Nucleus PLUS Internals

0001027-001 Rev. 102

Accelerated Technology
Embedded Systems Division
of Mentor Graphics
720 Oak Circle Dr. E.
Mobile, AL 36609
(251) 661-5770

All You NEED in an RTOS. Royalty Free.

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®

Related Documentation

Nucleus PLUS Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus PLUS kernel.

Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - `Courier New`
Filenames - `COURIER NEW, ALL CAPS`
Interactive Command Lines - **`Courier New, Bold`**
Menu Items/Buttons – *Times New Roman Italic*

Trademarks

MS-DOS is a trademark of Microsoft Corporation
UNIX is a trademark of X/Open
IBM PC is a trademark of International Business Machines, Inc.

Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology
720 Oak Circle Drive, East
Mobile, AL 36609
800-468-6853
251-661-5770
251-661-5788 (fax)

support@acceleratedtechnology.com
http://www.acceleratedtechnology.com

Document Part Number: 0001027-001 Rev. 102
Last Revised: May 13, 2002

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®

Contents

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®

1

Introduction

Purpose of Manual

Nucleus PLUS is delivered in source code form. Since the source code for Nucleus PLUS is quite large, a typical user would have a difficult time making any sense out of it. This manual is designed to help Nucleus PLUS users understand the source code.

About Nucleus PLUS

Nucleus PLUS is a real-time, preemptive, multitasking kernel designed for time-critical embedded applications. Approximately 95% of Nucleus PLUS is written in ANSI C. Because of this, Nucleus PLUS is extremely portable and is currently available for use with most microprocessor families.

Nucleus PLUS is typically implemented as a C library. Real-time Nucleus PLUS applications are linked with the Nucleus PLUS library. The resulting object may be downloaded to the target or placed in ROM. In a typical target environment, the binary image of the Nucleus PLUS instruction area, assuming all services are used, requires roughly 20 Kbytes of memory.

Nucleus PLUS Construction

Accelerated Technology's software development practices facilitate clarity, modularity, reliability, reusability, and ease of maintenance. Nucleus PLUS is comprised of multiple software components. Each software component has a unique purpose and a specific external interface to other components. The composition of each Nucleus PLUS software component is discussed in greater detail in subsequent chapters.

2

Implementation Conventions

Components

Accelerated Technology (ATI) uses a software component methodology. A software component has a single, clear purpose. Software components are typically comprised of several C and/or assembly language files. Each software component provides a well-defined external interface. Utilization of a component is accomplished through use of its external interface. With few exceptions, access to global data structures within a component is not allowed outside of the component. Because of this component methodology, Nucleus PLUS software components are both easy to replace and easy to re-use

Component Composition

A software component is typically comprised of an include file for data type definitions and constants, an include file for the component's external interfaces, and one or more C and/or assembly files. Component file names conform to the following conventions:

File	**Meaning**
`XX_DEFS.H`	Component constants and data structures are defined in this file.
`XX_EXTR.H`	External interfaces to the component are defined in this file. These interfaces are defined in terms of function prototypes.
`XXD.C`	Static and global data structures within the component are defined in this file. With few exceptions, data structures of one component are only accessed from functions within the component.
`XXI.C`	The component initialization function is defined in this file.
`XXF.C`	This file contains functions that provide status information about objects managed by the component.
`XXC.C`	This file contains the core functions of the component.
`XXCE.C`	This file contains the error-checking shell functions for the core functions.
`XXS.C`	Supplemental functions for the component are defined in this file.
`XXSE.C`	Error-checking functions for the supplemental component functions are defined in this file.

NOTE: xx represents the two-letter name of the component. A component does not necessarily have every possible type of file.

Format

All software source files have the same fundamental format. The first part of the file contains general information about the file and is called the prologue. The second part of the files is dedicated to internal data declarations and internal function prototyping. The remaining part of the file contains the actual functions.

Prologue

The purpose of the prologue is to describe the contents of the file, identify ATI as the owner of the file, and to provide information about revisions to the file.

An example of the prologue format follows:

```
/*************************************************************************/
/*                                                                       */
/*Copyright (c) 199x by Accelerated Technology                           */
/*                                                                       */
/*  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]       */
/*************************************************************************/
```

After the Prologue

The area after the prologue is reserved for constants, global data structure definitions, and inter-component function prototypes. Of course, include files only define component data structure types or external interfaces.

Remainder of File

The remainder of a software component file consists of C or assembly language functions. Each function is preceded by a description block. The format of a function description block follows:

```
/*************************************************************************/
/* 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]   */
/*                                                                       */
/*************************************************************************/
```

Naming Conventions

Naming conventitons are intended to make examination of the ATI source code less painful by incorporating the first three or four characters of the file name into global variable and function names. Of course, all names correspond to their usage. Detailed descriptions of the naming conventions are described in the following sub-sections.

Component Names

Component names are generally limited to two characters. The component name is used as the first two characters of each file that makes up the component.

Example:

Dynamic Memory Management Component Name: DM
Files that comprise DM:

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

#define Names

Defines are comprised of underscores, capital letters, and numeric characters. The maximum length of a define is 31 characters. Additionally, the first three characters of a define are "`CC_`" where "`CC`" is the same as the first two letters of the file name where the define is located.

Example (for file `EX_DEFS.H`):

```
#define EX_MY_CONSTANT      10
```

Structure Names

Structure names are comprised of underscores, capital letters, and numeric characters. The maximum length of a structure name is 31characters. Additionally, the first three characters of a structure name are "`CC_`" where "`CC`" is the same as the first two letters of the file name where the structure is defined.

Example (for file `EX_DEFS.H`):

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

Typedef Names

Typedef names are comprised of underscores, capital letters, and numeric characters. The maximum length of a typedef name is 31characters. Additionally, the first three characters of a typedef name are "`CC_`" where "`CC`" is the same as the first two letters of the file name the typedef is defined in.

Example (for file `EX_DEFS.H`):

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

Structure Member Names

Structure member names are comprised of underscores, lower-case letters, and numeric characters. The maximum length of structure member names is 31characters. Additionally, the first three characters of a structure member are defined as "`CC_`" where "`CC`" is the same as the first two letters of the "`CC_DEFS.H`" file that contains the structure definition.

Example (for file `EX_DEFS.H`):

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

Global Variable Names

Nucleus Plus global variable names are comprised of underscores, a single upper case character following each underscore, lower case characters, and numeric characters. The maximum length of a global variable name is 31characters. Additionally, the first three letters of a global variable name are defined as "`CCC`" where "`ccc`" is the same as the first three letters of the "`CCC.C`" file that contains the actual variable declaration.

Example (for file `EXD.C`):

```
int EXD_Global_Integer;
```

Local Variable Names

Local variable names (names for variables defined inside the context of a C function) are comprised of lower-case characters and possibly underscores and/or numeric characters. The maximum length of a local variable name is 31characters. Local variable names are not required to take the first three characters of the file they are defined in.

Example (for file `EXD.C`):

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

Function Names

Nucleus Plus function names are comprised of underscores, a single upper case character following each underscore, lower case characters, and numeric characters. The maximum length of a function name is 31characters. Additionally, the first three characters of a function name are the same as those of the file that contains the function definition.

Example (for file `EXD.C`):

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

Indentation

The basic unit of indentation is 4 spaces. Function declarations, variable declarations, and conditional compilation constructs start at column 1. Actual instructions start at column 4.

NOTE: the braces { and } are on separate lines. The { brace has the same indentation as the previous line, while the } brace lines up with the previous { brace.

Example (for file `EXD.C`):

```
void  EXD_Example_Function(int  i,  int  b)
{

unsigned int   a;
char         b;

    /* Actual instructions start.  */
    i =  0;
    while (i < 100)
    {
         /* Increment i.  */
         i =  i + 1;
    }
}
```

Comments

Comments are one of the most important features of the Nucleus PLUS source code. They are used in a meaningful and plentiful manner. There are two principal types of comments in ATI software. The first type of comment starts at the current indentation, while the second type of comment starts at column 45.

Example:

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

Accelerated Technology®

Embedded Systems Division of

Mentor Graphics®

3

Software Overview

Basic Usage

Data Types

Service Call Mapping

Environment Dependencies

Version Control

Basic Usage

Nucleus PLUS is typically implemented as a C library. Real-time Nucleus PLUS applications are linked with the Nucleus PLUS library. The resulting object may then be downloaded to the target or placed in ROM.

`PLUS.LIB` is typically the file name of the Nucleus PLUS library. This is built with the batch file `PLUS.BAT`. The contents of `PLUS.BAT` are specific to the development tool being used.

Operation Mode

In processor architectures that have both supervisor and user modes of operation, Nucleus PLUS application tasks typically run in supervisor mode. This is because application tasks make direct calls to operating system services that utilize privileged instructions. This method reduces service call overhead and is also much easier to implement. Of course, this method allows the tasks to access anything and everything.

Application Initialization

The user is responsible for providing its own initialization routine, which is called `Application_Initialize.` This routine should create the tasks, queues, and other system objects that are required when the system starts. If the application does not utilize dynamic creation/deletion of system objects during run-time, all of the required system objects may be created in `Application_Initialize`. Multitasking begins immediately after the user's `Application_Initialize` routine returns.

In some target environments, the low-level system initialization files, `INT.S`, `INT.ASM`, or `INT.SRC` may require modification. These files initialize the system timer interrupt, available memory, and other entities that are inherently processor or board specific.

Include File

All user code that references Nucleus PLUS services and/or data types, must include the file `NUCLEUS.H`. This file contains data type definitions, constant definitions, and function prototypes for all of the Nucleus PLUS services. This file is specific to each port of Nucleus PLUS.

Data Types

Nucleus PLUS defines several standard data types in the file `NUCLEUS.H`. These data types are guaranteed to remain constant in capability by assigning the appropriate target C compiler's basic data type. Therefore, Nucleus PLUS can perform in an identical manner on a variety of target environments.

