Nucleus 操作系统内部参考手册

注：由于译者水平有限，翻译难免有错误，

阅读时请参照英文文档，并欢迎批评指正！

Translator：LMF

E-mail:lmfjkemail◎sohu.com

北海　　2004.1.10

概要

第一章 介绍

这章描述了该手册的目的，提供了 Nucleus PLUS 内核的概述及它的结构。

章节

1.1 手册目的

Nucleus PLUS 以源码交付给用户使用。由于 Nucleus PLUS 源代码很大，典型用户很难对它加工。因此，该手册是用于帮助 Nucleus PLUS 用户了解源代码。

1.2 关于 Nucleus PLUS

Nucleus PLUS 是为实时嵌入式应用而设计的一个抢先式多任务操作系统内核，其 95%的代码是用 ANSIC 写成的，因此非常便于移植并能够支持大多数类型的处理器。从实现角度来看，Nucleus PLUS 是一组 C 函数库，应用程序代码与核心函数库连接在一起，生成一个目标代码，下载到目标板的 RAM 中或直接烧录到目标板的 ROM 中执行。在典型的目标环境中，Nucleus PLUS 核心代码区一般不超过 20K 字节大小。

1.3 Nucleus PLUS 的结构

Accelerated Technologys 软件开发实践证明具有帮助清晰，组件性，可靠性，可再用性及维护容易性。Nucleus PLUS 采用了软件组件的方法。每个组件具有单一而明确的目的，通常由几个 C 及汇编语言模块构成，提供清晰的外部接口，对组件的引用就是通过这些接口完成的。Nucleus PLUS 的软件组件的构成在接下来的章节做更加详细的描述。

第二章工具约定

这章描述了在 Nucleus PLUS 软件工具中软件代码的约定。

章节

2.1 组件

Nucleus PLUS 采用了软件组件的方法。每个组件具有单一而明确的目的，通常由几个 C 及汇编语言模块构成，提供清晰的外部接口，对组件的引用就是通过这些接口完成的。除了少数一些特殊情况外，不允许从外部对组件内的全局进行访问。由于采用了软件组件的方法，Nucleus PLUS 各个组件非常易于替换和复用。

2.2 组件组成

软件组成部分包含文件：数据类型的定义和常量文件，外部接口文件，一个或更多的 c 和汇编文件。具体文件如下列表：

文件名	功能说明	备注
XX_DEFS.H	定义常量和数据结构	
XX_EXTR.H	定义外部接口，是按照原形函数定义的	
XXD.C	定义静态和全局数据结构	
XXI..C	定义组件的初始化函数	
XXF.C	该文件的常量函数，提供对象的静态信息	
XXC.C	包含组件的核心函数	
XXCE.C	包含对核心函数错误检测的外壳函数	
XXS.C	定义补充的函数	
XXSE.C	定义查错函数	

注：xx 代表组件名的两个字母。每个组件不必每种文件类型都有。

格式

所有的软件代码文件都有相同的基本格式。文件的第一部分包含了标题的通用信息，作为序言调用的开端。第二部分包含了专用于内部数据声明和内部函数的说明。剩下的部分包含了文件的实际函数。

序言

序言的目的用于描述文件的目录。与 ATI 的自己的文件相关联，并提供了文件修订版的相关信息。

一个序言格式例子，如下所示：

```
/************************************************************************/
/*                                                                      */
/*    Copyright (c) 199x by Accelerated Technology, Inc.                */
/*                                                                      */
/*  PROPRIETARY RIGHTS of Accelerated Technology are  involved in       */
/*  the subject matter of this material.  All manufacturing,            */
/*  reproduction, use, and sales rights pertaining to this subject      */
/*  matter are  governed by the license agreement.  The recipient of    */
/*  this software implicitly accepts the terms of the license.          */
/*                                                                      */
/*                                                                      */
/************************************************************************/
/************************************************************************/
/* FILE NAME                                          VERSION           */
/*                                                                      */
/*    [name of this file]                               n.n             */
/*                                                                      */
/* COMPONENT                                                            */
/*                                                                      */
/*    [identifies the component]                                        */
/*                                                                      */
/* DESCRIPTION                                                          */
/*                                                                      */
/*    [general description of this file]                                */
/*                                                                      */
/* AUTHOR                                                               */
/*                                                                      */
/*    [author's name]                                                   */
/*                                                                      */
/* DATA STRUCTURES                                                      */
/*                                                                      */
/*    [global component data structures defined in this file]           */
/*                                                                      */
/* FUNCTIONS                                                            */
/*                                                                      */
/*    [functions defined in this file]                                  */
/*                                                                      */
/* DEPENDENCIES                                                         */
/*    [other file dependencies]                                         */
/*                                                                      */
/* HISTORY                                                              */
/*                                                                      */
/*    NAME          DATE      REMARKS                                   */
/*    [information about revising and verifying changes to this file]   */
/************************************************************************/
```

序言之后

序言后半部分是给常量、全局数据结构定义及内部组件－函数原型保留的。当然包括文件唯一的组件数据结构类型及外部接口的定义。

文件剩余部分

软件组件剩余部分由 C 或汇编语言函数组成。每个函数处于描述块的前面。一个函数的描述块格式如下所示：

```
/************************************************************************/
/*  FUNCTION                                                            */
/*                                                                      */
/*      [name of the function]                                          */
/*                                                                      */
/*  DESCRIPTION                                                         */
/*                                                                      */
/*      [general description of function]                               */
/*                                                                      */
/*  AUTHOR                                                              */
/*                                                                      */
/*      [author's  name]                                                */
/*                                                                      */
/*  CALLED BY                                                           */
/*                                                                      */
/*      [functions that call this function]                             */
/*                                                                      */
/*  CALLS                                                               */
/*                                                                      */
/*      [functions called by this function]                             */
/*                                                                      */
/*  INPUTS                                                              */
/*                                                                      */
/*      [inputs to the function]                                        */
/*                                                                      */
/*  OUTPUTS                                                             */
/*                                                                      */
/*      [outputs of this function]                                      */
/*                                                                      */
/*  HISTORY                                                             */
/*                                                                      */
/*  NAME            DATE        REMARKS                                 */
/*                                                                      */
/*  [information about revising and verifying changes to this function]*/
/*                                                                      */
/************************************************************************/
```

2. 3 命名约定

1、 组件命名

组件名字一般限于两个字符。组件名字作为构造组件的每个文件名的首两个字符。例如：

动态内存管理组件名：DM

构成 DM 的文件：

DM_DEFS.H
DM_EXTR.H
DMC.C
DMCE.C
DMI.C
DMF.C
DMD.C

2、**宏名命名** #define

宏名由下划线、大写字母和数字字符构成。最大支持长度为 31 个字符。此外一个宏定义

首 3 个字符是 CC_ ，其中“ CC ”与定义该宏的文件名的首 2 个字符相同。

例如：（对于 EX_DEFS ． H 文件）

```
#define EX_MY_CONSTANT 10
```

3、结构体命名

结构体名字由下划线、大写字母和数字字符构成。最大支持长度为 31 个字符。此外，一个结构体名的首 3 个字符是 CC_ ，其中“ CC ”与定义该结构体的文件名的首 2 个字符相同。

例如：（对于 EX_DEFS.H 文件）

```
 struct EX_MY_STRUCT
{
  int ex_member_a;
  int ex_member_b;
  int ex_member_c;
};
```

4、用 typedef 定义新结构体名命名

typedef 定义新结构体名字由下划线、大写字母和数字字符构成。最大支持长度为 31 个字符。此外，一个由 typedef 定义新结构体名字的首 3 个字符是 CC_ ，其中“ CC ”与定义由 typedef 命名结构体的文件名的首 2 个字符相同。

例如：（对于 EX_DEFS.H 文件）

```
typedef  struct EX_MY_STRUCT
 {
   int ex_member_a;
   int ex_member_b;
   int ex_member_c;
 }; EX_MY_MY_TYPEDEF
```

5、结构体成员命名

结构体成员名字由下划线、大写字母和数字字符构成，最大支持长度为 31 个字符。此外，一个结构体成员名字的首 3 个字符是 CC_ ，其中“ cc ”与包含该结构体定义的文件“ cc_defs ”的首 2 个字符相同。

例如：（对于 EX_DEFS.H 文件）

```
 struct EX_MY_STRUCT
{
  int ex_member_a;
  int ex_member_b;
  int ex_member_c;
};
```

6、全局变量命名

Nucleus PLUS 全局变量名由下划线以及紧跟着每个下划线的一个大写字母、小写字母、数字字符组成，最大支持长度为 31 个字符。此外，一个全局变量名字的首 3 个字符是 CCC ，其中“ CCC ”与包含实际变量描述的文件“ ccc.C ”的首 3 个字符相同。

例如：（对于 EXD.C 文件）

```
int EXD_Global_Integer;
```

7、局部变量命名

局部变量名由小写字母、有可能是下划线或数字字符组成，最大支持长度为 31 个字符。局部变量名字没有要求包含定义它们的文件的首 3 个字符。

例如：（对于 EXD.C 文件）

```
/* Assume the following declaration is inside a function. */
   int i;
```

8、函数命名

Nucleus PLUS 函数名由下划线以及紧跟着每个下划线的一个大写字母、小写字母、数字字符组成，最大支持长度为 31 个字符。此外，一个函数名的首 3 个字符与包含该函数定义的文件名相同。

例如：（对于 EXD.C 文件）

```
void EXD_My_Function(unsigned int i)
{
  .
  .
 }
```

2. 4 缩进

以 4 个空格为缩进的基本单位。函数描述，变量描述，条件编译结构起始于第一列。实际指令起始于第四列。注：花括号{ }间以行相隔。花括号{和前一行具有相同的缩减。同时花括号}行和前个花括号{对齐。

例如：（对于 EXD.C 文件）

```
void EXD_Example_Function(int i, int b)
 {
 unsigned int a;
 char b;
 /* Actual instruct tions star. */
      i= 0;
    while (i < 100)
    {
     /* Increment i. */
      i= i + 1;
    }
 }
```

2. 5 注释

注释部分是 Nucleus PLUS 源代码具有最重要的特征之一。不但意义深长而且表达丰富。这些是 ATI 公司软件注释部分两大最重要的类型。注释的第一种类型起始于当前缩进，而注释的第二种类型起始于第 45 列。

例如：

```
/* This is the first type of meaningful comment. */
 i= 10;
  j ++;    /* This is the second type of comment. */
```

第三章 软件概述

该章包含了与 Nucleus PLUS 软件相关的各种各样重要的讨论。

章节

3.1 基本用法

Nucleus PLUS是一组C函数库。 实时Nucleus PLUS应用将连接Nucleus PLUS 库文件。Nucleus PLUS 库文件：NUCLEUS.LI，它用于建立批处理文件 BUILD_LI.BAT，开发工具将使用 BUILD_LI.BAT 文件

操作方式

处理机体系结构有管态和用户态两种操作方式。Nucleus PLUS 应用层任务运行在管态模式。这便于应用层任务直接调用操作系统服务的特权指令。这种方法有效地降低服务程序的调用的频繁性，也易于执行。当然，这种方法允许任务访问任何事件。

应用程序初始化

Application_Initialize。系统开始运行时，它用于创建任务、队列、其他系统目标。在某些目标环境中，低级系统初始化文件 INT.S 、INT.ASM 、INT.SRC 根据要求修改。这些文件初始化系统时钟中断、有效内存、其他内部处理机或专用目标板。

包含文件 NUCLEUS.H

NUCLEUS.H 包含数据类型、常量、原始函数定义。

3.2 数据类型

NUCLEUS PLUS 操作系统定义的数据类型如下：

数据类型名	功能说明	
UNSIGNED	用于定义 32 位无符号整型变量	常用于定义无符号长型 C 类数据类型。
SIGNED	用于定义 32 带符号整型变量	
OPTION	最小的数据类型	
DATA_ELEMENT	与 OPTION 相同	
UNSIGNED_CHAR	用于定义 8 位无符号字符变量	
CHAR	用于定义 8 位带符号字符变量	
STATUS	用于定义有符号的整型变量	
INT	用于定义一个字的整型数据变量	
VOID	用于定义无返回值变量	
UNSIGNED_PTR	用于定义无符号数据类型的指针	
BYTE_PTR	用于定义 8 位无符号字符变量的指针	

3.3 服务调用映像

NUCLEUS PLUS 主要包含的文件 NUCLEUS.H,它包含了在 NUCLEUS PLUS 参考手册中定义函数的原型。但是 NU_*函数并不真实存在。对于 NUCLEUS PLUS 内核服务，在调用真实函数服务之前，根据用户的请求，确实存在一个函数在工作和一个外壳（shell）在检测错误。根据

错误检测条件的定义 NU_NO_ERROR_CHECKING，Nucleus PLUS 内核服务被映像，并通过宏的替换来给于合适的基本函数。在不请求时，这完全有利于消除错误检测。

3.3.1 错误检测

如果 NU_NO_ERROR_CHECKING 标记未定义（默认条件下），定义在 NUCLEUS PLUS 参考手册中的服务 NU_*调用被影像到下列内部函数。

NUCLEUS PLUS 服务函数	内部函数
NU_Activate_HISR	TCCE_Activate_HISR
NU_Allocate_Memory	DMCE_Allocate_Memory
NU_Allocate_Partition	DMCE_Allocate_Partition
NU_Broadcast_TO_Mailbox	MBSE_Broadcast_TO_Mailbox
NU_Broadcast_TO_Pipe	PISE_Broadcast_TO_Pipe
NU_Broadcast_TO_Queue	QUSE_Broadcast_TO_Queue
NU_Change_Preemption	TCSE_Change_Preemption
NU_Change_Priority	TCSE_Change_Preemption
NU_Change_Time_Slice	TCSE_Change_Time_Slice
NU_Check_Stack	TCT_Check_Stack
NU_Control_Interrupts	TCT_Control_Interrupts
NU_Control_Signals	TCSE_Control_Signals
NU_Control_Timer	TMSE_Control_Timer
NU_Create_Driver	IOCE_Create_Driver
NU_Create_Event_Group	EVCE_Create_Event_Group
NU_Create_HISR	TCCE_Create_HISR
NU_Create_Mailbox	MBCE_Create_Mailbox
NU_Create_Memory_Pool	DMCE_Create_Memory_Pool
NU_Create_Partition_Pool	PMCE_Create_Partition_Pool
NU_Create_Pipe	PICE_Create_Pipe
NU_Create_Queue	QUCE _Create_Queue
NU_Create_Semaphore	SMCE_Create_Semaphore
NU_Current_HISR_Pointer	TCF_Current_HISR_Pointer
NU_Current_Task_Pointer	TCC_Current_Task_Pointer
NU_Deallocate_Memory	DMCE_Deallocate_Memory
NU_Deallocate_Partition	PMCE_Deallocate_Partition
NU_Delete_Driver	IOCE_Delete_Driver
NU_Delete_Event_Group	EVCE_Delete_Event_Group
NU_Delete_HISR	TCCE_Delete_HISR
NU_Delete_Mailbox	MBCE_Delete_Mailbox
NU_Delete_Memory_Pool	DMCE_Delete_Memory_Pool
NU_Delete_Partition_Pool	PMCE_Delete_Partition_Pool
NU_Delete_Pipe	PICE_Delete_Pipe
NU_Delete_Queue	QUCE_Delete_Queue
NU_Delete_Semaphore	SMCE_Delete_Semaphore
NU_Delete_Task	TCCE_Delete_Task
NU_Delete_Timer	TMSE_Delete_Timer
NU_Disable_History_Saving	HIC_Disable_History_Saving
NU_Driver_Pointers	IOF_Driver_Pointers

NU_Enable_History_Saving	HIC_Enable_History_Saving
NU_Established_Drivers	IOF_Established_Drivers
NU_Established_Event_Groups	EVF_Established_Event_Groups
NU_Established_HISRs	TCF_Established_HISRs
NU_Established_Mailboxes	MBF_Established_Mailboxes
NU_EsLablished_Memory_Pools	DMF_Established_Memory_Pools
NU_Established_Fartition_Fools	PMF_Established_PartiLion_Pools
NU_Established_Pipes	PIF_Established_Pipes
NU_Established_Queues	QUF_Established_Queues
NU_Established_Semaphores	SMF_Established_Semaphores
NU_Established_Tasks	TCF_Established_Tasks
NU_Established_Timers	TMF_Established_Timers
NU_Event_Group_Information	EVF_Event_Group_Information
NU_Event_Group_Pointers	EVF_Event_Group-Pointers
NU_HISR_Information	TCF_HISR_Information
NU_HISR_Pointers	TCF_HISR_Pointers
NU_License_Information	LIC_License_Information
NU_Local_Control_Interrupts	TCT_Local_Control_Interrupts
NU_Mailbox_Information	MBF_Mailbox_Information
NU_Mailbox_Pointers	MBF_Mailbox_Pointers
NU_Make_History_Entry	HIC_Make_History_Entry_Service
NU_Memory_Pool_Information	DMF_Memory_Pool_Information
NU_Memory_Pool_Pointers	DMF_Memory_Pool_Pointers
NU_Obtain_Semaphore	SMCE_Obtain_Semaphore
NU_Partition_Pool_Information	PMF_Partition_Pool_Information
NU_Partition_Pool_Pointers	PMF_Partition_Pool_Pointers
NU_Pipe_Information	PIF_Pipe_Information
NU_Pipe_Pointers	PIF_Pipe_Pointers
NU_Protect	TCT_Protect
NU_Queue_Information	QUF_Queue_Information
NU_Queue_Pointers	QUF_Queue_Pointers
NU_Receive_From_Mailbox	MBCE_Receive_From_Mailbox
NU_Receive_From_Pipe	PICE_Receive_Frnm_Pipe
NU_Receive_From_Queue	QUCE_Receive_From_Queue
NU_Receive_Signals	TCSE_Receive_Signals
NU_Register_LISR	TCC_Register_LISR
NU_Register_Signal_Handler	TCSE_Register_Signal_Handler
NU_Release_Information	RLC_Release_Information
NU_Release_semaphore	SMCE_Release_Semaphore
NU_Relinquish	TCCE_Relinquish
NU_Request_Driver	IOCE_Request_Driver
NU_Reset_Mailbox	MBSE_Reset_Mailbox
NU_Reset_Pipe	PISE_Reset_Pipe
NU_Reset_Queue	QUSE_Reset_Queue
NU_Reset_Semaphore	SMSR_Reset_Semaphore

NU_Reset_Task	TCCE_Reset_Task
NU_Reset_Timer	TMSE_Reset_Timer
NU_Restore_Interrupts	TCT_Restore_Interrupts
NU_Resume_Driver	IOCE_Resume_Driver
NU_Resume_Task	TCCE_Resume_Service
NU_Retrieve_Clock	TMT_Retrieve_Clock
NU_Retrieve_Events	EVCE_Retrieve_Events
NU_Retrieve_History_Entry	HIC_Retrieve_History_Entry
NU_Semaphore_Information	SMF_Semaphore_Information
NU_Semaphore_Pointers	SMF_Semaphore_Pointers
NU_Send_Signals	TCSE_Send_Signals
NU_Send_To_Front_Of_Pipe	PISE_Send_To_Front_Of_Pipe
NU_Send_To_Front_Of_Queue	QUSE_Send_To_Front_Of_Queue
NU_Send_To_Mailbox	MBCE_Send_To_Mailbox
NU_Send_To_Pipe	PICE_Send_To_Pipe
NU_Send_To_Queue	QUCE_Send_To_Queue
NU_Set_Clock	TMT_Set_Clock
NU_Set_Events	EVCE_Set_Events
NU_Setup_Vector	INT_Setup_Vector
NU_Sleep	TCCE_Task_Sleep
NU_Suspend_Driver	IOCE_Suspend_Driver
NU_Suspend_Task	TCCE_Suspend_Service
NU_Task_Information	TCF_Task_Information
NU_Task_Pointers	TCF_Task_Pointers
NU_Terminate_Task	TCCE_Terminate_Task
NU_Timer_Information	TMF_Timer_Information
NU_Timer_Pointers	TMF_Timer_Pointers
NU_Unprotect	TCT_Unprotect

3.3.2 无错误检测

如果 NU_NO_ERROR_CHECKING 标记未定义（通常以－D 命令编译），定义在 NUCLEUS PLUS 参考手册中的服务 NU_*调用被影像到下列内部函数。

NUCLEUS PLUS 服务函数	内部函数
NU_Activate_HISR	TCC_Activate_HISR
NU_Allocate_Memory	DMC_Allocate_Memory
NU_Allocate_Partition	DMC_Allocate_Partition
NU_Broadcast_TO_Mailbox	MBS_Broadcast_TO_Mailbox
NU_Broadcast_TO_Pipe	PIS_Broadcast_TO_Pipe
NU_Broadcast_TO_Queue	QUS_Broadcast_TO_Queue
NU_Change_Preemption	TCS_Change_Preemption
NU_Change_Priority	TCS_Change_Preemption
NU_Change_Time_Slice	TCS_Change_Time_Slice
NU_Check_Stack	TCT_Check_Stack
NU_Control_Interrupts	TCT_Control_Interrupts
NU_Control_Signals	TCS_Control_Signals

NU_Control_Timer	TMS_Control_Timer
NU_Create_Driver	IOC_Create_Driver
NU_Create_Event_Group	EVC_Create_Event_Group
NU_Create_HISR	TCC_Create_HISR
NU_Create_Mailbox	MBC_Create_Mailbox
NU_Create_Memory_Pool	DMC_Create_Memory_Pool
NU_Create_Partition_Pool	PMC_Create_Partition_Pool
NU_Create_Pipe	PIC_Create_Pipe
NU_Create_Queue	QUC_Create_Queue
NU_Create_Semaphore	SMC_Create_Semaphore
NU_Current_HISR_Pointer	TCF_Current_HISR_Pointer
NU_Current_Task_Pointer	TCC_Current_Task_Pointer
NU_Deallocate_Memory	DMC_Deallocate_Memory
NU_Deallocate_Partition	PMC_Deallocate_Partition
NU_Delete_Driver	IOC_Delete_Driver
NU_Delete_Event_Group	EVC_Delete_Event_Group
NU_Delete_HISR	TCC_Delete_HISR
NU_Delete_Mailbox	MBC_Delete_Mailbox
NU_Delete_Memory_Pool	DMC_Delete_Memory_Pool
NU_Activate_HISR	TCC_Activate_HISR
NU_Allocate_Memory	DMC_Allocate_Memory

NU_Delete_Partition_Pool	PMC_Delete_Partition_Pool
NU_Delete_Pipe	PIC_Delete_Pipe
NU_Delete_Queue	QUC_Delete_Queue
NU_Delete_Semaphore	SMC_Delete_Semaphore
NU_Delete_Task	TCC_Delete_Task
NU_Delete_Timer	TMS_Delete_Timer
NU_Disable_History_Saving	HIC_Disable_History_Saving
NU_Driver_Pointers	IOF_Driver_Pointers
NU_Enable_History_Saving	HIC_Enable_History_Saving
NU_Established_Drivers	IOF_Established_Drivers
NU_Established_Event_Groups	EVF_Established_Event_Groups
NU_Established_HISRs	TCF_Established_HISRs
NU_Established_Mailboxes	MBF_Established_Mailboxes
NU_EsLablished_Memory_Pools	DMF_Established_Memory_Pools
NU_Established_Fartition_Fools	PMF_Established_PartiLion_Pools
NU_Established_Pipes	PIF_Established_Pipes
NU_Established_Queues	QUF_Established_Queues
NU_Established_Semaphores	SMF_Established_Semaphores
NU_Established_Tasks	TCF_Established_Tasks
NU_Established_Timers	TMF_Established_Timers
NU_Event_Group_Information	EVF_Event_Group_Information
NU_Event_Group_Pointers	EVF_Event_Group_Pointers
NU_HISR_Information	TCF_HISR_Information

NU_HISR_Pointers	TCF_HISR_Pointers
NU_License_Information	LIC_License_Information
NU_Local_Control_Interrupts	TCT_Local_Control_Interrupts
NU_Mailbox_Information	MBF_Mailbox_Information
NU_Mailbox_Pointers	MBF_Mailbox_Pointers
NU_Make_History_Entry	HIC_Make_History_Entry_Service
NU_Memory_Pool_Information	DMF_Memory_Pool_Information
NU_Memory_Pool_Pointers	DMF_Memory_Pool_Pointers
NU_Obtain_Semaphore	SMCE_Obtain_Semaphore
NU_Partition_Pool_Information	PMF_Partition_Pool_Information
NU_Partition_Pool_Pointers	PMF_Partition_Pool_Pointers
NU_Pipe_Information	PIF_Pipe_Information
NU_Pipe_Pointers	PIF_Pipe_Pointers
NU_Protect	TCT_Protect
NU_Queue_Information	QUF_Queue_Information
NU_Queue_Pointers	QUF_Queue_Pointers
NU_Receive_From_Mailbox	MBC_Receive_From_Mailbox

NU_Receive_From_Pipe	PIC_Receive_Frnm_Pipe
NU_Receive_From_Queue	QUC_Receive_From_Queue
NU_Receive_Signals	TCS_Receive_Signals
NU_Register_LISR	TCC_Register_LISR
NU_Register_Signal_Handler	TCS_Register_Signal_Handler
NU_Release_Information	RLC_Release_Information
NU_Release_semaphore	SMC_Release_Semaphore
NU_Relinquish	TCC_Relinquish
NU_Request_Driver	IOC_Request_Driver
NU_Reset_Mailbox	MBS_Reset_Mailbox
NU_Reset_Pipe	PIS_Reset_Pipe
NU_Reset_Queue	QUS_Reset_Queue
NU_Reset_Semaphore	SMS_Reset_Semaphore
NU_Reset_Task	TCC_Reset_Task
NU_Reset_Timer	TMS_Reset_Timer
NU_Restore_Interrupts	TCT_Restore_Interrupts
NU_Resume_Driver	IOC_Resume_Driver
NU_Resume_Task	TCC_Resume_Service
NU_Retrieve_Clock	TMT_Retrieve_Clock
NU_Retrieve_Events	EVC_Retrieve_Events
NU_Retrieve_History_Entry	HIC_Retrieve_History_Entry
NU_Semaphore_Information	SMF_Semaphore_Information
NU_Semaphore_Pointers	SMF_Semaphore_Pointers
NU_Send_Signals	TCS_Send_Signals
NU_Send_To_Front_Of_Pipe	PIS_Send_To_Front_Of_Pipe
NU_Send_To_Front_Of_Queue	QUS_Send_To_Front_Of_Queue
NU_Send_To_Mailbox	MBC_Send_To_Mailbox

NU_Send_To_Pipe	PIC_Send_To_Pipe
NU_Send_To_Queue	QUC_Send_To_Queue
NU_Set_Clock	TMT_Set_Clock
NU_Set_Events	EVC_Set_Events
NU_Setup_Vector	INT_Setup_Vector
NU_Sleep	TCC_Task_Sleep
NU_Suspend_Driver	IOC_Suspend_Driver
NU_Suspend_Task	TCC_Suspend_Service
NU_Task_Information	TCF_Task_Information
NU_Task_Pointers	TCF_Task_Pointers
NU_Terminate_Task	TCC_Terminate_Task
NU_Timer_Information	TMF_Timer_Information
NU_Timer_Pointers	TMF_Timer_Pointers
NU_Unprotect	TCT_Unprotect

3.4 条件编译

1、库文件条件式编译标志

编译标志	说明
NU_ENABLE_HISTORY	在指定文件中允许存储历史事件，只有**C.C 格式的文件受到该属性的影响。
NU_ENABLE_STACK_CHECK	在指定文件中每个函数的开始允许堆栈检测，只有**C.C 格式的文件受到该标志的影响
NU_ERROR_STRING	在发生严重的系统错误时，允许产生 ASCII 错误字符串。该标志应用于 ERD.C、ERI.C 和 ERC.C 文件中。
NU_NO_ERROR_CHECKING	只在 TMI.C 文件创建时钟高级中断 HISR 时，禁止错误检查
NU_DEBUG	该标志允许直接检查所有的 NUCLEUS PLUS 操作系统数据结构
NU_INLINE	在对直接插入代码处理过程中取代某些连接列表，目的提高执行效率。该标志应用于**C.C 或**S.C 格式文件。

2、库文件条件式值

除了外部定义条件编译标志外，还有条件编译值（定义在 NUCLEUS.H 中）。这些条件编译值是为每个端口建立的。改变这些值（除了 R1、R2、R3、R4 的值）将产生警告。条件编译值如下列表：

编译标志	说明
NU_POINTER_ACCESS	该值指定分割存储器存取通道数，用于读取和存储数据指针。
PAD_1	该值指定在一个结构体中的单字符之后应该填充多少个字节
PAD_2	该值指定在一个结构体中的两个连续的字符之后应该填充多少个字节。
PAD_3	该值指定在一个结构体中的三个连续的字符之后应该填充多少个字节。
R1、R2、R3、R4	该值用于存放NUCLEUS PLUS中频繁使用的变量之前的寄存器修改值。R1被用于修改最频繁使用的变量。By defining any of these to register.the corresponding variable in the source code is assigned register status.

