TIMERSTART
ZERR_TASK_GETTASK
ZERR_TASK_GETSELFID
ZERR_TASK_STATE
ZERR_TASK_QUEUECREATE
ZERR_TASK_QUEUEGET
ZERR_TASK_QUEUEPOP
ZERR_TASK_QUEUEPUSH
ZERR_TASK_QTIMERINIT
ZERR_TASK_QTIMERCREATE
ZERR_TASK_QTIMERSTART
ZERR_TASK_SPAWN
ZERR_TASK_DELETE
ZERR_TASK_SUSPEND
ZERR_TASK_RESUME
ZERR_TASK_RESTART
ZERR_TASK_HASHINSERT
ZERR_TASK_MSGALLOC
ZERR_TASK_MSGSEND
ZERR_TASK_MSGLEN
ZERR_TASK_PRIORITYSET
ZERR_TASK_NULLENTRY
ZERR_TASK_CREATEMUTEX
ZERR_TASK_CREATTHREAD
```

```
/* error code of msg module */
ZERR_MSG_NULLP
ZERR_MSG_POOLCREATE
ZERR_MSG_POOLALLOC
ZERR_MSG_INVLEN
ZERR_MSG_INVSTENTRY

/* error code of time module */
ZERR_TIME_INIT
ZERR_TIME_NULLP
ZERR_TIME_GETEPOCH
ZERR_TIME_SETEPOCH
ZERR_TIME_MKTIME
ZERR_TIME_LOCALTIME
ZERR_TIME_GETDAYOFWK
ZERR_TIME_CLOCKTICK
ZERR_TIME_OPENCPUINFO
ZERR_TIME_CLOCKGETTIME
ZERR_TIME_CLOCKSETTIME
ZERR_TIME_CLOCKGETRES
ZERR_TIME_CONVERTSYSTIME
ZERR_TIME_SETSYSTIME
ZERR_TIME_SETTIMEOFDAY

/* error code of timer module */
ZERR_TIMER_INIT
ZERR_TIMER_NULLP
ZERR_TIMER_TASKPROCESS
ZERR_TIMER_CREATE
ZERR_TIMER_DELETE
ZERR_TIMER_START
ZERR_TIMER_STOP
ZERR_TIMER_TIMELEN

/* error code of string module */
ZERR_STRING_NULLP
ZERR_STRING_INV

/* error code of socket module */
ZERR_SOCKET_NULLP
ZERR_SOCKET_INV

/* error code of inet module */
ZERR_INET_NULLP
```

```
ZERR_INET_ADDR
ZERR_INET_WSSTARTUP
ZERR_INET_WSCLEANUP

/* error code of list module */
ZERR_LIST_NULLP
ZERR_LIST_FULL
ZERR_LIST_EMPTY
ZERR_LIST_NOTIN

/* error code of queue module */
ZERR_QUEUE_NULLP
ZERR_QUEUE_INVP
ZERR_QUEUE_MUTEXCREATE
ZERR_QUEUE_SEMACREATE
ZERR_QUEUE_MALLOC
ZERR_QUEUE_FULL

/* error code of qtimer module */
ZERR_QTIMER_NULLP
ZERR_QTIMER_INVNUM
ZERR_QTIMER_INVTMRID
ZERR_QTIMER_MALLOC
ZERR_QTIMER_MUTEXCREATE
ZERR_QTIMER_EMPTY
ZERR_QTIMER_STATE
ZERR_QTIMER_INSERT
ZERR_QTIMER_MSGALLOC
ZERR_QTIMER_MSGSEND

/* error code of rtimer module */
ZERR_RTIMER_NULLP
ZERR_RTIMER_INVNUM
ZERR_RTIMER_INVTMRID
ZERR_RTIMER_MALLOC
ZERR_RTIMER_MUTEXCREATE
ZERR_RTIMER_EMPTY
ZERR_RTIMER_STATE
ZERR_RTIMER_INSERT
ZERR_RTIMER_MSGALLOC
ZERR_RTIMER_MSGSEND
ZERR_RTIMER_BASSETIMER

/* error code of pool module */
```

```
ZERR_POOL_NULLP
ZERR_POOL_INV
ZERR_POOL_SIZE2OBIG
ZERR_POOL_HEAPALLOC
ZERR_POOL_INVBKTID
ZERR_POOL_INVMAGICID
ZERR_POOL_INVREDZONE
ZERR_POOL_INVSOMEID
ZERR_POOL_INVBKTSIZE
ZERR_POOL_FORBIDINC
ZERR_POOL_BKTCREATE
ZERR_POOL_BKTGRPCREATE
ZERR_POOL_MUTEXCREATE
ZERR_POOL_EMPTY
ZERR_POOL_INVCOUNT