The following data types are defined by Nucleus PLUS:

Data Type	Meaning
`UNSIGNED`	This is required to be a 32-bit unsigned integer. It is usually defined as an unsigned long C data type.
`SIGNED`	This is required to be a 32-bit signed integer. It is usually defined as a signed long C data type.
`OPTION`	Smallest data type that is easily manipulated - usually an unsigned char C data type.
`DATA_ELEMENT`	Same as the previous `OPTION` data type.
`UNSIGNED_CHAR`	This data type is required to be an 8-bit unsigned character.
`CHAR`	This data type is required to be an 8-bit character.
`STATUS`	Equivalent to target C compiler's signed int data type.
`INT`	An integer data type that corresponds to the natural word size of the underlying architecture.
`VOID`	Equivalent to target C compiler's void data type.
`UNSIGNED_PTR`	This data type is a pointer to an `UNSIGNED` data type.
`BYTE_PTR`	This data type is a pointer to an `UNSIGNED_CHAR` data type.

Service Call Mapping

The main Nucleus PLUS include file, `NUCLEUS.H`, contains function prototypes that match those defined in the *Nucleus PLUS Reference Manual*. However, the `NU_*` functions do not really exist. For most Nucleus PLUS services, there exists a function that really does the work and a "shell" function that checks for errors in the user's request before calling the real function. Depending on the error checking conditional define, `NU_NO_ERROR_CHECKING`, the Nucleus PLUS service call is mapped, through macro substitution, to the appropriate underlying function. This facilitates complete elimination of error checking when it is not required.

Error Checking

If the `NU_NO_ERROR_CHECKING` flag is **not** defined (default condition), the `NU_*` service calls defined in the *Nucleus PLUS Reference Manual* are *mapped* to the following internal functions:

Nucleus PLUS Service	Internal Function
`NU_Activate_HISR`	`TCCE_Activate_HISR`
`NU_Allocate_Memory`	`DMCE_Allocate_Memory`
`NU_Allocate_Partition`	`PMCE_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_Priority`
`NU_Change_Time_Slice`	`TCSE_Change_Time_Slice`

Nucleus PLUS Service	Internal Function
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_Create_Task	TCCE_Create_Task
NU_Create_Timer	TMSE_Create_Timer
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_Established_Memory_Pools	DMF_Established_Memory_Pools
NU_Established_Partition_Pools	PMF_Established_Partition_Pools
NU_Established_Pipes	PIF_Established_Pipes
NU_Established_Queues	QUF_Established_Queues
NU_Established_Semaphores	SMF_Established_Semaphores

Nucleus PLUS Service	Internal Function
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_Get_Remaining_Time	TMF_Get_Remaining_Time
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_From_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	SMSE_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

Nucleus PLUS Service	Internal Function
`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`

No Error Checking

If the `NU_NO_ERROR_CHECKING` flag is defined (usually with a -D compilation switch), the `NU_*` service calls defined in the *Nucleus PLUS Reference Manual* are *mapped* to the following internal functions:

Nucleus PLUS Service	Internal Function
NU_Activate_HISR	TCC_Activate_HISR
NU_Allocate_Memory	DMC_Allocate_Memory
NU_Allocate_Partition	PMC_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_Priority
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_Create_Task	TCC_Create_Task
NU_Create_Timer	TMS_Create_Timer
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_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

Nucleus PLUS Service	Internal Function
NU_Established_HISRs	TCF_Established_HISRs
NU_Established_Mailboxes	MBF_Established_Mailboxes
NU_Established_Memory_Pools	DMF_Established_Memory_Pools
NU_Established_Partition_Pools	PMF_Established_Partition_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_Get_Remaining_Time	TMF_Get_Remaining_Time
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	SMC_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_From_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

Nucleus PLUS Service	**Internal Function**
`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`

Conditional Compilation

The Nucleus PLUS source code has a limited number of conditional compilation options. There are several options available during compilation of application code. However, most options are applicable to the creation of the Nucleus PLUS library.

Library Conditional Flags

Conditional compilation flags for the Nucleus PLUS library are usually specified in the `PLUS.BAT` batch file. These conditional compilation flags enable various features within the Nucleus PLUS library source. The conditional compilation options are as follows:

Compilation Flag	**Meaning**
`NU_ENABLE_HISTORY`	Enable history saving in the specified file. Note: only files of the form `**C.C` are affected by this option.
`NU_ENABLE_STACK_CHECK`	Enable stack checking at the beginning of each function in the specified file. Note: only files of the form `**C.C` are affected by this option.
`NU_ERROR_STRING`	Enable making an ASCII error string if a fatal system error occurs. This flag is applicable to the compilation of `ERD.C`, `ERI.C`, and `ERC.C`.
`NU_NO_ERROR_CHECKING`	Disable error-checking shell on creation of the timer HISR in `TMI.C`. Not applicable to any other Nucleus PLUS library compilation.
`NU_DEBUG`	Maps the application data structures defined in `NUCLEUS.H` to the actual internal data structures used in Nucleus PLUS. This option allows the user to examine all Nucleus PLUS data structures directly. All library files and application files should either use or not use this option.
`NU_INLINE`	Replaces some linked-list processing with in-line code in order to improve performance. This option is applicable to any or all `**C.C` or `**S.C` files.

Library Conditional Values

In addition to the externally defined conditional compilation flags, there are several conditional compilation values defined in `NUCLEUS.H`. These values are set up specifically for each port. Changing any of these values (except the `R1`, `R2`, `R3`, `R4` options) should be done with caution. The conditional values are defined as follows:

Compilation Value	**Meaning**
`NU_POINTER_ACCESS`	This value specifies how many separate memory accesses are required to load and store a data pointer. A value of one allows an in-line optimization. Any value greater than one uses a function to load/store certain data pointers under protection from interrupts.
`PAD_1`	This value specifies how many bytes of padding should be added after a single character in a structure.
`PAD_2`	This value specifies how many bytes of padding should be added after two consecutive characters in a structure.
`PAD_3`	This value specifies how many bytes of padding should be added after three consecutive characters in a structure.
`R1, R2, R3, R4`	These values are used to place the "register" modifier in front of frequently used variables in Nucleus PLUS. R1 is used to modify the most frequently used variable. By defining any of these to "register" the corresponding variable in the source code is assigned register status.

Application Conditional Flags

There are several conditional flags available when compiling application programs. Nucleus PLUS application elements may disable error checking on parameters supplied to Nucleus PLUS services by defining **`NU_NO_ERROR_CHECKING`** with a compiler command line option. This results in a substantial increase in run-time performance, and also reduces code size.

Application data structures defined in `NUCLEUS.H` may be mapped directly to the internal Nucleus PLUS data structures by defining the `NU_DEBUG` option during compilation. This allows the user to directly examine the internals of each Nucleus PLUS data structure within a source-level debugging environment. If the `NU_DEBUG` option is used, it is often a good idea to re-build all of the Nucleus PLUS source code swith the same `NU_DEBUG` option.

Environment Dependencies

Processor and development tool dependencies in Nucleus PLUS have been isolated to four files. Three of the files `(INT.?`, `TCT.?`, and `TMT.?)` are written in assembly language. These files provide the low-level, run-time environment for the underlying target environment. The third file `(NUCLEUS.H)` is included, either directly or indirectly, by all of the files in the system. This file defines various data types and other processor and development tool specific information.

Initialization

The `INT.[S, ASM, or SRC]` file is responsible for providing low-level initialization and services for accessing the processor's interrupt vector table. This file also contains default Interrupt Service Routine (ISR) handlers. The function `INT_Initialize` is specific to a given target board. For example, if the target processor is not able to generate an internal timer interrupt, setting up the timer interrupt becomes board specific. This means that a modified version of `INT` might be necessary for different boards even though they share the same processor architecture.

Thread Control

The `TCT.[S, ASM, or SRC]` file is primarily responsible for transferring control between threads and the system. A thread is defined as either a Nucleus PLUS task or a Nucleus PLUS HISR. This file contains all of the code necessary to perform context switches between tasks and HISRs. Additionally, this file contains code necessary for handling protection conflicts and task signals.

Timer Management

The `TMT.[S, ASM,` or `SRC]` file is primarily responsible for handling Nucleus PLUS timer services, including the timer interrupt handler. In most ports, the timer interrupt handler is designed for low-overhead operation when no timer has expired.

Nucleus PLUS Include File

The `NUCLEUS.H` include file is included by all Nucleus PLUS source files - either directly or indirectly. Application files that reference Nucleus PLUS services and/or data types must also include `NUCLEUS.H`. This file defines a variety of data types, interrupt lockout/enable values, the number of interrupt vectors, the size of system control blocks, and other target specific information.

Version Control

There are several different version layers in a Nucleus PLUS system. The system version is defined by the ASCII string `RLD_Release_String` in the file `RLD.C`. This version contains the current version of the generic C source code as well as the version of the target specific code. For example, the release string for version 1 of the MS-DOS/Borland target specific code with version 1.13 of the generic code would be:

“`Copyright (c) 200x ATI - Nucleus PLUS - DOS Borland C Version 1.13.1`”

Nucleus PLUS also has version information for each file. The version information in the header block of each file identifies the version of that specific file. In many cases, the version in the file header is quite different than the version of the system. Each function in a file also contains version information in its header. The version information specified near the bottom of each function’s header indicates what changes were made to the function and which version of the file they apply to.

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®

4

Component Descriptions

Nucleus PLUS Internals

This chapter describes various software components of the Nucleus PLUS system. Each component's files, data structures, and functions are described. Nucleus PLUS is comprised of 16 distinct components.

Common Services Component (CS)

The Common Services Component (CS) is responsible for providing other Nucleus PLUS components with linked list facilities. Each Common Service node data structure is included within other system data structures.

Common Services Files

The Common Services Component (CS) consists of three files. Each source file of the Common Services Component is defined below.