3、应用条件标志

当编译应用程序时，这里有几个条件标志有效。通过编译命令行选项定义NUCLEUS PLUS

服务参数NU_NO_ERROR_CHECKING来禁止NUCLEUS PLUS应用基础进行错误检测。这将导致在一个实时执行中时间真正地增加，并降低代码大小。

在编译过程中，通过定义NU_DEBUG选项使定义在*NUCLEUS.H*中的应用数据结构被直接影像到内部的NUCLEUS PLUS数据结构里。在一个源级调试环境中，这也允许用户直接检查每个NUCLEUS PLUS数据结构的内部。如果使用了NU_DEBUG选项，能够非常好地重建所有的NUCLEUS PLUS源代码。

3.5 附加环境

处理器和开发工具取决NUCLEUS PLUS中四个绝对的文件。其中这三个文件(INT.?,TCT.?,and TMT.?)是用汇编写的。这些文件为基本的目标环境提供低级、运行期环境。第三个文件NUCLEUS.H直接或非直接包含在系统的所有文件中。该文件定义了变量数据、类型以及其他处理器和开发工具指定的信息。

1、初始化

INT（.S、ASM、SRC）文件负责低级初始化、中断向量表存取服务。它也包含缺省的中断服务例程处理程序。函数 INT_Initialize 专用于已知的目标板的初始化。例如，目标处理机不能产生内部时钟中断，就应建立目标板专用时钟中断。也就是说尽管他们使用相同的处理机体系结构，但是对于不同的目标板有必要对初始化文件 INT 作相应的修改。

2、线程控制

TCT(.S、.ASM、.SRC) 文件主要负责线程与系统间传输。一个线程定义任何一个 NUCLEUS PLUS 操作系统任务和高级中断。该文件包含所有在任务和高级中断之间能执行上下毫不相关事件的代码。另外，它还包含处理保护冲突和任务信号的代码。

3、定时器管理

TMS(.S、.ASM、.SRC) 文件主要负责 NUCLEUS PLUS 操作系统时钟的服务，其中包含中断定时处理程序。在许多端口中，当没有时钟发出时，中断定时处理程序用于处理频繁的低级操作。

4、NUCLEUS PLUS 操作系统头文件

NUCLEUS.H 文件通过 NUCLEUS PLUS 操作系统源文件直接或间接包含。以 NUCLEUS PLUS 操作系统服务或数据类型为参考编写应用文件必须包含 NUCLEUS.H 文件。该文件定义变量数据类型、中断关闭或开放值、中断向量数目、系统控制块大小以及其他目标指定信息。

3.6 版本控制

操作系统版本控制文件 RLD.C。该系统版本是在 RLD.C 文件中通过 ASCII 码字符串 RLD_Release_String 来定义的。该版本包含了通用 C 语言源代码的当前版本以及目标板指定代码的版本。例如，基于 DOS 1.1 版本/以通用 C 代码版本 2.2 编写的 Borland 目标指定代码。

Copyright (c) 199x ATT - Nucleus PLUS - DOS Rot-land C Version 1.1 .G2.2 ”

NUCLEUS PLUS 的每个文件也都包含版本信息。版本信息包含在每个文件的头块，它关联到指定文件的版本。在许多例子中，在文件头的版本与系统版本很不相同。在文件中的每个函数也在它的头文件中包含版本信息。版本信息指定邻近的每个函数基部说明了函数的做了那些改动以及将应用那些版本的文件。

第四章 组件描述

章节

4.1 公共服务（CS）P39

公共服务区（CS）负责提供其他 NUCLEUS PLUS 操作系统组件与链接表的设备连接。在其他系统数据结构中都包含每个公共服务节点。

4.1.1 公共服务文件

公共服务部分（CS）有三个文件组成，具体说明如下：

文件	说明描述
CS_DEFS.H	该文件包含了 CS 专用的常量和数据结构的定义
CS_EXTR.H	该文件定义了 CS 所有的外部接口
CSC.C	该文件包含了 CS 的所有核心函数。核心函数负责对基本放置列表和移动列表处理。

4.1.2 公共服务数据结构

公共服务控制块

公共服务控制块 CS_NODE 包含了连接公共服务节点的前一个和后一个指针以及对公共服务请求处理的其他字段。

字段声明

```
struct CS_NODE_STRUCT  *cs_previous          /*前一个指针*/
struct CS_NODE_STRUCT  *cs_next              /*后一个指针*/
DATA_ELEMENT cs_priority                   /*代表任务或高级中断的优先级*/
DATA_ELEMENT cs_padding[PAD-1]/*调整公共服务结构边界，某些端口该字段不用。*/
```

4.1.3 公共服务函数

这部分是对公共服务区（CS）的函数作简要的概述。推荐复习实际的代码能获的更多的信息。

CSC_Place_On_List P41

功能：该函数在指定的双连循环表末端放置指定的结点。

语法：VOID CSC_Place_On_List(CS_NODE **head, CS_NODE *new_node)

调用函数：无

CSC_Priority_Place_On_List P41

功能：该函数基于优先级在列表中放置指定结点。在列表中，该结点放置在相对于它具有等同或更高级别的其他结点之后。注意：低数字值表示更高级别。

语法：VOID CSC_Priority_ Place_On_List(CS_NODE **head, CS_NODE *new_node)

调用函数：无

CSC_Remove_From_List P41

功能：从指定的链接表中移除指定结点。

语法：VOID CSC_Remove_From_List(CS_NODE **head, CS_NODE *node)

调用函数：无

4.2 初始化（IN）P42

初始化组件负责对 NUCLEUS PLUS 操作系统的初始化。其中有两个初始化步骤：首先是对独立目标板初始化，接着是初始化 NUCLEUS PLUS 操作系统的每一组件。最后调用初始化函数 Appliation_Initialize（该函数具体操作用户定义）。初始化完成后，转向执行任务循环调度函数 TCT_Shedule 函数。请参阅 NUCLEUS PLUS 操作系统参考手册第三节，了解更多的初始化信息。

4.2.1 初始化文件

初始化模块（IN）由 3 个文件组成。每个初始化模块的源文件定义如下：

文件	说明描述
IN_EXTR.H	该文件定义了 IN 所有外部接口。
INC.C	该文件包含了 IN 的核心函数，核心函数用于处理基本系统的初始化。
INT.[S,ASM,SRC]	该文件包含了 IN 所有目标独立函数。

4.2.2 初始化函数

这部分是对初始化（IN）的函数作简要的概述。推荐复习实际的代码能获的更多的信息。

INC_Initialize

功能：函数 INC_Initialize( )主要初始化系统各功能模块。在系统初始化完成后，调用应用程序初始化函数 Application_Initialize( )，在所有的初始化完成后，函数 Application_Initialize( )调用任务调度列表 TCT_Schedule( )，开始调度任务。

语法：VOID INC_Initialize(VOID *first_available_memory)

调用函数：

```
Application_Initialize( )
RLC_Release_Information( )
LIC_License_Information( )
ERI_Initialize( )
HII_Initialize( )
TCI_Initialize( )
MBI_Initialize( )
QUI_Initialize( )
PI1_Initialize( )
SMI_Initialize( )
EVI_Initialize( )
PMI_Initialize( )
DMI_Initialize( )
```

TMI_Initialize( )
IOI_Initialize( )
TCT_Schedule( )

INT_Initialize( ) 汇编语言编写 P43

功能：该函数 INT_Initialize( )处理低级的、目标板相关的初始化。一旦这个函数执行完，控制权将会被转交给独立的目标板初始化函数 INC_Initialize( )，函数 INT_Initialize( )功能如下：

- 安装必要的处理器/系统控制寄存器
- 初始化向量表
- 安装系统堆栈指针
- 安装定时器中断
- 计算定时器高级中断堆栈各优先级
- 计算第一个可用内存地址
- 转交控制权给函数 INC_Initialize( )去初始化所有的操作系统模块

语法：VOID INT_Initialize(void)

调用函数：INC_Initialize( )

INT_Vectors_Loaded( ) 汇编语言编写 P44

功能：函数 INT_Vectors_Loaded()的返回标志表示是否装载所有缺省的向量表。如果它出错，每个低级中断寄存器另外把状态寄存器 ISR 的外壳（Shell）加载入实际的向量表。

语法：INT INT_Vectors_Loaded(void)

调用函数：无

INT_Setup_Vector

功能：函数 INT_Setup_Vector（）用于建立指定的新中断向量值，返回旧的中断向量值。

语法：VOID *INT_SetuP_Vector(INT vector, VOID *new)

调用函数：无

4.3 线程控制（TC）P45

线程控制组件（TC）负责管理具有竞争、实时 Nucleus PLUS 操作系统任务和高级中断服务例程（HISRS）的执行。一个 Nucleus PLUS 操作系统任务是具有专用目的的半独立程序段。大多数应用层具有多任务，为了控制执行进程，通常给任务分配优先级。Nucleus PLUS 操作系统的优先级范围是 0 到 255 级，0 代表最高级别，255 代表最低级别。高优先级的任务比低优先级的任务首先执行。此外，一个低优先级的任务可以被一个准备就绪的高优先级任务抢占，除非抢占被禁止。任务通常处于其中五个状态之一。这五个状态为：运行、就绪、挂起、终止、完成。

4.3.1 线程控制文件

线程控制模块（TC）由 9 个文件组成。具体描述如下：

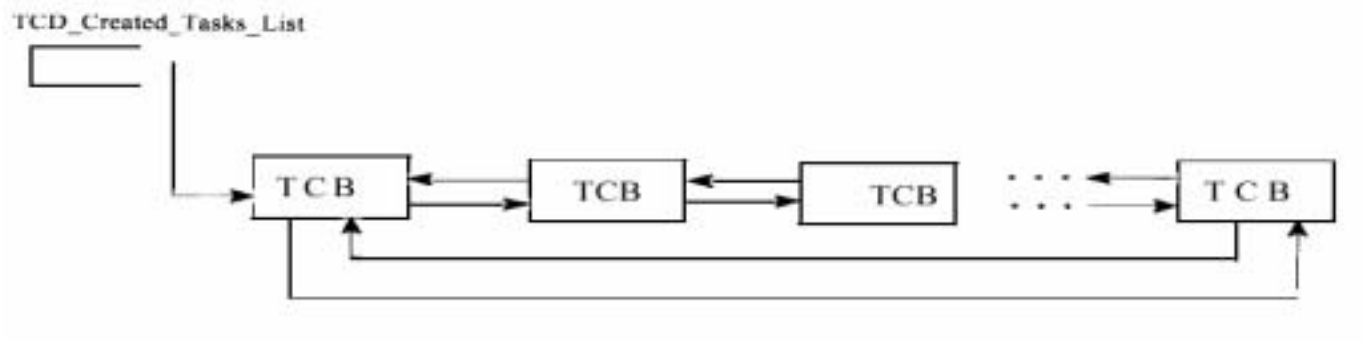
文件	说明
TC_DEFS.H	该文件包含线程控制模块（TC）的常量和数据结构定义。
TC_EXTR.H	该文件包含线程控制模块（TC）所有外部接口函数。
TCD.C	该文件定义了线程控制模块（TC）全局数据结构体。
TCI.C	该文件包含线程控制模块（TC）初始化函数。
TCF.C	该文件包含线程控制模块（TC）信息收集函数。
TCC.C	该文件包含线程控制模块（TC）核心函数，它用于处理基本创建的任务和删除的任务服务。
TCS.C	该文件包含线程控制模块（TC）的补充函数，它使用的频率比核心函数

	少。
TCCE.C	该文件包含为核心函数(TCC.C)提供的错误检测函数接口。
TCSE.C	该文件包含为补充函数(TCS.C)提供的错误检测函数接口。
TCT.[S,ASM,SRC]	该文件包含线程控制模块（TC）所有的目标板独立函数。

4.3.2 线程控制数据结构

创建任务列表

可以动态地创建和删除 NUCLEUS PLUS 操作系统任务。每个任务的线程控制模块（TCB）被保存在双链循环表中。最新建的任务被放置在列表的末端，删除的任务被从列表中移除。任务列表的头指针是 TCD_Created_Tasks_List 。



总任务数

当前创建的 NUCLEUS PLUS 操作系统任务总数包含在变量 TCD_Total_Tasks 中。变量 TCD_Total_Tasks 的值与在已创建列表中的线程控制模块（TCBS）数目相等。在 TCD_List_Protect 保护下可以操作变量。

创建任务列表保护

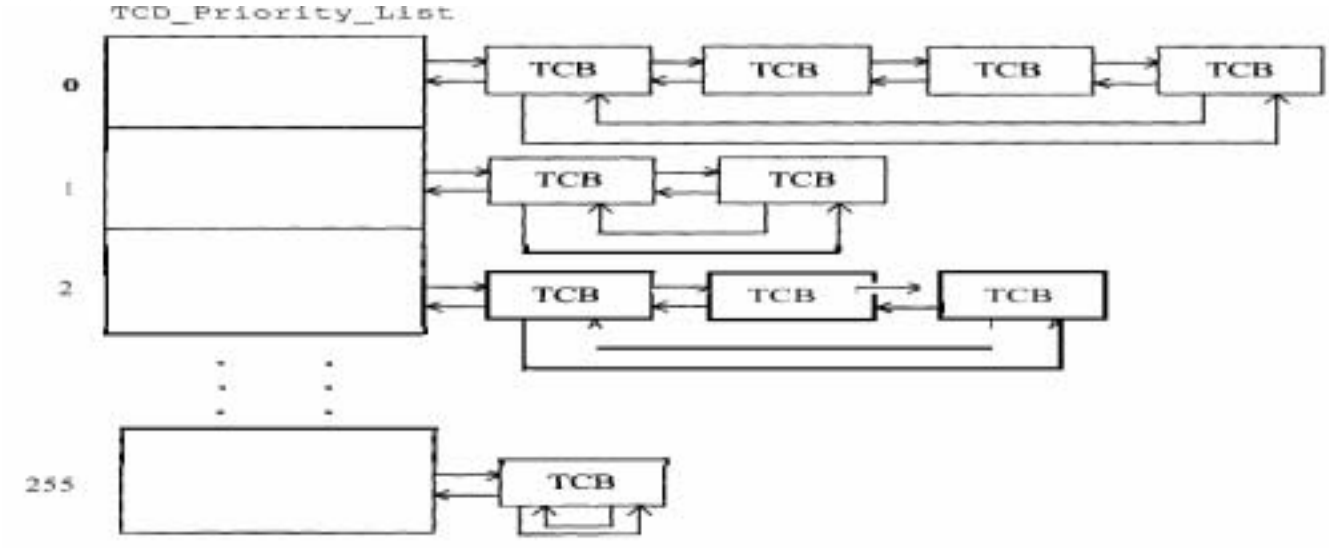
NUCLEUS PLUS 操作系统保护来自任务竞争或高级优先中断所创建任务列表的完整性。这是通过使用内部数据保护结构体 TCD_List_Protect 来实现的。所有任务的创建和删除都受到 TCD_List_Protect 保护。

字段声明

```
TC_TCB  tc_tcb_pointer          /*确认线程当前受到保护*/
UNSIGNED  cc_thread_waiting     /*表示一个或更多的线程正在等待保护*/
```

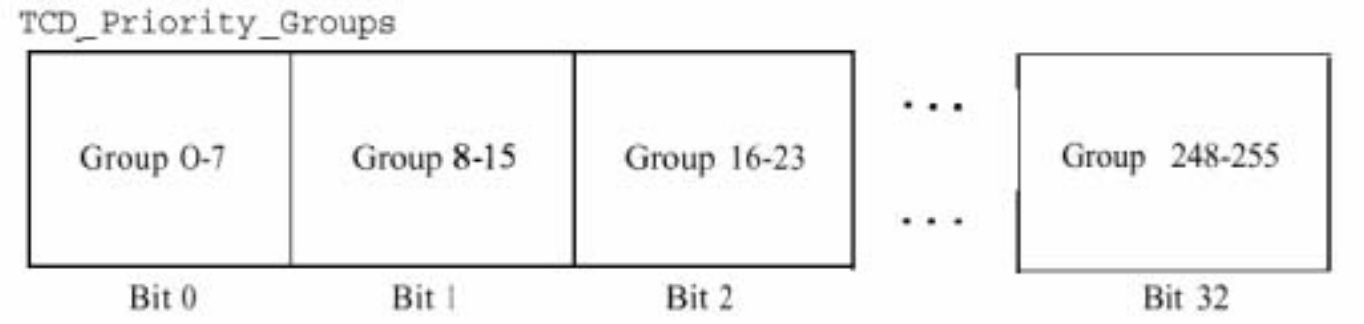
优先级列表

TCD_Priority_List是一个TCB指针数组。每个TCB指针数组的元素为任务列表的头指针由优先级决定是否准备好执行。如果优先级为NULL，表示没有任务准备好执行。该队列按优先级排列。



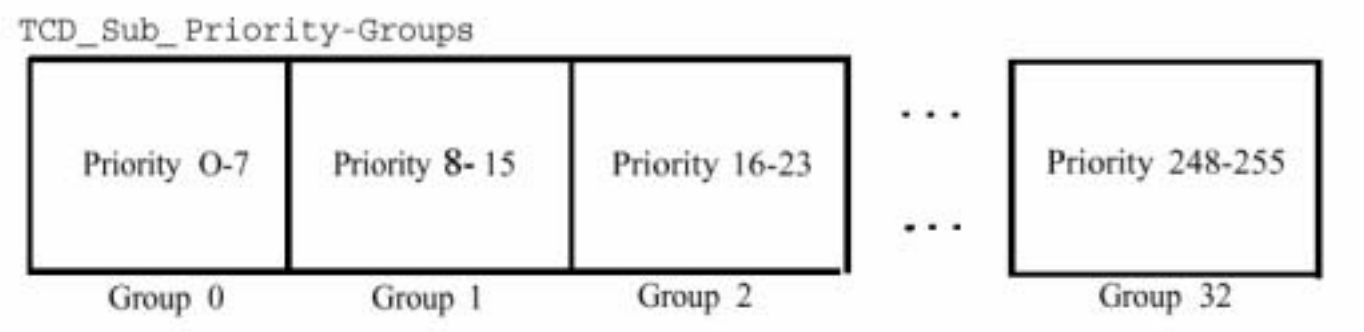
优先级组

TCD_Priority_Groups是32位无符号整数用于二进制图，每一个位相当于子优先级组。例如，如果设置bit 0，那么最少一个任务有0～8个优先级准备好执行。



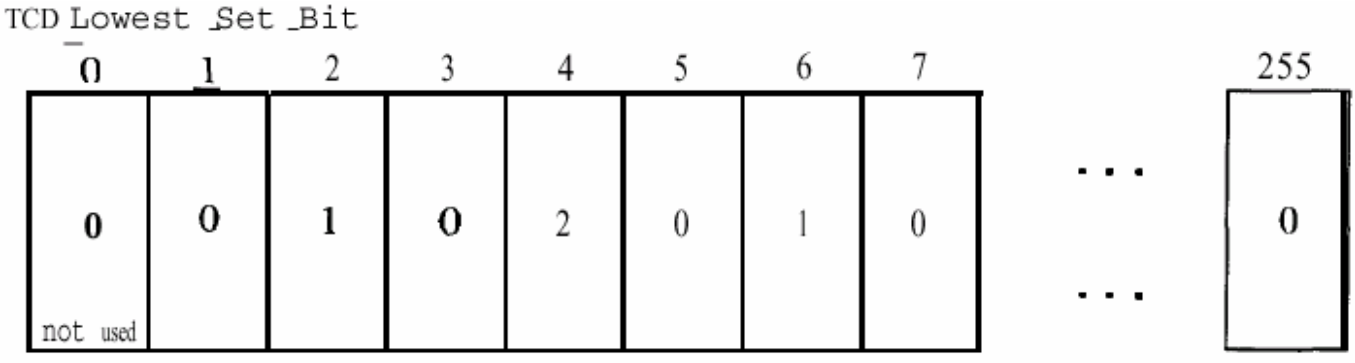
子级队列

NUCLEUS PLUS操作系统在位图中使用子优先级来代表执行实际的优先级。TCD_Sub_Priority_Groups 是子优先级队列数组。在数组中，Index 0相当于0～8优先级，每个元素的Bit 0代表0级别，Bit 7代表7级别。



最低位设置 P49

TCD_Lowest_Set位不超过标准的查找表范围。该表的索引值范围从0到225。在表中的任何位置索引值包含最低位设置值，这是用来决定最高优先级任务，表示在前一个已定义位图中。例如，7的最低位值包含在TCD_Lowest_Set索引值7中。在下面的表中，7相对的位值是0。



执行任务指针 P49

NUCLEUS PLUS操作系统执行任务的指针是*TCD_Execute_Task。注意*TCD_Execute_Task并不定指向当前执行的任务。在任务抢先占用性和任务保护冲突过程中，将有两种情况形成。当两种情况存在时，系统将产生几个指针。

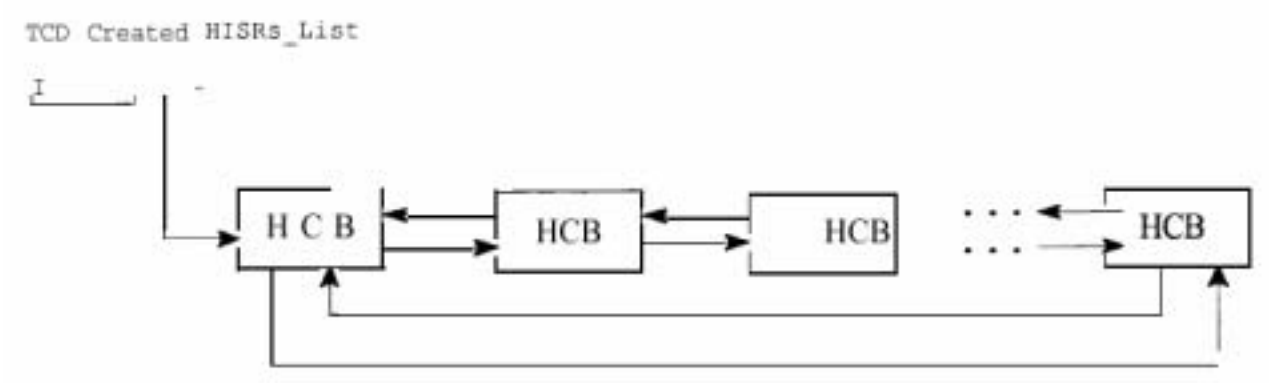
最高优先级变量 P49

NUCLEUS PLUS操作系统的变量TCD_Highest_Priority表示执行就绪任务最高优先级。注意这并不定代表当前执行任务。如果当前执行任务的抢先占用性被禁止，该变量为True。没有任务执行时，该变量被设为最大优先级。

创建高级中断服务程序列表指针 P50

创建高级中断服务程序列表的头指针是*TCD_ Created_HISRs_List，该指针为NULL时，

表示当前没有高级中断（HISRS）服务程序被创建。



高级中断总数目变量 P50

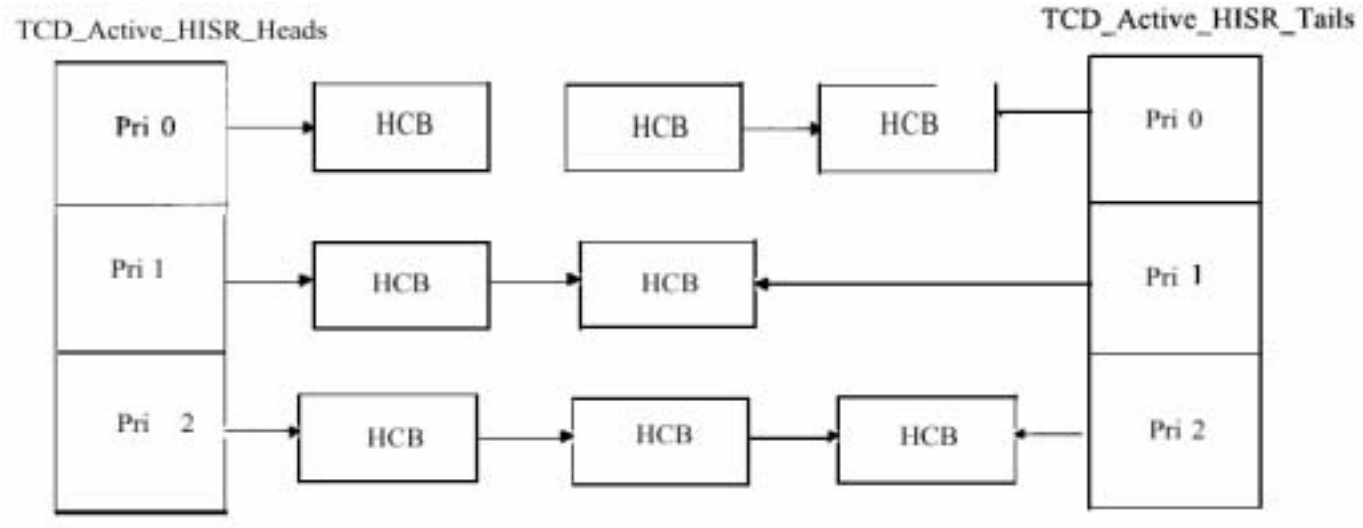
变量TCD_Total_HISRs表示当前创建NUCLEUS PLUS操作系统高级中断的总数目。该变量值等于在已创建列表中的HCBS的数目。对该变量值的操作受到TCD_HISR_Protect的保护。

有效高级中断头 P50

NUCLEUS PLUS操作系统保存有效高级中断列表头指针数组。该列表TCD_Active_HISR_Heads,在该列表有三个优先级。高级中断优先级标志在列表中。

有效高级中断尾 P51

NUCLEUS PLUS操作系统保存有效高级中断列表尾指针数组。该列表是TCD_Active_HISR_Tails,在该列表有三个优先级。高级中断优先级标志在列表中。如图所示，表中0代表最高优先级，2代表最低优先级。



执行高级中断指针

*TCD_Execute_HISR是执行最高优先级中断的指针。如果该指针为NULL，表示当前没有高级中断有效。注意当前线程指针并非一直是该指针。

当前线程

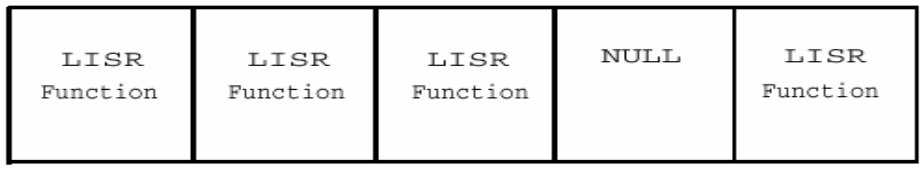
变量TCD_Current_Thread用于存储当前执行线程的控制块。因此，该变量的指针在TC_TCB或TC_THB结构体中。除了初始化外，在独立目标板部分模块，该变量被设置或清除。

注册低级中断（LISRS）

注册低级中断列表TCD_Registered_LISRs，它表示是否把一个给定的中断向量注册为低级中断。

低级中断指针 P52

*TCD_LISR_Pointers是一个列表的低级中断指针，它用来说明当中断发生时有低级中断函数调用。如果入口为NULL，说明没有低级中断函数调用，该入口还可用。



中断计数 P52

变量TCD_Interrupt_Count表示当前进程中中断服务程序的数目。如果该变量为0，说明当前进程中，没有中断产生。如果该变量值大于1，表示当前进程中有嵌套中断。

堆栈开关 P52

该标志TCD_Stack_Switched 表示在线程上下环境被保存之后，是否切换系统堆栈。某些端口没有用该变量。

系统保护 P53

NUCLEUS PLUS操作系统是保护来自任务竞争或高级优先中断系统结构的完整性。这是通过内部保护结构TCD_System_Protect来实现的。执行所有系统的创建和删除受到TCD_System_Protect保护。

字符段声明：

```
TC_TCB *tc_tcb_pointer            /*确认线程当前受到保护*/
UNSIGNED tc_thread_waiting        /*标志一个或多个线程正在等待保护*/
```

系统堆栈 P53

*TCB_System_Protect表示系统堆栈基指针。当系统处于空闲或中断处理过程中，说明系统堆栈在使用中。在目标板独立初始化过程中，该变量被建立。

低级中断（LISR）保护 P54

NUCLEUS PLUS操作系统是保护来自线程竞争或高级优先中断的低级中断的完整性。这是通过内部保护结构TCD_LISR_Protect来实现的 。执行所有LISR的创建和删除受到TCD_ LISR _Protect保护。

字符段声明：

```
TC_TCB *tc_tcb_pointer            /*确认线程当前受到保护*/
UNSIGNED tc_thread_waiting    /*标志一个或多个线程正在等待保护*/
```

高级中断（HISR）保护 P54

NUCLEUS PLUS操作系统是通过线程竞争或高级优先中断来保护高级中断的完整性。这是通过内部保护结构TCD_HISR_Protect来实现的 。执行所有HISR的创建和删除受到TCD_ HISR _Protect保护。

字符段声明：

```
TC_TCB *tc_tcb_pointer            /*确认线程当前受到保护*/
UNSIGNED tc_thread_waiting    /*标志一个或多个线程正在等待保护*/
```

中断级别 P54

变量TCD_Interrupt_Level表示系统允许中断状态。在端口，该变量是布尔型。

未处理中断 P54

变量TCD_Unhandled_Interrupts保存系统发生错误时未处理的中断向量个数。

任务控制模块 P55

任务控制模块TC_TCB包含任务的优先级和处理任务控制请求。

字符段声明：

```
CS_NODE tc_created
UNSIGNED tc_id
CHAR tc_name[NU_MAX_NAME]
DATA_ELEMENT tc_status
DATA_ELEMENT tc_delayed_suspend
DATA_ELEMENT tc_priority
```

```
DATA_ELEMENT tc_preemption
UNSIGNED tc_scheduled
UNSIGNED tc_cur_ time_slice
VOID *tc_stack_start
VOID *tc_stack_end
VOID *tc_stack_pointer
UNSIGNED tc_stack_size
UNSIGNED tc_stack_minimum
struct TC_PROTECT_STRUCT  *tc_current_protect
VOID *tc_saved_stack_ ptr
UNSIGNED tc_time_slice
struct TC_TCB_STRUCT *tc_ ready_previous
struct TC_TCB_STRUCT *tc_ready_next
UNSIGNED tc_priority_group
TC_TCB_STRUCT **tc_priority_head
DATA_ELEMENT *tc_sub_priority_ptr
DATA_ELEMENT tc_gub_ priority
DATA_ELEMENT tc_saved_status
DATA_ELEMENT tc_signal_active
DATA_ELEMENT tc_padding[PAD_3]
VOID (*tc entry) (UNSIGNED, VOID *)
UNSIGNED tc_argc
VOID *tc_argv
VOID (*tc_cleanup) (VOID *)
VOID *tc_cleanup_info
struct TC_PROTECT_STRUCT *tc_suspend_protect
INT tc_timer_active
TM TCB tm_timer_control
UNSIGNED tc_signals
UNSIGNED tc_enabled_signals
VOID (*tc_signal_handler) (UNSIGNED)
UNSIGNED tc_system_reserved_1
UNSIGNED tc_system_reserved_2
UNSIGNED tc_system_reserved_3
UNSIGNED tc_app_reserved_1
```

变量	描述
tc_created	任务的连接节点结构。它是双循环链表，用于连接已创建的任务列表。
tc_id	控制内部任务标识0x5441534B，它相当于ASCII TASK。
tc_name	由用户指定8个字符任务名字。
tc_status	任务的当前状态。
tc_delayed_suspend	表示任务是否挂起的标志。
tc_priority	任务当前优先级。
tc_preemption	该标志决定是否使能抢占方式。
tc_scheduled	表示任务调度计数。
tc_cur_ time_slice	当前时间片的值。
*tc_stack_start	指向任务堆栈起始地址的指针。
*tc_stack_end	指向任务堆栈结束地址的指针。

*tc_stack_pointer	通过应用使用的一个保留字。
tc_stack_size	存储任务堆栈大小。
tc_stack_minimum	任务允许最小的堆栈大小。
*tc_current_protect	指向任务当前保护结构的指针。
*tc_saved_stack_ptr	任务先前堆栈指针。
tc_time_slice	任务当前时间片的值。
*tc_ready_previous	指向先前就绪TCB的指针。
*tc_ready_next	在就绪列表中指向下一个TCB的指针。
tc_priority_group	优先级群位的当前表征码。
**tc_priority_head	指向优先级列表头块的指针。
*tc_sub_prioirty_ptr	指向优先级子群的指针。
tc_sub_priority	优先级子群位的当前表征码
tc_saved_status	先前任务的状态。
tc_signal_active	该标志表示信号是否有效。
tc_padding	用于在偶数临界上调整任务结构。在某些端口不用。
(*tc_entry)(UNSIGNED, VOID *)	任务入口函数。
tc_argc	表示一个可选任务变元。
*tc_argv	表示一个可选任务变元。
(*tc_cleanup) (VOID *)	任务清除服务例程。
tc_cleanup_info	指向任务清除信息的指针。
*tc_suspend_protect	在任务挂起的同时，指向保护结构的指针。
tc_timer_active	该标志决定定时器是否有效。
tc_timer_control	定时器控制块。
tc_signals	包含当前信号。
tc_enabled_signals	包含使能信号。
(*tc_signal_handler)(UNSIGNED)	信号处理例程。
tc_system_reserved_1	通过系统使用的一个保留字。
tc_system_reserved_2	通过系统使用的一个保留字。
tc_system_reserved_3	通过系统使用的一个保留字。
tc_app_reserved_1	通过应用使用的一个保留字。