/* error code of mutex module */
ZERR_MUTEX_NULLP

/* error code of module module */
ZERR_MODULE_NULLP
ZERR_MODULE_INVTASKID
ZERR_MODULE_NOTINIT
ZERR_MODULE_NOTREG
ZERR_MODULE_REGED
ZERR_MODULE_TASKSPAWN
ZERR_MODULE_MALLOC
ZERR_MODULE_NOQUEUE

/* error code of memory module */
ZERR_MEM_NULLP
ZERR_MEM_POOLCREATE
ZERR_MEM_POOLALLOC
ZERR_MEM_POOLGETSIZE
ZERR_MEM_INVSIZE
ZERR_MEM_MEMCHK

/* error code of hash module */
ZERR_HASH_NULLP
ZERR_HASH_MALLOC
ZERR_HASH_ITEMEXIST
ZERR_HASH_ITEMNOTEXIST
ZERR_HASH_FULL
```

```
/* error code of fsm module */
ZERR_FSM_NULLP
ZERR_FSM_UNKSTATE
ZERR_FSM_UNKEVNT
ZERR_FSM_INVEVNT
ZERR_FSM_ERREVNT

/* error code of dbuf module */
ZERR_DBUF_NULLP
ZERR_DBUF_INV
ZERR_DBUF_ALLOCDATA
ZERR_DBUF_POOLCREATE
ZERR_DBUF_INVTYPE
ZERR_DBUF_INVBLKSIZE
ZERR_DBUF_ALLOCRES
ZERR_DBUF_BUFREUSE
ZERR_DBUF_INVLEN
ZERR_DBUF_DBUFCREATE
ZERR_DBUF_INVOFFSET
ZERR_DBUF_INVSRCBUF
ZERR_DBUF_NOROOM
ZERR_DBUF_ADDDATA
ZERR_DBUF_NODATA
ZERR_DBUF_NOTINBUF

/* error code of abnf module */
ZERR_ABNF_NULLP
ZERR_ABNF_INV
ZERR_ABNF_MALLOC
ZERR_ABNF_INVBITARRAYSIZE
ZERR_ABNF_INVMAGICID
ZERR_ABNF_HASHCREATE
ZERR_ABNF_HASHINSERT
ZERR_ABNF_TKNFULL
ZERR_ABNF_PREADDDATA
ZERR_ABNF_PSTADDDATA
ZERR_ABNF_PREDELDATA
ZERR_ABNF_PSTDELDATA
ZERR_ABNF_DBUFCOPY
ZERR_ABNF_DBUFCAT

/* error code of idle module */
ZERR_IDLE_NULLP
ZERR_IDLE_INV
```

ZERR\_IDLE\_TASKSPAWN

## 10.6 ZOS 日志操作宏

```
/* zos log macros */  
#define ZOS_LOG_FATAL(_args) ZXX_LOG_FATAL(ZOS_LOGID, _args)  
#define ZOS_LOG_ERROR(_args) ZXX_LOG_ERROR(ZOS_LOGID, _args)  
#define ZOS_LOG_WARNING(_args) ZXX_LOG_WARNING(ZOS_LOGID, _args)  
#define ZOS_LOG_INFO(_args) ZXX_LOG_INFO(ZOS_LOGID, _args)  
#define ZOS_LOG_DBG(_args) ZXX_LOG_DBG(ZOS_LOGID, _args)
```

## 10.7 ZOS 单链表操作宏

```
#define ZOS_SLIST_SIZE(_slist) \
    ((_slist)->dwCount)

#define ZOS_SLIST_ISFULL(_slist) \
    ((_slist)->dwCount >= (_slist)->dwMaxNum)

#define ZOS_SLIST_ISEMPY(_slist) \
    ((_slist)->pstHead == ZNULL)

#define ZOS_SLIST_HEAD_NODE(_slist) (_slist)->pstHead
#define ZOS_SLIST_TAIL_NODE(_slist) (_slist)->pstTail

#define ZOS_SLIST_NODE_NEXT(_node) (_node)->pstNext
#define ZOS_SLIST_NODE_DATA(_node) (_node)->pData

#define FOR_ALL_NODE_IN_SLIST(_slist, _node) \
    for (_node = ZOS_SLIST_HEAD_NODE(_slist); \
        _node != ZNULL; \
        _node = ZOS_SLIST_NODE_NEXT(_node))

/* zos slist create */
#define ZOS_SLIST_CREATE(_slist, _size) \
    Zos_SlistCreate(_slist, _size)