File	**Description**
`CS_DEFS.H`	This file contains constants and data structure definitions specific to the CS.
`CS_EXTR.H`	All external interfaces to the CS are defined in this file.
`CSC.C`	This file contains all of the core functions of the CS. Functions that handle basic place-on-list and remove-from-list services are defined in this file.

Common Services Control Block

The Common Services Control Block `CS_NODE` contains the previous and next pointers to link the Common Services nodes together, and other fields necessary for processing Common Services requests.

Field Declarations

```
struct CS_NODE_STRUCT       *cs_previous
struct CS_NODE_STRUCT       *cs_next
DATA_ELEMENT        cs_priority
DATA_ELEMENT        cs_padding[PAD_1]
```

Field Summary

Field	Description
`*cs_previous`	This is a link in the current node to the previous node structure for Common Services. It is part of the Common Services list, which is a doubly linked, circular list.
`*cs_next`	This is a link in the current node to the next node structure for Common Services. It is part of the Common Services list, which is a doubly linked, circular list.
`cs_priority`	Denotes the task or HISR priority.
`cs_padding`	This is used to align the Common Services structure on an even boundary. In some ports this field is not used.

Common Services Functions

The following sections provide a brief description of the functions in the Common Services Component (CS). Review of the actual source code is recommended for further information.

CSC_Place_On_List

This function places the specified node at the end of the specified doubly linked circular list.

```
VOID CSC_Place_On_List(CS_NODE **head, CS_NODE *new_node)
```

Functions Called

None

CSC_Priority_Place_On_List

This function places the specified node on the list based upon its priority. The node is placed after all other nodes on the list of equal or greater priority. Note that lower numerical values indicate greater priority.

```
VOID CSC_Priority_Place_On_List(CS_NODE **head, CS_NODE *new_node)
```

Functions Called

None

`CSC_Remove_From_List`

This function removes the specified node from the specified linked list.

```
VOID CSC_Remove_From_List(CS_NODE **head, CS_NODE *node)
```

Functions Called

None

Initialization Component (IN)

The Initialization Component (IN) is responsible for initializing the Nucleus PLUS system. There are typically two parts to the initialization process. The target-dependent portion is initialized first and then each Nucleus PLUS component is initialized. The last initialization routine called is `Application_Initialize`. The user defines its contents. After initialization is complete, control is transferred to the scheduling loop, `TCT_Schedule`. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about initialization.

Initialization Files

The Initialization Component (IN) consists of three files. Each source file of the Initialization Component is defined below.

File	**Description**
`IN_EXTR.H`	All external interfaces to the IN are defined in this file.
`INC.C`	This file contains the core function of the IN. The function that handles the basic system initialization service is defined in this file.
`INT.[S,ASM,SRC]`	This file contains all of the target dependent functions of the IN. A sample initialization file is also provided for user customization purposes.

Initialization Functions

The following sections provide a brief description of the functions in the Initialization Component (IN). Review of the actual source code is recommended for further information.

`INC_Initialize`

This function is the main initialization function of the system. All components are initialized by this function. After system initialization is complete, the `Application_Initialize` routine is called. After all initialization is complete, this function calls `TCT_Schedule` to start scheduling tasks.

```
VOID INC_Initialize(VOID  *first_available_memory)
```

Functions Called

```
Application_Initialize
RLC_Release_Information
LIC_License_Information
ERI_Initialize
HII_Initialize
TCI_Initialize
MBI_Initialize
QUI_Initialize
```

`INT_Initialize`

This is an assembly language function that handles all low-level, target dependent initialization. Once this function is complete, control is transferred to the target independent initialization function, `INC_Initialize`. Responsibilities of this function include the following:

- Setup of necessary processor/system control registers.
- Initialization of the vector table.
- Setup of the system stack pointer.
- Setup of the timer interrupt.
- Calculation of the timer HISR stack and priority.
- Calculation of the first available memory address.
- Transfer of control to `INC_Initialize` to initialize all of the system components.

```
VOID INT_Initialize(void)
```

Functions Called

```
INC_Initialize
```

INT_Vectors_Loaded

This is an assembly language function, which returns the flag that indicates whether or not all the default vectors have been loaded. If it is false, each LISR register also loads the ISR shell into the actual vector table.

```
INT INT_Vectors_Loaded(void)
```

Functions Called

None

INT_Setup_Vector

This is an assembly language function, which sets up the specified vector with the new vector value. The previous vector value is returned to the caller.

```
VOID *INT_Setup_Vector(INT vector, VOID *new)
```

Functions Called

None

Thread Control Component (TC)

The Thread Control Component (TC) is responsible for managing the execution of competing, real-time Nucleus PLUS tasks and High Level Interrupt Routines (HISRs). A Nucleus Plus task is a semi-independent program segment with a dedicated purpose. Most applications have multiple tasks. In order to control the execution process, tasks are usually assigned a priority. Nucleus PLUS priorities range from 0 to 255, where 0 is the highest priority and 255 is the lowest priority. Higher priority tasks are executed before lower priority tasks. Additionally, a lower priority task may be preempted when a higher priority task becomes ready, unless preemption has been disabled. Tasks are always in one of five states: *executing, ready, suspended, terminated or finished.*

A Nucleus PLUS HISR is a scheduled piece of an ISR that is allowed to interact with Nucleus PLUS services. HISRs have priorities ranging from 0 to 2, where 0 is the highest priority. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about the Thread Control Component.

Thread Control Files

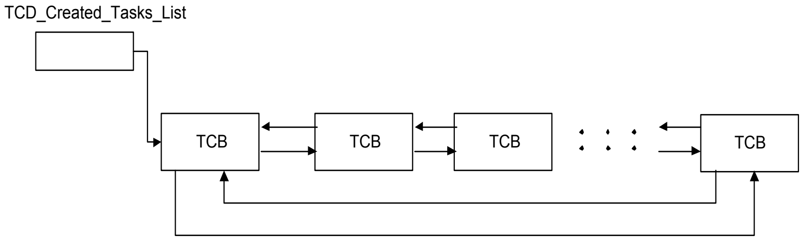
The Thread Control Component (TC) consists of nine files. Each source file of the Thread Control Component is defined below.

File	**Description**
`TC_DEFS.H`	This file contains constants and data structure definitions specific to the TC.
`TC_EXTR.H`	All external interfaces to the TC are defined in this file.
`TCD.C`	Global data structures for the TC are defined in this file.
`TCI.C`	This file contains the initialization function for the TC.
`TCF.C`	This file contains the information gathering functions for the TC.
`TCC.C`	This file contains all of the core functions of the TC. Functions that handle basic create-task and delete-task services are defined in this file.
`TCS.C`	This file contains supplemental functions of the TC. Functions contained in this file are typically used less frequently than the core functions.
`TCCE.C`	This file contains the error checking function interfaces for the core functions defined in `TCC.C`.
`TCSE.C`	This file contains the error checking function interfaces for the supplemental functions defined in `TCS.C`.
`TCT.[S,ASM,SRC]`	This file contains all of the target dependent functions of the TC.

Thread Control Data Structures

Created Tasks List

Nucleus PLUS Tasks may be created and deleted dynamically. The Thread Control Block (TCB) for each created task is kept on a doubly linked, circular list. Newly created tasks are placed at the end of the list, while deleted tasks are completely removed from the list. The head pointer of this list is `TCD_Created_Tasks_List`.



Total Tasks

The total number of currently created Nucleus PLUS tasks is contained in the variable `TCD_Total_Tasks`. The contents of this variable correspond to the number of TCBs on the created list. Manipulation of this variable is also done under the protection of `TCD_List_Protect`.

Created Task List Protection

Nucleus PLUS protects the integrity of the Created Tasks List from competing tasks and/or HISRs. This is done by using an internal protection structure called `TCD_List_Protect`. All task creation and deletion is done under the protection of `TCD_List_Protect`.

Field Declarations

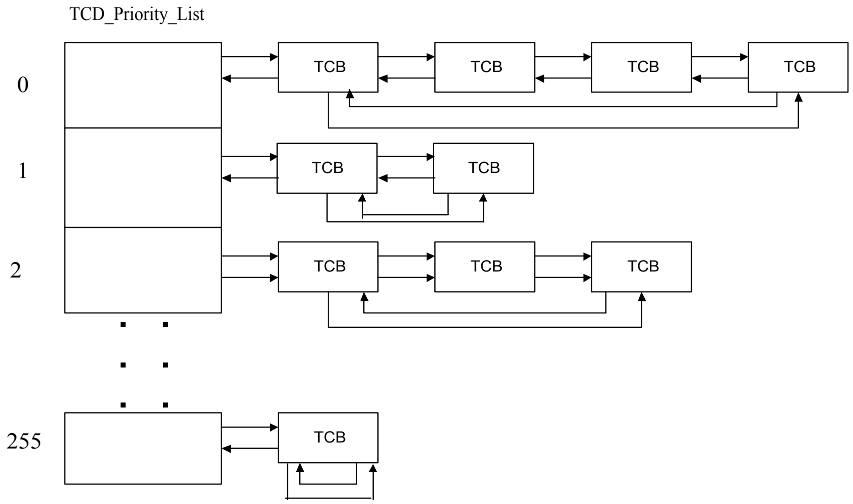
```
TC_TCB        *tc_tcb_pointer
UNSIGNED      tc_thread_waiting
```

Field Summary

Field	Description
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

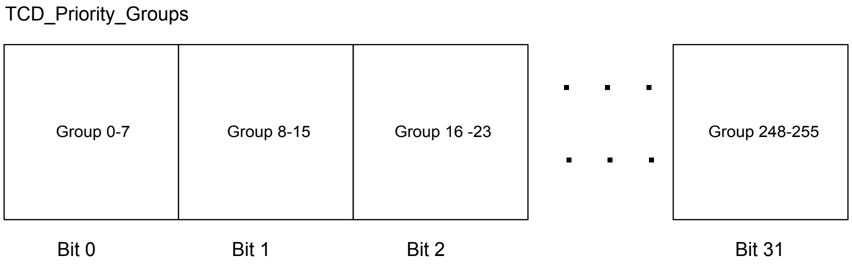
Priority List

`TCD_Priority_List` is an array of TCB pointers. Each element of the array is the head pointer of the list of tasks ready for execution at that priority. If the priority is `NULL`, there are no tasks ready for execution at that priority. The array is indexed by priority.