高级中断控制模块 P58

高级中断控制模块TC_HCB包含高级中断的优先级和其他处理任务控制请求。

字符段声明：

```
CS_NODE tc_created
UNSIGNED tc_id
CHAR tc_name[NU_MAX NAME]
DATA_ELEMENT tc_not_used_1
DATA_ELEMENT tc_not_used_2
DATA_ELEMENT tc_priority
DATA_ELEMENT tc_not_used_3
UNSIGNED tc_scheduled
UNSIGNED tc_cur_time_slice
VOID  *tc_stack_start
VOID  *tc_stack_end
```

```
VOID  *tc_stack_pointer
UNSIGNED tc_stack_size
UNSIGNED tc_stack_minimum
struct TC_PROTECT_STRUCT *tc_current_Protect
struct TC_HCB_STRUCT *tc_actrve_next
UNSIGNED tc_activation_count
VOID (*tc_entry)(VOID)
UNSIGNED tc_system_reserved_1
UNSIGNED tc_system_reserved_2
UNSIGNED tc_system_reserved_3
UNSIGNED tc_aPP_reserved_1
```

变量	说明
tc_created	HISRs的连接节点结构。它是双循环链表，用于连接已创建的HISRs列表。
tc_id	控制内部HISR标识0x48495352，它相当于ASCII HISR。
tc_name	由用户指定有8个字符HISR名字。
tc_not_used_1	This field is a place holder and is not used.
tc_not_used_2	This field is a place holder and is not used.
tc_Priority	HISR的优先级。
tc_not_used_3	This field is a place holder and is not used.
tc_scheduled	HISR调度计数。
tc_cur_time_slice	当前时间片的值。
*tc_stack_start	指向HISRs堆栈起始地址的指针。
*tc_stack_end	指向HISRs堆栈结束地址的指针。
*tc_stack_Pointer	HISRs堆栈指针。
tc_stack_size	存储 HISRs堆栈大小。
tc_stack_minimum	HISRs 允许最小的堆栈大小。
*tc_current_Protect	指向当前保护结构指针
*tc_active_next	指向下一个有效的HISR指针。
tc_activation_count	HISR有效计数器
(*tc_entry) (VOID)	HISRs 入口函数。
tc_system_reserved_1	通过系统使用的一个保留字。
tc_system_reserved_2	通过系统使用的一个保留字。
tc_system_reserved_3	通过系统使用的一个保留字。
tc_app_reserved_1	通过应用使用的一个保留字。

保护模块 P58

Nucleus PLUS操作系统是通过任务竞争或高级中断来保护操作系统数据结构的完整性。这是通过内部保护结构TC_Protect来实现的。所有Nucleus PLUS操作系统的数据结构的创建和删除，都在TC_Protect保护下操作的。

字符段声明：

```
TC_TCB *tc_tcb_Pointer
UNSIGNED tc_thread_waiting
```

变量	说明
*tc_tcb_Pointer	确认当前线程已受到保护。
tc_thread_waiting	该标志表示一个或更多的线程正在等待保护。

4.3.3线程控制函数 P59

下面部分是对线程控制模块（TC）做简要的概述。

TCC_Create_Task P59

功能：该函数创建一个任务，并把任务放在创建任务列表中。

语法：

```
STATUS  TCC_Create_Task(NU_TASK *task_ptr,
                        CHAR *name,
                        VOID(*task_entry)(UNSIGNED, VOID *),
                        UNSIGNED argc,
                        VOID *argv,
                        VOID *stack_address,
                        UNSIGNED stack_size,
                        OPTION priority,
                        UNSIGNED time_slice,
                        OPTION preempt,
                        OPTION auto_start)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
TCT_Control_To_System
TCT_Protect
TCC_Resume_Task
TCT_Build_Task_Stack
TCT_Unprotect
TMC_Init_Task_Timer
[TCT_Check_Stack]
```

TCC_Delete_Task P60

功能：删除一个任务并把该任务从任务列表中移除。它是通过该函数来完成和终止任务的。注意该函数没有空闲内存与任务控制模块或堆栈相关联。它是负责应用层。

语法：`STATUS TCC_Delete_Task(NU_TASK *task_ptr)`

调用函数：

```
CSC_Remove_From_List
[HIE_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCC_Create_HISR P61

功能：创建一个高级中断服务程序（HISR）并把它放在创建高级中断列表中。该函数支持所有系统资源创建高级中断。经常在待用状态下，创建高级中断。

语法：

```
STATUS TCC_Create_HISR(NU_HISR *hisr_ptr,
                       CHAR *name,
                       VOID (*hisr_entry) (VOID),
                       OPTION priority,
                       VOID *stack_address,
```

UNSIGNED stack_size)

调用函数：

CSC_Place_On_List
[HIC_Make_History_Entry]
TCT_Build_HISR_Stack
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCC_ Delete_HISR

功能：删除一个高级中断并该高级中断从创建高级中断列表中移除。假定在一个闲置状态使用该函数，应注意该函数没有空闲内存与高级中断控制模块或堆栈相关联。它是负责应用层的。

语法：STATUS TCC_Delete_HISR(NU_HISR *hisr_ptr)

调用函数：

CSC_Remove_From_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCC_Reset_Task

功能：该函数用于重置指定任务。在一个任务完成或结束状态下，才能执行重置任务。在一个非条件暂停状态下，任务被停止。

语法：STATUS TCC_Reset_Task(NU_TASK *task_ptr, UNSIGNED argc,VOID *argv)

调用函数：

[HIC_Make_History_Entry]
TCT_Build_Task_Stack
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCC_Terminate_Task P62

功能：该函数用于终止指定任务。如果任务已经终止，该函数不起作用。如果任务处于悬挂状态时，调用指定的任务清除服务程序清除悬挂数据结构。

语法：STATUS TCC_Terminate_Task(NU_TASK *task_ptr)

调用函数：

[HIC_Make_History_Entry]
TCC_Suspend_Task
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
TCT_Unprotect_Specific
TMC_Stop_Task_Timer

TCC_Resume_Task

功能：该函数用于重新恢复前一个挂起任务。该任务必须正是由于相同的请求而导致暂停状态。如果重新恢复的任务优先级比正在调用的任务和目前抢占的任务优先级

高，该函数就返回NU_TRUE的值。如果没有任务抢占请求，也返回一个NU_TRUE值。

语法：STATUS TCC_Resume_Task(NU_TASK *task_ptr, OPTION suspend_type)

调用函数：

[TCT_Check_Stack]
TCT_Set_Current_Protect
TCT_Set_Execute_Task
TMC_Stop_Task_Timer

TCC_Resume_Service P63

功能：该函数提供一个适当的接口给实际的服务用于重新恢复任务。

语法：STATUS TCC_Resume_Service(NU_TASK *task_ptr)

调用函数：

[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Unprotect

TCC_Suspend_Task

功能：该函数挂起指定任务。如果指定任务正在被调用，就返回控制信息给系统。

语法：

```
VOID TCC_Suspend_Task(NU_TASK *task_ptr,
                      OPTION Suspend_type,
                      VOID (*cleanup)(VOID*),
                      VOID *information,
                      UNSIGNED timeout)
```

调用函数：

[HIC_Make_History_Entry]
TCT_Control_To_System
TCT_Protect
TCT_Set_Execute_Task
TCT_Protect_Switch
TCT_Unprotect
TMC_Start_Task_Timer

TCC_Suspend_Service P64

功能：该函数提供一个适当的接口给实际的服务用于挂起任务。

语法：STATUS TCC_Suspend_Service(NU_TASK *task_ptr)

调用函数：

[HIC_Make_History_Entry]
TCC_Suspend_Task
[TCT_Check_stack]
TCT_Protect
TCT_Unprotect

TCC_Task_Timeout　P65

功能：该函数处理任务挂起时间情况。注意任务睡眠请求也被认为是任务暂停挂起情况。
语法：VOID TCC_Task_Timeout(NU_TASK *task_ptr)
调用函数：（调用清除函数）

```
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Protect
TCT_Set_Current_Protect
TCT_Unprotect
TCT_Unprotect_Specific
```

TCC_Task_Sleep　P65

功能：该函数提供任务睡眠挂起。它的基本目的是提供一个接口给实际的任务挂起函数。
语法：VOID TCC_Task_Sleep(UNSIGNED ticks)
调用函数：

```
[HIC_Make_History_Entry]
TCC_Suspend_Task
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCC_Relinquish

功能：在处于相同优先级别时，该函数把正在调用的任务移到其他任务末端。直到处于相同优先级的所有其他任务获取一个机会执行后，正在调用的任务才能被执行。
语法：VOID TCC_Relinquish(VOID)
调用函数:

```
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Set_Execute_Task
TCT_Unprotect
```

TCC_Time_Slice　P66

功能：在处于相同优先级别时，该函数把指定的任务移到其他任务末端。如果指定任务长时间没有准备好，任务请求就会被忽略。
语法：VOID TCC_Time_Slice(NU_TASK *task_Ptr)
调用函数:

```
[TCT_Check_Stack]
TCT_Protect
TCT_Set_Execute_Task
TCT_UnProtect
```

TCC_Current_Task_Pointer

功能：该函数返回一个当前正在执行的任务的指针。如果当前线程不是任务线程，就返回一个NU_NULL的值。

语法：NU_TASK *TCC_Current_Task_Pointer(VOID)
调用函数：无

TCC_Current_HISR_Pointer P67

功能：该函数返回一个当前正在执行的高级中断指针。如果当前线程不是高级中断线程，就返回一个NU_NULL的值。
语法：NU HISR *TCC_Current_HISR_Pointer(VOID)
调用函数：无

TCC_Task_Shell

功能:该函数是开始执行所有应用任务的外壳。当控制从应用任务返回的时候，外空程序使任务该任务完成。该外空程序也传送变元argc和argv 给任务入口函数。
语法：VOID TCC_Task_Shell(VOID)
调用函数：

```
Task Entry Function
TCC_Suspend_Task
TCT_Protect
```

TCC_Signal_Shell P68

功能：该函数通过调用任务指定的信号处理函数来处理信号。当信号处理完成时，任务被置于适当的状态。
语法：VOID TCC_Signal_Shell(VOID)
调用函数：

```
tasks signal handling routine
[TCT_Check_Stack]
TCT_Signal_Exit
TCT_Protect
TCT_Set_Execute_Task
TCT_UnProtect
```

TCC_Dispatch_LISR P69

功能：该函数调用与指定中断向量相关的低级中断（LISR）。注意在线程中断过程中，该函数被调用。
语法: VOID TCC_Dispatch_LISR(INT vector)
调用函数：

```
application LISR
ERC_System_Error
```

TCC_Register_LISR

功能：该函数以所提供的向量号注册所提供的低级中断LISR。如果所提供的LISR为NU_NULL,所提供的向量未注册。先前已注册的LISR返回完成状态给调用者。
语法：

```
STATUS TCC_Register_LISR(INT vector, VOID(*new_lisr)(INT),
                         VOID( **old_lisr)(INT))
```

调用函数：

```
[HIC_Make_History_Entry]
INT_Retrieve_Shell
```

```
INT_Setup_Vector
INT_Vectors_Loaded
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCCE_Create_Task P70

功能：该函数执行与创建任务函数相关参数的错误检测。
语法:

```
STATUS TCCE_Create_Task(NU_TASK *task_ptr,
                        CHAR name,
                        VOID(*task_entry)(UNSIGNED, VOID *),
                        UNSIGNED argc,
                        VOID *argv,
                        VOID *stack_address,
                        UNSIGNED stack_size,
                        OPTION priority,
                        UNSIGNED time_slice,
                        OPTION Preempt,
                        OPTION auto_start)
```

调用函数：TCC_Create

TCCE_Create_HISR

功能：该函数执行与创建高级中断函数相关参数的错误检测。
语法:

```
STATUS TCCE_Create_HISR(NU_HISR *hisr_ptr,
                        CHAR *name,
                        VOID(*hisr_entry) (VOID),
                        OPTION priority,
                        VOID *stack_address,
                        UNSIGNED stack_size)
```

调用函数：TCC_Create_HISR

TCCE_Delete_HISR　P71

功能：该函数执行与删除高级中断函数相关参数的错误检测。
语法: STATUS TCCE_Delete_HISR(NU_HISR *hisr_ptr)
调用函数：TCC_Delete_HISR

TCCE_Delete_Task

功能：该函数执行与删除任务函数相关参数的错误检测。
语法：STATUS TCCE_Delete_Task(NU_TASK *task_ptr)
调用函数：TCC_Delete_Task

TCCE_Reset_Task

功能：该函数执行与重启任务函数相关参数的错误检测。
语法:STATUS TCCE_Reset_Task(NU_TASK *task_ptr,UNSIGNED argc,VOID *argv)
调用函数：TCC_Reset_Task

TCCE_Terminate_Task　P72

功能：该函数执行与终止任务函数相关参数的错误检测。
语法:

```
STATUS TCCE_Terminate_Task(NU_TASK *task_ptr)
```

调用函数：

```
TCC_Terminate_Task
```

TCCE_Resume_Service

功能：该函数执行与恢复任务函数相关参数的错误检测。
语法:`STATUS TCCE_Resume_Service(NU_TASK *task_ptr)`
调用函数：

```
TCC_Delete_Task
TCCE_Validate_Resume
TCC_Resume_Service
```

TCCE_Suspend_Service

功能：该函数执行与暂停服务函数相关参数的错误检测。
语法：`STATUS TCCE_Suspend_Service(NU_TASK *task_ptr)`
调用函数：`TCC_Suspend_Service`

TCCE_Relinquish　P73

功能：该函数执行放弃函数的错误检测。如果当前线程不是一个任务，将忽略该请求。
语法：`VOID TCCE_Relinquish(VOID)`
调用函数：`TCC_Relinquish`

TCCE_ask_Sleep

功能：该函数执行任务睡眠函数的错误检测。
语法：`VOID TCCE_Task_Sleep(UNSIGNED ticks)`
调用函数：`TCC_Task_Sleep`

TCCE_SusPend_Error

功能：该函数检测一个挂起请求错误。只有任务线程才允许挂起请求，而来自其他任何线程的挂起请求将产生一个错误。
语法：`INT TCCE_Suspend_Error(VOID)`
调用函数：无

TCCE_Activate_HISR　P74

功能：该函数执行与激活的高级中断函数相关参数的错误检测。
语法：

```
STATUS TCCE_Activate_HISR(NU_HISR *hisr_ptr)
```

调用函数：

```
TCT_Activate_HISR
```

TCCE_Validate_Resume

功能：该函数使恢复服务有效，并恢复围绕任务状态检测调度保护所调用的驱动器。
语法：`STATUS TCCE_Validate_Resume(OPTION resume_type,NU_TASK *task_ptr)`

调用函数：

```
TCT_Set_Current_Protect
TCT_Sysfem_Protect
TCT_System_Unprotect
TCT_Unprotect
```

TCF_Established_Tasks　P75

功能：返回建立任务的当前数目。已删除的前一个任务不再被认为建立。

语法：`UNSIGNED TCF_Established_Tasks(VOID)`

调用函数：`[TCT_Check_Stack]`

TCF_Established_HISRs

功能：返回建立高级中断（HISRS）的当前数目。已删除的前一个任务不再被认为建立。

语法：`UNSIGNED TCF_Established_HISRs(VOID)`

调用函数：

```
[TCT_Check_Stack]
```

TCF_Task_Pointers

功能：在一个指定的位置，开始建立任务指针列表。放置在列表中的任务指针数量等于任务的总数或在指定的调用中最大的指针数目。

语法：

```
UNSIGNED TCF_Task_Pointers(NU_TASK **Pointer_list,UNSIGNED
                           maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TCF_HISR_Pointers P76

功能：在一个指定的位置，开始建立高级中断指针列表。放置在列表中的高级中断指针数量等于任务的总数或在指定的调用中最大的指针数目。

语法：

```
UNSIGNED TCF_HISR_Pointers(NU_HISR **pointer_list, UNSIGNED
                           maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCF_Task_Information

功能：返回指定任务的信息。然而，如果给定的任务指针无效，该函数只返回一个错误状态。

语法：

```
STATUS TCF_Task_Information(NU_TASK *task_ptr,
                            CHAR *name,
                            DATA ELEMENT *status,
                            UNSIGNED *scheduled_count,
```

```
                                    DATA_ELEMENT *priority,
                                    OPTION preempt,
                                    UNSIGNED *time_slice,
                                    VOID **stack_base,
                                    UNSIGNED *stack_size,
                                    UNSIGNED *minimum_stack)
```

调用函数：

```
        [TCT_Check_Stack]
        TCT_System_Protect
        TCT_Unprotect
```

TCF_HISR_Information　P77

功能：返回指定高级中断（HISR）的信息。然而，如果给定的HISR指针无效，该函数只返回一个错误状态。

语法：

```
    STATUS TCF_HISR_Information(NU_HISR *hisr_ptr,
                               CHAR *name,
                               UNSIGNED *scheduled_count,
                               DATA_ELEMENT *priority,
                               VOID **stack_base,
                               UNSIGNED *stack_size,
                               UNSIGNED minimum_stack)
```

调用函数：

```
        [TCT_Check_Stack]
        TCT_System_Protect
        TCT_Unprotect
```

TCI_Initialize

功能：该函数初始化控制TC模块操作的数据结构。系统被初始化为空闲。

语法：`VOID TCI_Initialize(VOID)`

调用函数：无

TCS_Change_Priority P78

功能：该函数改变指定任务的优先级。挂起任务或就绪的任务的优先级也能被它改变。如果新的优先级请求场景切换，控制权就返回给系统。

语法：`OPTION TCS_Change_Priority(NU_TASK *task_ptr, OPTION new_priority)`

调用函数：

```
        [HIC_Make_History_Entry]
        [TCT_Check_Stack]
        TCT_Control_To_System
        TCT_Protect
        TCT_Set_Execute_Task
        TCT_Unprotect
```

TCS_ Change_Preemption

功能：该函数能改变正在调用任务抢占的状态。任务的抢占性可以允许或禁止。如果它被禁止，任务一直运行到它挂起或放弃。如果抢占处于未决状态，调用该函数使

抢占引起一个场景切换。

语法：OPTION TCS_Change_Preemption(OPTION preempt)

调用函数：

[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Set_Execute_Task
TCT_Unprotect

TCS_Change_Time_Slice

功能：该函数改变指定任务的时间片。时间片值0表示禁止使用时间片。

语法：UNSIGNED TCS_Change_Time_Slice(NU_TASK *task_ptr,UNSIGNED time_slice)

调用函数：

[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCS_ Control_Signals

功能：该函数允许指定信号并返回前一个允许信号的值用于调用。如果一个新的允许信号存在并已注册信号处理，那么信号处理开始。

语法：UNSIGNED TCS_Control_Signals(UNSIGNED enable_signal_mask)

调用函数：

[HIC_Make_History_Entry]
TCC_signal_Shell
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCS_Receive_Signals P80

功能：该函数返回目前的信号给调用程序。注意该信号能自动被清除。

语法：UNSIGNED TCS_Receive_Signals(VOID)

调用函数：

[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

TCS_Register_Signal_Handler

功能：该函数给正在调用的任务注册一个信号处理程序。注意如果一个允许信号存在并且是第一次注册的信号处理调用，那立刻处理该信号。

语法：STATUS TCS_Register_Signal_Handler(VOID (*signal_handler)(UNSIGNED))

调用函数：

[HIC_Make_History_Entry]
TCC_Signal_Shell
[TCT_Check_Stack]

TCT_Protect
TCT_Unprotect

TCS_Send_Signals P81

功能：该函数返回指定任务的指定信号。如果允许，就建立指定任务来处理信号。
语法：STATUS TCS_Send_Signals(NU_TASK *task_ptr, UNSIGNED signals)
调用函数：

[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Signal_Shell
TCT_Build_Signal_Frame
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Unprotect

TCSE_Change_Priority

功能：该函数执行改变优先级服务的错误检测。如果检测到一个错误，该服务被忽略和将返回请求优先级。
语法：OPTION TCSE_Change_Priority(NU_TASK *task_ptr,OPTION new_priority)
调用函数：TCS_Change_Priority

TCSE_Change_Preemption P82

功能：该函数执行抢占服务的错误检测。如果当前线程不是任务线程，该请求将被忽略。
语法：OPTION TCSE_Change_Preemption(OPTION Preempt)
调用函数：TCS_Change_Preemption

TCSE_Change_Time_Slice

功能：该函数执行抢占服务的错误检测。如果指定的指针无效，该请求将被忽略。
语法：UNSIGNED TCS_Change_Time_Slice(NU_TASK *task_ptr, UNSIGNED time_slice)
调用函数：TCS_Change_Time_Slice

TCSE_Control_Signais

功能：该函数查看该调用程序是否由一个非空任务线程产生。如果是，该请求被忽略。
语法：UNSIGNED TCSE_Control_Signals(UNSIGNED enable_signal_mask)
调用函数：TCS_Control_Signals

TCSE_Receive_Signals P83

功能：该函数确定该调用程序是否由一个执行任务的线程产生。如果不是，该调用被忽略。
语法：UNSIGNED TCSE_Receive_Signals(VOID)
调用函数：TCS_Receive_Signals

TCSE_Register_Signal_Handler

功能：该函数确定该调用是否为一个任务。如果该调用不是一个任务或给定的信号处理函数指针为空（NULL）,将返回一个当的错误状态。
语法：STATUS TCSE_Register_Signal_Handler(VOID (*signal_handler)(UNSIGNED))

调用函数：TCS_Register_Signal_Handler

TCSE_Send_Signals

功能：该函数检测一个无效的任务。如果选择了一个无效的任务，将返回一个错误。
语法：STATUS TCSE_Send_Signals(NU_TASK *task_ptr, UNSIGNED signals)
调用函数：TCS_Send_Signals

TCT_Control_Interrupts P84 汇编语言编写

功能：该函数用于允许和禁止指定调用的中断。通过调用来禁止中断，一直保持禁止中断状态到另一个调用来使能它们。
语法：INT TCT_Control_Interrupts(new_level)
调用函数：无

TCT_Local_Control_Interrupts 汇编语言编写

功能：该函数用于允许和禁止指定的中断。
语法：INT TCT_Local_Control_Interrupts(new_level)
调用函数：无

TCT_Restore_Interrupts 汇编语言编写

功能：该函数在指定的全局变量TCD_Interrupt_Level中恢复中断。
语法：VOID TCT_Restore_Interrupts(VOID)
调用函数：无

TCT_Build_Task_Stack 汇编语言编写P85

功能：该函数给一个任务建立一个初始化堆栈帧。初始化堆栈包含与寄存器初始化值相关和入口任务指针的信息。而且，初始化堆栈帧和中断堆栈帧具有相同的格式。
语法：VOID TCT_Build_Task_Stack(TC_TCB *task)
调用函数：无

TCT_Build_HISR_Stack 汇编语言编写

功能：该函数建立一个高级中断堆栈帧目的是允许很快地调度高级中断。
语法：VOID TCT_Build_HISR_Stack(TC_HCB *hisr)
调用函数：无

TCT_Build_Signal_Frame 汇编语言编写

功能：该函数在任务堆栈顶端建立一个帧。这将引起任务信号处理程序执行下一次已经执行的任务。
语法：VOID TCT_Build_Signal_Frame(TC_TCB *task)
调用函数：无

TCT_Check_Stack 汇编语言编写 P86

功能：该函数检测当前堆栈溢出情况。除此之外，该函数保留正在调用线程的堆栈空间最小数目并返回当前有效值的堆栈空间。
语法：UNSIGNED TCT_Check_Stack(void)
调用函数：ERC_System_Error

TCT_Schedule 汇编语言编写

功能：该函数等待一个线程就绪。一旦线程就绪，该函数对线程的控制转移初始化。
语法：VOID TCT_Schedule(void)
被调用函数：TCT_Control_To_Thread

TCT_Control_To_Thread 汇编语言编写

功能：控制指定线程的转移。每次控制一个线程的转移，它的调度计数器计数增加。除此之外，在过程中，任务线程的时间片被允许。该函数建立 TCD_Current_Thread 指针。
语法：VOID TCT_Control_To_Thread(TC_TCB *thread)
调用函数：无

TCT_Control_To_System 汇编语言编写

功能：该函数从一个线程控制给系统。注意该服务是在请求方式下被调用的，而不是从中断线程中调用的。编译程序要求通过函数的约束保护寄存器被保存在该例行程序中。注意这通常是有效寄存器总数的一个子集。
语法：VOID TCT_Control_To_System(void)
调用函数：TCT_Schedule

TCT_Signal_Exit 汇编语言编写

功能：该函数从信号处理程序中退出。该函数的基本目的是清除调度保护并返回堆栈指针给常规任务堆栈指针。
语法：VOID TCT_Control_To_System(void)
调用函数：TCT_Schedule

TCT_Current_Thread 汇编语言编写

功能：返回当前线程指针。
语法：VOID *TCT_Current_Thread(void)
调用函数：无

TCT_Set_Execute_Task 汇编语言编写 P88

功能：在保护下设置当前任务执行变量，这与中断系统相反。
语法：VOID TCT_Set_Execute_Task(TC_TCB *task)
调用函数：无

TCT_Protect 汇编语言编写

功能：该函数避免多线程的存取。如果另外一个任务或高级中断线程控制请求保护结构，那么该线程被调度运行直到释放保护结构才停止。这时，线程被挂起，控制返回给线程，将产生TCT_Protect调用。这阻止高优先级的低优先级任务企图获得保护结构。
语法：VOID TCT_Protect(TC_PROTECT *protect)
调用函数：无

TCT_Unprotect 汇编语言编写

功能：该函数释放当前活动线程的保护。如果当前调用的不是活动线程，该请求被忽略。
语法：VOID TCT_Unprotect(void)
被调用函数：无

TCT_Unprotect_Specific 汇编语言编写 P89

功能：释放指定的保护结构。
语法：VOID TCT_Unprotect_Specific(TC_PROTECT *protect)

调用函数：无

TCT_Set_Current_Protect 汇编语言编写

功能：该函数给指定保护指针设置当前线程控制模块的当前保护现场。
语法：VOID TCT_Set_Current_Protect(TC_PROTECT *protect)
调用函数：无

TCT_Protect_Switch 汇编语言编写

功能：该汇编函数一直等待到一个指定的任务不再有保护和它相关联。这是很必要的，除非它们释放了所有它们的保护，否则任务不能挂起或终止。
语法：VOID TCT_Protect_Switch(VOID *thread)
调用函数：无

TCT_Schedule_Protected 汇编语言编写　P90

功能：该函数保存一个线程的最小场景。 然而，它直接调度另外已优于该线程调用服务例程的线程。
语法：VOID TCT_Schedule_Protected(VOID *thread)
调用函数：TCT_Control_To_Thread

TCT_Interrupt_Context_Save 汇编语言编写

功能：该函数保存中断线程场景。也支持套嵌中断。如果一个任务或高级中断服务例程的线程被中断，在保存场景之后，堆栈指针就被切换到系统堆栈。
语法：VOID TCT_Interrupt_Context_Save(vector)
调用函数：无

TCT_Interrupt_Context_Restore 汇编语言编写

功能：如果存在一个 嵌套中断条件存在，该函数就恢复中断场景。否则该服务例程把控制权转交给调度函数。
语法：VOID TCT_Interrupt_Context_Restore(void)
调用函数：TCT_Schedule

TCT_Activate_HISR 汇编语言编写　P91

功能：该函数激活指定的HISR。如果HISR已处于激活状态，HISR的有效计数器加1。否则，HISR被放在适当HISR的优先级列表。HISR的优先级列表处于准备好执行状态。
语法：STATUS TCT_Activate_HISR(TC_HCB *hisr)
调用函数：无

TCT_HISR_Shell 汇编语言编写

功能：该汇编函数每个HISR的执行外壳（shell）。如果HISR已经完成它的处理，该外壳服务例程退回系统。否则，连续调用HISR服务例程，直到有效计数为0。
语法：VOID TCT_HISR_Shell(void)
调用函数：hisr->tc_entry

TCT_Check_For_Preemption 汇编语言编写 P92

功能：该汇编函数在一个最小的ISR处理过程中检测是否有某些其他中断情况发生。如果是这样，就执行整个上下环境存储和恢复，目的是处理抢先占有。否则，控制权返回中断点。

语法：VOID TCT_Check_For_Preemption(void)
调用函数：
TCT_Interrupt_Context_Save
TCT_Interrupt_Context_Restore

4.4定时器(TM) P94

定时器模块（TM）负责处理 NUCLEUS PLUS 操作系统所有定时器功能。Nucleus PLUS 操作系统定时器时间基本单位是一个滴答声(tick)，相当于单个硬件定时器中断。Nucleus PLUS 操作系统定时器在应用级用于在定时器溢出时执行特殊的服务例程。定时器也可以应用于任务和提供任务睡眠以及挂起时间溢出。请参阅 Nucleus PLUS 操作系统手册第三章，了解更多有关的定时器的信息。

4.1.1 定时器文件

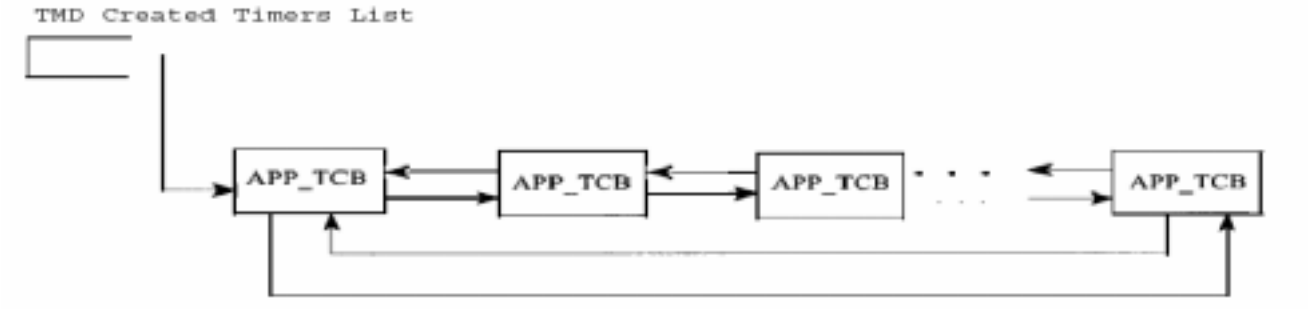
定时器模块由 9 个文件组成。定时器模块的源文件定义如下：

文件	说明
TM_DEFS.H	该文件包含定时器控制模块（TM）的常量和数据结构定义。
TM _EXTR.H	该文件包含定时器控制模块（TM）所有外部接口函数。
TMD.C	该文件定义了定时器控制模块（TM）全局数据结构体。
TMI.C	该文件包含定时器控制模块（TM）初始化函数。
TMF.C	该文件包含定时器控制模块（TM）信息收集函数。
TMC.C	该文件包含定时器控制模块（TM）核心函数，它用于处理基本定时器的启动和定时器的停止。
TMS.C	该文件包含定时器控制模块（TM）的补充函数，它使用的频率比核心函数少。
TMSE.C	该文件包含为补充函数(TMS.C)提供的错误检测函数接口。
TMT.[S,ASM,SRC]	该文件包含定时器控制模块（TM）所有的目标板独立函数。

4.1.2 定时器数据结构

创建定时器列表

可以动态创建和删除 Nucleus PLUS 操作系统函数应用层定时器。每个创建的定时器控制模块（APP_TCB）被保存在双链循环表中。最新建的定时器被放置在列表的末端，删除的定时器被从列表中移除。定时器列表的头指针是 TCD_Created_Times_List 。创建的定时器被专用于应用层定时器。



创建定时器列表保护

NUCLEUS PLUS 操作系统保护来自任务竞争或高级优先中断所创建定时器列表的完整性。这是通过使用内部数据保护结构体 TMD_List_Protect 来实现的。所有定时器的创建和删除都受到 TMD_List_Protect 保护。

字段描述:

```
TC_TCB *tc_tcb_pointer        /*确认当前线程已受到保护*/
UNSIGNED tct_head_waiting    /*标志一个或更多线程已正在等待保护*/
```

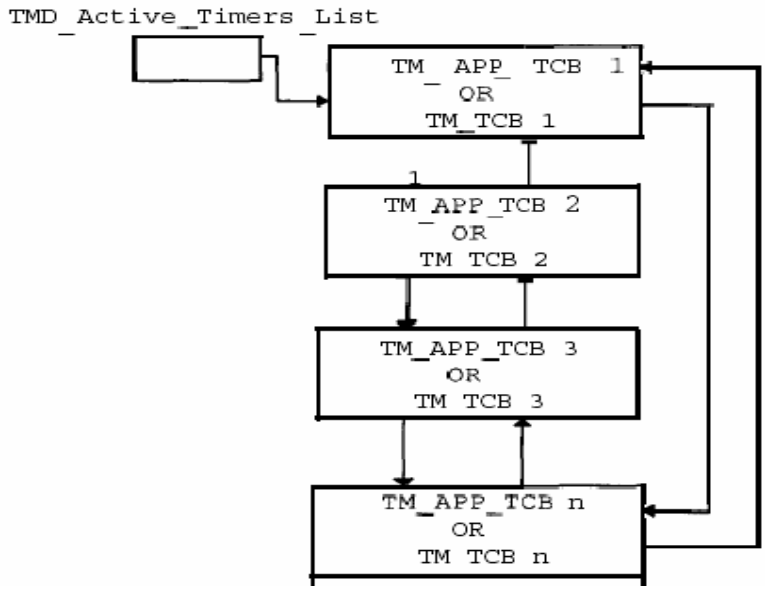
定时器总数目

变量TMD_Total_Timers表示当前创建NUCLEUS PLUS操作系统定时器的总数目。该变量值等于在已创建列表中的TCBS的数目。对该变量值的操作受到TMD_Created_List_Protect的保护。

有效的定时器列表 P95

NUCLEUS PLUS 操作系统有效定时器保存在双链循环表中。TMD_Active_Timers_List 表示列表的头指针。如果指针为 NULL，表示定时器无效。定时器列表支持应用层定时和任务定时。任务定时器结构在任务控制块 TCB 中。该定时器列表按顺序保留溢出时间。

剩余时间是溢出增量，非绝对时间。这样做的目的是避免在每个定时器中断时调整入口列表。定时器剩余时间为 0 时，被认为是时间用完。



有效列表忙

NUCLEUS PLUS 操作系统保护来自任务竞争或高级优先中断所创建定时器列表的完整性。通过使用保护标志 TMD_Active_List_Busy 来实现。所有有效定时器列表的增加和删除受到 TMD_Active_List_Busy 保护。

系统时钟

NUCLEUS PLUS 操作系统函数 TMD_System_C1ock 保持一个连续增加的系统时钟。每个定时器中断一次，时钟计数加一次。

定时器启动

NUCLEUS PLUS 操作系统在变量 TMD_Timer_Start 中存储最后定时器请求值。

定时器

变量 TMD_Timer 是一个倒数计数的定时器，它用于代表在系统中最小有效定时器值。当定时器溢出时，该变量值为零。

定时器状态

TMD_Timer_State 表示定时器状态变量值。如果状态有效，该定时器计数器递减。如果状态溢出，定时器 HISR 和定时器任务被启动去处理溢出。

时间片

NUCLEUS PLUS 操作系统使用变量 TMD_Time_S1ice 作为当前执行任务时间片倒计数值。当 TMD_Time_S1ice 值变零时，时间片处理启动。

时间片任务

TMD_Time_Slice_Task 是一个指向任务控制块 TCB 的时间片指针。当时间片定时器溢出时，该指针在定时器中断中建立。

时间片状态

NUCLEUS PLUS 操作系统使用 TMD_Time_S1ice_state 表示时间片变量状态。如果该状态

有效，时间片计数器递减。如果状态溢出，定时器 HISR 被启动去处理溢出。如果状态表示时间片没有效，就忽略时间片计数器。

HISR

TMD_HISR 是定时器高级中断服务例程 HISR 的控制块。

HISR 堆栈指针

TMD_HISR_Stack_Ptr 指向为定时器 HISR 保留的存储区。注意：这在 INT_Initialize 中建立。

HISR 堆栈大小

NUCLEUS PLUS 操作系统变量 TMD_HISR_Stack_size 确定分配给定时器 HISR 堆栈大小。注意：这在 INT_Initialize 中建立。

HISR 优先级

TMD_HISR_Priority 表示定时器 HISR 的优先级。优先级范围从 0 到 2，0 代表最高优先级。注意：这在 INT_Initialize 中建立。

定时器控制块

定时器控制块 TM_TCB 包含剩余时间和处理定时器请求的其他字段。

字段描述：

```
INT tm_timer_type
UNSIGNED tm_remaining_time
VOID *tm_information_struct
TM_TCB STRUCT *tm_next_timer
TM_TCB STRUCT *tm_previous_timer
```

字段概要：

字段	描述
tm_timer_type	表示如果定时器为了应用或一个任务。
tm_remaining_time	在前一个定时器溢出发生后，存储剩余时间数
*tm_information_struct	关于定时器信息的指针。这准确的溢出值是在有效列表中所有前一定时器的剩余时间总值。
*tm_next_timer	列表中下一个定时器的指针
*tm_previous_timer	列表中前一个定时器的指针

应用层定时器控制块

应用层定时器控制块 TM_APP_TCB 包含定时器满期服务例程指针和处理应用层定时器请求的其他字段

字段描述：

```
CS_NODE tm_created
UNSIGNED tm_id
CHAR tm_name[NU_MAX_NAME]
VOID (*tm_expiration_routine) (UNSIGNED)
UNSIGNED tm_expiration_id
INT tm_enabled
UNSIGNED tm_expirations
UNSIGNED tm_initial_time
UNSIGNED tm_reschedule_time
TM_TCB tm_actual_timer
```

字段概述：

字段	描述
tm_created	这是给定时器连接的节点结构体。它被连接到已创建的定时列表。该列表是一个双循环链表。

tm_id	这容纳内部定时器标识 0x54494D45，相当于 ASCII 时间。
tm_name	用户指定定时器名（八个字符）
*tm_expiration_routine	定时器溢出函数指针
tm_expiration_id	定时器溢出名
tm_enabled	如果定时器使能，就确立一个标志。
tm_expirations	这用于存储定时器溢出时间数。
tm_initial_time	用于存储定时器初始启动时间。
tm_reschedule_time	用于存储定时器重新预定时间。
tm_actual_timer	实际的定时器控制块 TCB

4.1.3 定时器函数

下面部分是对定时器控制模块（TM）做简要的概述。建议阅读源代码有助于获得更多的信息。

TMC_Init_Task_Timer

功能：该函数负责对挂起的任务定时器初始化。

语法：VOID TMC_Init_Task_Timer(TM_TCB *timer, VOID *information)

调用函数：无

TMC_Start_Tas k_Timer

功能：该函数负责启动一个任务定时器。注意由于该函数从任务控制块中被调用，这里有一些特殊的保护补偿。

语法：VOID TMC_Start_Task_Timer(TM_TCB *timer, UNSIGNED time)

调用函数：TMC_Start_Timer

TMC_Stop_Task_Timer

功能：该函数负责停止一个任务定时器。注意由于该函数从任务控制块中被调用，这里有一些特殊的保护补偿。

语法：VOID TMC_Stop_Task_Timer(TM_TCB *timer, UNSIGNED time)

调用函数：TMC_Stop_Timer

TMC_Start_Timer

功能：该函数负责启动应用和任务定时器。

语法：VOID TMC_Start_Timer(TM_TCB *timer, UNSIGNED time)

调用函数：

TMT_Read_Timer
TMT_Adjust_Timer
TMT_Enable_Timer

TMC_Stop_Timer　P101

功能：该函数负责停止应用和任务定时器。

语法：VOID TMC_Stop_Timer(TM_TCB *timer)

调用函数：TMT_Disable_Timer

TMC_Timer_HISR

功能：该函数负责对定时器溢出高级中断的处理。如果一个应用层定时器溢出，就调用定时器溢出函数。否则，如果时间片定时器溢出，就激活时间片处理。

语法：VOID TMC_Timer_HISR(VOID)

调用函数：

TCC_Time_Slice

```
TMC_Timer_Expiration
TMT_Retrieve
```

TMC_Timer_Expiration P102

功能：该函数负责对所有任务定时器溢出处理。这包含用于任务睡眠和时间溢出的应用层定时器和基本的任务定时器。

语法：`VOID TMC_Timer_Expiration(VOID)`

调用函数：

```
expiration_function
TMC_Stop_Timer
TCC_Task_Timeout
TMC_Start_Timer
TCT_System_Protect
TMT_Disable_Timer
TCT_Unprotect
TMT_Enable_Timer
```

TMF_Established_Timers

功能：该函数返回已建立定时器当前数。以前删除的定时器不再认为是已建立的定时器。

语法：`UNSIGNED TMF_Established_Timers(VOID)`

调用函数：`[TCT_Check_Stack]`

TMF_Timer_Pointers

功能：建立一个定时器指针列表。放在该列表中的定时器指针数等于定时器总数或在调用中指定的最大指针数。

语法：

```
UNSIGNED TMF_Timer_Pointers(NU_TIMER **pointer_list,
                            UNSIGNED maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TMF_Timer_Information P104

功能：该函数返回指定定时器的相关信息。然而，如果提供的定时器指针无效，该函数只返回一个错误状态。

语法：

```
STATUS TMF_Timer_Information(NU_TIMER *timer_ptr,
                             CHAR *name,
                             OPTION *enable,
                             UNSIGNED *expirations,
                             UNSIGNED *id,
                             UNSIGNED *initial_time,
                             UNSIGNED *reschedule_time)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TMI_Intialize

功能：该函数初始化控制时钟管理模块操作的数据结构。这里最初没有应用时钟被创建。
语法：VOID TMI_Initialize(VOID)
调用函数：

```
ERC_System_Error
TCC_Create_HISR
TCCE_Create_HISR
```

TMS_Create_Timer

功能：该函数创建一个应用定时器并把它放在已创建的定时器队列中。可以通过指定使能参数来使该定时器有效。
语法：

```
STATUS TMS_Create_Timer(NU_TIMER *timer_ptr,
                        CHAR *name,
                        VOID (*expiration_routine)(UNSIGNED),
                        UNSIGNED id,
                        UNSIGNED initial_time,
                        UNSIGNED reschedule_time,
                        OPTION enable)
```

调用函数：

```
CSC_Place_On_List
[HIC_MakeIHistory_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
TMS_Control_Timer
```

TMS_Delete_Timer　P105

功能：该函数删除一个应用定时器并把它从已创建的定时器队列中移除。使该定时器有效。
语法：STATUS TMS_Delete_Timer(NU_TIMER *timer_ptr)
调用函数：

```
CSC_Remove_From_List
[HIC_Make_iistory_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_System_Protect
TCT_Unprotect
```

TMS_Reset_Timer

功能：该函数复位指定的应用定时器。注意在禁止状态下的定时器优于该调用。如果使能参数指定为自动激活，复位之后该定时器有效。
语法：

```
STATUS TMS_Reset_Timer(NU_TIMER *timer_ptr,
                       VOID (*expiration_routine)(UNSIGNED),
                       UNSIGNED initial_time,
```

```
                            UNSIGNED Reschedule_time,
                            OPTION enable)
```

调用函数：

```
        [HIC_Make_History_Entry]
        [TCT_Check_Stack]
        TCT_System_Protect
        TCT_Unprotect
        TMS_Control_Timer
```

TMS_Control_Timer　P106

功能：该函数既能使能也可以禁止指定定时器。如果该定时器已经在所期望的状态下，就保持原样（simply leave it alone.）。

语法：`STATUS TMS_Control_Timer(NU_TIMER *app_timer, OPTION enable)`

调用函数：

```
       [HIC_Make_History_Entry]
       [TCT_Check_Stack]
       TCT_System_Protect
       TCT_Unprotect
       TMC_Start_Timer
       TMC_Stop_Timer
```

TMSE_Create_Timer

功能：该函数通过参数来对创建定时器函数进行错误检测。

语法：

```
    STATUS TMSE_Create_Timer(NU_TIMER *timer_ptr,
                             CHAR *name,
                             VOID (*expiration_routine)(UNSIGNED),
                             UNSIGNED id,
                             UNSIGNED initial_time,
                             UNSIGNED reschedule_time,
                             OPTION enable)
```

调用函数：

```
        TMS_Create_Timer
```

TMSE_Delete_Timer　P107

功能：该函数通过参数来对删除定时器函数进行错误检测。

语法：`STATUS TMSE_Delete_Timer(NU_TIMER *timer_ptr)`

调用函数：`TMS_Delete_Timer`

TMSE_Reset_Timer

功能：该函数通过参数来对复位定时器函数进行错误检测。

语法：

```
   STATUS TMSE_Reset_Timer(NU_TIMER *timer_ptr,
                           VOID (*expiration_routine)(UNSIGNED),
                           UNSIGNED initial_time,
                           UNSIGNED Reschedule_time,
                           OPTION enable)
```

调用函数：
TMS_Reset_Timer

TMSE_Control_Timer

功能：该函数通过参数来对控制定时器函数进行错误检测。
语法：STATUS TMSE_Control_Timer(NU_TIMER *timer_ptr,OPTION enable)
调用函数：TMS_Control_Timer

TMT_Set_Clock　P108

功能：该汇编函数给系统时钟设置指定值。
语法：VOID TMT_Set_Clock(UNSIGNED new_value)
调用函数：无

TMT_Retrieve_Clock

功能：该汇编函数返回系统时钟当前值。
语法：UNSIGNED TMT_Retrieve_Clock(void)
调用函数：无

TMT_Read_Timer

功能：该汇编函数返回倒计数定时器当前值。
语法：UNSIGNED_TMT_Read_Timer(void)
调用函数：无

TMT_Enabie_Timer　P109

功能：该汇编函数使能已指定值的倒计数定时器。
语法：VOID TMT_Enable_Timer(UNSIGNED time)
调用函数：无

TMT_Adjust_Timer

功能：如果新值比当前的值小，该汇编函数调整已指定值的倒计数定时器。
语法：VOID TMT_Adjust_Timer(UNSIGNED time)
调用函数：无

TMT_Disable_Timer

功能：该汇编函数禁止倒计数定时器。
语法：VOID TMT_Disable_Timer(void)
调用函数：无

TMT_Retrieve_TS_Task　P110

功能：该汇编函数返回时间片任务指针。
语法：NU_TASK *TMT_Retrieve_TS_Task(VOID)
调用函数：无

TMT_Timer_Interrupt

功能：该汇编函数处理实际的硬件中断。处理包括更新系统时钟、倒计数定时器和时间片定时器。如果一个或两个定时器期满，定时器HISR就有效。
语法：VOID TMT_Timer_Interrupt(void)

调用函数：

TCT_Activate_HISR
TCT_Interrupt_Context_Save
TCT_Interrupt_Context_Restore

4.5邮箱(MB)

邮箱组件负责对所有的NUCLEUS PLUS 邮箱设备进行处理。一个NUCLEUS PLUS 邮箱是为内部任务之间通信提供低消耗机制。每个邮箱能容纳控制一条消息。一条邮箱消息由4个32位字组成。任务从一个空邮箱中等待一条消息时，就会挂起。相反地，当任务试图给一个已有消息的邮箱发送消息时，任务也会挂起。邮箱可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解邮箱更加详细的内容。

4.5.1邮箱文件

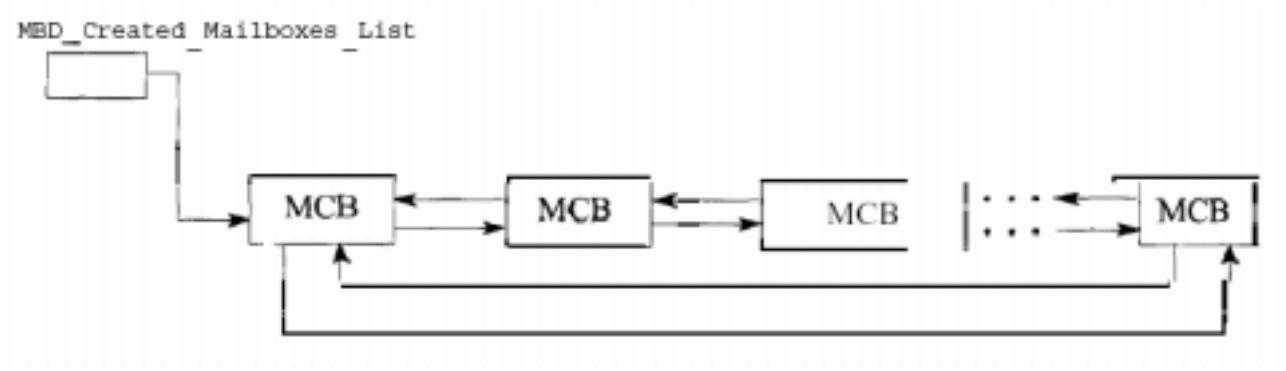
邮箱组件由9个文件构成。邮箱组件每个源文件定义如下：

文件	描述
MB_DEFS.H	该文件包含MB特殊的常量和数据结构的定义。
MB_EXTR.H	MB的所有外部接口函数都定义在该文件里。
MBD.C	MB的全局数据结构定义在这个文件里。
MBI.C	该文件包含了MB的初始化函数。
MBF.C	该文件包含了获取MB信息的函数。
MBC.C	该文件包含了MB所有核心函数。处理基本的发送给邮箱和从邮箱接收服务的函数定义在这个文件里。
MBS.C	该文件包含了MB的补充函数。该函数使用频率比核心函数少。
MBCE.C	该文件包含了定义在MBC.C中的核心函数与错误检测函数的接口。
MBSE.C	该文件包含了定义在MBS.C中的补充函数与错误检测函数的接口。

4.5.2邮箱数据结构

创建邮箱队列

NUCLEUS PLUS邮箱可以动态创建和删除。每个已创建邮箱的控制块被保存在双链循环表中。新创建的邮箱被放在列表的末端，而删除的邮箱被完全从列表中移除。邮箱列表的头指针是MBD_Created_Mailboxes_List。



创建邮箱队列保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建邮箱时列表的完整性。这是通过使用内部保护结构MBD_List_Protect来实现。所有的邮箱创建和删除是在MBD_List_Protect的保护下操作。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述:

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

邮箱总数

当前已创建的NUCLEUS PLUS邮箱的总数存放在变量MBD_Total_Mailboxes中。该变量的内容等于已创建列表中MCBS的数目。该变量的操作受到MBD_List_Protect的保护。

邮箱控制模块

邮箱控制块MB_MCB包含邮箱信息区域（4个32位无字符字）。以及处理邮箱请求所需的字符段。

字符段描述：

```
CS_NODE mb_created
UNSIGNED mb_id
CHAR mb_name[NU_MAX_NAME]
DATA_ELEMENT mb_message_present
DATA_ELEMENT mb_fifo_suspend
DATA_ELEMENT mb_padding[PAD_2]
UNSIGNED mb_tasks_waiting
UNSIGNED mb_message_area[MB_MESSAGE_SIZE]
struct MB_SUSPEND_STRUCT *mb_suspension_list
```

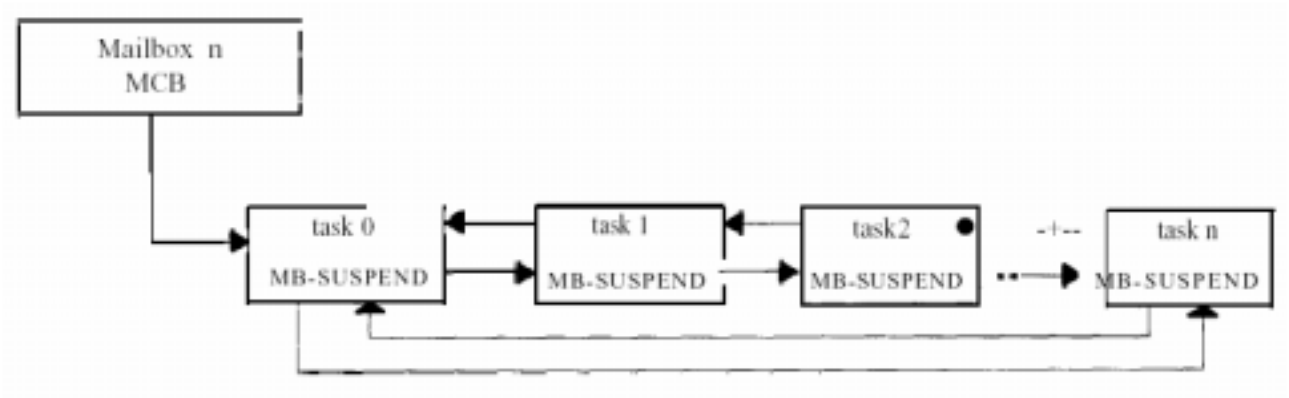
字符段概述：

字符段	描述
mb_created	它为邮箱提供连接节点结构。它是一个双链循环表，用于连接已经创建的邮箱。
mb_id	这是用来控制内部邮箱0x4D424F58的标识，相当于ASCII邮箱。
mb_name	这是由用户为邮箱指定的，8个字符名。
mb_message_present	表示目前有一条消息在邮箱中的标志。
mb_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
mb_padding	这是用于在一个偶数临界边缘上，调整邮箱结构。在某些端口，该字符端不用。
mb_tasks_waiting	表示在邮箱上当前被挂起的任务数。
mb_message_area	邮箱存储区域。
*mb_suspension_list	邮箱挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

邮箱挂起结构　P114

任务可以在邮箱为空或满的时候挂起。在挂起过程中，一个MB_SUSPEND结构被建立。该结构包含了在任务和任务邮箱请求挂起时的有关信息。在双链循环表中该挂起结构被连接到MCB，并分配正在挂起任务的堆栈。这里为每一个在邮箱上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于邮箱的创建。如果有邮箱以FIFO挂起，挂起块被加到列表的末端。否则，如果邮箱以优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
CS_NODE mb_suspend_link
MB_MCB *mb_mailbox
TC_TCB *mb_suspended_task
UNSIGNED *mb_message_area
STATUS mb_return_status
```

字符段概述：

字符段	描述
mb_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*mb_mailbox	一个邮箱结构指针。
*mb_suspended_task	已挂起的任务的任务控制块的指针。
*mb_message_area	该指针表示已挂起的任务消息缓冲的位置。
mb_return_status	表示任务在邮箱上挂起的完成状态。

4.5.3邮箱函数

下面部分对邮箱组件（MB）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

MBC_Create_Mailbox　P115

功能：创建一个邮箱并把它放在已创建邮箱列表中。

语法：

```
STATUS MBC_Create_Mailbox(NU_MAILBOX *mailbox_ptr,
                          CHAR *name,
                          OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIz_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

MBC_Delete_Mailbox　P116

功能：该函数删除一个邮箱并把它从已创建的邮箱列表中移除。所有在邮箱上挂起的任务被恢复。注该函数并没有释放与邮箱控制块相关联的内存。这是由应用层负责的。

语法：`STATUS MBC_Delete_Mailbox(NU_MAILBOX *mailbox_ptr)`

调用函数：

```
CSC_Remove_From_List            TCT_Protect
[HIC_Make_History_Entry]        TCT_Set_Current_Protect
TCC_Resume_Task                 [TCT_Check_Stack]
TCT_System_Protect              TCT_System_Unprotect
TCT_Control_To_System           TCT_Unprotect
```

MBC_Send_To_Mailbox

功能：该函数发送一条4个字的消息给指定的邮箱。如果在邮箱上有一个或多个任务为了一条消息挂起，当这条消息拷贝到第一任务消息区域时，该任务就恢复。如果邮箱满，正在调用的任务就有可能挂起。

语法：

```
STATUS MBC_Send_To_Mailbox(NU_MAILBOX *mailbox_ptr,
                           VOID *message,
                           UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List
CSC_Priority_Place_On_List
TCC_Task_Priority
CSC_Remove_From_List
[TCT_Check_Stack]
[HIC_Make_History_Entry]
TCT_Control_To_System
TCT_Current_Thread
TCC_Resume_Task
TCC_Suspend_Task
TCT_System_Protect
TCT_Unprotect
```

MBC_Receive_From_Mailbox　P117

功能：该函数从指定的邮箱中接收一条消息。如果当前有一条消息在邮箱中，该消息被从邮箱中移出并放入调用区域。否则，如果目前邮箱中没有消息，正在调用的任务就有可能挂起。

语法：

```
STATUS MBC_Receive_From_Mailbox(NU_MAILBOX *mailbox_ptr,
                                VOID *message,
                                UNSIGNED suspend)
```

调用函数:

```
CSC_Place_On_List
CSC_Priority_Place_On_List
TCC_Task_Priority
CSC_Remove_From_List
[TCT_Check_Stack]
[HIC_Make_History_Entry]
TCT_Control_To_System
TCT_Current_Thread
TCC_Resume_Task
TCC_Suspend_Task
```

```
TCT_System_Protect
TCT_Unprotect
```

MBC_Cleanup

功能：该函数负责从一个邮箱中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：`VOID MBC_Cleanup(VOID *information)`

调用函数：`CSC_Remove_From_List`

MBCE_Create_Mailbox　P118

功能：该函数通过参数对邮箱创建函数执行错误检测。

语法：

```
STATUS MBCE_Create_Mailbox(NU_MAILBOX *mailbox_ptr,
                           CHAR *name,
                           OPTION suspend_type)
```

调用函数：

```
MBC_Create_Mailbox
```

MBCE_Delete_Mailbox

功能：该函数通过参数对实际删除邮箱函数执行错误检测。

语法：

```
STATUS MBCE_Delete_Mailbox(NU_MAILBOX *mailbox_ptr)
```

调用函数：

```
MBC_Delete_Mailbox
```

MBCE_Send_To_Mailbox　P119

功能：该函数通过参数对发送给邮箱函数执行错误检测。

语法：

```
STATUS MBCE_Send_To_Mailbox(NU_MAILBOX *mailbox_ptr,
                            VOID *message,
                            UNSIGNED suspend)
```

调用函数：

```
MBC_Sent_To_Mailbox
TCCE_Suspend_Error
```

MBCE_Receive_From_Mailbox

功能：该函数通过参数对从邮箱接收函数执行错误检测。

语法：

```
STATUS MBCE_Receive_From_Mailbox(NU_MAILBOX *mailbox_ptr,
                                 VOID *message,
                                 UNSIGNED suspend)
```

调用函数：

```
MBC_Receive_From_Mailbox
TCCE_Suspend_Error
```

MBF_Established_Mailboxes　P120

功能：返回已建立邮箱的当前数目。以前删除的邮箱不再被认为已建立。

语法：UNSIGNED MBF_Established_Mailboxes(VOID)
调用函数：[TCT_Check_Stack]

MBF_Mailbox_Pointers

功能：建立起始于指定位置的邮箱指针列表。放在邮箱列表中邮箱个数等于邮箱总数或是在调用中指定的指针最大个数。
语法：

```
UNSIGNED MBF_Mailbox_Pointers(NU_MAILBOX **pointer_list,
                              UNSIGNED maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

MBF_Maiibox_Information　P121

功能：返回指定邮箱的相关信息。然而，如果所支持的邮箱指针无效，该函数只返回一个错误状态。
语法：

```
STATUS MBF_Mailbox_Information(NU_MAILBOX *mailbox_ptr,
                               CHAR *name,
                               OPTION *suspend_type,
                               DATA_ELEMENT *message_present,
                               UNSIGNED *tasks_waiting,
                               NU_TASK **first_task)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

MBI_Initialize

功能：该函数初始化控制邮箱模块操作的数据结构。这里最初没有邮箱。
语法：VOID MBI_Initialize(VOID)
调用函数：无

MBS_Reset_Mailbox　P122

功能：该函数把邮箱复位到初始状态。在邮箱中任一消息就会丢失。在邮箱上挂起的任务也被恢复到复位完成状态。
语法：STATUS MBS_Reset_Mailbox(NU_MAILBOX *mailbox_ptr)
调用函数：

```
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect
```

MBS_Broadcast_To_Mailbox

功能：该函数给目前正在等待邮箱消息的所有任务发送一条消息。如果没有任务等待，该服务就像通常的发送例程。

语法：
```
STATUS MBS_Broadcast_To_Mailbox(NU_MAILBOX *mailbox_ptr,
                                VOID *message
                                UNSIGNED suspend)
```

调用函数：

CSC_Place_On_List	CSC_Priority_Place_On_List
CSC_Remove_From_List	[HIC_Make_History_Entry]
TCC_Resume_Task	TCC_Suspend_Task
TCC_Task_Priority	[TCT_Check_Stack]
TCT_Control_To_System	TCT_Current_Thread
TCT_System_Protect	TCT_Unprotect

MBSE_Reset_Mailbox　P123

功能：该函数通过参数对实际复位邮箱函数执行错误检测。

语法：
```
STATUS MBSE_Reset_Mailbox(NU_MAILBOX *mailbox_ptr)
```

调用函数：

MBS_Reset_Mailbox

MBSE_Broadcast_To_Mailbox

功能：该函数通过参数对邮箱广播函数执行错误检测。

语法：
```
STATUS MBSE_Broadcast_To_Mailbox(NU_MAILBOX *mailbox_ptr,
                                 VOID *message,
                                 UNSIGNED suspend)
```

调用函数：

MBS_Broadcast_To_Mailbox
TCCE_Suspend_Error

4.6 队列(QM)

队列组件负责对所有的NUCLEUS PLUS队列设备进行处理。一个NUCLEUS PLUS 队列是为任务之间通信提供机制。每个队列能容纳控制多条消息。一条队列消息由一个或多个32位字组成。任务从一个空队列中等待一条消息时，就会挂起。相反地，当任务试图给一个已满队列发送消息时，任务也会挂起。队列可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解队列更加详细的内容。

4.6.1队列文件

队列组件（QU）由9个文件构成。队列组件每个源文件定义如下：

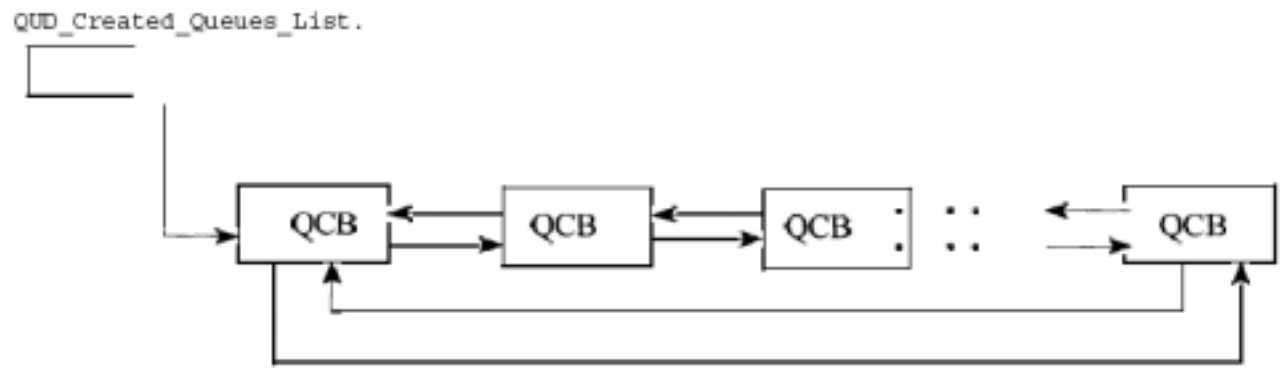
文件	描述
QU_DEFS.H	该文件包含QU特殊的常量和数据结构的定义。
QU_EXTR.H	QU的所有外部接口函数都定义在该文件里。
QUD.C	QU的全局数据结构定义在这个文件里。
QUI.C	该文件包含了QU的初始化函数。
QUF.C	该文件包含了获取QU信息的函数。
QUC.C	该文件包含了QU所有核心函数。它用于处理基本的队列发送和队列接收服务。

QUS.C	该文件包含了QU的补充函数。该函数使用频率比核心函数少。
QUCE.C	该文件包含了定义在QUC.C中的核心函数与错误检测函数的接口。
QUSE.C	该文件包含了定义在QUS.C中的补充函数与错误检测函数的接口。

4.6.2队列数据结构

创建队列列表

NUCLEUS PLUS队列可以动态创建和删除。每个已创建队列的控制块（QCB）被保存在双链循环表中。新创建的队列被放在列表的末端，而删除的队列被完全从列表中移除。队列列表的头指针是QUD_Created_Queues_List。



创建队列列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建队列时列表的完整性。这是通过使用内部保护结构QUD_List_Protect来实现。所有的队列创建和删除是在QUD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

队列总数

当前已创建的NUCLEUS PLUS队列的总数存放在变量QUD_Total_Queues中。该变量的内容等于已创建列表中QCBS的数目。该变量的操作受到QUD_List_Protect的保护。

队列控制模块

队列控制块QU_QCB包含队列信息区域（一个或多个32位无符号字）。以及处理队列请求所需的字符段。

字符段描述：