/* zos slist delete */
#define ZOS_SLIST_DELETE(_slist) \
    Zos_SlistDelete(_slist)

/* zos slist node init */
#define ZOS_SLIST_NODE_INIT(_node, _data) do { \
    (_node)->pstNext = ZNULL; \
    (_node)->pData = (ZCHAR *)(_data); \
} while (0)

/* zos insert node before head in slist, seemed as a function */
#define Zos_SlistAdd2Head(_slist, _node) \
    Zos_SlistInsert(_slist, ZNULL, _node)

/* zos insert node after tail in slist, seemed as a function */
#define Zos_SlistAdd2Tail(_slist, _node) \
    Zos_SlistInsert(_slist, (_slist)->pstTail, _node)
```

```

/* zos insert node before head in slist */
#define ZOS_SLIST_ADD2HEAD(_slist, _node) \
    Zos_SlistInsert(_slist, ZNULL, _node)

/* zos insert node after tail in slist */
#define ZOS_SLIST_ADD2TAIL(_slist, _node) \
    Zos_SlistInsert(_slist, (_slist)->pstTail, _node)

/* zos insert node after previou node in slist */
#define ZOS_SLIST_INSERT(_slist, _prevnode, _node) \
    Zos_SlistInsert(_slist, _prevnode, _node)

/* zos dequeue node from the first node in slist */
#define ZOS_SLIST_DEQUEUE(_slist, _node) \
    _node = Zos_SlistDequeue(_slist)

/* zos remove one node */
#define ZOS_SLIST_REMOVE(_slist, _node) \
    Zos_SlistRemove(_slist, _node)

/* zos find node by index */
#define ZOS_SLIST_FIND_BY_INDEX(_slist, _index, _node) \
    _node = Zos_SlistFindByIndex(_slist, _index)

/* zos typedef slist with specific name */
#define ZOS_TYPEDEF_SLIST(_name) \
    /* single list node */ \
    typedef struct tag##_name##_LST_NODE \
    { \
        struct tag##_name##_LST_NODE *pstNext; /* next slist node */ \
        ST_##_name *pData; /* slist node data */ \
    } ST_##_name##_LST_NODE; \
    /* single list */ \
    typedef struct tag##_name##_LST \
    { \
        ZULONG dwMaxNum; /* maximum number of slist nodes */ \
        ZULONG dwCount; /* actual count of slist nodes */ \
        ST_##_name##_LST_NODE *pstHead; /* slist node head */ \
        ST_##_name##_LST_NODE *pstTail; /* slist node tail */ \
    } ST_##_name##_LST

```

```
/* typedef slist macros */
#define ZOS_TYPEDEF_SLIST_SIZE(_list) \
    ZOS_SLIST_SIZE((ST_ZOS_SLIST *)(_list))

#define ZOS_TYPEDEF_SLIST_ISFULL(_list) \
    ZOS_SLIST_ISFULL((ST_ZOS_SLIST *)(_list))

#define ZOS_TYPEDEF_SLIST_ISEMPY(_list) \
    ZOS_SLIST_ISEMPY((ST_ZOS_SLIST *)(_list))

#define ZOS_TYPEDEF_SLIST_HEAD_NODE(_list) \
    ZOS_SLIST_HEAD_NODE((ST_ZOS_SLIST *)(_list))

#define ZOS_TYPEDEF_SLIST_TAIL_NODE(_list) \
    ZOS_SLIST_TAIL_NODE((ST_ZOS_SLIST *)(_list))

#define ZOS_TYPEDEF_SLIST_NODE_NEXT(_node) \
    ZOS_SLIST_NODE_NEXT((ST_ZOS_SLIST_NODE *)(_node))

#define ZOS_TYPEDEF_SLIST_NODE_DATA(_node) \
    ZOS_SLIST_NODE_DATA((ST_ZOS_SLIST_NODE *)(_node))

#define FOR_ALL_NODE_IN_TYPEDEF_SLIST(_list, _node) \
    FOR_ALL_NODE_IN_SLIST((ST_ZOS_SLIST *)(_list), _node)

/* zos typedef slist create */
#define ZOS_TYPEDEF_SLIST_CREATE(_list, _size) \
    ZOS_SLIST_CREATE((ST_ZOS_SLIST *)(_list), _size)

/* zos typedef slist delete */
#define ZOS_TYPEDEF_SLIST_DELETE(_list) \
    ZOS_SLIST_DELETE((ST_ZOS_SLIST *)(_list))

/* zos typedef slist node init */
#define ZOS_TYPEDEF_SLIST_NODE_INIT(_node, _data) \
    ZOS_SLIST_NODE_INIT((ST_ZOS_SLIST_NODE *)(_node), _data)