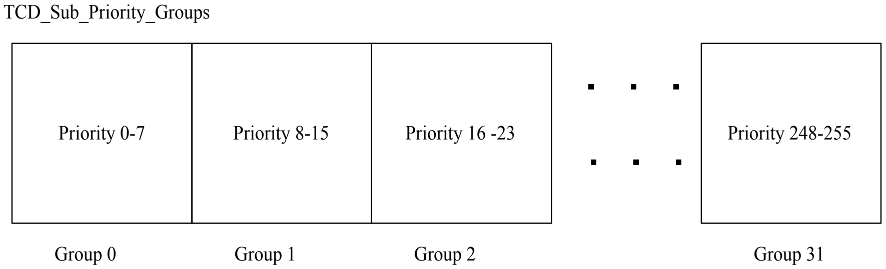
Priority Groups

`TCD_Priority_Groups` is a 32-bit unsigned integer that is used as a bit map. Each bit corresponds to an 8-priority group. For example, if bit 0 is set, at least one task of priority 0 through 8 is ready for execution.



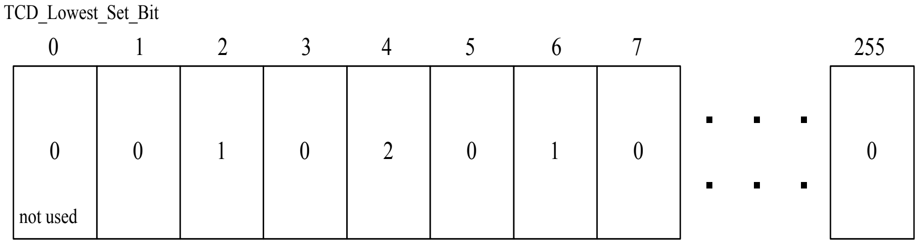
Sub-Priority Groups

Nucleus PLUS uses sub-priorities to determine the exact priority represented in the bit map. `TCD_Sub_Priority_Groups` is an array of sub-priority groups. In this array, index 0 corresponds to priorities 0 through 8. Bit 0 of this element represents priority 0, while bit 7 represents priority 7.



Lowest Bit Set

`TCD_Lowest_Set_Bit` is nothing more than a standard look-up table. The table is indexed by values ranging from 1 to 255. The value at any position in the table contains the lowest set bit for that value. This is used to determine the highest priority task represented in the previously defined bit maps. For example, the lowest bit set for the value of 7 is contained in index 7 of the `TCD_Lowest_Set_Bit` array. In the table below, the value of 7 is shown to have bit 0 set, which is correct.



Execute Task

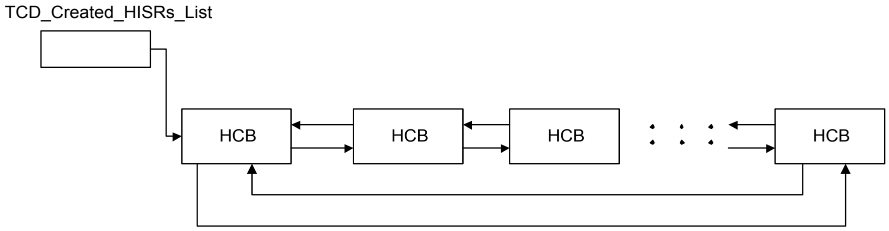
Nucleus PLUS maintains a pointer to the task to execute. This pointer is called `TCD_Execute_Task`. Note that `TCD_Execute_Task` does not necessarily point to the currently executing task. There are several points in the system where this is true. Two common situations arise during task preemption and during task protection conflicts.

Highest Priority

The Nucleus PLUS variable `TCD_Highest_Priority` contains the highest task priority ready for execution. Note that this does not necessarily represent the priority of the currently executing task. This is true if the currently executing task has preemption disabled. If no tasks are executing, this variable is set to the maximum priority.

Created HISRs List

`TCD_Created_HISRs_List` is the head pointer of the list of created High-Level Interrupt Service Routines (HISR). If this pointer is `NU_NULL`, there are no HISRs currently created.



Total HISRs

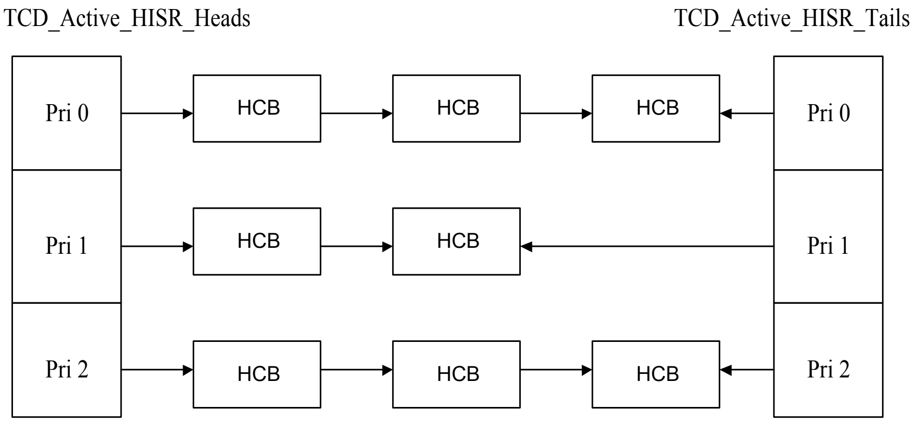
The total number of currently created Nucleus PLUS HISRs is contained in the variable `TCD_Total_HISRs`. The contents of this variable correspond to the number of HCBs on the created list. Manipulation of this variable is also done under the protection of `TCD_HISR_Protect`.

Active HISR Heads

Nucleus PLUS keeps an array of active HISR list head pointers. This list is called `TCD_Active_HISR_Heads`. There are three HISR priorities available. The HISR priority is an index into this table. Priority/index 0 represents the highest priority and priority/index 2 represents the lowest priority.

Active HISR Tails

Nucleus PLUS keeps an array of active HISR list tail pointers. There are three HISR priorities available. The HISR priority is an index into this table. Priority/index 0 represents the highest priority and priority/index 2 represents the lowest priority.



Execute HISR

`TCD_Execute_HISR` contains a pointer to the highest priority HISR to execute. If this pointer is `NU_NULL`, no HISRs are currently activated. Note that the current thread pointer is not always equal to this pointer.

Current Thread

The control block of the currently executing thread is stored in the variable `TCD_Current_Thread`. Therefore, this variable points at either a `TC_TCB` or a `TC_HCB` structure. Except for initialization, this variable is set and cleared in the target dependent portion of this component.

Registered LISRs

Nucleus PLUS maintains a list, called `TCD_Registered_LISRs`, that specifies whether or not a LISR is registered for a given interrupt vector. Values in the list that are indexed by non-zero vectors can be used as an index into the list of LISR pointers. The actual registered LISR can be found by referencing the LISR pointer list at the specified index.

LISR	LISR	LISR	LISR	LISR

LISR Pointers

`TCD_LISR_Pointers` is a list of LISR pointers that indicate the LISR function to call when the interrupt occurs. If the entry is `NULL`, there is no specified LISR function to call, and that entry is available for use.

LISR Function	LISR Function	LISR Function	NULL	LISR Function

Interrupt Count

The number of Interrupt Service Routines (ISRs) currently in progress is contained in the variable `TCD_Interrupt_Count`. If the contents of this variable are zero, then no interrupts are in progress. If the contents are greater than 1, nested interrupts are in progress.

Stack Switched

`TCD_Stack_Switched` is a flag that indicates if the system stack was switched to after the thread's context was saved. Some ports do not use this variable.

System Protect

Nucleus PLUS protects the integrity of the system structures from competing tasks and/or HISRs. This is done by using an internal protection structure called `TCD_System_Protect`. All system creation and deletion is done under the protection of `TCD_System_Protect`.

Field Declarations

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

Field Summary

Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

System Stack

`TCD_System_Stack` contains the system stack base pointer. When the system is idle or in interrupt processing, the system stack is in use. This variable is usually set up during target dependent initialization.

LISR Protect

Nucleus PLUS protects the integrity of LISRs from competing threads and/or HISRs. This is done by using an internal protection structure called `TCD_LISR_Protect`. All LISR creation and deletion is done under the protection of `TCD_LISR_Protect`.

Field Declarations

```
TC_TCB       *tc_tcb_pointer
UNSIGNED     tc_thread_waiting
```

Field Summary

Field	**Descripton**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

HISR Protect

Nucleus PLUS protects the integrity of HISRs from competing threads and/or HISRs. This is done by using an internal protection structure called `TCD_HISR_Protect`. All HISR creation and deletion is done under the protection of `TCD_HISR_Protect`.

Field Declarations

```
TC_TCB       *tc_tcb_pointer
UNSIGNED     tc_thread_waiting
```

Field Summary

Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Interrupt Level

`TCD_Interrupt_Level` is a variable that contains the enabled interrupt posture of the system. In ports, this variable is a Boolean.

Unhandled Interrupts

Nucleus PLUS maintains a variable that contains the last unhandled interrupt vector number in system error conditions. This variable is `TCD_Unhandled_Interrupts`.

Task Control Block

The Task Control Block `TC_TCB` contains the task's priority and other fields necessary for processing task control requests.