```
CS_NODE qu_created
UNSIGNED qu_id
CHAR qu_name[NU_MAX_NAME]
DATA_ELEMENT qu_fixed_size
DATA_ELEMENT qu_fifo_suspend
DATA_ELEMENT qu_padding
UNSIGNED qu_queue_size
```

```
UNSIGNED qu_message
UNSIGNED qu_message_size
UNSIGNED qu_available
UNSIGNED_PTR qu_start
UNSIGNED_PTR qu_end
UNSIGNED_PTR qu_read
UNSIGNED_PTR qu_write
UNSIGNED qu_tasks_waiting
UNSIGNED qu_message_area[QU_MESSAGE_SIZE]
struct QU_SUSPEND_STRUCT *qu_urgent_list
struct QU_SUSPEND_STRUCT *qu_suspension_list
```

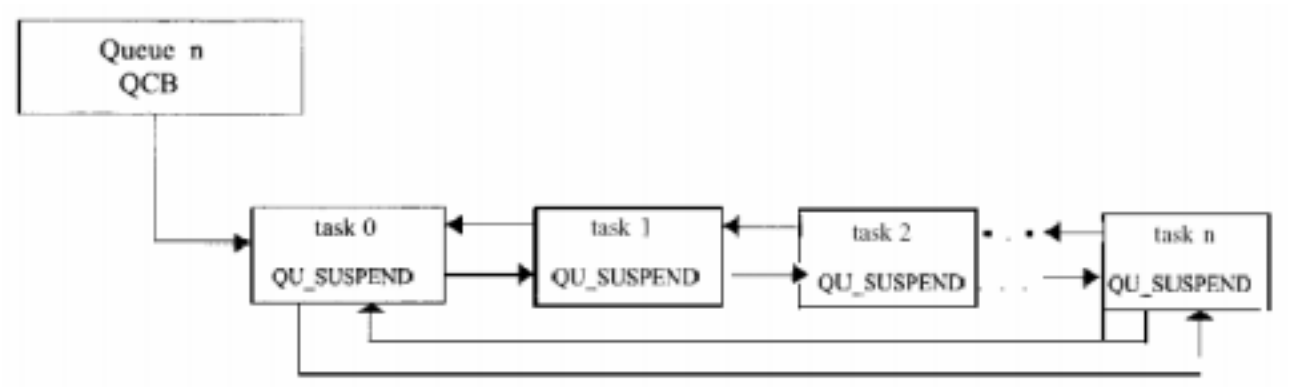
字符段概述：

字符段	描述
qu_created	它为队列提供连接节点结构。它是一个双链循环表，用于连接已经创建的队列。
qu_id	这是用来控制内部队列0x4D424F58的标识，相当于ASCII队列。
qu_name	这是由用户为队列指定的，8个字符名。
qu_fixed_size	该标志表示队列的大小是固定的还是可变的。
qu_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
qu_padding	这是用于在一个偶数临界边缘上，调整队列结构。在某些端口，该字符端不用。
qu_queue_size	队列总大小。
qu_message	该标志表示队列中是否有一条消息。
qu_message_size	控制队列消息的大小。
qu_available	说明在队列中有多少个字节有效。
qu_start	存储队列的起始端。
qu_end	存储队列的末端。
qu_read	读指针
qu_write	写指针
qu_tasks_waiting	表示当前在队列上被挂起的任务数。
*qu_urgent_list	表示紧急消息挂起队列指针。
*qu_suspension_list	队列挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

队列挂起结构　P128

任务可以在队列为空或满的情况下挂起。在挂起过程中，一个QU_SUSPEND结构被建立。该结构包含了在任务和任务队列请求挂起时的有关信息。在双链循环表中该挂起结构被连接到QCB，并分配正在挂起任务的堆栈。这里为每一个在队列上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于队列的创建。如果队列选择以先进先出FIFO挂起，挂起块被增加到列表的末端。否则，如果队列选择以优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
QU_SUSPEND_STRUCT
CS_NODE qu_suspend_link
QU_QCB *qu_queue
TC_TCB *qu_suspended_task
UNSIGNED_PTR *qu_message_area
UNSIGNED qu_message_size
UNSIGNED qu_actual_size
STATUS qu_return_status
```

字符段概述：

字符段	描述
qu_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*qu_queue	一个队列结构指针。
*qu_suspended_task	已挂起的任务的任务控制块的指针。
*qu_message_area	该指针表示已挂起的任务消息缓冲的位置。
qu_message_size	存储请求消息的大小。
qu_actual_size	存储消息的实际大小。
qu_return_status	表示任务在队列上挂起的完成状态。

4.6.3 队列函数

下面部分对队列组件（QU）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

QUC_Create_Queue　P129

功能：该函数创建一个队列并把它放到已创建队列列表中。

语法：

```
STATUS QUC_Create_Queue(NU_QUEUE *queue_ptr,
                        CHAR *name,
                        VOID *start_address,
                        UNSIGNED queue_size,
                        OPTION message_type,
                        UNSIGNED message_size,
                        OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
```

```
TCT_Unprotect
```

QUC_Delete_Queue

功能：该函数删除一个队列并把它从已创建的队列列表中移除。所有在队列上挂起的任务被恢复。注该函数并没有释放与队列控制块相关联的内存。这是由应用层负责的。

语法：`STATUS QUC_Delete_Queue(NU_QUEUE *queue_ptr)`

调用函数：

```
CSC_Remove_From_List            TCT_Protect
[HIC_Make_History_Entry]        TCT_Set_Current_Protect
TCC_Resume_Task                 [TCT_Check_Stack]
TCT_System_Protect              TCT_System_Unprotect
TCT_Control_To_System           TCT_Unprotect
```

QUC_Send_To_Queue

功能：该函数发送一条消息给指定的队列。消息长度取调用。决如果在队列上有一个或多个任务为了一条消息挂起，当这条消息拷贝到第一任务消息区域时，该任务就恢复。如果队列不能控制消息，正在调用的任务对调用者有可选性。

语法：

```
STATUS QUC_Send_To_Queue(NU_QUEUE *queue_ptr,
                         VOID *message,
                         UNSIGNED size,
                         UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List               TCC_Task_Priority
CSC_Priority_Place_On_List      [TCT_Check_Stack]
CSC_Remove_From_List            TCT_Control_To_System
[HIC_Make_History_Entry]        TCT_Current_Thread
TCC_Resume_Task                 TCT_System_Protect
TCC_Suspend_Task                TCT_Unprotect
```

QUC_Receive_From_Queue　P131

功能：该函数从指定的队列中接收一条消息。消息大小在调用时指定。如果当前有一条消息在队列中，该消息被从队列中移出并放入调用区域。如果该请求没有得到满足，就有可能挂起。

语法：

```
STATUS QUC_Receive_From_Queue(NU_QUEUE *queue_ptr,
                              VOID *message,
                              UNSIGNED size,
                              UNSIGNED *actual size,
                              UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List               TCC_Task_Priority
CSC_Priority_Place_On_List      [TCT_Check_Stack]
CSC_Remove_From_List            TCT_Control_To_System
[HIC_Make_History_Entry]        TCT_Current_Thread
TCC_Resume_Task                 TCT_System_Protect
```

TCC_Suspend_Task　　TCT_Unprotect

QUC_Cleanup

功能：该函数负责从一个队列中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：VOID QUC_Cleanup(VOID *information)

调用函数：CSC_Remove_From_List

QUCE_Create_Queue　P132

功能：该函数通过参数对队列创建函数执行错误检测。

语法：

```
STATUS QUCE_Create_Queue (NU_QUEUE *queue_ptr,
                          CHAR *name,
                          VOID *start_address,
                          UNSIGNED queue_size,
                          OPTION message_type,
                          UNSIGNED message_size,
                          OPTION suspend_type)
```

调用函数：

QUC_Create_Queue

QUCE_Delete_Queue

功能：该函数通过参数对队列删除函数执行错误检测。

语法：

```
STATUS QUCE Delete_Queue(NU_QUEUE *queue_ptr)
```

调用函数：

QUC_Delete_Queue

QUCE_Send_To_Queue　P133

功能：该函数通过参数对给队列发送消息的函数执行错误检测。

语法：

```
STATUS QUCE_Send_To_Queue(NU_QUEUE *queue_ptr,
                          VOID *message,
                          UNSIGNED size,
                          UNSIGNED suspend)
```

调用函数：

QUC_Send_To_Queue
TCCE_Suspend_Error

QUCE_Receive_From_Queue

功能：该函数通过参数对从队列接收消息的函数执行错误检测。

语法：

```
STATUS QUCE_Receive_From_Queue(NU_QUEUE *queue_ptr,
                               VOID *message,
                               UNSIGNED size,
                               UNSIGNED *actual_size,
                               UNSIGNED suspend)
```

调用函数：

QUC_Receive_From_Queue
CCE_Suspend_Error

QUF_Established_Queues　P134

功能：该函数返回已建立队列的当前数目。以前删除的队列不再被认为已建立。
语法：UNSIGNED QUF_Established_Queues(VOID)
调用函数：[TCT_Check_Stack]

QUF_Queue_Information

功能：该函数返回指定队列的相关信息。然而，如果所支持的队列指针无效，该函数只返回一个错误状态。
语法：

```
STATUS QUF_Queue_Information(NU_QUEUE *queue_ptr,
                             CHAR *name,
                             VOID **start_address,
                             UNSIGNED *queue_size,
                             UNSIGNED "available,
                             UNSIGNED *messages,
                             OPTION *message_type,
                             UNSIGNED *message_size,
                             OPTION *suspend_type,
                             UNSIGNED *tasks_waiting,
                             NU_TASK **first_task)
```

调用函数：

[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect

QUF_Queue_Pointers　P135

功能：建立起始于指定位置的队列指针列表。放在队列列表中队列个数等于队列总数或是在调用中指定的指针最大个数。
语法：

```
UNSIGNED QUF_Queue_Pointers(NU_QUEUE **pointer_list,
                            UNSIGNED maximum_pointers)
```

调用函数：

[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

QUI_Initialize

功能：该函数初始化控制队列模块操作的数据结构。这里最初没有队列。
语法：VOID QUI_Initialize(VOID)
调用函数：无

QUS_Reset_Queue　P136

功能：该函数把队列复位到初始状态。在队列中任一消息就会丢失。在队列上挂起的任务

也被恢复到复位完成状态。

语法：STATUS QUS_Reset_Queue(NU_QUEUE *queue_ptr)

调用函数：

[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect

QUS_Send_To_Front_Of_Queue

功能：该函数发送一条消息到指定消息队列的前端。消息的长度在调用时决定。如果在队列上有一个或多个任务为了一条消息挂起，当这条消息拷贝到第一任务消息区域时，该任务就恢复。如果队列中有足够的空间，该消息就被拷贝到所有其他消息的前面。如果队列中没有足够的空间，调用就有可能挂起。

语法：

```
STATUS QUS_Send_To_Front_Of_Queue(NU_QUEUE *queue_ptr,
                                  VOID *message,
                                  UNSIGNED size,
                                  UNSIGNED suspend)
```

调用函数：

CSC_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Current_Thread
TCT_System_Protect
TCT_Unprotect

QUS_Broadcast_To_Queue　P137

功能：该函数给目前正在等待指定队列消息的所有任务发送一条消息。如果没有任务等待，该服务就像标准的发送例程。

语法：

```
STATUS QUS_Broadcast_To_Queue(NU_QUEUE *queue_ptr,
                              VOID *message,
                              UNSIGNED size,
                              UNSIGNED suspend)
```

调用函数：

CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
TCC_Task_Priority
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Current_Thread
TCT_System_Protect
TCT_Unprotect

QUSE_Reset_Queue

功能：该函数通过参数对队列复位的函数执行错误检测。

语法：STATUS QUSE_Reset_Queue(NU_QUEUE *queue_ptr)

调用函数：

QUS_Reset_Queue

QUSE_Send_To_Front_Of_Queue P138

功能：该函数通过参数对发送消息到队列广播消息前端的函数执行错误检测。

语法：

```
STATUS QUSE_Send_To_Front_Of_Queue(NU_QUEUE *queue_ptr,
                                   VOID *message,
                                   UNSIGNED size,
                                   UNSIGNED suspend)
```

调用函数：

QUS_Send_To_Front_Of_Queue
TCCE_Suspend_Error

QUSE_Broadcast_To_Queue

功能：该函数通过参数对队列广播消息的函数执行错误检测。

语法：

```
STATUS QUSE_Broadcast_To_Queue(NU_QUEUE *queue_ptr,
                               VOID *message,
                               UNSIGNED size,
                               UNSIGNED suspend)
```

调用函数：

QUS_Broadcast_To_Queue
TCCE_Suspend_Error

4.7 管道(PI)

管道组件负责对所有的NUCLEUS PLUS管道设备进行处理。一个NUCLEUS PLUS 管道是为任务之间通信提供一种机制。每个管道能容纳控制多条消息。一条管道消息由一个或多个字节组成。任务从一个空管道中等待一条消息时，就会挂起。相反地，当任务试图给一个已满管道发送消息时，任务也会挂起。管道可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解管道更加详细的内容。

4.7.1管道文件

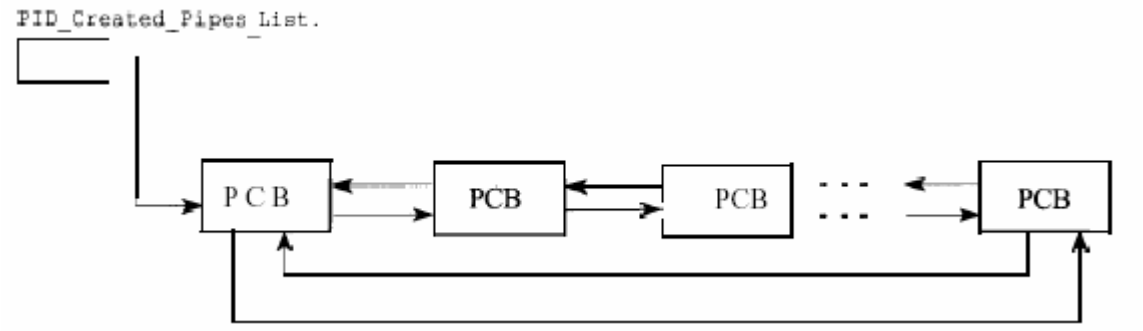
管道组件（PI）由9个文件构成。管道组件每个源文件定义如下：

文件	描述
PI_DEFS.H	该文件包含PI特殊的常量和数据结构的定义。
PI_EXTR.H	PI的所有外部接口函数都定义在该文件里。
PID.C	PI的全局数据结构定义在这个文件里。
PII.C	该文件包含了PI的初始化函数。
PIF.C	该文件包含了获取PI信息的函数。
PIC.C	该文件包含了PI所有核心函数。它用于处理基本的管道发送和管道接收服务。
PIS.C	该文件包含了PI的补充函数。该函数使用频率比核心函数少。
PICE.C	该文件包含了定义在PIC.C中的核心函数与错误检测函数的接口。
PISE.C	该文件包含了定义在PIS.C中的补充函数与错误检测函数的接口。

4.7.2管道数据结构

创建管道列表

NUCLEUS PLUS管道可以动态创建和删除。每个已创建管道的控制块（PCB）被保存在双链循环表中。新创建的管道被放在列表的末端，而删除的管道被完全从列表中移除。管道列表的头指针是PID_Created_Pipes_List。



创建管道列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建管道时列表的完整性。这是通过使用内部保护结构PID_List_Protect来实现。所有的管道创建和删除是在PID_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述：

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

管道总数

当前已创建的NUCLEUS PLUS管道的总数存放在变量PID_Total_Pipes中。该变量的内容等于已创建列表中PCBS的数目。该变量的操作受到PID_List_Protect的保护。

管道控制模块

管道控制块PI_PCB包含管道信息区域（一个或多个字节）。以及处理管道请求所需的字符段。

字符段描述：

```
CS_NODE pi_created
UNSIGNED pi_id
CHAR pi_name[NU_MAX_NAME]
DATA_ELEMENT pi_fixed_size
DATA_ELEMENT pi_fifo_suspend
DATA_ELEMENT pi_padding[PAD_2]
UNSIGNED pi_pipe_size
UNSIGNED pi_message_size
UNSIGNED pi_available
BYTE_PTR pi_start
BYTE_PTR pi_end
```

```
    BYTE_PTR pi_read
    BYTE_PTR pi_write
    UNSIGNED pi_tasks_waiting
    UNSIGNED pi_messages
    UNSIGNED pi_message_area[PI_MESSAGE_SIZE]
    struct PI_SUSPEND_STRUCT *pi_urgent_list
    struct PI_SUSPEND_STRUCT *pi_suspension_list
```

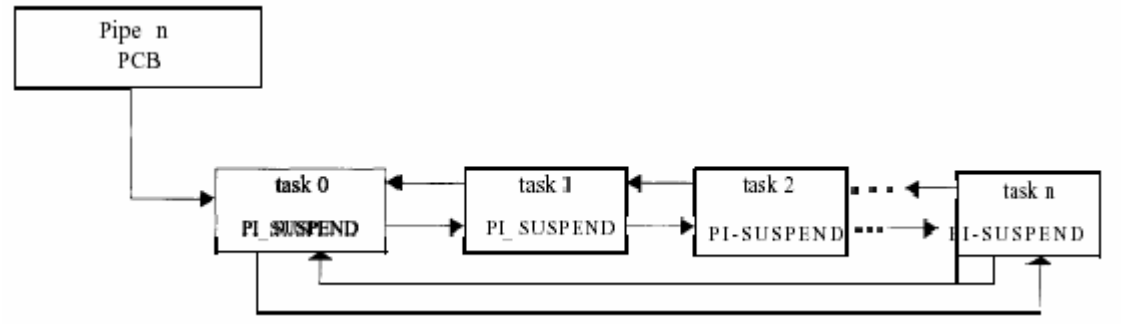
字符段概述：

字符段	描述
pi_created	它为管道提供连接节点结构。它是一个双链循环表，用于连接已经创建的管道。
pi_id	这是用来控制内部管道0x4D424F58的标识，相当于ASCII管道。
pi_name	这是由用户为管道指定的，8个字符名。
pi_fixed_size	该标志表示管道的大小是固定的还是可变的。
pi_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
pi_padding	这是用于在一个偶数临界边缘上，调整管道结构。在某些端口，该字符端不用。
pi_pieue_size	管道总大小。
pi_messages	该标志表示管道中是否有一条消息。
pi_message_size	控制管道消息的大小。
pi_available	说明在管道中有多少个字节有效。
pi_start	存储管道的起始端。
pi_end	存储管道的末端。
pi_read	读指针
pi_write	写指针
pi_tasks_waiting	表示当前在管道上被挂起的任务数。
*pi_urgent_list	表示紧急消息挂起管道指针。
*pi_suspension_list	管道挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

管道挂起结构　P143

任务能在管道为空或满的时候挂起。在挂起过程中，一个PI_SUSPEND结构被建立。该结构包含了在任务和任务管道请求挂起时的有关信息。在双链循环表中该挂起结构被连接到PCB，并分配正在挂起任务的堆栈。这里为每一个在管道上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于管道的创建。如果有管道以FIFO挂起，挂起块被加到列表的末端。否则，如果管道以优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
CS_NODE pi_suspend_link
PI_PCB *pi_pipe
TC_TCB *pi_suspended_task
BYTE_PTR *pi_message_area
UNSIGNED pi_message_size
UNSIGNED pi_actual_size
STATUS pi_return_status
```

字符段概述：

字符段	描述
pi_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*pi_pieue	一个管道结构指针。
*pi_suspended_task	一个指向已挂起任务的任务控制块的指针。
*pi_message_area	该指针表示已挂起的任务消息缓冲的位置。
pi_message_size	存储请求消息的大小。
pi_actual_size	存储消息的实际大小。
pi_return_status	表示任务在管道上挂起的完成状态。

4.7.3 管道函数

下面部分对管道组件（PI）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

PIC_Create_Pipe　P144

功能：创建一个管道并把它放在已创建管道队列中。

语法：

```
STATUS PIC_Create_Pipe(NU_PIPE *pipe_ptr,
                       CHAR *name,
                       VOID *start_address,
                       UNSIGNED pipe_size,
                       OPTION message_type,
                       UNSIGNED message_size,
                       OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

PIC_Delete_Pipe

功能：该函数删除一个管道并把它从已创建的管道列表中移除。所有在管道上挂起的任务被恢复。注该函数并没有释放与管道相关联的内存。这是由应用层负责的。

语法：`STATUS PIC_Delete_Pipe(NU_PIPE *pipe_ptr)`

调用函数：

```
CSC_Remove_From_List            TCT_Protect
[HIC_Make_History_Entry]        TCT_Set_Current_Protect
TCC_Resume_Task                 TCT_System_Protect
[TCT_Check_Stack]               TCT_System_Unprotect
```

```
TCT_Control_To_System            TCT_Unprotect
```

PIC_Send_To_Pipe P145

功能：该函数发送一条消息给指定的管道。消息长度取决于调用。如果在管道上有一个或多个任务为了一条消息挂起，当这条消息拷贝到第一任务消息区域时，该任务就恢复。如果管道不能控制消息，正在调用的任务对调用者有可选性。

语法：

```
STATUS PIC_Send_To_Pipe(NU_PIPE *pipe_ptr,
                        VOID *message,
                        UNSIGNED size,
                        UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List                TCC_Task_Priority
CSC_Priority_Place_On_List       [TCT_Check_Stack]
CSC_Remove_From_List             TCT_Control_To_System
[HIC_Make_History_Entry]         TCT_Current_Thread
TCC_Resume_Task                  TCT_System_Protect
TCC_Suspend_Task                 TCT_Unprotect
```

PIC_Receive_From_Pipe

功能：该函数从指定的管道中接收一条消息。消息大小在调用时指定。如果当前有一条消息在管道中，该消息被从管道中移出并放入调用区域。如果该请求没有得到满足，就有可能挂起。

语法：

```
STATUS PIC_Receive_From_Pipe(NU_PIPE *pipe_ptr,
                             VOID *message,
                             UNSIGNED size,
                            UNSIGNED *actual_size,
                            UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List                [TCT_Check_Stack]
CSC_Remove_From_List             TCT_Control_To_System
[HIC_Make_History_Entry]         TCT_Current_Thread
TCC_Resume_Task                  TCT_System_Protect
TCC_Suspend_Task                 TCT_Unprotect
TCC_Task_Priority
```

PIC_Cleanup P146

功能：该函数负责从一个管道中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：`VOID PIC_Cleanup(VOID *information)`

调用函数：`CSC_Remove_From_List`

PICE_Create_Pipe

功能：该函数通过参数对管道创建函数执行错误检测。

语法：

```
STATUS PICE_Create_Pipe(NU_PIPE *pipe_ptr,
```

```
                                CHAR *name,
                                VOID *start_address,
                                UNSIGNED pipe_size,
                                OPTION message_type,
                                UNSIGNED message_size,
                                OPTION suspend_type)
```

调用函数：

```
        PIC_Create_Pipe
```

PICE_Delete_Pipe　P147

功能：该函数通过参数对管道删除函数执行错误检测。

语法：

```
    STATUS PICE_Delete_Pipe(NU_PIPE *pipe_ptr)
```

调用函数：

```
        PIC_Delete_Pipe
```

PICE_Send_To_Pipe

功能：该函数通过参数对给管道发送消息的函数执行错误检测。

语法：

```
    STATUS PICE_Send_To_Pipe(NU_PIPE *pipe_ptr,
                             VOID *message,
                             UNSIGNED size,
                             UNSIGNED suspend)
```

调用函数：

```
        PIC_Send_To_Pipe
        TCCE_Suspend_Error
```

PICE_Receive_From_Pipe　P148

功能：该函数通过参数对从管道接收消息的函数执行错误检测。

语法：

```
    STATUS PICE_Receive_From_Pipe(NU_PIPE *pipe_ptr,
                                  VOID *message,
                                  UNSIGNED size,
                                  UNSIGNED *actual_size,
                                  UNSIGNED suspend)
```

调用函数：

```
        PIC_Receive_From_Pipe
        TCCE_Suspend_Error
```

PIF_Established_Pipes

功能：该函数返回已建立管道的当前数目。以前删除的管道不再被认为已建立。

语法：`UNSIGNED PIF_Established_Pipes(VOID)`

调用函数：`[TCT_Check_Stack]`

PIF_Pipe_lnformation　P149

功能：该函数返回指定管道的相关信息。然而，如果所支持的管道指针无效，该函数只返回一个错误状态。

语法：

```
STATUS PIF_Pipe_Information(NU_PIPE *pipe_ptr,
                            CHAR *name,
                            VOID **start_address,
                            UNSIGNED *pipe_size,
                            UNSIGNED "available,
                            UNSIGNED *messages,
                            OPTION *message_type,
                            UNSIGNED *message_size,
                            OPTION *suspend_type,
                            UNSIGNED *tasks_waiting,
                            NU_TASK **first_task)
```

调用函数：

[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect

PIF_Pipe_Pointers

功能：建立起始于指定位置的管道指针列表。放在管道列表中管道个数等于管道总数或是在调用中指定的指针最大个数。

语法：

```
UNSIGNED PIF_Pipe_Pointers(NU_PIPE **pointer_list,
                           UNSIGNED maximum_pointers)
```

调用函数：

[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

PII_Initialize　P150

功能：该函数初始化控制管道模块操作的数据结构。这里最初没有管道。

语法：`VOID PII_Initialize(VOID)`

调用函数：无

PIS_Reset_Pipe

功能：该函数把管道复位到初始状态。在管道中任一消息就会丢失。在管道上挂起的任务也被恢复到复位完成状态。

语法：`STATUS PIS_Reset_Pipe(NU_PIPE *pipe_ptr)`

调用函数：

[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect

PIS_Send_To_Front_Of_Pipe　P151

功能：该函数发送一条消息到指定消息管道的前端。消息的长度在调用时决定。如果在管道上有一个或多个任务为了一条消息挂起，当这条消息拷贝到第一任务消息区域时，该任务就恢复。如果管道中有足够的空间，该消息就被拷贝到所有其他消息的前面。如果管道中没有足够的空间，调用就有可能挂起。

语法：

```
STATUS PIS_Send_To_Front_Of_Pipe(NU_PIPE *pipe_ptr,
                                 VOID *message,
                                 UNSIGNED size,
                                 UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List                [TCT_Check_Stack]
CSC_Remove_From_List             TCT_Control_To_System
[HIC_Make_History_Entry]         TCT_Current_Thread
TCC_Resume_Task                  TCT_System_Protect
TCC_Suspend_Task                 TCT_Unprotect
```

PIS_Broadcast_To_Pipe

功能：该函数给目前正在等待指定管道消息的所有任务发送一条消息。如果没有任务等待，该服务就像标准的发送例程。

语法：

```
STATUS PIS_Broadcast_To_Pipe(NU_PIPE *pipe_ptr,
                             VOID *message,
                             UNSIGNED size,
                             UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List                TCC_Task_Priority
CSC_Priority_Place_On_List       [TCT_Check_Stack]
CSC_Remove_From_List             TCT_Control_To_System
[HIC_Make_History_Entry]         TCT_Current_Thread
TCC_Resume_Task                  TCT_System_Protect
TCC_Suspend_Task                 TCT_Unprotect
```

PISE_Reset_Pipe　P152

功能：该函数通过参数对管道复位的函数执行错误检测。

语法：`STATUS PISE_Reset_Pipe(NU_PIPE *pipe_ptr)`

调用函数：

```
PIS_Reset_Pipe
```

PISE_Send_To_Front_Of_Pipe

功能：该函数通过参数对发送消息到管道前端的函数执行错误检测。

语法：

```
STATUS PISE_Send_To_Front_Of_Pipe(NU_PIPE *pipe_ptr,
                                  VOID *message,
                                  UNSIGNED size,
                                  UNSIGNED suspend)
```

调用函数：

```
PIS_Send_To_Front_Of_Pipe
```

TCCE_Suspend_Error

PISE_Broadcast_To_Pipe　P153

功能：该函数通过参数对管道广播消息的函数执行错误检测。

语法：

```
STATUS PISE_Broadcast_To_Pipe(NU_PIPE *pipe_ptr,
                              VOID *message,
                              UNSIGNED size,
                              UNSIGNED suspend)
```

调用函数：

PIS_Broadcast_To_Pipe
TCCE_Suspend_Error

4.8 信号量(SM)

信号量组件负责对所有的NUCLEUS PLUS信号量设备进行处理。一个NUCLEUS PLUS 信号量是为在应用过程中不同任务执行同步提供一种机制。NUCLEUS PLUS提供信号量计数范围值是0～4,294,967,294。任务等待一个非零信号量时，就会挂起。信号量可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解信号量更加详细的内容。