/* zos insert node before head in typedef slist */
#define ZOS_TYPEDEF_SLIST_ADD2HEAD(_list, _node) \
    ZOS_SLIST_ADD2HEAD((ST_ZOS_SLIST *)(_list), _node)

/* zos insert node after tail in typedef slist */
#define ZOS_TYPEDEF_SLIST_ADD2TAIL(_list, _node) \
    ZOS_SLIST_ADD2TAIL((ST_ZOS_SLIST *)(_list), _node)
```

```

/* zos insert node after previou node in typedef slist */
#define ZOS_TYPEDEF_SLIST_INSERT(_list, _prevnode, _node) \
    ZOS_SLIST_INSERT((ST_ZOS_SLIST *)(_list), _prevnode, _node)

/* zos dequeue node from the first node in typedef slist */
#define ZOS_TYPEDEF_SLIST_DEQUEUE(_list, _node) \
    ZOS_SLIST_DEQUEUE((ST_ZOS_SLIST *)(_list), _node)

/* zos remove one node */
#define ZOS_TYPEDEF_SLIST_REMOVE(_list, _node) \
    Zos_SlistRemove((ST_ZOS_SLIST *)(_list), (_node))

/* zos find node by index */
#define ZOS_TYPEDEF_SLIST_FIND_BY_INDEX(_list, _index, _node) \
    ZOS_SLIST_FIND_BY_INDEX((ST_ZOS_SLIST *)_list, _index, _node)

```

## 10.8 ZOS 双链表操作宏

```

#define ZOS_DLIST_SIZE(_dlist) \
    ((_dlist)->dwCount)

#define ZOS_DLIST_ISFULL(_dlist) \
    ((_dlist)->dwCount >= (_dlist)->dwMaxNum)

#define ZOS_DLIST_ISEMPY(_dlist) \
    ((_dlist)->pstHead == ZNULL)

#define ZOS_DLIST_HEAD_NODE(_dlist) (_dlist)->pstHead
#define ZOS_DLIST_TAIL_NODE(_dlist) (_dlist)->pstTail

#define ZOS_DLIST_NODE_NEXT(_node) (_node)->pstNext
#define ZOS_DLIST_NODE_DATA(_node) (_node)->pData

#define FOR_ALL_NODE_IN_DLIST(_dlist, _node) \
    for (_node = ZOS_DLIST_HEAD_NODE(_dlist); \
        _node != ZNULL; \
        _node = ZOS_DLIST_NODE_NEXT(_node))

/* zos dlist create */
#define ZOS_DLIST_CREATE(_dlist, _size) \
    Zos_DlistCreate(_dlist, _size)

/* zos dlist delete */

```

```
#define ZOS_DLIST_DELETE(_dlist) \  
    Zos_DlistDelete(_dlist)  
  
/* zos dlist node init */  
#define ZOS_DLIST_NODE_INIT(_node, _data) do { \  
    (_node)->pstPrev = ZNULL; \  
    (_node)->pstNext = ZNULL; \  
    (_node)->pData = (ZCHAR *)_data; \  
} while (0)  
  
/* zos insert node before head in dlist, seemed as a function */  
#define Zos_DlistAdd2Head(_dlist, _node) \  
    Zos_DlistInsert(_dlist, ZNULL, _node)  
  
/* zos insert node after tail in dlist, seemed as a function */  
#define Zos_DlistAdd2Tail(_dlist, _node) \  
    Zos_DlistInsert(_dlist, (_dlist)->pstTail, _node)  
  
/* zos insert node before head in dlist */  
#define ZOS_DLIST_ADD2HEAD(_dlist, _node) \  
    Zos_DlistInsert(_dlist, ZNULL, _node)  
  
/* zos insert node after tail in dlist */  
#define ZOS_DLIST_ADD2TAIL(_dlist, _node) \  
    Zos_DlistInsert(_dlist, (_dlist)->pstTail, _node)  
  
/* zos insert node after the after previou node in dlist */  
#define ZOS_DLIST_INSERT(_dlist, _prevnode, _node) \  
    Zos_DlistInsert(_dlist, _prevnode, _node)
```

```

/*lint -save -e* */

/* zos dequeue node from the first node in dlist */
#define ZOS_DLIST_DEQUEUE(_dlist, _node) \
    _node = Zos_DlistDequeue(_dlist)

/* zos remove one node */
#define ZOS_DLIST_REMOVE(_dlist, _node) \
    Zos_DlistRemove(_dlist, _node)