Field Declarations

```
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_sub_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
```

Field Summary

Field	**Description**
`tc_created`	This is the link node structure for tasks. It is linked into the created tasks list, which is a doubly linked, circular list.
`tc_id`	This holds the internal task identification of 0x5441534B, which is an equivalent to ASCII TASK.
`tc_name`	This is the user-specified, 8 character name for the task.
`tc_status`	This is the task's current status.
`tc_delayed_suspend`	A flag that indicates if task is suspended.
`tc_priority`	The current priority of the task.
`tc_preemption`	A flag that determines if preemption is enabled.
`tc_scheduled`	This indicates the task's scheduled count.
`tc_cur_time_slice`	This is the value of the current time slice.
`*tc_stack_start`	A pointer to the starting address for the task's stack.
`*tc_stack_end`	A pointer to the ending address for the task's stack.
`*tc_stack_pointer`	This is the task's stack pointer.
`tc_stack_size`	Stores the task's stack size.
`tc_stack_minimum`	The task's minimum allowable stack size.
`*tc_current_protect`	A pointer to the task's current protection structure.
`*tc_saved_stack_ptr`	The task's previous stack pointer.
`tc_time_slice`	The value for the task's current time-slice.
`*tc_ready_previous`	A pointer to the previously ready TCB.
`*tc_ready_next`	A pointer to the TCB that is next on the ready list.
`tc_priority_group`	The current mask of the priority group bit.
`**tc_priority_head`	A pointer to the head of the priority list.
`*tc_sub_prioirty_ptr`	A pointer to the priority sub-group.
`tc_sub_priority`	The current mask of the priority sub-group bit.
`tc_saved_status`	This is the previous task's status.
`tc_signal_active`	A flag indicating if the signal is active or not.
`tc_padding`	This is used to align the task structure on an even boundary. In some ports this field is not used.
`(*tc_entry)(UNSIGNED, VOID *)`	This is the task entry function.
`tc_argc`	An optional task argument.

Field	Description
`*tc_argv`	An optional task argument.
`(*tc_cleanup)(VOID *)`	This is the task clean-up routine.
`*tc_cleanup_info`	This is a pointer to task clean-up information.
`*tc_suspend_protect`	A pointer to the protection structure at the time of task suspension.
`tc_timer_active`	A flag that determines if the timer is active.
`tc_timer_control`	The timer control block.
`tc_signals`	Contains the current signals.
`tc_enabled_signals`	Contains the enabled signals.
`(*tc_signal_handler)`	
`(UNSIGNED)`	This is the signal handling routine.
`tc_system_reserved_1`	This is a reserved word for use by the system.
`tc_system_reserved_2`	This is a reserved word for use by the system.
`tc_system_reserved_3`	This is a reserved word for use by the system.
`tc_app_reserved_1`	This is a reserved word for use by the application.

HISR Control Block

The HISR Control Block `TC_HCB` contains the HISR's priority and other fields necessary for processing task control requests.

Field Declarations

```
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_active_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
```

Field Summary

Field	Description
`tc_created`	This is the link node structure for HISRs. It is linked into the created HISRs list, which is a doubly linked, circular list.
`tc_id`	This holds the internal HISR identification of 0x48495352, which is an equivalent to ASCII HISR.
`tc_name`	This is the user-specified, 8 character name for the HISR.
`tc_not_used_1`	This field is a placeholder and is not used.
`tc_not_used_2`	This field is a placeholder and is not used.
`tc_priority`	This is the priority of the HISR.
`tc_not_used_3`	This field is a placeholder and is not used.
`tc_scheduled`	This is the HISR scheduled count.
`tc_cur_time_slice`	This is the value of the current time slice.
`*tc_stack_start`	A pointer to the starting address for the HISR's stack.
`*tc_stack_end`	A pointer to the ending address for the HISR's stack.
`*tc_stack_pointer`	This is the HISR's stack pointer.
`tc_stack_size`	Stores the HISR's stack size.
`tc_stack_minimum`	The HISR's minimum allowable stack size.
`*tc_current_protect`	A pointer to the HISR's current protection structure.
`*tc_active_next`	A pointer to the next activated HISR.
`tc_activation_count`	The activation counter for the HISR.
`(*tc_entry)(VOID)`	This is the HISR's entry function.
`tc_system_reserved_1`	This is a reserved word for use by the system.
`tc_system_reserved_2`	This is a reserved word for use by the system.
`tc_system_reserved_3`	This is a reserved word for use by the system.
`tc_app_reserved_1`	This is a reserved word for use by the application.

Protection Block

Nucleus PLUS protects the integrity of Nucleus PLUS data structures from competing tasks and/or HISRs. This is done by using an internal protection structure called `TC_Protect`. All Nucleus PLUS data structure creation and deletion, and any list access is done under the protection of `TC_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	**Descripton**
`*tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Thread Control Functions

The following sections provide a brief description of the functions in the Thread Control Component (TC). Review of the actual source code is recommended for further information.

TCC_Create_Task

This function creates a task and then places it on the list of created tasks. All the resources necessary to create the task are supplied to this routine. If specified, the newly created task is started. Otherwise, it is left in a suspended state.

```
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)
```

Functions Called

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

TCC_Delete_Task

This function deletes a task and removes it from the list of created tasks. It is assumed by this function that the task is in a finished or terminated state. Note that this function does not free memory associated with the task's control block or its stack. That is the responsibility of the application.

```
STATUS  TCC_Delete_Task(NU_TASK *task_ptr)
```

Functions Called

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

TCC_Create_HISR

This function creates a High-Level Interrupt Service Routine (HISR) and then places it on the list of created HISRs. All the resources necessary to create the HISR are supplied to this routine. HISRs are always created in a dormant state.

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

Functions Called

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

TCC_Delete_HISR

This function deletes a HISR and removes it from the list of created HISRs. It is assumed by this function that the HISR is in a non-active state. Note that this function does not free memory associated with the HISR's control block or its stack. This is the responsibility of the application.

```
STATUS TCC_Delete_HISR(NU_HISR *hisr_ptr)
```

Functions Called

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

TCC_Reset_Task

This function resets the specified task. Note that a task reset can only be performed on tasks in a finished or terminated state. The task is left in an unconditional suspended state.

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

Functions Called

```
[HIC_Make_History_Entry]
TCT_Build_Task_Stack
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCC_Terminate_Task

This function terminates the specified task. If the task is already terminated, this function does nothing. If the task to terminate is currently suspended, the specified cleanup routine is also invoked to clean up suspension data structures.

```
STATUS  TCC_Terminate_Task(NU_TASK *task_ptr)
```

Functions Called

```
Cleanup routine
[HIC_Make_History_Entry]
TCC_Suspend_Task
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
TCT_Unprotect_Specific
TMC_Stop_Task_Timer
```

TCC_Resume_Task

This function resumes a previously suspended task. The task must currently be suspended for the same reason indicated by this request. If the task resumed is of higher priority than the calling task and the current task is preemptable, this function returns a value of `NU_TRUE`. If no preemption is required, a `NU_FALSE` is returned.

```
STATUS  TCC_Resume_Task(NU_TASK *task_ptr,
                        OPTION suspend_type)
```

Functions Called

```
[TCT_Check_Stack]
TCT_Set_Current_Protect
TCT_Set_Execute_Task
TMC_Stop_Task_Timer
```

TCC_Resume_Service

This function provides a suitable interface to the actual service to resume a task.

```
STATUS TCC_Resume_Service(NU_TASK *task_ptr)
```

Functions Called

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

TCC_Suspend_Task

This function suspends the specified task. If the specified task is the calling task, control is transferred back to the system.

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

Functions Called

```
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

This function provides a suitable interface to the actual service to suspend a task.

```
STATUS TCC_Suspend_Service(NU_TASK *task_ptr)
```

Functions Called

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

TCC_Task_Timeout

This function processes task suspension timeout conditions. Note that task sleep requests are also considered a timeout condition.

```
VOID  TCC_Task_Timeout(NU_TASK *task_ptr)
```

Functions Called

```
Caller’s cleanup function
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Protect
TCT_Set_Current_Protect
TCT_Unprotect
TCT_Unprotect_Specific
```

TCC_Task_Sleep

This function provides task sleep suspensions. Its primary purpose is to interface with the actual task suspension function.

```
VOID  TCC_Task_Sleep(UNSIGNED ticks)
```

Functions Called

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

TCC_Relinquish

This function moves the calling task to the end of other tasks at the same priority level. The calling task does not execute again until all the other tasks of the same priority get a chance to execute.

```
VOID TCC_Relinquish(VOID)
```

Functions Called

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

TCC_Time_Slice

This function moves the specified task to the end of the other tasks at the same priority level. If the specified task is no longer ready, this request is ignored.

```
VOID  TCC_Time_Slice(NU_TASK *task_ptr)
```

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Set_Execute_Task
TCT_Unprotect
```

TCC_Current_Task_Pointer

This function returns the pointer of the currently executing task. If the current thread is not a task thread, a `NU_NULL` is returned.

```
NU_TASK *TCC_Current_Task_Pointer(VOID)
```

Functions Called

None

TCC_Current_HISR_Pointer

This function returns the pointer of the currently executing HISR. If the current thread is not a HISR thread, a `NU_NULL` is returned.

```
NU_HISR *TCC_Current_HISR_Pointer(VOID)
```

Functions Called

None

TCC_Task_Shell

This function is a shell from which all application tasks are initially executed. The shell causes the task to finish when control is returned from the application task. Also, the shell passes `argc` and `argv` arguments to the task's entry function.

```
VOID TCC_Task_Shell(VOID)
```

Functions Called

```
Task Entry Function
TCC_Suspend_Task
TCT_Protect
```

TCC_Signal_Shell

This function processes signals by calling the task-supplied signal handling function. When signal handling is completed, the task is placed in the appropriate state.

```
VOID TCC_Signal_Shell(VOID)
```

Functions Called

```
task’s signal handling routine
[TCT_Check_Stack]
TCT_Signal_Exit
TCT_Protect
TCT_Set_Execute_Task
TCT_Unprotect
```

TCC_Dispatch_LISR

This function dispatches the LISR associated with the specified interrupt vector. Note that this function is called during the interrupt thread.