4.8.1信号量文件

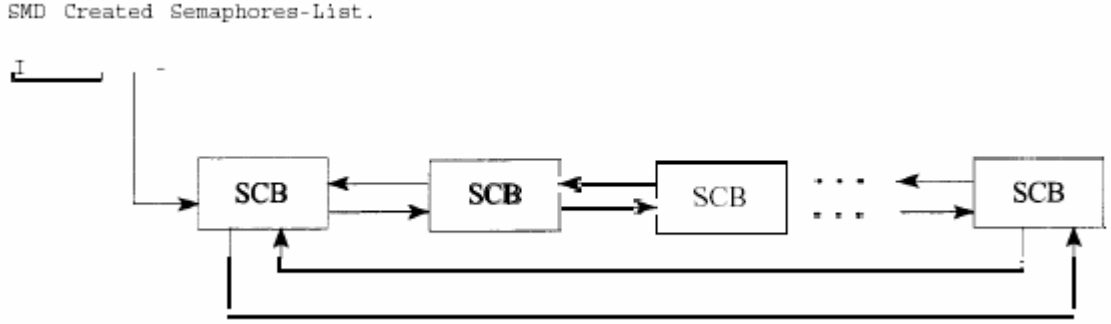
信号量组件（SM）由9个文件构成。信号量组件每个源文件定义如下：

文件	描述
SM_DEFS.H	该文件包含SM特殊的常量和数据结构的定义。
SM_EXTR.H	SM的所有外部接口函数都定义在该文件里。
SMD.C	SM的全局数据结构定义在这个文件里。
SMI.C	该文件包含了SM的初始化函数。
SMF.C	该文件包含了获取SM信息的函数。
SMC.C	该文件包含了SM所有核心函数。它用于处理基本的信号量发送和信号量接收服务。
SMS.C	该文件包含了SM的补充函数。该函数使用频率比核心函数少。
SMCE.C	该文件包含了定义在SMC.C中的核心函数与错误检测函数的接口。
SMSE.C	该文件包含了定义在SMS.C中的补充函数与错误检测函数的接口。

4.8.2信号量数据结构

创建信号量列表

NUCLEUS PLUS信号量可以动态创建和删除。每个已创建信号量的控制块（SCB）被保存在双链循环表中。最新创建的信号量被放在列表的末端，而删除的信号量被完全从列表中移除。信号量列表的头指针是SMD_Created_Semaphores_List。



创建信号量列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建信号量时列表的完整性。这是通过使用内部保护结构SMD_List_Protect来实现。所有的信号量创建和删除是在SMD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

信号量总数

当前已创建的NUCLEUS PLUS信号量的总数存放在变量SMD_Total_Smpes中。该变量的内容等于已创建列表中SCBS的数目。该变量的操作受到SMD_List_Protect的保护。

信号量控制模块

信号量控制块SM_SCB包含信号量计数。以及处理信号量请求所需的字符段。

字符段描述：

```
CS_NODE sm_created
UNSIGNED sm_id
CHAR sm_name[NU_MAX_NAME]
UNSIGNED sm_semaphore_count
DATA_ELEMENT sm_fifo_suspend
DATA_ELEMENT sm_padding[PAD_1]
UNSIGNED sm_tasks_waiting
struct SM_SUSPEND_STRUCT *sm_suspension_list
```

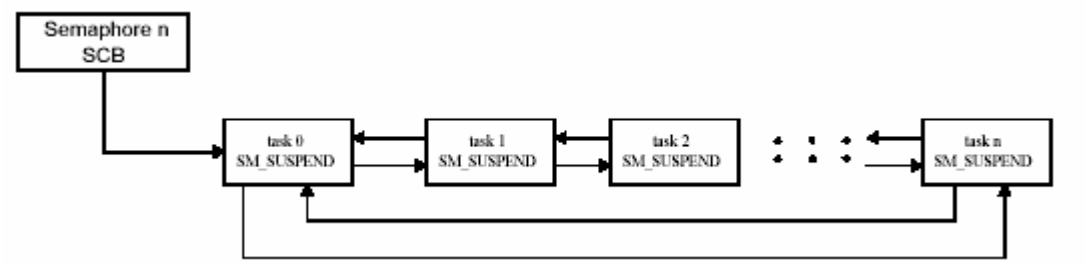
字符段概述：

字符段	描述
sm_created	它为信号量提供连接节点结构。它是一个双链循环表，用于连接已经创建的信号量。
sm_id	这是用来控制内部信号量0x53454D41的标识，相当于ASCII信号量。
sm_name	这是由用户为信号量指定的，8个字符名。
sm_semaphore_count	存储当前信号量计数值。
sm_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
sm_padding	这是用于在一个偶数临界边缘上，调整信号量结构。在某些端口，该字符端不用。
sm_tasks_waiting	表示当前在信号量上被挂起的任务数。
*sm_suspension_list	信号量挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

信号量挂起结构　P143

任务能可以在一个信号量当前计数为0的情况下挂起。在挂起过程中，一个SM_SUSPEND结构被建立。该结构包含了在任务和任务信号量请求挂起时的有关信息。在双链循环表中该挂起结构被连接到SCB，并分配正在挂起任务的堆栈。这里为每一个在信号量上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于信号量的创建。如果信号量选择先进先出（FIFO）挂起，则挂起块被增加到列表的末端。否则，如果信号量选择优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
CS_NODE sm_suspend_link
SM_SCB *sm_smpe
TC_TCB *sm_suspended_task
BYTE_PTR *sm_message_area
UNSIGNED sm_message_size
UNSIGNED sm_actual_size
STATUS sm_return_status
```

字符段概述：

字符段	描述
sm_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*sm_smeue	一个信号量结构指针。
*sm_suspended_task	一个指向已挂起任务的任务控制块的指针。
*sm_message_area	该指针表示已挂起的任务消息缓冲的位置。
sm_message_size	存储请求消息的大小。
sm_actual_size	存储消息的实际大小。
sm_return_status	表示任务在信号量上挂起的完成状态。

4.8.3 信号量函数

下面部分对信号量组件（SM）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

SMC_Create_Semaphore

功能：创建一个信号量并把它放在已创建信号量列表中。

语法：

```
STATUS SMC_Create_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                            CHAR *name,
                            UNSIGNED initial_count,
                            OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

SMC_Delete_Semaphore

功能：该函数删除一个信号量并把它从已创建的信号量列表中移除。所有在信号量上挂

起的任务被恢复。注该函数并没有释放与信号量控制块相关联的内存。这是由应用层负责的。

语法：STATUS SMC_Delete_Semaphore(NU_SEMAPHORE *semaphore_ptr)

调用函数：

CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System

SMC_Release_Semaphore

功能：该函数释放一个以前获得的信号量。如果有一个或多个任务正在等待一个信号量，当第一个任务得到信号量时，就释放信号量请求。否则信号量计数只是增加。

语法 STATUS SMC_Release_Semaphore(NU_SEMAPHORE *semaphore_ptr)

调用函数：

CSC_Remove_From_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect

SMC_Cleanup

功能：该函数负责从一个信号量中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：VOID SMC_Cleanup(VOID *information)

调用函数：CSC_Remove_From_List

SMCE_Create_Semaphore

功能：该函数通过参数对信号量创建函数执行错误检测。

语法：

```
STATUS SMCE_Create_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                             CHAR *name,
                             UNSIGNED initial_count,
                             OPTION suspend_type)
```

调用函数：SMC_Create_Semaphore

SMCE_Delete_Semaphore

功能：该函数通过参数对实际删除信号量函数执行错误检测。

语法：STATUS SMCE_Delete_Semaphore(NU_SEMAPHORE *semaphore_ptr)

调用函数：SMC_Delete_Semaphore

SMCE_Obtain_Semaphore

功能：该函数通过参数对获得信号量的函数执行错误检测。

语法：

```
STATUS SMCE_Obtain_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                             UNSIGNED suspend)
```

调用函数：

```
SMC_Obtain_Semaphore
TCCE_Suspend_Error
```

SMCE_Release_Semaphore

功能：该函数通过参数对释放信号量函数执行错误检测。

语法：

```
STATUS SMCE_Release_Semaphore(NU_SEMAPHORE *semaphore_ptr)
```

调用函数：

```
SMC_Release_Semaphore
TCCE_Suspend_Error
```

SMF_Established_Semaphores

功能：该函数返回已建立信号量的当前数目。以前删除的信号量不再被认为已建立。

语法：`UNSIGNED SMF_Established_Semaphorees(VOID)`

调用函数：`[TCT_Check_Stack]`

SMF_Semaphore_Pointers

功能：建立起始于指定位置的信号量指针列表。放在信号量列表中信号量个数等于信号量总数或是在调用中指定的指针最大个数。

语法：

```
UNSIGNED SMF_Semaphore_Pointers(NU_SEMAPHORE **pointer_list,
                                UNSIGNED maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

SMF_Semaphore_Information

功能：该函数返回指定信号量的相关信息。然而，如果所支持的信号量指针无效，该函数只返回一个错误状态。

语法：

```
STATUS SMF_Semaphore_Information(NU_SEMAPHORE *semaphore_ptr,
                                 CHAR *name,
                                 UNSIGNED *current_count,
                                 OPTION *suspend_type,
                                 UNSIGNED *tasks_waiting,
                                 NU_TASK **first_task)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

SMI_Initialize

功能：该函数初始化控制信号量模块操作的数据结构。这里最初没有信号量。

语法：`VOID SMI_Initialize(VOID)`

调用函数：无

SMS_Reset_Semaphore

功能：该函数把信号量复位到初始状态。在信号量上挂起的任务也被恢复到复位完成状态。

语法：
```
STATUS SMS_Reset_Semaphore(NU_SEMAPHORE *semaphore_ptr
                           UNSIGNED *current_count)
```

调用函数：

```
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect
```

SMSE_Reset_Semaphore

功能：该函数通过参数对实际复位信号量函数执行错误检测。

语法：
```
STATUS SMSE_Reset_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                            UNSIGNED *current_count)
```

调用函数：SMS_Reset_Semaphore

4.9 事件模块(EV)

事件组件负责对所有的NUCLEUS PLUS事件组设备进行处理。一个NUCLEUS PLUS 事件是为了表示一个确立的事件已发生而提供的一种机制。事件组中一个位代表一个事件。这个位叫作事件标志。每个事件组中有32个事件标志。任务可以在等待一个事件特殊设置的情况下挂起。事件组可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解事件更加详细的内容。

4.9.1事件文件

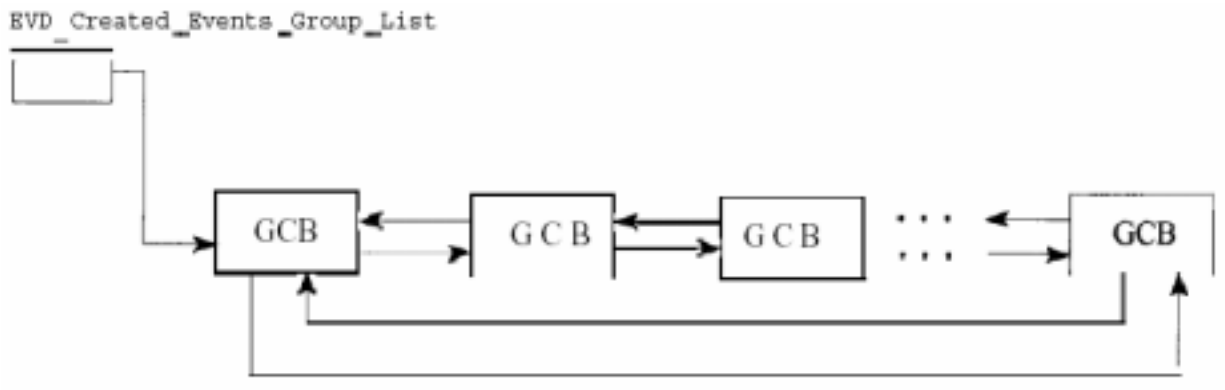
事件组件（EV）由7个文件构成。事件组件每个源文件定义如下：

文件	描述
EV_DEFS.H	该文件包含事件组（EV）特殊的常量和数据结构的定义。
EV_EXTR.H	事件组（EV）的所有外部接口函数都定义在该文件里。
EVD.C	事件组（EV）的全局数据结构定义在这个文件里。
EVI.C	该文件包含了事件组（EV）的初始化函数。
EVF.C	该文件包含了获取事件组（EV）信息的函数。
EVC.C	该文件包含了事件组（EV）所有核心函数。它用于处理基本的事件发送和事件接收服务。
EVCE.C	该文件包含了定义在EVC.C中的核心函数与错误检测函数的接口。

4.9.2事件组数据结构

创建事件组列表

NUCLEUS PLUS事件可以动态创建和删除。每个已创建事件的控制块（GCB）被保存在双链循环表中。最新创建的事件组被放在列表的末端，而删除的事件组被完全从列表中移除。事件列表的头指针是EVD_Greated_Events_Group_List。



创建事件列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建事件时列表的完整性。这是通过使用内部保护结构EVD_List_Protect来实现。所有的事件创建和删除是在EVD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述：

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

事件总数

当前已创建的NUCLEUS PLUS事件的总数存放在变量EVD_Total_Event_Groups中。该变量的内容等于已创建列表中GCBS的数目。该变量的操作受到EVD_List_Protect的保护。

事件组控制模块

事件控制块EV_GCB包含事件计数。以及处理事件请求所需的字符段。

字符段描述：

```
CS_NODE ev_created
UNSIGNED ev_id
CHAR ev_name[NU_MAX_NAME]
UNSIGNED ev_current_events
UNSIGNED ev_tasks_waiting
struct EV_SUSPEND_STRUCT *ev_suspension_list
```

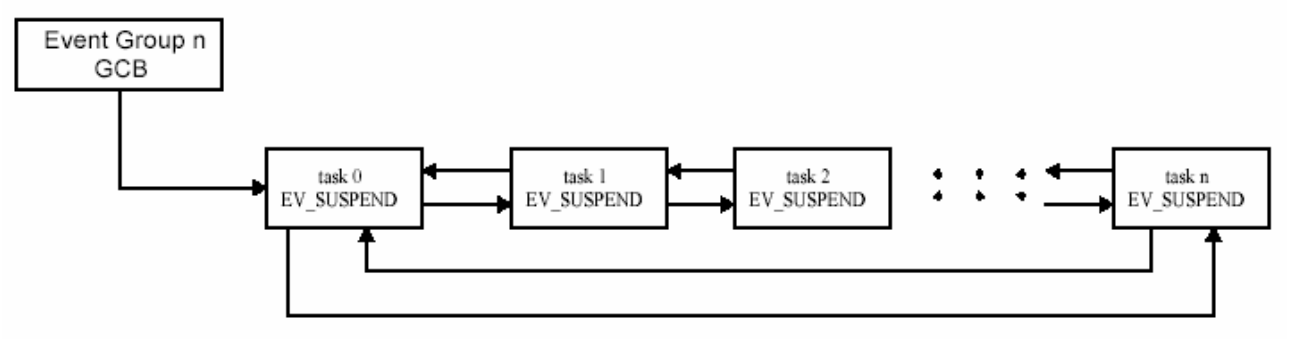
字符段概述：

字符段	描述
ev_created	它为事件提供连接节点结构。它是一个双链循环表，用于连接已经创建的事件。
ev_id	这是用来控制内部事件0x45564E54的标识，相当于ASCII事件。
ev_name	这是由用户为事件指定的，8个字符名。
ev_current_events	包含当前事件标志。
ev_tasks_waiting	表示当前在事件上被挂起的任务数。
*ev_suspension_list	事件挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

事件组挂起结构

任务可以在一个事件组不符合用户指定事件标志组合的情况下挂起。在挂起过程中，一

个EV_SUSPEND结构被建立。该结构包含了在任务和任务事件组请求挂起时的有关信息。在双链循环表中该挂起结构被连接到GCB，并分配正在挂起任务的堆栈。这里为每一个在事件上挂起的任务提供一个挂起块。



字符段描述：

```
CS_NODE ev_suspend_link
EV_GCB *ev_event_group
UNSIGNED ev_requested_events
DATA_ELEMENT ev_operation
DATA_ELEMENT ev_padding[PAD_l]
TC_TCB *ev_suspended_task
STATUS ev_return_status
UNSIGNED ev_actual_events
```

字符段概述：

字符段	描述
ev_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*ev_event_group	一个事件组结构指针。
ev_requested_events	表示该事件组已被请求。
ev_operation	该操作类型在事件组上被请求。特别是在某些AND/OR端口上。
ev_padding	用于调整在偶临界上挂起事件组结构。某些端口该域不用。
*ev_suspended_task	一个指向已挂起任务的任务控制块的指针。
ev_return_status	表示任务在事件组上挂起的完成状态。
ev_actual_events	通过请求返回实际的事件标志设置。

4.9.3 事件组函数

下面部分对事件组组件（EV）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

EVC_Create_Event_Group

功能：创建一个事件组并把它放在已创建事件组列表中。

语法：

```
STATUS EVC_Create_Event_Group(NU_EVENT_GROUP *event_group_ptr,
                              CHAR *name)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

EVC_Delete_Event_Group

功能：该函数删除一个事件组并把它从已创建的事件组列表中移除。所有在事件组上挂起的任务被恢复。注该函数并没有释放与事件组控制块相关联的内存。这是由应用层负责的。

语法：STATUS EVC_Delete_Event_group(NU_EVENT_GROUP *event_group_ptr)

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_Unprotect
```

EVC_Set_Events

功能：该函数设置指定事件标志组中的事件标志。事件标志可以与事件组中当前事件与/或。当满足事件请求时，在一个组上挂起的任务被恢复。

语法：

```
STATUS EVC_Set_Events (NU_EVENT_GROUP *event_group_ptr,
                       UNSIGNED events,
                       OPTION operation)
```

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect
```

EVC_Retrieve_Events

功能：从指定事件组中检索事件标志不同的组合。如果事件组没有包含必需标志，正在调用的任务就有可能挂起。

语法：

```
STATUS EVC_Retrieve_Events (NU_EVENT_GROUP *event_group_ptr,
                            UNSIGNED requested_events,
                            OPTION operation,
                            UNSIGNED *retrieved_events,
                            UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
TCC_Suspend_Task
[TCT_Check_Stack]
TCT_Current_Thread
TCT_System_Protect
TCT_Unprotect
```

EVC_Cleanup

功能：该函数负责从一个事件组中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：VOID EVC_Cleanup(VOID *information)

调用函数：CSC_Remove_From_List

EVCE_Create_Event_group

功能：该函数通过参数对事件组创建函数执行错误检测。

语法：

```
STATUS EVCE_Create_Event_group(NU_EVENT_GROUP *event_group_ptr,
                               CHAR *name)
```

调用函数：EVC_Create_Event_group

EVCE_Delete_Event_group

功能：该函数通过参数对实际删除事件组函数执行错误检测。

语法：STATUS EVCE_Delete_Event_group(NU_EVENT_GROUP *event_group_ptr)

调用函数：EVC_Delete_Event_group

EVCE_Set_Events

功能：该函数通过参数对设置事件组函数执行错误检测。

语法：

```
STATUS EVCE_Set_Events (NU_EVENT_GROUP *event_group_ptr,
                        UNSIGNED events,
                        OPTION operation)
```

调用函数：EVC_Set_Events

EVCE_Retrieve_Events

功能：该函数通过参数对检索事件函数执行错误检测。

语法：

```
STATUS EVCE_Retrieve_Events (NU_EVENT_GROUP *event_group_ptr,
                             UNSIGNED requested_events,
                             OPTION operation,
                             UNSIGNED *retrieved_events,
                             UNSIGNED suspend)
```

调用函数：

EVC_Retrieve_Events
TCCE_Suspend_Error

EVF_Established_Event_groups

功能：该函数返回已建立事件组的当前数目。以前删除的事件组不再被认为已建立。

语法：UNSIGNED EVF_Established_Event_groupes(VOID)

调用函数：[TCT_Check_Stack]

EVF_Event_group_Pointers

功能：建立起始于指定位置的事件组指针列表。放在事件组列表中事件组个数等于事件组总数或是在调用中指定的指针最大个数。

语法：

```
UNSIGNED EVF_Event_group_Pointers(NU_EVENT_GROUP **pointer_list,
                                  UNSIGNED maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

EVF_Event_group_Information

功能：该函数返回指定事件组的相关信息。然而，如果所支持的事件组指针无效，该函数只返回一个错误状态。

语法：

```
STATUS EVF_Event_group_Information(NU_EVENT_GROUP *event_group_ptr,
                                   CHAR *name,
                                   UNSIGNED *event_flags,
                                   UNSIGNED *tasks_waiting,
                                   NU_TASK **first_task)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

EVI_Initialize

功能：该函数初始化控制事件组模块操作的数据结构。这里最初没有事件组。

语法：`VOID EVI_Initialize(VOID)`

调用函数：无

4.10 分区内存组件(PM)

内存分区组件（PM）负责处理所有的NUCLEUS PLUS内存分区设备。一个NUCLEUS PLUS 分区内存池包含指定固定大小的内存池。任务可以在一个空池中等待一个内存分区的情况下挂起。分区池可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解分区内存池更加详细的内容。

4.10.1分区内存文件

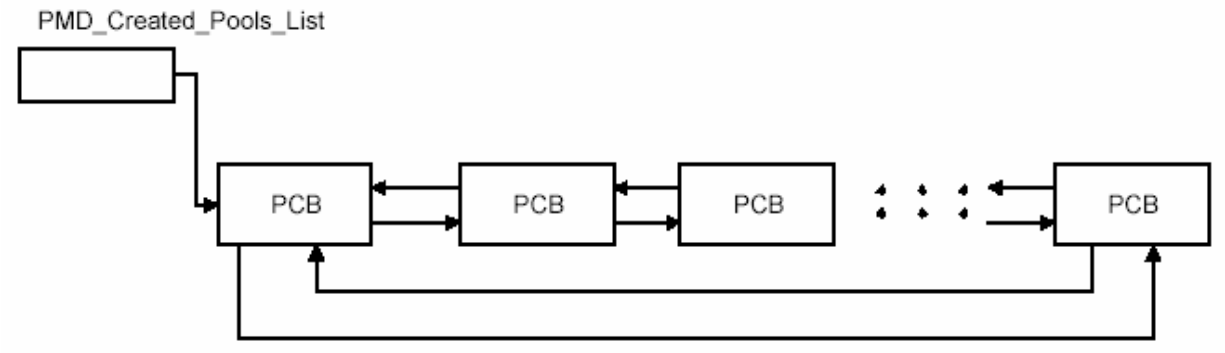
分区内存组件（PM）由7个文件构成。内存分区组件每个源文件定义如下：

文件	描述
PM_DEFS.H	该文件包含PM特殊的常量和数据结构的定义。
PM_EXTR.H	PM的所有外部接口函数都定义在该文件里。
PMD.C	PM的全局数据结构定义在这个文件里。
PMI.C	该文件包含了PM的初始化函数。
PMF.C	该文件包含了获取PM信息的函数。
PMC.C	该文件包含了PM所有核心函数。它用于处理基本的内存分区和内存释放服务。
PMCE.C	该文件包含了定义在PMC.C中的核心函数与错误检测函数的接口。

4.10.2内存分区数据结构

创建内存分区列表

NUCLEUS PLUS内存分区池可以动态创建和删除。每个已创建分区内存池的分区内存控制块（PCB）被保存在双链循环表中。最新创建的分区内存池被放在列表的末端，而删除的分区内存池被完全从列表中移除。分区内存列表的头指针是PMD_Created_Pools_List。



创建分区内存列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建分区内存池时列表的完整性。这是通过使用内部保护结构PMD_List_Protect来实现。所有的分区内存创建和删除是在PMD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述：

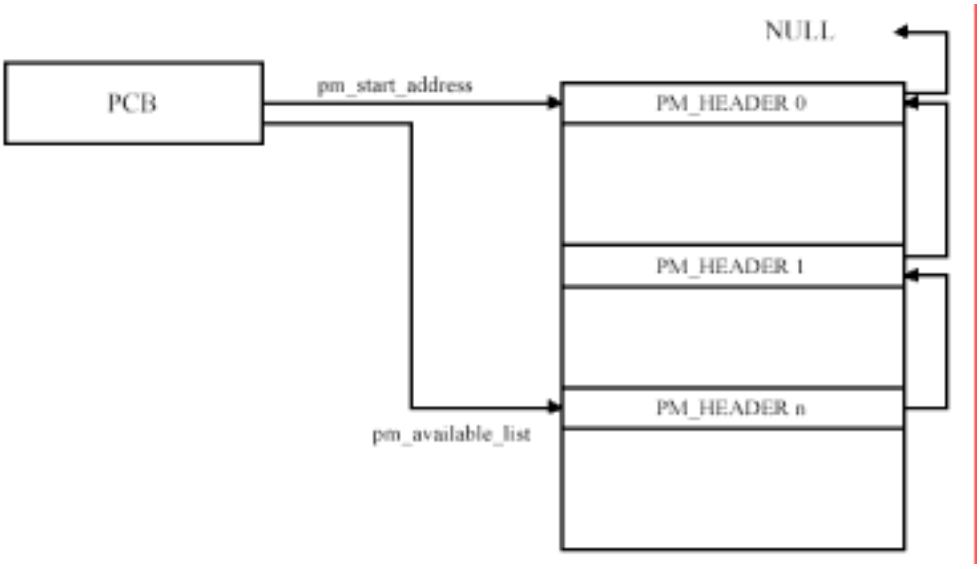
tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

分区池总数

当前已创建的NUCLEUS PLUS分区内存池的总数存放在变量PMD_Total_Pools中。该变量的内容等于已创建列表中PCBS的数目。该变量的操作受到PMD_List_Protect的保护。

有效的分区列表

如果有效分区列表是一个单连接的NULL，就结束列表。PCB含有列表起始地址的指针，就如列表中下一个有效分区一样。已分配的分区被从列表前端移出，释放的分区被放置在列表的分区。每个分区都有一个用于连接分区的头控制块。对于端口该字符段没用。



分区内存控制模块

分区内存池控制块PM_PCB包含当前内存池起始地址和处理分区池请求所必须的其他区域。

字符段描述：

```
CS_NODE pm_created
UNSIGNED pm_id
CHAR pm_name[NU_MAX_NAME]
VOID *pm_start_address
UNSIGNED pm_pool_size
UNSIGNED pm_partition_size
UNSIGNED pm_available
UNSIGNED pm_allocated
struct PM_HEADER_STRUCT *pm_available_list
DATA_ELEMENT pm_fifo_suspend
DATA_ELEMENT pm_padding[PAD_1]
UNSIGNED pm_tasks_waiting
struct PM_SUSPEND_STRUCT *pm_suspension_list
```

字符段概述：

字符段	描述
pm_created	它为分区内存池提供连接节点结构。它是一个双链循环表，用于连接已经创建的分区内存池。
pm_id	这是用来控制内部内存分区0x50415254的标识，相当于ASCII部分。
pm_name	这是由用户为分区内存池指定的，8个字符名。
*pm_start_address	当前分区内存池起始地址。
pm_pool_size	控制分区内存池大小。
pm_partition_size	当前分区内存池的大小。
pm_available	用在当前内存池中有效分区数。
pm_allocated	控制已分配的分区数。
*pm_available_list	当前内存池有效分区的列表。
pm_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
pm_padding	这是用于在一个偶数临界边缘上，调整分区内存池结构。在某些端口，该字符端不用。
pm_tasks_waiting	表示当前在分区内存池上被挂起的任务数。
*pm_suspension_list	分区内存池挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

分区内存池头结构

分区头结构PM_HEADER被放在每个有效分区开始端。除了最后一个分区指向结束符NULL外，其他每个分区头包含指向下一个有效分区的指针。每个分区头也包含指向它的分区内存池控制块（PCB）的指针。

字符段描述

```
struct PM_HEADER_STRUCT *pm_next_available
PM_PCB *pm_partition_pool
```

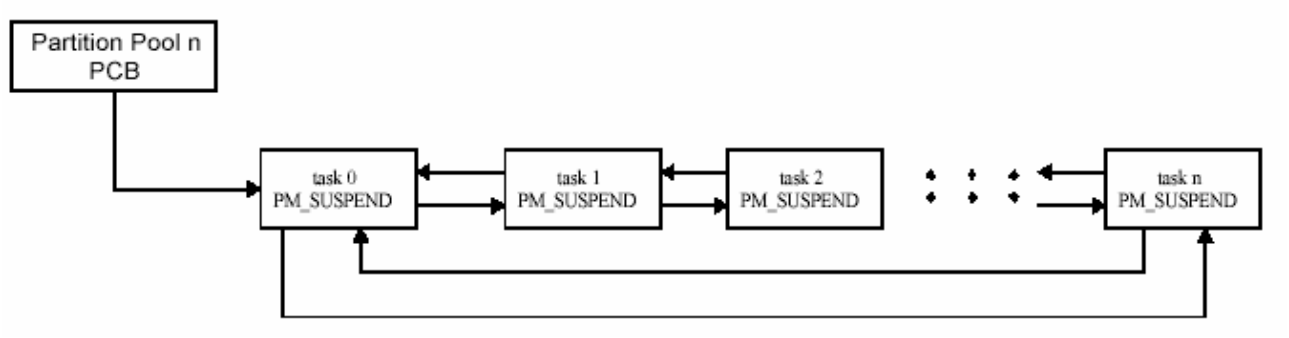
字符概述：

*pm_next_available	在有效列表中指向下一个分区的指针。
*pm_partition_pool	指向分区PCB的指针。

分区内存池挂起结构

任务能可以在分区内存池为空和满的情况下挂起。在挂起过程中，一个PM_SUSPEND结构被建立。该结构包含了在任务和任务分区内存请求挂起时的有关信息。在双链循环表中该挂起结构被连接到PCB，并分配正在挂起任务的堆栈。这里为每一个在分区内存池上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于分区池的创建。如果分区内存池选择先进先出（FIFO）挂起，则挂起块被增加到列表的末端。否则，如果分区内存池选择优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
CS_NODE pm_suspend_link
PM_PCB *pm_partition_pool
TC_TCB *pm_suspended_task
VOID *pm_return_pointer
STATUS pm_return_status
```

字符段概述：

字符段	描述
pm_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*pm_partition_pool	一个分区内存池结构指针。
*pm_suspended_task	一个指向已挂起任务的任务控制块的指针。
*pm_return_pointer	返回已经请求的内存地址。
pm_return_status	表示任务在分区池上挂起的完成状态。

4.10.3 分区内存函数

下面部分对分区内存组件（PM）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

PMC_Create_Partition_Pool

功能：创建一个内存分区池并把它放在已创建分区池列表中。

语法：

```
STATUS PMC_Create_Partition_Pool(NU_PARTITION_POOL *pool_ptr,
                                 CHAR *name,
                                 VOID *start_address,
                                 UNSIGNED pool_size,
                                 UNSIGNED partition_size,
                                 OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
```

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

PMC_Delete_Partition_Pool

功能：该函数删除一个内存分区池并把它从已创建的分区池列表中移除。所有在分区池上挂起的任务被恢复。注该函数并没有释放与池区域或池控制块相关联的内存。这是由应用层负责的。

语法：`STATUS PMC_Delete_Partition_Pool (NU_PARTITION_POOL *pool_ptr)`

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_Unprotect
```

PMC_Allocate_Partition

功能：该函数从指定的内存分区池分配一个内存区。如果内存分区当前有效，该函数立即完成。否则，如果分区无效，就有可能挂起。

语法：

```
STATUS PMC_Allocate_Partition(NU_PARTITION_POOL *pool_ptr,
                              VOID **return_pointer,
                              UNSIGNED suspend)
```

调用函数：

```
CSC_Place_On_List
CSC_Priority_Place_On_List
[HIC_Make_History_Entry]
TCC_Suspend_Task
TCC_Task_Priority
[TCT_Check_Stack]
TCT_Current_Thread
TCT_System_Protect
TCT_Unprotect
```

PMC_Deallocate_Partition

功能：该函数释放一个先前已分配的分区。如果有一个任务正在等待一个分区，则该分区只被给予正在等待的任务，并且正在等待的任务被恢复。否则，分区返回分区池。

语法：`STATUS PMC_Deallocate_Partition(VOID *partition)`

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
```

```
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_System_Protect
TCT_Unprotect
```