/* zos find node by index */
#define ZOS_DLIST_FIND_BY_INDEX(_dlist, _index, _node) \
    _node = Zos_DlistFindByIndex(_dlist, _index)

/*lint -restore */

/* zos typedef dlist with specific name */
#define ZOS_TYPEDEF_DLIST(_name) \
    /* double list node */ \
    typedef struct tag##_name##_LST_NODE \
    { \
        struct tag##_name##_LST_NODE *pstNext; /* next dlist node */ \
        struct tag##_name##_LST_NODE *pstPrev; /* previous dlist node */ \
        ST_##_name *pData; /* dlist node data */ \
    } ST_##_name##_LST_NODE; \
    /* double list */ \
    typedef struct tag##_name##_LST \
    { \
        ZULONG dwMaxNum; /* maximum number of dlist nodes */ \
        ZULONG dwCount; /* actual count of dlist nodes */ \
        ST_##_name##_LST_NODE *pstHead; /* dlist node head */ \
        ST_##_name##_LST_NODE *pstTail; /* dlist node tail */ \
    } ST_##_name##_LST

/* typedef dlist macros */
#define ZOS_TYPEDEF_DLIST_SIZE(_list) \
    ZOS_DLIST_SIZE((ST_ZOS_DLIST *)(_list))

#define ZOS_TYPEDEF_DLIST_ISFULL(_list) \
    ZOS_DLIST_ISFULL((ST_ZOS_DLIST *)(_list))

#define ZOS_TYPEDEF_DLIST_ISEMPY(_list) \
    ZOS_DLIST_ISEMPY((ST_ZOS_DLIST *)(_list))

```

```
#define ZOS_TYPEDEF_DLIST_HEAD_NODE(_list) \
    ZOS_DLIST_HEAD_NODE((ST_ZOS_DLIST *)(_list))

#define ZOS_TYPEDEF_DLIST_TAIL_NODE(_list) \
    ZOS_DLIST_TAIL_NODE((ST_ZOS_DLIST *)(_list))

#define ZOS_TYPEDEF_DLIST_NODE_NEXT(_node) \
    ZOS_DLIST_NODE_NEXT((ST_ZOS_DLIST_NODE *)(_node))

#define ZOS_TYPEDEF_DLIST_NODE_DATA(_node) \
    ((ST_ZOS_DLIST_NODE *)(_node))->pData

#define FOR_ALL_NODE_IN_TYPEDEF_DLIST(_list, _node) \
    FOR_ALL_NODE_IN_DLIST((ST_ZOS_DLIST *)(_list), _node)

/* zos typedef dlist create */
#define ZOS_TYPEDEF_DLIST_CREATE(_list, _size) \
    ZOS_DLIST_CREATE((ST_ZOS_DLIST *)(_list), _size)

/* zos typedef dlist delete */
#define ZOS_TYPEDEF_DLIST_DELETE(_list) \
    ZOS_DLIST_DELETE((ST_ZOS_DLIST *)(_list))

/* zos typedef node init */
#define ZOS_TYPEDEF_DLIST_NODE_INIT(_node, _data) \
    ZOS_DLIST_NODE_INIT((ST_ZOS_DLIST_NODE *)_node, _data)

/* zos insert node before head in typedef dlist */
#define ZOS_TYPEDEF_DLIST_ADD2HEAD(_list, _node) \
    ZOS_DLIST_ADD2HEAD((ST_ZOS_DLIST *)(_list), _node)

/* zos insert node after tail in typedef dlist */
#define ZOS_TYPEDEF_DLIST_ADD2TAIL(_list, _node) \
    ZOS_DLIST_ADD2TAIL((ST_ZOS_DLIST *)(_list), _node)

/* zos insert node after the after previou node in typedef dlist */
#define ZOS_TYPEDEF_DLIST_INSERT(_dlist, _prevnode, _node) \
    ZOS_DLIST_INSERT((ST_ZOS_DLIST *)(_dlist), _prevnode, _node)

/* zos dequeue node from the first node in typedef dlist */
#define ZOS_TYPEDEF_DLIST_DEQUEUE(_list, _node) \
    ZOS_DLIST_DEQUEUE((ST_ZOS_DLIST *)(_list), _node)
```

```
/* zos remove one node */
#define ZOS_TYPEDEF_DLIST_REMOVE(_list, _node) \
    ZOS_DLIST_REMOVE((ST_ZOS_DLIST *)(_list), _node)

/* zos typedef dlist find node data on index location */
#define ZOS_TYPEDEF_DLIST_FIND_BY_INDEX(_list, _index, _node) \
    ZOS_DLIST_FIND_BY_INDEX((ST_ZOS_DLIST *)_list, _index, _node)
```