```
VOID TCC_Dispatch_LISR(INT vector)
```

Functions Called

```
application LISR
ERC_System_Error
```

TCC_Register_LISR

This function registers the supplied LISR with the supplied vector number. If the supplied LISR is `NU_NULL`, the supplied vector is de-registered. The previously registered LISR is returned to the caller, along with the completion status.

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

Functions Called

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

TCCE_Create_Task

This function performs error checking on the parameters supplied to the create task function.

```
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)
```

Functions Called

```
TCC_Create_Task
```

TCCE_Create_HISR

This function performs error checking on the parameters supplied to the create HISR function.

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

Functions Called

`TCC_Create_HISR`

TCCE_Delete_HISR

This function performs error checking on the parameters supplied to the delete HISR function.

```
STATUS  TCCE_Delete_HISR (NU_HISR *hisr_ptr)
```

Functions Called

`TCC_Delete_HISR`

TCCE_Delete_Task

This function performs error checking on the parameters supplied to the delete task function.

```
STATUS  TCCE_Delete_Task(NU_TASK *task_ptr)
```

Functions Called

TCC_Delete_Task

TCCE_Reset_Task

This function performs error checking on the parameters supplied to the reset task function.

```
STATUS TCCE_Reset_Task(NU_TASK* task_ptr,UNSIGNED argc,
                       VOID *argv)
```

Functions Called

TCC_Reset_Task

TCCE_Terminate_Task

This function performs error checking on the parameters supplied to the terminate task function.

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

Functions Called

TCC_Terminate_Task

TCCE_Resume_Service

This function performs error checking on the parameters supplied to the resume task function.

```
STATUS  TCCE_Resume_Service(NU_TASK *task_ptr)
```

Functions Called

```
TCCE_Validate_Resume
TCC_Resume_Service
```

TCCE_Suspend_Service

This function performs error checking on the suspend service.

```
STATUS  TCCE_Suspend_Service(NU_TASK *task_ptr)
```

Functions Called

```
TCC_Suspend_Service
```

TCCE_Relinquish

This function performs error checking for the relinquish function. If the current thread is not a task, this request is ignored.

```
VOID  TCCE_Relinquish(VOID)
```

Functions Called

```
TCC_Relinquish
```

TCCE_Task_Sleep

This function performs error checking for the task sleep function. If the current thread is not a task, this request is ignored.

```
VOID  TCCE_Task_Sleep(UNSIGNED ticks)
```

Functions Called

`TCC_Task_Sleep`

TCCE_Suspend_Error

This function checks for a suspend request error. Suspension requests are only allowed from task threads. A suspend request from any other thread is an error.

```
INT TCCE_Suspend_Error(VOID)
```

Functions Called

None

TCCE_Activate_HISR

This function performs error checking on the parameters supplied to the activate HISR function.

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

Functions Called

`TCT_Activate_HISR`

TCCE_Validate_Resume

This function validates the resume service and resume driver calls with scheduling protection around the examination of the task status.

```
STATUS  TCCE_Validate_Resume(OPTION resume_type,
                             NU_TASK *task_ptr)
```

Functions Called

```
TCT_Set_Current_Protect
TCT_System_Protect
TCT_System_Unprotect
TCT_Unprotect
```

TCF_Established_Tasks

Returns the current number of established tasks. Tasks previously deleted are no longer considered established.

```
UNSIGNED  TCF_Established_Tasks(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

TCF_Established_HISRs

Returns the current number of established HISRs. HISRs previously deleted are no longer considered established.

```
UNSIGNED  TCF_Established_HISRs(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

TCF_Task_Pointers

Builds a list of task pointers, starting at the specified location. The number of task pointers placed in the list is equivalent to the total number of tasks or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TCF_HISR_Pointers

Builds a list of HISR pointers, starting at the specified location. The number of HISR pointers placed in the list is equivalent to the total number of HISRs or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCF_Task_Information

Returns information about the specified task. However, if the supplied task pointer is invalid, the function simply returns an error status.

```
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)
```

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TCF_HISR_Information

Returns information about the specified HISR. However, if the supplied HISR pointer is invalid, the function simply returns an error status.

```
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)
```

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TCI_Initialize

This function initializes the data structures that control the operation of the TC component. The system is initialized as idle.

```
VOID  TCI_Initialize(VOID)
```

Functions Called

None

TCS_Change_Priority

This function changes the priority of the specified task. The priority of a suspended or a ready task can be changed. If the new priority requires a context switch, control is transferred back to the system.

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

Functions Called

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

TCS_Change_Preemption

This function changes the preemption posture of the calling task. Preemption for a task may be enabled or disabled. If it is disabled, the task runs until it suspends or relinquishes. If a preemption is pending, a call to this function to enable preemption causes a context switch.

```
OPTION   TCS_Change_Preemption(OPTION preempt)
```

Functions Called

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

TCS_Change_Time_Slice

This function changes the time slice of the specified task. A time slice value of 0 disables time slicing.

```
UNSIGNED     TCS_Change_Time_Slice(NU_TASK *task_ptr,
                                   UNSIGNED time_slice)
```

Functions Called

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

TCS_Control_Signals

This function enables the specified signals and returns the previous enable signal value back to the caller. If a newly enabled signal is present and a signal handler is registered, signal handling is started.

```
UNSIGNED  TCS_Control_Signals(UNSIGNED enable_signal_mask)
```

Functions Called

```
[HIC_Make_History_Entry]
TCC_Signal_Shell
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCS_Receive_Signals

This function returns the current signals back to the caller. Note that the signals are cleared automatically.

```
UNSIGNED  TCS_Receive_Signals(VOID)
```

Functions Called

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

TCS_Register_Signal_Handler

This function registers a signal handler for the calling task. Note that if an enabled signal is present and this is the first registered signal handler call, the signal is processed immediately.

```
STATUS  TCS_Register_Signal_Handler(VOID (*signal_handler)
                                    (UNSIGNED))
```

Functions Called

```
[HIC_Make_History_Entry]
TCC_Signal_Shell
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TCS_Send_Signals

This function sends the specified task the specified signals. If enabled, the specified task is setup in order to process the signals.

```
STATUS  TCS_Send_Signals(NU_TASK *task_ptr, UNSIGNED signals)
```

Functions Called

```
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Signal_Shell
TCT_Build_Signal_Frame
[TCT_Check_Stack]
```

TCSE_Change_Priority

This function performs error checking for the change priority service. If an error is detected, this service is ignored and the requested priority is returned.

```
OPTION  TCSE_Change_Priority(NU_TASK *task_ptr,
                             OPTION new_priority)
```

Functions Called

`TCS_Change_Priority`

TCSE_Change_Preemption

This function performs error checking on the change preemption service. If the current thread is not a task thread, this request is ignored.

```
OPTION   TCSE_Change_Preemption(OPTION preempt)
```

Functions Called

`TCS_Change_Preemption`

TCSE_Change_Time_Slice

This function performs error checking on the change time slice service. If the specified task pointer is invalid, this request is ignored.

```
UNSIGNED     TCSE_Change_Time_Slice(NU_TASK *task_ptr,
                                    UNSIGNED time_slice)
```

Functions Called

`TCS_Change_Time_Slice`

TCSE_Control_Signals

This function checks to see if the call is being made from a non-task thread. If so, the request is simply ignored.

```
UNSIGNED  TCSE_Control_Signals(UNSIGNED enable_signal_mask)
```

Functions Called

`TCS_Control_Signals`

TCSE_Receive_Signals

This function determines whether or not the call is being made from a task thread of execution. If not, the call is ignored.

```
UNSIGNED  TCSE_Receive_Signals(VOID)
```

Functions Called

`TCS_Receive_Signals`

TCSE_Register_Signal_Handler

This function determines whether or not the caller is a task. If the caller is not a task and/or if the supplied signal handling function pointer is `NULL,` an appropriate error status is returned.

```
STATUS  TCSE_Register_Signal_Handler(VOID (*signal_handler)
                                        (UNSIGNED))
```

Functions Called

`TCS_Register_Signal_Handler`

TCSE_Send_Signals

This function checks for an invalid task. If an invalid task is selected an error is returned.

```
STATUS TCSE_Send_Signals(NU_TASK *task_ptr, UNSIGNED signals)
```

Functions Called

`TCS_Send_Signals`

TCT_Control_Interrupts

This is an assembly language function that enables and disables interrupts as specified by the caller. Interrupts disabled by this call are left disabled until another call is made to enable them.

```
INT  TCT_Control_Interrupts(new_level)
```

Functions Called

None

TCT_Local_Control_Interrupts

This is an assembly language function, which enables and disables interrupts as specified by the caller.

```
INT  TCT_Local_Control_Interrupts(new_level)
```

Functions Called

None

TCT_Restore_Interrupts

This is an assembly language function that restores interrupts to that specified in the global `TCD_Interrupt_Level` variable.

```
VOID  TCT_Restore_Interrupts(VOID)
```

Functions Called

None

TCT_Build_Task_Stack

This is an assembly language function, which builds an initial stack frame for a task. The initial stack contains information concerning initial values of registers and the task's point of entry. Furthermore, the initial stack frame is in the same form as an interrupt stack frame.

```
VOID  TCT_Build_Task_Stack(TC_TCB *task)
```

Functions Called

None

TCT_Build_HISR_Stack

This is an assembly language function. It builds an HISR stack frame that allows quick scheduling of the HISR.

```
VOID  TCT_Build_HISR_Stack(TC_HCB *hisr)
```

Functions Called

None

TCT_Build_Signal_Frame

This is an assembly language function that builds a frame on top of the task's stack. This causes the task's signal handler to execute the next time the task is executed.

```
VOID  TCT_Build_Signal_Frame(TC_TCB *task)
```

Functions Called

None

TCT_Check_Stack

This assembly language function checks the current stack for overflow conditions. Additionally, this function keeps track of the minimum amount of stack space for the calling thread and returns the current available stack space.