PMC_Cleanup

功能：该函数负责从一个分区池中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：`VOID PMC_Cleanup(VOID *information)`

调用函数：`CSC_Remove_From_List`

PMCE_Create_Partition_Pool

功能：该函数通过参数对分区池创建函数执行错误检测。

语法：

```
STATUS PMCE_Create_Partition_Pool(NU_PARTITION_POOL *pool_ptr,
                                  CHAR *name,
                                  VOID *start_address,
                                  UNSIGNED pool_size,
                                  UNSIGNED partition_size,
                                  OPTION suspend_type)
```

调用函数：`PMC_Create_Partition_Pool`

PMCE_Delete_Partition_Pool

功能：该函数通过参数对删除分区池函数执行错误检测。

语法：`STATUS PMCE_Delete_Partition_Pool(NU_PARTITION_POOL *pool_ptr)`

调用函数：`PMC_Delete_Partition_Pool`

PMCE_Allocate_Partition

功能：该函数通过参数对分配分区函数执行错误检测。

语法：

```
STATUS PMCE_Allocate_Partition(NU_PARTITION_POOL *pool_ptr,
                               VOID **return_pointer,
                               UNSIGNED suspend)
```

调用函数：

```
PMC_Allocate_Partition
TCCE_Suspend_Error
```

PMCE_Deallocate_Partition

功能：该函数通过参数对释放分区函数执行错误检测。

语法：`STATUS PMCE_Deallocate_Partition(VOID *partition)`

调用函数：

```
PMC_Deallocate_Partition
TCCE_Suspend_Error
```

PMF_Established_Partition_Pools

功能：该函数返回已建立分区池的当前数目。以前删除的分区池不再被认为已建立。

语法：`UNSIGNED PMF_Established_Partition_Pools(VOID)`

调用函数：[TCT_Check_Stack]

PMF_Partition_Pool_Pointers

功能：建立起始于指定位置的池指针列表。放在分区池列表中池个数等于池总数或是在调用中指定的指针最大个数。

语法：

```
UNSIGNED PMF_ Partition_Pool_Pointers(NU_PARTITION_POOL **pointer_list,
                                      UNSIGNED maximum_pointers)
```

调用函数：

[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

PMF_Partition_Pool_Information

功能：该函数返回指定分区池的相关信息。然而，如果所支持的分区池指针无效，该函数只返回一个错误状态。

语法：

```
STATUS PMF_Partition_Pool_Information(NU_PARTITION_POOL *pool_ptr,
                                      CHAR *name,
                                      VOID **start_address,
                                      UNSIGNED *pool_size,
                                      UNSIGNED *partition_size,
                                      UNSIGNED *available,
                                      UNSIGNED *allocated,
                                      OPTION *suspend_type,
                                      UNSIGNED *tasks_waiting,
                                      NU_TASK **first_task)
```

调用函数：

[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect

PMI_Initialize

功能：该函数初始化控制分区内存池组件操作的数据结构。这里最初没分区池。

语法：VOID PMI_Initialize(VOID)

调用函数：无

4.11 动态内存组件(DM)

动态内存组件（DM）负责处理所有的NUCLEUS PLUS动态内存设备。一个NUCLEUS PLUS 动态内存池包含了用户指定的字节数。池的内存位置取决于应用时指定。任务可以在一个等待足够大的动态内存变为有效的情况下挂起。动态池可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解动态内存池更加详细的内容。

4.11.1动态内存文件

动态内存组件（DM）由7个文件构成。动态内存组件每个源文件定义如下：

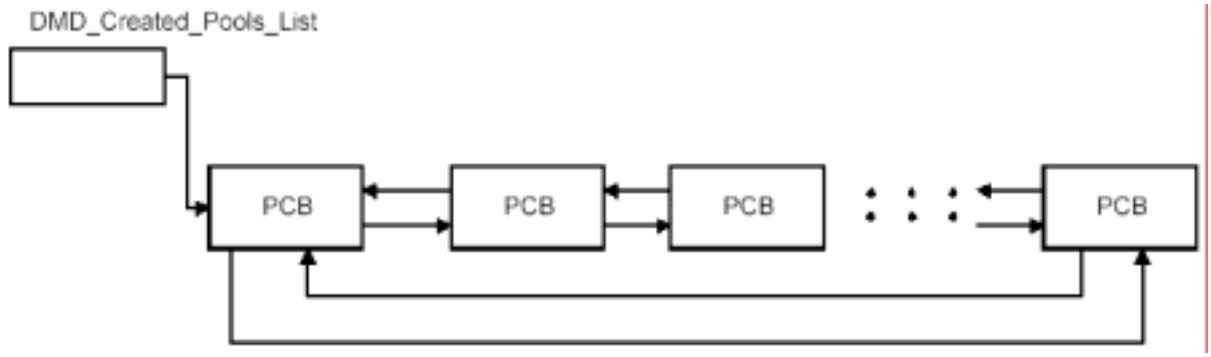
文件	描述
DM_DEFS.H	该文件包含DM特殊的常量和数据结构的定义。

DM_EXTR.H	DM的所有外部接口函数都定义在该文件里。
DMD.C	DM的全局数据结构定义在这个文件里。
DMI.C	该文件包含了DM的初始化函数。
DMF.C	该文件包含了获取DM信息的函数。
DMC.C	该文件包含了DM所有核心函数。它用于处理基本的内存分区和内存释放服务。
DMCE.C	该文件包含了定义在DMC.C中的核心函数与错误检测函数的接口。

4.11.2动态内存数据结构

创建动态内存列表

NUCLEUS PLUS动态内存池可以动态创建和删除。每个已创建动态内存池的动态内存控制块（PCB）被保存在双链循环表中。最新创建的动态内存池被放在列表的末端，而删除的动态内存池被完全从列表中移除。动态内存列表的头指针是DMD_Created_Pools_List。



创建动态内存列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建动态内存时列表的完整性。这是通过使用内部保护结构DMD_List_Protect来实现。所有的动态内存创建和删除是在DMD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述

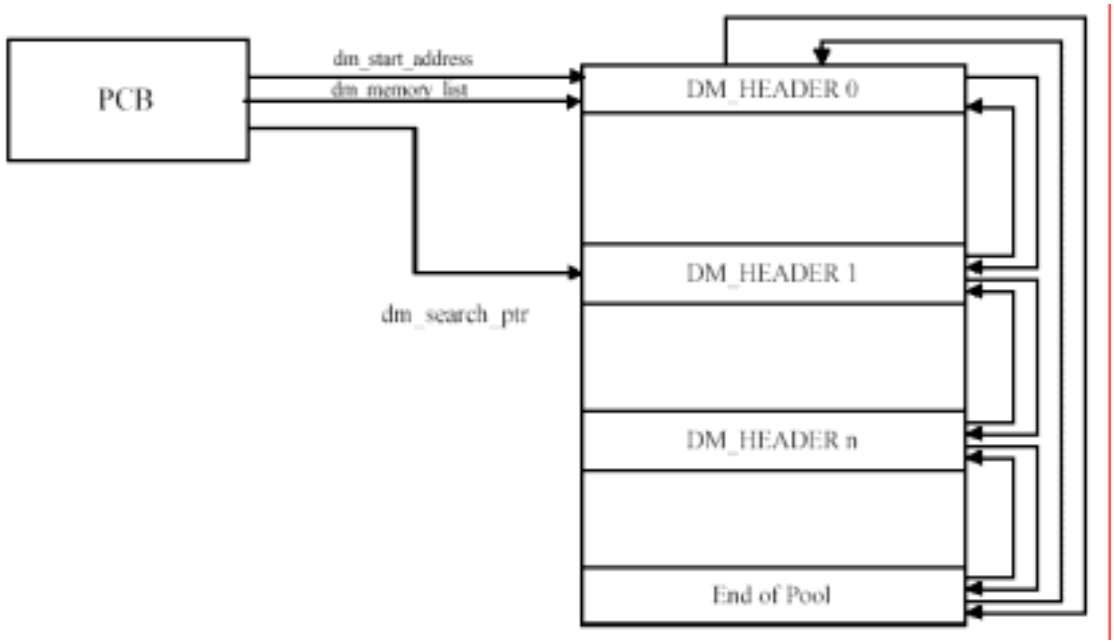
tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

动态池总数

当前已创建的NUCLEUS PLUS动态内存池的总数存放在变量DMD_Total_Pools中。该变量的内容等于已创建列表中PCBS的数目。该变量的操作受到DMD_List_Protect的保护。

有效的内存列表

如果有效内存列表是一个双循环连接表，以NULL结束列表。它包含有效的动态内存控制块（PCB）。PCB含有列表起始地址的指针，就如列表中下一个有效块一样。PCB中还包含一个搜索指针，它采用线性搜索和累计有效内存块，目的是满足内存请求。已分配的块被从列表前端移出，释放的块被放回列表中它原来所在的位置。每个块都包含一个用于连接不同块的头。对于端口该字符段没用。



动态池控制模块

分区内存池控制块DM_PCB包含当前内存池起始地址和处理动态内存池请求所必须的其他区域。

字符段描述：

```
CS_NODE dm_created
TC_PROTECT dm_protect
UNSIGNED dm_id
CHAR dm_name[NU_MAX_NAME]
VOID *dm_start_address
UNSIGNED dm_pool_size
UNSIGNED dm_min_allocation
UNSIGNED dm_available
struct DM_HEADER_STRUCT *dm_memory_list struct
DM_HEADER_STRUCT *dm_search_ptr
DATA_ELEMENT dm_fifo_suspend
DATA_ELEMENT dm_padding[PAD_1]
UNSIGNED dm_tasks_waiting
struct DM_SUSPEND_STRUCT *dm_suspension_list
```

字符段概述：

字符段	描述
dm_created	它为分区内存池提供连接节点结构。它是一个双链循环表，用于连接已经创建的动态池。
dm_protect	动态内存池保护结构指针。
dm_id	这是用来控制内部动态内存池0x44594E41的标识，相当于ASCII DYNA.。
dm_name	这是由用户为动态内存池指定的，8个字符名。
*dm_start_address	当前动态内存池起始地址。
dm_pool_size	控制动态内存池大小。
dm_min_allocation	在一个控制块中分配的最小字节数。
dm_available	用在当前动态内存池中总的有效字节数。

*dm_memory_list	在当前内存池的一个内存块列表。
*dm_search_ptr	用于搜索一个动态内存池头的指针
*dm_available_list	当前内存池有效分区的列表。
dm_fifo_suspend	该标志决定任务是按先进先出还是优先级顺序挂起。
dm_padding	这是用于在一个偶数临界边缘上，调整动态内存池结构。在某些端口，该字符端不用。
dm_tasks_waiting	表示当前在分区内存池上被挂起的任务数。
*dm_suspension_list	动态内存池挂起列表头指针。如果没有任务挂起，该指针为空（NULL）。

动态内存池头结构

分区头结构DM_HEADER被放在每个有效分区开始端。除了最后一个分区指向结束符NULL外，其他每个分区头包含指向下一个有效分区的指针。每个分区头也包含指向它的分区内存池控制块（PCB）的指针。

字符段描述

```
struct DM_HEADER_STRUCT *dm_next_memory
struct DM_HEADER_STRUCT *dm_previous_memory
DATA_ELEMENT dm_memory_free
DM_PCB *dm_memory_pool
```

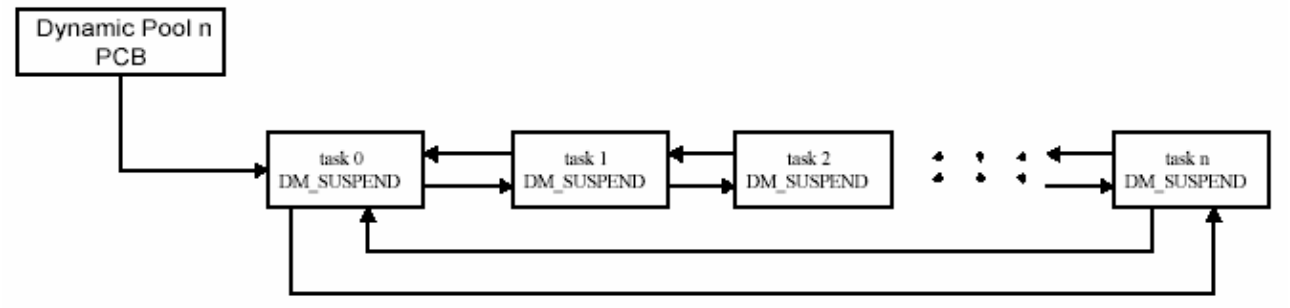
字符概述：

*dm_next_ memory	在有效列表中指向下一个内存块的指针。
*dm_previous_memory	在有效列表中指向前一个内存块的指针。
dm_memory_free	该标志表示当前内存块是否空闲。
*dm_partition_pool	指向PCB的指针。

动态内存池挂起结构

任务能可以在动态内存池为空和满的情况下挂起。在挂起过程中，一个DM_SUSPEND结构被建立。该结构包含了在任务和任务动态内存请求挂起时的有关信息。在双链循环表中该挂起结构被连接到PCB，并分配正在挂起任务的堆栈。这里为每一个在动态内存池上挂起的任务提供一个挂起块。

挂起块在挂起列表放置的顺序取决于分区池的创建。如果动态内存池选择先进先出（FIFO）挂起，则挂起块被增加到列表的末端。否则，如果动态内存池选择优先级挂起，挂起块被放在具有相同或优先级高的挂起块之后。



字符段描述：

```
CS_NODE dm_suspend_link
DM_PCB *dm_memory_pool
UNSIGNED dm_request_size
TC_TCB *dm_suspended_task
VOID *dm_return_pointer
STATUS dm_return_status
```

字符段概述：

字符段	描述
dm_suspend_link	表示一个与其他挂起块相连接的连接节点结构。
*dm_memory_pool	一个动态内存池结构指针。
dm_request_size	包含已请求内存块大小。
*dm_suspended_task	一个指向已挂起任务的任务控制块的指针。
*dm_return_pointer	返回已经请求的内存地址。
dm_return_status	表示任务在动态池上挂起的完成状态。

4.11.3 动态内存函数

下面部分对动态内存组件（DM）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

DMC_Create_Memory_Pool

功能：创建一个动态内存池并把它放在已创建动态内存池列表中。

语法：

```
STATUS DMC_Create_Memory_Pool(NU_MEMORY_POOL *pool_ptr,
                              CHAR *name,
                              VOID *start_address,
                              UNSIGNED pool_size,
                              UNSIGNED min_allocation,
                              OPTION suspend_type)
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

DMC_Delete_Memory_Pool

功能：该函数删除一个动态内存池并把它从已创建的动态池列表中移除。所有在内存池上挂起的任务被恢复。注该函数并没有释放与池区域或池控制块相关联的内存。这是由应用层负责的。

语法：`STATUS DMC_Delete_Memory_Pool (NU_MEMORY_POOL *pool_ptr)`

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Protect
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_Unprotect
```

DMC_Allocate_Memory

功能：该函数从指定的动态内存池分配内存。如果动态内存当前有效，该函数立即完成。

否则，如果动无效，任务就有可能挂起。

语法：

```
STATUS DMC_Allocate_Memory(NU_MEMORY_POOL *pool_ptr,
                           VOID **return_pointer,
                           UNSIGNED size,
                           UNSIGNED suspend)
```

调用函数：

CSC_Place_On_List
[HIC_Make_History_Entry]
TCC_Suspend_Task
TCC_Task_Priority
[TCT_Check_Stack]
TCT_Current_Thread
TCT_Protect
TCT_Set_Suspend_Protect
TCT_System_Protect
TCT_Unprotect
TCT_Unprotect_Specific

DMC_Deallocate_Memory

功能：该函数释放一个先前已分配的动态内存块。已释放的动态内存块会和邻近的合并。这就保证在池中没有空闲内存连续块。如果有一个任务正在等待动态内存，在释放完成之后就决定现在是否满足该请求。

语法：`STATUS DMC_Deallocate_Memory(VOID *memory)`

调用函数：

CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_protect
TCT_Unprotect

DMC_Cleanup

功能：该函数负责从一个内存池中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：`VOID DMC_Cleanup(VOID *information)`

调用函数：`CSC_Remove_From_List`

DMCE_Create_Memory_Pool

功能：该函数通过参数对动态内存池创建函数执行错误检测。

语法：

```
STATUS DMCE_Create_Memory_Pool(NU_MEMORY_POOL *pool_ptr,
```

```
                                    CHAR *name,
                                    VOID *start_address,
                                    UNSIGNED pool_size,
                                    UNSIGNED min_allocation,
                                    OPTION suspend_type)
```

调用函数：DMC_Create_Memory_Pool

DMCE_Delete_Memory_Pool

功能：该函数通过参数对删除动态内存池函数执行错误检测。
语法：STATUS DMCE_Delete_Memory_Pool(NU_MEMORY_POOL *pool_ptr)
调用函数：DMC_Delete_Memory_Pool

DMCE_Allocate_Memory

功能：该函数通过参数对分配动态函数执行错误检测。
语法：

```
STATUS DMCE_Allocate_Memory(NU_MEMORY_POOL *pool_ptr,
                            VOID **return_pointer,
                            UNSIGNED size
                            UNSIGNED suspend)
```

调用函数：

```
DMC_Allocate_Memory
TCCE_Suspend_Error
```

DMCE_Deallocate_Memory

功能：该函数通过参数对释放内存函数执行错误检测。
语法：STATUS DMCE_Deallocate_Memory(VOID *memory)
调用函数：

```
DMC_Deallocate_Memory
TCCE_Suspend_Error
```

DMF_Established_Memory_Pools

功能：该函数返回已建立内存池的当前数目。以前删除的池不再被认为已建立。
语法：UNSIGNED DMF_Established_Memory_Pools(VOID)
调用函数：[TCT_Check_Stack]

DMF_Memory_Pool_Pointers

功能：建立起始于指定位置的内存池指针列表。放在内存池列表中池个数等于池总数或是在调用中指定的指针最大个数。
语法：

```
UNSIGNED DMF_Memory_Pool_Pointers(NU_MEMORY_POOL **pointer_list,
                                  UNSIGNED maximum_pointers)
```

调用函数：

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

DMF_Memory_Pool_Information

功能：该函数返回指定内存池的相关信息。然而，如果所支持的内存池指针无效，该函数只返回一个错误状态。

语法：

```
STATUS DMF_Memory_Pool_Information(NU_MEMORY_POOL *pool_ptr,
                                   CHAR *name,
                                   VOID **start_address,
                                   UNSIGNED *pool_size,
                                   UNSIGNED *min_allocation,
                                   UNSIGNED *available,
                                   UNSIGNED *allocated,
                                   OPTION *suspend_type,
                                   UNSIGNED *tasks_waiting,
                                   NU_TASK **first_task)
```

调用函数：

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

DMI_Initialize

功能：该函数初始化控制动态内存池组件操作的数据结构。这里最初没有动态内存池。

语法：VOID DMI_Initialize(VOID)

调用函数：无

4.12 输入输出驱动器(IO)

输入输出驱动组件（IO）负责处理所有的NUCLEUS PLUS输入输出驱动设备。一个NUCLEUS PLUS 输入输出驱动为初始化、分配、释放、输入、输出、状态和结束请求提供了标准的I/O驱动接口。该接口具有公共的控制结构，这使得应用层在相似但不相同的方式下能够处理各种各样的外设。任务可以在等待一个外设变为有效的情况下挂起。I/O可由用户动态创建和删除。请参阅Nucleus PLUS reference manual 第三章，了解输入输出驱动更加详细的内容。

4.12.1输入输出驱动器文件

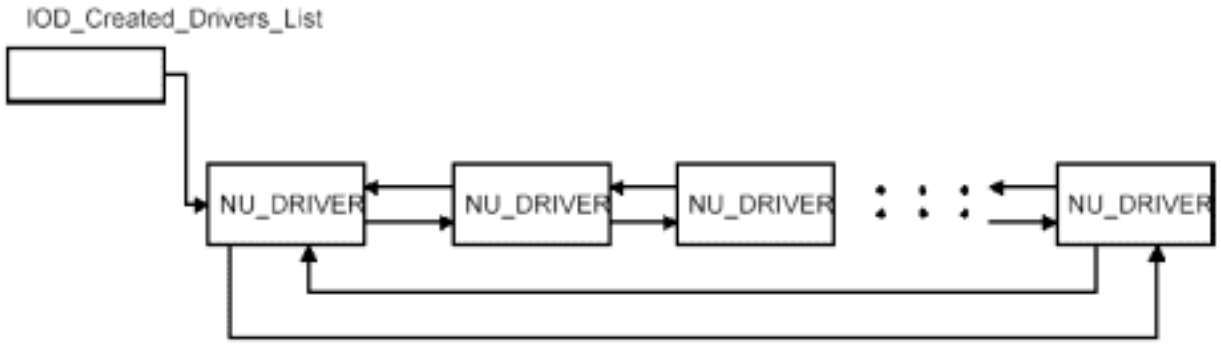
输入输出驱动器组件（IO）由7个文件构成。输入输出驱动器组件每个源文件定义如下：

文件	描述
IO_DEFS.H	该文件包含IO特殊的常量和数据结构的定义。
IO_EXTR.H	IO的所有外部接口函数都定义在该文件里。
IOD.C	IO的全局数据结构定义在这个文件里。
IOI.C	该文件包含了IO的初始化函数。
IOF.C	该文件包含了获取IO信息的函数。
IOC.C	该文件包含了IO所有核心函数。它用于处理基本的输入和输出服务。
IOCE.C	该文件包含了定义在IOC.C中的核心函数与错误检测函数的接口。

4.12.2输入输出数据结构

创建输入输出列表

NUCLEUS PLUS输入输出驱动可以动态创建和删除。每个已创建输入输出驱动的输入输出控制块(NU_DRIVER)被保存在双链循环表中。最新创建的输入输出驱动被放在列表的末端，而删除的输入输出驱动被完全从列表中移除。列表的头指针是IOD_Created_Drivers_List。



创建输入输出驱动列表保护

NUCLEUS PLUS保护在任务或HISRs竞争已创建输入输出驱动时列表的完整性。这是通过使用内部保护结构IOD_List_Protect来实现。所有的输入输出驱动创建和删除是在IOD_List_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

输入输出驱动总数

当前已创建的NUCLEUS PLUS输入输出驱动的总数存放在变量IOD_Total_Drivers中。该变量的内容等于已创建列表中NU_Drivers的数目。该变量的操作受到IOD_List_Protect的保护。

输入输出驱动请求结构

输入输出驱动请求结构NU_DRIVER_REQUEST负责传输必要的信息给一个已创建的I/O驱动或从它中取出。在请求结构中，请求信息类型通过字符段nu_function来指定。当然，该结构精确的解析取决于特殊的驱动器。

字符段描述：

```
INT nu_function
UNSIGNED nu_timeout
STATUS nu_status
UNSIGNED nu_supplemental
VOID nu_supplemental_ptr
union NU_REQUEST_INFO_UNION nu_request_info
```

字符段概述：

字符段	描述
nu_function	这是I/O请求函数代码。它有7个由请求时决定的值。
	NU_INITIALIZE 1
	NU_ASSIGN 2
	NU_RELEASE 3
	NU_INPUT 4
	NU_OUTPUT 5
	NU_STATUS 6
	NU_TERMINATE 7

nu_timeout	控制请求时间超时。
nu_status	包含请求状态。
nu_supplemental	包含用户提供的补充信息。
*nu_supplemental_ptr	一个指向驱动器指定补充信息的指针。
nu_request_info	一个用于驱动器请求的结构集合。这些请求包括初始化、分配、释放、输入、输出、状态和结束。

输入输出驱动初始化请求

使用初始化结构NU_INITIALIZE_STRUCT创建I/O驱动初始化请求。该结构包含驱动器基本地址和中断向量的信息。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_INITIALIZE来指定。

字符段描述：

```
VOID *nu_io_address
UNSIGNED nu_logical_units
VOID *nu_memory
INT nu_vector
```

字符段概述：

字符段	描述
*nu_io_address	一个指向驱动器I/O基本地址。
nu_logical_units	包含驱动器中逻辑单元个数。
*nu_memory	一个通用内存指针。
nu_vector	包含驱动器中断向量个数。

输入输出驱动作业请求

使用结构NU_ASSIGN_STRUCT创建I/O驱动器作业请求。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_ASSIGN来指定。

字符段描述：

```
UNSIGNED nu_logical_units
INT nu_assign_info
```

字符段概述：

字符段	描述
nu_logical_units	包含驱动器中逻辑单元个数。
nu_assign_info	该变量用于附加的I/O驱动器作业信息。

输入输出驱动器释放请求

使用结构NU_RELEASE_STRUCT创建I/O驱动器释放请求。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_RELEASE来指定。

字符段描述：

```
UNSIGNED nu_logical_units
INT nu_release_info
```

字符段概述：

字符段	描述
nu_logical_units	包含驱动器中逻辑单元个数。
nu_release_info	该变量用于附加的I/O驱动器释放信息。

输入输出驱动器输入请求

使用结构NU_INPUT_STRUCT创建I/O驱动器输入请求。该结构包含了发送给驱动器处理数据的有关信息。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_INPUT来指定。

字符段描述：

```
UNSIGNED nu_logical_unit
UNSIGNED nu_offset
UNSIGNED nu_request_size
UNSIGNED nu_actual_size
Void *nu_buffer_ptr
```

字符段概述：

字符段	描述
nu_logical_unit	包含驱动器中逻辑单元个数。一个指向驱动器I/O基本地址。
nu_offset	用于表示设备基本I/O地址的I/O偏移或输入缓冲偏移。
nu_request_size	I/O驱动器输入数据已请求的大小。
nu_actual_size	I/O驱动器输入数据已请求的实际大小。
*nu_buffer_ptr	I/O驱动器输入数据缓冲指针。

输入输出驱动器输出请求

使用结构NU_OUTPUT_STRUCT创建I/O驱动器输出请求。该结构包含了驱动器接收数据的有关信息。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_OUTPUT来指定。

字符段描述：

```
UNSIGNED nu_logical_unit
UNSIGNED nu_offset
UNSIGNED nu_request_size
UNSIGNED nu_actual_size
Void *nu_buffer_ptr
```

字符段概述：

字符段	描述
nu_logical_unit	包含驱动器中逻辑单元个数。一个指向驱动器I/O基本地址。
nu_offset	用于表示设备基本I/O地址的I/O偏移或输入缓冲偏移。
nu_request_size	I/O驱动器输出数据已请求的大小。
nu_actual_size	I/O驱动器输出数据已请求的实际大小。
*nu_buffer_ptr	I/O驱动器输出数据缓冲指针。

输入输出驱动器状态请求

使用结构NU_STATUS_STRUCT创建I/O驱动器状态请求。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_STATUS来指定。

字符段描述：

```
UNSIGNED nu_logical_units
INT nu_extra_status
```

字符段概述：

字符段	描述
nu_logical_units	包含驱动器中逻辑单元个数。
nu_extra_status	附加状态信息指针。

输入输出驱动器结束请求

使用结构NU_STATUS_STRUCT创建I/O驱动器结束请求。该请求通过结构NU_DRIVER_REQUEST中的字符段nu_function的值NU_TERMINATE来指定。

字符段描述：

```
UNSIGNED nu_logical_units
```

字符段概述：

字符段	描述
nu_logical_units	包含驱动器中逻辑单元个数。

4.12.3 输入输出驱动器函数

下面部分对输入输出驱动组件（IO）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

IOC_Create_Driver

功能：创建一个 I/O 驱动器并把它放在已创建 I/O 驱动器列表中。注该函数实际上并没有调用该驱动器。

语法：

```
STATUS IOC_Create_Driver(NU_DRIVER *driver,
                         CHAR *name,
                         VOID (*driver_entry) (NU_DRIVER*,
                         NU_DRIVER_REQUEST *))
```

调用函数：

```
CSC_Place_On_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

IOC_Delete_Driver

功能：该函数删除一个 I/O 驱动器并把它从已创建的驱动器列表中移除。注该函数实际上并没有调用该驱动器。

语法：

```
STATUS IOC_Delete_Driver (NU_DRIVER *driver)
```

调用函数：

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

IOC_Request_Driver

功能：该函数发送一个用户请求给指定的I/O驱动器。

语法：

```
STATUS IOC_Request_Driver(NU_DRIVER *driver,
                          NU_DRIVER_REQUEST *request)
```

调用函数：

```
[HIC_Make_History_Entry]
[TCT_Check_Stack]
```

IOC_Resume_Driver

功能：恢复在在一个I/O驱动器中以前挂起一个任务。特别是，该函数从I/O驱动器中调用。

语法：STATUS IOC_Resume_Driver(NU_TASK *task)

调用函数：

```
[HIC_Make_History_Entry]
TCC_Resume_Task
TCT_Control_To_System
[TCT_Check_Stack]
TCT_Get_Current_Protect
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_Unprotect
TCT_Unprotect_Specific
```

IOC_Suspend_Driver

功能：该函数在一个I/O驱动器中挂起一个任务。它的负责保持跟踪在I/O驱动器中正在等待的任务。

语法：

```
STATUS IOC_Suspend_Driver(VOID (*terminate_routine)(VOID *),
                          VOID *information,
                          UNSIGNED timeout)
```

调用函数：

```
[HIC_Make_History_Entry]
TCC_Suspend_Task
TCT_Current_Thread
[TCT_Check_Stack]
TCT_Get_Current_Protect
TCT_Set_Suspend_Protect
TCT_System_Protect
TCT_Unprotect_Specific
```

IOC_Cleanup

功能：该函数负责从一个内存池中清除挂起的块。除非在进行中超时或任务终止，它是不会被调用。注保护（和中止时间一样）已经有效。

语法：VOID IOC_Cleanup(VOID *information)

调用函数：CSC_Remove_From_List

IOCE_Create_Driver

功能：该函数通过参数对I/O驱动器创建函数执行错误检测。

语法：