```
UNSIGNED  TCT_Check_Stack(void)
```

Functions Called

`ERC_System_Error`

TCT_Schedule

This assembly language function waits for a thread to become ready. Once a thread is ready, this function initiates a transfer of control to that thread.

```
VOID  TCT_Schedule(void)
```

Functions Called

`TCT_Control_To_Thread`

TCT_Control_To_Thread

This is an assembly language function. It transfers control to the specified thread. Each time control is transferred to a thread, its scheduled counter is incremented. Additionally, time slicing for task threads is enabled in this routine. The `TCD_Current_Thread` pointer is set up by this function.

```
VOID  TCT_Control_To_Thread(TC_TCB *thread)
```

Functions Called

None

TCT_Control_To_System

This is an assembly language function that returns control from a thread to the system. Note that this service is called in a solicited manner, i.e., it is not called from an interrupt thread. Registers required by the compiler to be preserved across function boundaries are saved by this routine. Note that this is usually a subset of the total number of available registers.

```
VOID  TCT_Control_To_System(void)
```

Functions Called

`TCT_Schedule`

TCT_Signal_Exit

This assembly language function exits from a signal handler. The primary purpose of this function is to clear the scheduler protection and switch the stack pointer back to the normal task's stack pointer.

```
VOID  TCT_Control_To_System(void)
```

Functions Called

`TCT_Schedule`

TCT_Current_Thread

This is an assembly language function, which returns the current thread pointer.

```
VOID  *TCT_Current_Thread(void)
```

Functions Called

None

TCT_Set_Execute_Task

This assembly language function sets the current task to execute the variable under protection, which is against interrupts.

```
VOID  TCT_Set_Execute_Task(TC_TCB *task)
```

Functions Called

None

`TCT_Protect`

This assembly language function protects against multiple thread access. If another thread (TASK or HISR) owns the requested protection structure, then that thread will be scheduled to run until it releases the protection structure. At that time, the thread is suspended, and control is returned to the thread doing the `TCT_Protect` call. This prevents lower priority tasks from blocking higher priority threads trying to obtain a protection structure.

```
VOID  TCT_Protect(TC_PROTECT *protect)
```

Functions Called

None

`TCT_Unprotect`

This is an assembly language function that releases protection of the currently active thread. If the caller is not an active thread, then this request is ignored.

```
VOID  TCT_Unprotect(void)
```

Functions Called

None

TCT_Unprotect_Specific

This assembly language function releases a specific protection structure.

```
VOID  TCT_Unprotect_Specific(TC_PROTECT *protect)
```

Functions Called

None

TCT_Set_Current_Protect

This is an assembly language function, which sets the current protection field of the current thread's control block to the specified protection pointer.

```
VOID  TCT_Set_Current_Protect(TC_PROTECT *protect)
```

Functions Called

None

TCT_Protect_Switch

This is an assembly language function that waits until a specific task no longer has any protection associated with it. This is necessary since tasks cannot be suspended or terminated unless they have released all of their protection.

```
VOID  TCT_Protect_Switch(VOID *thread)
```

Functions Called

None

TCT_Schedule_Protected

This assembly language function saves the minimal context of a thread. Then it directly schedules another thread that has protection over the thread that called this routine.

```
VOID  TCT_Schedule_Protected(VOID *thread)
```

Functions Called

`TCT_Control_To_Thread`

TCT_Interrupt_Context_Save

This is an assembly language function that saves the interrupted thread's context. Nested interrupts are also supported. If a task or HISR thread was interrupted, the stack pointer is switched to the system stack after the context is saved.

```
VOID  TCT_Interrupt_Context_Save(vector)
```

Functions Called

None

TCT_Interrupt_Context_Restore

This assembly language function restores the interrupt context if a nested interrupt condition is present. Otherwise, this routine transfers control to the scheduling function.

```
VOID  TCT_Interrupt_Context_Restore(void)
```

Functions Called

`TCT_Schedule`

TCT_Activate_HISR

This is an assembly language function, which activates the specified HISR. If the HISR is already activated, the HISR's activation count is simply incremented. Otherwise, the HISR is placed on the appropriate HISR priority list in preparation for execution.

```
STATUS  TCT_Activate_HISR(TC_HCB *hisr)
```

Functions Called

None

TCT_HISR_Shell

This is an assembly language function that is the execution shell of each and every HISR. If the HISR has completed its processing, this shell routine exits back to the system. Otherwise, it sequentially calls the HISR routine until the activation count goes to zero.

```
VOID TCT_HISR_Shell(void)
```

Functions Called

`hisr -> tc_entry`

TCT_Check_For_Preemption

This is an assembly language function that checks to see if some other interrupt condition occurred while a minimal ISR was in process. If so, a full context save and restore is performed in order to process the preemption. Otherwise, control is transferred back to the point of interrupt.

```
VOID TCT_Check_For_Preemption(void)
```

Functions Called

```
TCT_Interrupt_Context_Save
TCT_Interrupt_Context_Restore
```

Timer Component (TM)

The Timer Component (TM) is responsible for processing all Nucleus PLUS timer facilities. The basic unit of time for a Nucleus PLUS timer is a tick, which corresponds to a single hardware timer interrupt. Nucleus PLUS timers can be applied at the application level to execute a particular routine at timer expiration. Timers can also apply to tasks and are used to provide task sleeping and service call suspension timeouts. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about timers.

Timer Files

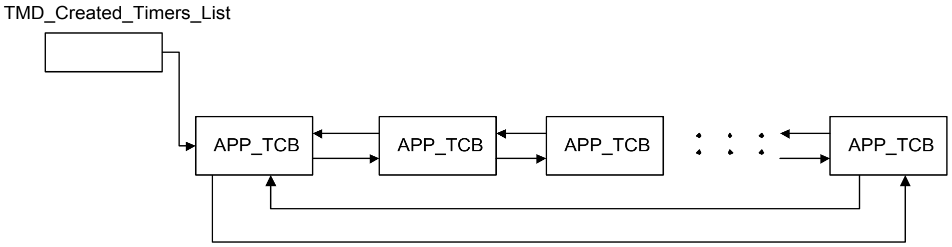
The Timer Component (TM) consists of nine files. Each source file of the Timer Component is defined below.

File	Description
`TM_DEFS.H`	This file contains constants and data structure definitions specific to the TM.
`TM_EXTR.H`	All external interfaces to the TM are defined in this file.
`TMD.C`	Global data structures for the TM are defined in this file.
`TMI.C`	This file contains the initialization function for the TM.
`TMF.C`	This file contains the information gathering functions for the TM.
`TMC.C`	This file contains all of the core functions of the TM. Functions that handle basic start-timer and stop-timer services are defined in this file.
`TMS.C`	This file contains supplemental functions of the TM. Functions contained in this file are typically used less frequently than the core functions.
`TMSE.C`	This file contains the error checking function interfaces for the supplemental functions defined in `TMS.C`.
`TMT.[S,ASM,SRC]`	This file contains all of the target dependent functions of the `TM`.

Timer Data Structures

Created Timers List

Nucleus PLUS application timers may be created and deleted dynamically. The Timer Control Block (`APP_TCB`) for each created timer is kept on a doubly linked, circular list. Newly created timers are placed at the end of the list, while deleted timers are completely removed from the list. The head pointer of this list is `TMD_Created_Timers_List`. The Created Timers List is used exclusively for application timers.



Created Timer List Protection

Nucleus PLUS protects the integrity of the Created Timers List from competing tasks and/or HISRs. This is done by using an internal protection structure called `TMD_Created_List_Protect`. All timer creation and deletion is done under the protection of `TMD_Created_List_Protect`.

Field Declarations

```
TC_TCB       *tc_tcb_pointer
UNSIGNED     tc_thread_waiting
```

Field Summary

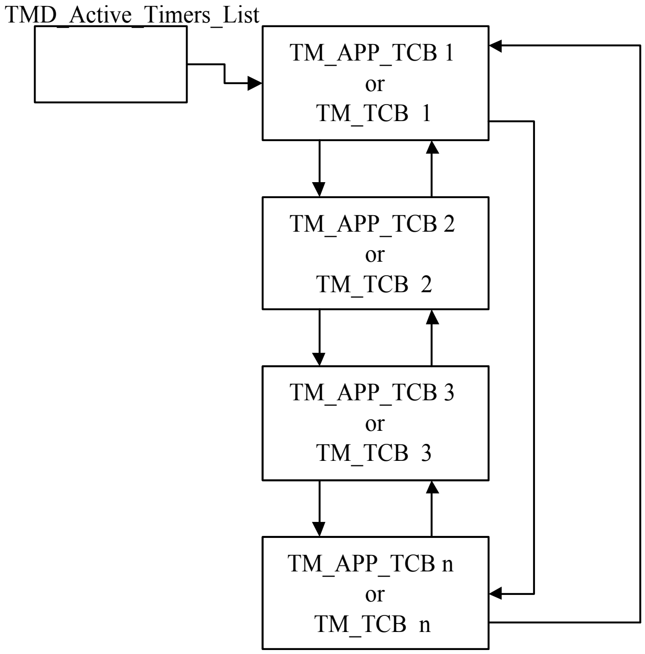
Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Timers

The total number of currently created Nucleus PLUS timers is contained in the variable `TMD_Total_Timers`. The contents of this variable correspond to the number of TCBs on the created list. Manipulation of this variable is also done under the protection of `TMD_Created_List_Protect`.

Active Timers List

Nucleus PLUS active timers are maintained on a doubly linked, circular list. `TMD_Active_Timers_List` is the head pointer to this list. If this pointer is `NULL`, there are no timers active. The timer list supports both application timers and task timers. Task timer structures reside in the task's TCB. The timer list is maintained in order of expiration time. The remaining time is in delta expirations, not absolute time. This is done in order to avoid adjusting the entire list on every timer interrupt. A timer with a remaining time of zero is considered to be expired.



Active List Busy

Nucleus PLUS protects the integrity of the Active Timers List from competing tasks and/or HISRs. This is done by using a protection flag called `TMD_Active_List_Busy`. All active timer list additions and deletions are done under the protection of `TMD_Active_List_Busy`.

System Clock

Nucleus PLUS maintains a continually incrementing system clock called `TMD_System_Clock`. The clock is incremented by one each timer interrupt.

Timer Start

Nucleus PLUS stores the value of the last timer request in the variable `TMD_Timer_Start`.

Timer

The variable `TMD_Timer` is a countdown timer that is used to represent the smallest active timer value in the system. When a timer expires, this variable has a value of zero.

Timer State

`TMD_Timer_State` indicates the state of the timer variable. If the state is active, the timer counter is decremented. If the state is expired, the timer HISR and timer task are initiated to process the expiration. If the state indicates that the timer is not active, the timer counter is ignored.

Time Slice

Nucleus PLUS uses the variable `TMD_Time_Slice` as a countdown value for the currently executing task's time slice. Time slice processing is started when the value of `TMD_Time_Slice` becomes zero.

Time Slice Task

`TMD_Time_Slice_Task` is a pointer to the TCB of the task to time-slice. This pointer is setup in the timer interrupt when a time-slice timer has expired.

Time Slice State

Nucleus PLUS indicates the state of the time slice variable using `TMD_Time_Slice_State`. If the state is active, the time slice counter is decremented. If the state is expired, the timer HISR is initiated to process the expiration. If the state indicates that the time slice is not-active, the time slice counter is ignored.

HISR

`TMD_HISR` is the timer HISR's control block.

HISR Stack Pointer

`TMD_HISR_Stack_Ptr` points to the memory area reserved for the timer HISR. Note that this is setup in `INT_Initialize`.

HISR Stack Size

Nucleus PLUS determines the size of the allocated timer HISR stack with the variable `TMD_HISR_Stack_Size`. Note that this is setup in `INT_Initialize`.

HISR Priority

`TMD_HISR_Priority` indicates the priority of the timer HISR. Priorities range from 0 to 2, where priority 0 is the highest. Note that this is also initialized in `INT_Initialize`.

Timer Control Block

The Timer Control Block `TM_TCB` contains the remaining time and other fields necessary for processing timer requests.

Field Declarations