```
STATUS IOCE_Create_Driver(NU_DRIVER *driver,
                          CHAR *name,
                          VOID (*driver_entry) (NU_DRIVER*,
                          NU_DRIVER_REQUEST *))
```

调用函数：IOC_Create_Driver

IOCE_DeIete_Driver

功能：该函数通过参数对I/O驱动器删除函数执行错误检测。
语法：STATUS IOCE_Delete_Driver(NU_DRIVER *driver)
调用函数：IOC_Delete_Driver

IOCE_Request_Driver

功能：该函数通过参数对I/O驱动器请求函数执行错误检测。
语法：STATUS IOC_Request_Driver(NU_DRIVER *driver,
NU_DRIVER_REQUEST *request)
调用函数：

IOCE_Resume_Driver

功能：该函数通过参数对I/O驱动器恢复函数执行错误检测。
语法：STATUS IOCE_Resume_Driver(NU_TASK *task)
调用函数：
IOC_Resume_Driver
TCCE_Validate_Resume

IOCE_Suspend_Driver

功能：该函数通过参数对I/O驱动器挂起函数执行错误检测。
语法：
STATUS IOCE_Suspend_Driver(Driver(NU_DRIVER *driver,
NU_DRIVER_REQUEST *request)
调用函数：IOC_Suspend_Driver

IOF_Established_Drivers

功能：返回已建立的I/O驱动器当前个数。以前删除的I/O驱动器不再被认为已建立。
语法：UNSIGNED IOF_Established_Drivers(VOID)
调用函数：[TCT_Check_Stack]

IOF_Driver_Pointers

功能：建立起始于指定位置的驱动器指针列表。放在列表中驱动器个数等于驱动器总数或是在调用中指定的指针最大个数。
语法：
UNSIGNED IOF_Driver_Pointers(NU_DRIVER **pointer_list,
UNSIGNED maximum_pointers)
调用函数：
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect

IOI_Initialize

功能：该函数初始化控制输入输出驱动器组件操作的数据结构。这里最初没有输入输出驱动器。

语法：VOID IOI_Initialize(VOID)
调用函数：无

4.13 历史组件(HI)

历史组件（HI）负责处理所有的NUCLEUS PLUS历史设备。NUCLEUS PLUS历史组件保持了不同的系统活动的循环记录。应用层任务和HISRs在历史记录中创建入口。历史记录中的每个入口包含了详细的NUCLEUS PLUS服务调用和调用者的相关信息。请参阅Nucleus PLUS reference manual 第14章，了解历史记录更加详细的内容。

4.13.1历史文件

历史组件（HI）由5个文件构成。历史组件每个源文件定义如下：

文件	描述
HI_DEFS.H	该文件包含HI特殊的常量和数据结构的定义。
HI_EXTR.H	HI的所有外部接口函数都定义在该文件里。
HID.C	HI的全局数据结构定义在这个文件里。
HII.C	该文件包含了HI的初始化函数。
HIC.C	该文件包含了HI所有核心函数。它用于处理基本的历史纪录使能和禁止服务。

4.13.2历史数据结构

历史使能

NUCLEUS PLUS历史入口可以动态创建。历史使能标志表示是否允许存储。如果该标志值为NU_FALSE,就禁止历史存储。否则，使能历史存储并在历史记录中按要求产生一个恰当的入口。

写索引

NUCLEUS PLUS历史表中的下一个入口索引包含在变量HID_Write_Index中。该变量的内容相当于在历史表中的下一个有效入口索引的位置。该变量的操作受到HID_History_Protect的保护。

读索引

NUCLEUS PLUS历史表中最旧的入口索引包含在变量HID_Read_Index中。该变量的内容相当于在历史表中最旧的入口索引的位置。该变量的操作受到HID_History_Protect的保护。

历史表保护

NUCLEUS PLUS保护在任务或HISRs竞争历史表的完整性。这是通过使用内部保护结构HID_History_Protect来实现。所有历史的使能和禁止是在HID_History_Protect的保护下操作的。

字符段描述：

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

字符段概述：

tc_tcb_pointer	确认当前线程已受保护。
tc_thread_waiting	该标志表示一个或多个线程正在等待保护。

总入口数

在NUCLEUS PLUS历史表中入口总数存放在变量HID_Entry_Count中。该变量的内容相当于在历史表中有效的入口个数。该变量的操作受到HID_History_Protect的保护。

历史表结构

历史表结构HI_HISTORY_ENTRY包含了当前历史入口的起始索引和处理历史请求所必需的

其他字符段。
字符段描述：

```
DATA_ELEMENT hi_id
DATA_ELEMENT hi_caller
UNSIGNED hi_param1
UNSIGNED hi_param2
UNSIGNED hi_param3
UNSIGNED hi_time
VOID *hi_thread
```

字符段概述：

字符段	描述
hi_id	这是用于在表中历史入口的索引。它只是一个由HI_HISTORY_ENTRY结构构成的阵列。
hi_caller	该实体在历史记录中产生入口。这可以是一个任务、一个HISR或初始化进程。
hi_param1	存储历史记录信息的第一个参数。
hi_param2	存储历史记录信息的第二个参数。
hi_param3	存储历史记录信息的第三个参数。
hi_time	在时钟片中的当前系统时间。
*hi_thread	调用线程的指针。

4.13.3 历史组件函数

下面部分对历史组件（HI）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

HIC_Disable_History_Saving

功能：该函数禁止历史存储函数。
语法：VOID HIC_Disable_History_Saving(VOID)
调用函数：

```
TCT_Protect
TCT_Unprotect
```

HIC_Enable_History_Saving

功能：该函数使能历史存储函数。
语法：VOID HIC_Enable_History_Saving(VOID)
调用函数：

```
TCT_Protect
TCT_Unprotect
```

HIC_Make_History_Entry_Service

功能：该函数在历史表中产生一个应用入口。
语法：

```
VOID HIC_Make_History_Entry_Service(UNSIGNED param1,
                                    UNSIGNEDparam2,
                                    UNSIGNED param3)
```

调用函数：HIC_Make_History_Entry

HIC_Make_History_Entry

功能：该函数在历史表中下一个有效位置产生一个入口（如果历史存储使能）。

语法：

```
VOID HIC_Make_History_Entry(DATA_ELEMENT id,
                            UNSIGNED param1,
                            UNSIGNED param2,
                            UNSIGNED param3)
```

调用函数：

```
TCC_Current_HISR_Pointer
TCC_Current_Task_Pointer
TCT_Get_Current_Protect
TCT_Protect
TCT_Set_Current_Protect
TCT_Unprotect
TCT_Unprotect_Specific
TMT_Retrieve_Clock
```

HIC_Retrieve_History_Entry

功能：该函数检索在历史表中下一个最旧的入口。如果没有更多的入口，就返回一个错误状态。

语法：

```
STATUS HIC_Retrieve_History_Entry(DATA_ELEMENT*id,
                                  UNSIGNED *param1,
                                  UNSIGNED *param2,
                                  UNSIGNED *param3,
                                  UNSIGNED *time,
                                  NU_TASK **task, NU_HISR **hisr)
```

调用函数：

```
TCT_Protect
TCT_Unprotect
```

HII_Initialize

功能：该函数初始化控制历史组件操作的数据结构。

语法：`VOID HII_Initialize(VOID)`

调用函数：无

4.14 错误组件(ER)

错误组件（ER）负责处理所有的NUCLEUS PLUS错误设备。NUCLEUS PLUS错误组件是一个公共错误处理服务例程，它处理致命系统情况。当发生致命错误的时候，系统处理被传送到该组件。这时该服务例程创建一个适当的ASCII错误信息，并告诉用户相关的错误类型。然后通过无限循环转移系统。请参阅Nucleus PLUS reference manual 第3章，了解相关错误管理更加详细的内容。

4.14.1错误组件文件

错误组件（ER）由4个文件构成。错误组件每个源文件定义如下：

文件	描述
ER_EXTR.H	ER的所有外部接口函数都定义在该文件里。
ERD.C	ER的全局数据结构定义在这个文件里。
ERI.C	该文件包含了ER的初始化函数。
ERC.C	该文件包含了ER所有核心函数。它用于处理基本的系统错误服务。

4.14.2错误数据结构

错误代码

通过错误代码的使用可以删除NUCLEUS PLUS错误。当系统判断错误代码存在的情况下，它通过一个错误代码的使用来判断错误的类型。错误代码值存放在变量ERD_Error_Code中。

代码	常量	描述
1	NU_ERROR_CREATING_TIMER_HISR	创建定时器HISR时发生的一个错误。
2	NU_ERROR_CREATING_TIMER_TASK	创建定时器任务时发生的一个错误。
3	NU_STACK_OVERFLOW	任务或堆栈溢出时发生的一个错误。
4	NU_UNHANDLED_INTERRUPT	一个中断发生优于一个 LISR 注册。

错误字符串

NUCLEUS PLUS以ASCII字符串格式报告错误。该字符串建立和存储在变量ERD_Error_String中。该变量的内容相当于在错误发生时铜通过系统报告的错误代码的ASCII码版本。如果条件编译标志NU_ERROR_STRING用于编译ERD.C、ERI.C、ERC.C文件，将产生该字符串。

4.14.3 错误组件函数

下面部分对错误组件（HI）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

ERC_System_Error

功能：该函数处理通过不同的系统组件检测到的系统错误。
语法：VOID ERC_System_Error(INT error_code)
调用函数：无

ERI_Initialize

功能：该函数初始化错误管理组件的数据结构。
语法：VOID ERI_Initialize(VOID)
调用函数：无

4.16 许可证组件(LI)

许可证组件（LI）负责处理所有的NUCLEUS PLUS许可证设备。NUCLEUS PLUS许可证组件是一个公共处理例程，它用于存储和报告客户许可证以及序列号相关信息。该信息包含NUCLEUS PLUS软件当前版本和发行号。请参阅Nucleus PLUS reference manual 第3章，了解相关许可证管理更加详细的内容。

4.15.1许可证组件文件

许可证组件（LI）由2个文件构成。许可证组件每个源文件定义如下：

文件	描述
LID.C	LI的全局数据结构定义在这个文件里。
LIC.C	该文件包含了LI所有核心函数。它用于处理基本的系统许可报告服务。

4.15.2许可证数据结构

许可证字符串

NUCLEUS PLUS以ASCII字符串格式报告发行信息。该字符串建立和存储在变量LID_License__String中。该变量内容包括了客户许可信息和序列号。

4.16.3 许可证组件函数

下面部分对许可证组件（LI）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

LIC_License_Information

功能：该函数返回一个许可信息字符串的指针。该信息字符串确认客户和NUCLEUS PLUS许可的产品线。

语法：CHAR *LIC_License_Information(VOID)

调用函数：无

4.16 发行组件(RL)

发行组件（RL）负责处理所有的NUCLEUS PLUS发行设备。NUCLEUS PLUS发行组件是一个例程，专用于存储和报告发行信息。该信息包含NUCLEUS PLUS软件当前版本和发行号。请参阅Nucleus PLUS reference manual 第3章，了解相关发行管理更加详细的内容。

4.16.1发行组件文件

发行组件（RL）由2个文件构成。发行组件每个源文件定义如下：

文件	描述
RLD.C	RL的全局数据结构定义在这个文件里。
RLC.C	该文件包含了RL所有核心函数。它用于处理基本的系统发行报告服务。

4.16.2发行数据结构

发行字符串

NUCLEUS PLUS以ASCII字符串格式报告发行信息。该字符串建立和存储在变量RLD_Release_String中。该变量包含了NUCLEUS PLUS软件当前发行版本的描述。

特殊字符串

NUCLEUS PLUS以ASCII字符串格式报告混杂信息。该字符串建立和存储在变量RLD_Special_String.中。该变量包含了NUCLEUS PLUS系统原始相关信息。

4.16.3 发行组件函数

下面部分对发行组件（RL）的函数进行简要的描述。推荐回顾实际源代码以获得更多信息。

RLC_Release_Information

功能：该函数返回一个发行信息字符串的指针。该信息字符串确认NUCLEUS PLUS当前版本。

语法：CHAR *RLC_Release_Information(VOID)

调用函数：无

附录 A Nuleus PLUS 常量

本附录包含了所有的Nucleus PLUS在Nucleus PLUS Reference Manual第四章中服务。

Nucleus PLUS常量（按字母顺序排列）

名字	十进制	十六进制
NU_ALLOCATE_MEMORY_ID	47	2F
NU_ALLOCATE_PARTITION_ID	43	2B
NU_AND	2	2
NU_AND_CONSUME	3	3
NU_BROADCAST_TO_MAILBOX_ID	16	10
NU_BROADCAST_TO_PIPE_ID	30	1E
NU_BROADCAST_TO_QUEUE_ID	23	17
NU_CHANGE_PREEMPTION_ID	11	B
NU_CHANGE_PRIORITY_ID	10	A
NU_CHANGE_TIME_SLICE_ID	65	41
NU_CONTROL_SIGNALS_ID	49	31
NU_CONTROL_TIMER_ID	58	3A
NU_CREATE_DRIVER_ID	60	3C
NU_CREATE_EVENT_GROUP_ID	37	25
NU_CREATE_HISR_ID	54	36
NU_CREATE_MAILBOX_ID	12	C
NU_CREATE_MEMORY_POOL_ID	45	2D
NU_CREATE_PARTITION_POOL_ID	41	29
NU_CREATE_PIPE_ID	25	19
NU_CREATE_QUEUE_ID	18	12
NU_CREATE_SEMAPHORE_ID	32	20
NU_CREATE_TASK_ID	2	2
NU_CREATE_TIMER_ID	56	38
NU_DEALLOCATE_MEMORY_ID	48	30
NU_DEALLOCATE_PARTITION_ID	44	2C
NU_DELETE_DRIVER_ID	61	3D
NU_DELETE_EVENT_GROUP_ID	38	26
NU_DELETE_HISR_ID	55	37
NU_DELETE_MAILBOX_ID	13	D
NU_DELETE_MEMORY_POOL_ID	46	2E
NU_DELETE_PARTITION_POOL_ID	42	2A
NU_DELETE_PIPE_ID	26	1A
NU_DELETE_QUEUE_ID	19	13
NU_DELETE_SEMAPHORE_ID	33	21
NU_DELETE_TASK_ID	3	3
NU_DELETE_TIMER_ID	57	39
NU_DISABLE_INTERRUPTS	[Port Specific]	
NU_DISABLE_TIMER	4	4
NU_DRIVER_SUSPEND	10	A
NU_ENABLE_INTERRUPTS	[Port Specific]	
NU_ENABLE_TIMER	5	5

NU_END_OF_LOG	-1	FFFFFFFF
NU_EVENT_SUSPEND	7	7
NU_FALSE	0	0
NU_FIFO	6	6
NU_FINISHED	11	B
NU_FIXED_SIZE	7	7
NU_GROUP_DELETED	-2	FFFFFFFE
NU_INVALID_DELETE	-3	FFFFFFFD
NU_INVALID_DRIVER	-4	FFFFFFFC
NU_INVALID_ENABLE	-5	FFFFFFFB
NU_INVALID_ENTRY	-6	FFFFFFFA
NU_INVALID_FUNCTION	-7	FFFFFFF9
NU_INVALID_GROUP	-8	FFFFFFF8
NU_INVALID_HISR	-9	FFFFFFF7
NU_INVALID_MAILBOX	-10	FFFFFFF6
NU_INVALID_MEMORY	-11	FFFFFFF5
NU_INVALID_MESSAGE	-12	FFFFFFF4
NU_INVALID_OPERATION	-13	FFFFFFF3
NU_INVALID_PIPE	-14	FFFFFFF2
NU_INVALID_POINTER	-15	FFFFFFF1
NU_INVALID_POOL	-16	FFFFFFF0
NU_INVALID_PREEMPT	-17	FFFFFFEF
NU_INVALID_PRIORITY	-18	FFFFFFEE
NU_INVALID_QUEUE	-19	FFFFFFED
NU_INVALID_RESUME	-20	FFFFFFEC
NU_INVALID_SEMAPHORE	-21	FFFFFFEB
NU_INVALID_SIZE	-22	FFFFFFEA
NU_INVALID_START	-23	FFFFFFE9
NU_INVALID_SUSPEND	-24	FFFFFFE8
NU_INVALID_TASK	-25	FFFFFFE7
NU_INVALID_TIMER	3	FFFFFFE6
NU_INVALID_VECTOR	-27	FFFFFFE5
NU_MAILBOX_DELETED	-28	FFFFFFE4
NU_MAILBOX_EMPTY	-29	FFFFFFE3
NU_MAILBOX_FULL	-30	FFFFFFE2
NU_MAILBOX_RESET	-31	FFFFFFE1
NU_MAILBOX_SUSPEND	3	3
NU_MEMORY_SUSPEND	9	9
NU_NO_MEMORY	-32	FFFFFFE0
NU_NO_MORE_LISRS	-33	FFFFFFDF
NU_NO_PARTITION	-34	FFFFFFDE
NU_NO_PREEMPT	8	8
NU_NO_START	9	9
NU_NO_SUSPEND	0	0
NU_NOT_DISABLED	-35	FFFFFFDD
NU_NOT_PRESENT	-36	FFFFFFDC

NU_NOT_REGISTERED	-37	FFFFFFDB
NU_NOT_TERMINATED	-38	FFFFFFDA
NU_NULL	0	0
NU_OBTAIN_SEMAPHORE_ID	35	23
NU_OR	0	0
NU_OR_CONSUME	1	1
NU_NO_SUSPEND	0	0

NU_PARTITION_SUSPEND	8	8
NU_PIPE_DELETED	-39	FFFFFFD9
NU_PIPE_EMPTY	-40	FFFFFFD8
NU_PIPE_FULL	-41	FFFFFFD7
NU_PIPE_RESET	-42	FFFFFFD6
NU_PIPE_SUSPEND	5	5
NU_POOL_DELETED	-43	FFFFFFD5
NU_PREEMPT	10	A
NU_PRIORITY	11	B
NU_PURE_SUSPEND	1	1
NU_QUEUE_DELETED	-44	FFFFFFD4
NU_QUEUE_EMPTY	-45	FFFFFFD3
NU_QUEUE_FULL	-46	FFFFFFD2
NU_QUEUE_RESET	-47	FFFFFFD1
NU_QUEUE_SUSPEND	4	4
NU_READY	0	0
NU_RECEIVE_FROM_MAILBOX_ID	17	11
NU_RECEIVE_FROM_PIPE_ID	31	1F
NU_RECEIVE_FROM_QUEUE_ID	24	18
NU_RECEIVE_SIGNALS_ID	50	32
NU_REGISTER_LISR_ID	53	35
NU_REGISTER_SIGNAL_HANDLER_ID	51	33
NU_RELEASE_SEMAPHORE_ID	36	24
NU_RELINQUISH_ID	8	8
NU_REQUEST_DRIVER_ID	62	3E
NU_RESET_MAILBOX_ID	14	E
NU_RESET_PIPE_ID	27	1B
NU_RESET_QUEUE_ID	20	14
NU_RESET_SEMAPHORE_ID	34	22
NU_RESET_TASK_ID	4	4
NU_RESET_TIMER_ID	59	3B
NU_RESUME_DRIVER_ID	63	3F
NU_RESUME_TASK_ID	6	6
NU_RETRIEVE_EVENTS_ID	40	28
NU_SEMAPHORE_DELETED	-48	FFFFFFD0
NU_SEMAPHORE_RESET	-49	FFFFFFCF
NU_SEMAPHORE_SUSPEND	6	6
NU_SEND_SIGNALS_ID	52	34

NU_SEND_TO_FRONT_OF_QUEUE_ID	21	15
NU_SEND_TO_FRONT_OF_PIPE_ID	28	1C
NU_SEND_TO_MAILBOX_ID	15	F
NU_SEND_TO_PIPE_ID	29	1D
NU_SEND_TO_QUEUE_ID	22	16
NU_SET_EVENTS_ID	39	27
NU_SLEEP_ID	9	9
NU_SLEEP_SUSPEND	2	2
NU_START	12	C
NU_SUCCESS	0	0
NU_SUSPEND	0xFFFFFFFFUL	FFFFFFFF
NU_SUSPEND_DRIVER_ID	64	40
NU_SUSPEND_TASK_ID	7	7
NU_TERMINATE_TASK_ID	5	5
NU_TERMINATED	12	C
NU_TIMEOUT	-50	FFFFFFCE
NU_TRUE	1	1
NU_UNAVAILABLE	-51	FFFFFFCD
NU_USER_ID	1	1
NU_VARIABLE_SIZE	13	D

NU_ENABLE_INTERRUPTS	[Port Specific]	
NU_DISABLE_INTERRUPTS	[Port Specific]	
NU_FALSE	0	0
NU_NO_SUSPEND	0	0
NU_NULL	0	0
NU_OR	0	
0		
NU_READY	0	0
NU_SUCCESS	0	0
NU_OR_CONSUME	1	1
NU_PURE_SUSPEND	1	1
NU_TRUE	1	1
NU_USER_ID	1	1
NU_AND	2	2
NU_CREATE_TASK_ID	2	2
NU_SLEEP_SUSPEND	2	2
NU_AND_CONSUME	3	3
NU_DELETE_TASK_ID	3	3
NU_MAILBOX_SUSPEND	3	3
NU_DISABLE_TIMER	4	4
NU_QUEUE_SUSPEND	4	4
NU_RESET_TASK_ID	4	4
NU_ENABLE_TIMER	5	5
NU_PIPE_SUSPEND	5	5

NU_TERMINATE_TASK_ID	5	5
NU_FIFO	6	6
NU_RESUME_TASK_ID	6	6
NU_SEMAPHORE_SUSPEND	6	6
NU_EVENT_SUSPEND	7	7
NU_FIXED_SIZE	7	7
NU_SUSPEND_TASK_ID	7	7
NU_NO_PREEMPT	8	8
NU_PARTITION_SUSPEND	8	8
NU_RELINQUISH_ID	8	8
NU_MEMORY_SUSPEND	9	9
NU_NO_START	9	9
NU_SLEEP_ID	9	9
NU_CHANGE_PRIORITY_ID	10	A
NU_DRIVER_SUSPEND	10	A
NU_PREEMPT	10	A
NU_RESUME_TASK_ID	6	6
NU_SEMAPHORE_SUSPEND	6	6
NU_EVENT_SUSPEND	7	7
NU_FIXED_SIZE	7	7
NU_SUSPEND_TASK_ID	7	7
NU_NO_PREEMPT	8	8
NU_PARTITION_SUSPEND	8	8
NU_RELINQUISH_ID	8	8
NU_MEMORY_SUSPEND	9	9
NU_NO_START	9	9
NU_SLEEP_ID	9	9
NU_CHANGE_PRIORITY_ID	10	A
NU_DRIVER_SUSPEND	10	A
NU_PREEMPT	10	A

NU_CHANGE_PREEMPTION_ID	11	B
NU_FINISHED	11	B
NU_PRIORITY	11	B
NU_CREATE_MAILBOX_ID	12	C
NU_START	12	C
NU_TERMINATED	12	C
NU_DELETE_MAILBOX_ID	13	D
NU_VARIABLE_SIZE	13	D
NU_RESET_MAILBOX_ID	14	E
NU_SEND_TO_MAILBOX_ID	15	F
NU_BROADCAST_TO_MAILBOX_ID	16	10
NU_RECEIVE_FROM_MAILBOX_ID	17	11
NU_CREATE_QUEUE_ID	18	12
NU_DELETE_QUEUE_ID	19	13
NU_RESET_QUEUE_ID	20	14

NU_SEND_TO_FRONT_OF_QUEUE_ID	21	15
NU_SEND_TO_QUEUE_ID	22	16
NU_BROADCAST_TO_QUEUE_ID	23	17
NU_RECEIVE_FROM_QUEUE_ID	24	18
NU_CREATE_PIPE_ID	25	19
NU_DELETE_PIPE_ID	26	1A
NU_RESET_PIPE_ID	27	1B
NU_SEND_TO_FRONT_OF_PIPE_ID	28	1C
NU_SEND_TO_PIPE_ID	29	1D
NU_BROADCAST_TO_PIPE_ID	30	1E
NU_RECEIVE_FROM_PIPE_ID	31	1F
NU_CREATE_SEMAPHORE_ID	32	20
NU_DELETE_SEMAPHORE_ID	33	21
NU_RESET_SEMAPHORE_ID	34	22
NU_OBTAIN_SEMAPHORE_ID	35	23
NU_RELEASE_SEMAPHORE_ID	36	24
NU_CREATE_EVENT_GROUP_ID	37	25
NU_DELETE_EVENT_GROUP_ID	38	26
NU_SET_EVENTS_ID	39	27
NU_RETRIEVE_EVENTS_ID	40	28
NU_CREATE_PARTITION_POOL_ID	41	29
NU_DELETE_PARTITION_POOL_ID	42	2A
NU_ALLOCATE_PARTITION_ID	43	2B
NU_DEALLOCATE_PARTITION_ID	44	2C
NU_CREATE_MEMORY_POOL_ID	45	2D
NU_DELETE_MEMORY_POOL_ID	46	2E
NU_ALLOCATE_MEMORY_ID	47	2F
NU_DEALLOCATE_MEMORY_ID	48	30
NU_CONTROL_SIGNALS_ID	49	31
NU_RECEIVE_SIGNALS_ID	50	32
NU_REGISTER_SIGNAL_HANDLER_ID	51	33
NU_SEND_SIGNALS_ID	52	34
NU_REGISTER_LISR_ID	53	35
NU_CREATE_HISR_ID	54	36
NU_DELETE_HISR_ID	55	37
NU_CREATE_TIMER_ID	56	38
NU_DELETE_TIMER_ID	57	39
NU_CONTROL_TIMER_ID	58	3A
NU_RESET_TIMER_ID	59	3B
NU_CREATE_DRIVER_ID	60	3C
NU_DELETE_DRIVER_ID	61	3D
NU_REQUEST_DRIVER_ID	62	3E
NU_RESUME_DRIVER_ID	63	3F
NU_SUSPEND_DRIVER_ID	64	40
NU_CHANGE_TIME_SLICE	65	41
NU_SUSPEND	0xFFFFFFFFUL	FFFFFFFF

NU_END_OF_LOG	-1	FFFFFFFF
NU_GROUP_DELETED	-2	FFFFFFFE
NU_INVALID_DELETE	-3	FFFFFFFD
NU_INVALID_DRIVER	-4	FFFFFFFC
NU_INVALID_ENABLE	-5	FFFFFFFB
NU_INVALID_ENTRY	-6	FFFFFFFA
NU_INVALID_FUNCTION	-7	FFFFFFF9
NU_INVALID_GROUP	-8	FFFFFFF8
NU_INVALID_HISR	-9	FFFFFFF7
NU_INVALID_MAILBOX	-10	FFFFFFF6
NU_INVALID_MEMORY	-11	FFFFFFF5
NU_INVALID_MESSAGE	-12	FFFFFFF4
NU_INVALID_OPERATION	-13	FFFFFFF3
NU_INVALID_PIPE	-14	FFFFFFF2
NU_INVALID_POINTER	-15	FFFFFFF1
NU_INVALID_POOL	-16	FFFFFFF0
NU_INVALID_PREEMPT	-17	FFFFFFEF
NU_INVALID_PRIORITY	-18	FFFFFFEE
NU_INVALID_QUEUE	-19	FFFFFFED
NU_INVALID_RESUME	-20	FFFFFFEC
NU_INVALID_SEMAPHORE	-21	FFFFFFEB
NU_INVALID_SIZE	-22	FFFFFFEA
NU_INVALID_START	-23	FFFFFFE9
NU_INVALID_SUSPEND	-24	FFFFFFE8
NU_INVALID_TASK	-25	FFFFFFE7
NU_INVALID_TIMER	-26	FFFFFFE6
NU_INVALID_VECTOR	-27	FFFFFFE5
NU_MAILBOX_DELETED	-28	FFFFFFE4
NU_MAILBOX_EMPTY	-29	FFFFFFE3
NU_MAILBOX_FULL	-30	FFFFFFE2
NU_MAILBOX_RESET	-31	FFFFFFE1
NU_NO_MEMORY	-32	FFFFFFE0
NU_NO_MORE_LISRS	-33	FFFFFFDF
NU_NO_PARTITION	-34	FFFFFFDE
NU_NOT_DISABLED	-35	FFFFFFDD
NU_NOT_PRESENT	-36	FFFFFFDC
NU_NOT_REGISTERED	-37	FFFFFFDB
NU_NOT_TERMINATED	-38	FFFFFFDA
NU_PIPE_DELETED	-39	FFFFFFD9
NU_PIPE_EMPTY	-40	FFFFFFD8
NU_PIPE_FULL	-41	FFFFFFD7
NU_PIPE_RESET	-42	FFFFFFD6
NU_POOL_DELETED	-43	FFFFFFD5
NU_QUEUE_DELETED	44	FFFFFFD4
NU_QUEUE_EMPTY	45	FFFFFFD3
NU_QUEUE_FULL	46	FFFFFFD2

NU_QUEUE_RESET	47	FFFFFFD1
NU_SEMAPHORE_DELETED	48	FFFFFFD0
NU_SEMAPHORE_RESET	49	FFFFFFCF
NU_TIMEOUT	50	FFFFFFCE
NU_UNAVAILABLE	51	FFFFFFCD

附录 B 致命系统错误

本附录包含了所有的Nucleus PLUS致命系统错误常量。如果一个致命系统错误发生，就有一个常量传输给致命处理函数ERC_System_Error。

如果系统错误是NU_STACK_OVERFLOW,就表示当前执行线程堆栈太小。当前线程通过全局变量TCD_Current_Thread的检查来确认。这包含了指向当前线程控制块的指针。

如果系统错误是NU_UNHANDLED_INTERRUPT,就接收到一个与LISR无关的中断。该中断向量引起存储在全局变量TCD_Unhandled_Interrupt中的系统错误。

Nucleus PLUS致命系统错误

名字	十进制	十六进制
NU_ERROR_CREATING_TIMER_HISR	1	1
NU_ERROR_CREATING_TIMER_TASK	2	2
NU_STACK_OVERFLOW	3	3
NU_UNHANDLED_INTERRUPT	4	4

附录 C I/O驱动器结构请求

本附录包含了所有标准的Nucleus PLUS I/O驱动器常量和请求结构。请参阅Nucleus PLUS Reference Manual第三和第五章关于I/O驱动器的讨论。

Nucleus PLUS I/O驱动器常量

名字	十进制	十六进制
NU_IO_ERROR	-1	FFFFFFFF
NU_INITIALIZE	1	1
NU_ASSIGN	2	2
NU_RELEASE	3	3
NU_INPUT	4	4
NU_OUTPUT	5	5
NU_STATUS	6	6

Nucleus PLUS I/O驱动器 C语言结构

```
/* Define I/O driver request structures. */
struct NU_INITIALIZE_STRUCT
{
VOID *nu_io_address; /* Base IO address */
UNSIGNED nu_logical_units; /* Number of logical units */
VOID *nu_memory; /* Generic memory pointer */
INT nu_vector; /* Interrupt vector number */
};
struct NU_ASSIGN_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
```

```
INT nu_assign_info; /* Additional assign info */
};
struct NU_RELEASE_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
INT nu_release_info; /* Additional release info */
};
struct NU_INPUT_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
UNSIGNED nu_offset; /* Offset of input */
UNSIGNED nu_request_size; /* Requested input size */
UNSIGNED nu_actual_size; /* Actual input size */
VOID *nu_buffer_ptr; /* Input buffer pointer */
};
struct NU_OUTPUT_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
UNSIGNED nu_offset; /* Offset of output */
UNSIGNED nu_request_size; /* Requested output size */
UNSIGNED nu_actual_size; /* Actual output size */
VOID *nu_buffer_ptr; /* Output buffer pointer */
};
struct NU_STATUS_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
VOID *nu_extra_status; /* Additional status ptr */
};
struct NU_TERMINATE_STRUCT
{
UNSIGNED nu_logical_unit; /* Logical unit number */
};
typedef struct NU_DRIVER_REQUEST_STRUCT
{
INT nu_function; /* I/O request function */
UNSIGNED nu_timeout; /* Timeout on request */
STATUS nu_status; /* Status of request */
UNSIGNED nu_supplemental; /* Supplemental information */
VOID *nu_supplemental_ptr; /* Supplemental info pointer*/
/* Define a union of all the different types of request
structures.*/
union NU_REQUEST_INFO_UNION
{
struct NU_INITIALIZE_STRUCT nu_initialize;
struct NU_ASSIGN_STRUCT nu_assign;
struct NU_RELEASE_STRUCT nu_release;
struct NU_INPUT_STRUCT nu_input;
```

```
struct NU_OUTPUT_STRUCT nu_output;
struct NU_STATUS_STRUCT nu_status;
struct NU_TERMINATE_STRUCT nu_terminate;
} nu_request_info;
} NU_DRIVER_REQUEST;
```