```
INT                        tm_timer_type
UNSIGNED                   tm_remaining_time
VOID                       *tm_information
struct TM_TCB_STRUCT       *tm_next_timer
struct TM_TCB_STRUCT       *tm_previous_timer
```

Field Summary

Field	**Description**
`tm_timer_type`	Indicates if the timer is for an application or a task.
`tm_remaining_time`	This stores the amount of time remaining after expiration of the previous timer occurs. The true expiration is the sum of all previous timer's remaining time on the active list
`*tm_information`	A pointer to general information about the timer.
`*tm_next_timer`	A pointer to the next timer in the list.
`*tm_previous_timer`	A pointer to the previous timer in the list.

Application's Timer Control Block

The Application's Timer Control Block `TM_APP_TCB` contains a pointer to the timer expiration routine and other fields necessary for processing application timer requests.

Field Declarations

```
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
```

Field Summary

Field	**Descripton**
`tm_created`	This is the link node structure for timers. It is linked into the created timers list, which is a doubly linked, circular list.
`tm_id`	This holds the internal timer identification of 0x54494D45, which is an equivalent to ASCII TIME.
`tm_name`	This is the user-specified, 8 character name for the timer.
`*tm_expiration_routine`	A pointer to the timer expiration function.
`tm_expiration_id`	This is the name of the expiration.
`tm_enabled`	A flag that determines if the timer is enabled.
`tm_expirations`	This stores the number of timer expirations.
`tm_initial_time`	Stores the initial starting time for the timer.
`tm_reschedule_time`	Stores the reschedule time for the timer.
`tm_actual_timer`	The actual timer TCB.

Timer Functions

The following sections provide a brief description of the functions in the Timer Component (TM). Review of the actual source code is recommended for further information.

TMC_Init_Task_Timer

This function is responsible for initializing the supplied task timer.

```
VOID  TMC_Init_Task_Timer(TM_TCB *timer, VOID *information)
```

Functions Called

None

TMC_Start_Task_Timer

This function is responsible for starting a task timer. Note that there are some special protection considerations since this function is called from the task control component.

```
VOID  TMC_Start_Task_Timer(TM_TCB *timer, UNSIGNED time)
```

Functions Called

`TMC_Start_Timer`

TMC_Stop_Task_Timer

This function is responsible for stopping a task timer. Note that there are some special protection considerations since this function is called from the task control component.

```
VOID  TMC_Stop_Task_Timer(TM_TCB *timer)
```

Functions Called

```
TMC_Stop_Timer
```

TMC_Start_Timer

This function is responsible for starting both application and task timers.

```
VOID  TMC_Start_Timer(TM_TCB *timer, UNSIGNED time)
```

Functions Called

```
TMT_Read_Timer
TMT_Adjust_Timer
TMT_Enable_Timer
```

TMC_Stop_Timer

This function is responsible for stopping both application and task timers.

```
VOID  TMC_Stop_Timer(TM_TCB *timer)
```

Functions Called

```
TMT_Disable_Timer
```

TMC_Timer_HISR

This function is responsible for High-Level interrupt processing of a timer expiration. If an application timer has expired, the timer expiration function is called. Otherwise, if the time-slice timer has expired, time-slice processing is invoked.

```
VOID  TMC_Timer_HISR(VOID)
```

Functions Called

```
TCC_Time_Slice
TMC_Timer_Expiration
TMT_Retrieve_TS_Task
```

TMC_Timer_Expiration

This function is responsible for processing all task timer expirations. This includes application timers and basic task timers that are used for task sleeping and timeouts.

```
VOID  TMC_Timer_Expiration(VOID)
```

Functions Called

```
expiration_function
TCC_Task_Timeout
TCT_System_Protect
TCT_Unprotect
```

TMF_Established_Timers

This function returns the current number of established timers. Timers previously deleted are no longer considered established.

```
UNSIGNED  TMF_Established_Timers(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

TMF_Get_Remaining_Time

This function retrieves the remaining time before the expiration of the specified timer.

```
UNSIGNED TMF_Get_Remaining_Time(NU_TIMER *timer,
                                UNSIGNED *remaining_time)
```

Functions Called

```
TCT_Protect
TCT_Unprotect
```

TMF_Timer_Pointers

Builds a list of timer pointers, starting at the specified location. The number of timer pointers placed in the list is equivalent to the total number of timers or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

TMF_Timer_Information

This function returns information about the specified timer. However, if the supplied timer pointer is invalid, the function simply returns an error status.

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

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

TMI_Initialize

This function initializes the data structures that control the operation of the Timer Management component. There are no application timers created initially.

```
VOID  TMI_Initialize(VOID)
```

Functions Called

```
ERC_System_Error
TCC_Create_HISR
TCCE_Create_HISR
```

TMS_Create_Timer

This function creates an application timer and places it on the list of created timers. The timer is activated if designated by the enable parameter.

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

Functions Called

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

TMS_Delete_Timer

This function deletes an application timer and removes it from the list of created timers.

```
STATUS  TMS_Delete_Timer(NU_TIMER *timer_ptr)
```

Functions Called

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

TMS_Reset_Timer

This function resets the specified application timer. Note that the timer must be in a disabled state prior to this call. The timer is activated after it is reset if the enable parameter specifies automatic activation.

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

Functions Called

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

TMS_Control_Timer

This function either enables or disables the specified timer. If the timer is already in the desired state, simply leave it alone.

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

Functions Called

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

TMSE_Create_Timer

This function performs error checking on the parameters supplied to the create timer function.

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

Functions Called

```
TMS_Create_Timer
```

TMSE_Delete_Timer

This function performs error checking on the parameters supplied to the delete timer function.

```
STATUS  TMSE_Delete_Timer(NU_TIMER *timer_ptr)
```

Functions Called

`TMS_Delete_Timer`

TMSE_Reset_Timer

This function performs error checking on the parameters supplied to the reset timer function.

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

Functions Called

`TMS_Reset_Timer`

TMSE_Control_Timer

This function performs error checking on the parameters supplied to the control timer function.

```
STATUS  TMSE_Control_Timer(NU_TIMER *timer_ptr, OPTION enable)
```

Functions Called

`TMS_Control_Timer`

TMT_Set_Clock

This assembly language function sets the system clock to the specified value.

```
VOID  TMT_Set_Clock(UNSIGNED new_value)
```

Functions Called

None

TMT_Retrieve_Clock

This is an assembly language function that returns the current value of the system clock.

```
UNSIGNED  TMT_Retrieve_Clock(void)
```

Functions Called

None

TMT_Read_Timer

This is an assembly language function, which returns the current value of the countdown timer.

```
UNSIGNED  TMT_Read_Timer(void)
```

Functions Called

None

TMT_Enable_Timer

This is an assembly language function that enables the countdown timer with the specified value.

```
VOID  TMT_Enable_Timer(UNSIGNED time)
```

Functions Called

None

TMT_Adjust_Timer

This is an assembly language function that adjusts the countdown timer with the specified value - if the new value is less than the current.

```
VOID  TMT_Adjust_Timer(UNSIGNED time)
```

Functions Called

None

TMT_Disable_Timer

This assembly language function disables the countdown timer.

```
VOID  TMT_Disable_Timer(void)
```

Functions Called

None

TMT_Retrieve_TS_Task

This is an assembly language function that returns the time-sliced task pointer.

```
NU_TASK  *TMT_Retrieve_TS_Task(VOID)
```

Functions Called

None

TMT_Timer_Interrupt

This assembly language function processes the actual hardware interrupt. Processing includes updating the system clock and the countdown timer and the time-slice timer. If one or both of the timers expire, the timer HISR is activated.

```
VOID  TMT_Timer_Interrupt(void)
```

Functions Called

```
TCT_Activate_HISR
TCT_Interrupt_Context_Save
TCT_Interrupt_Context_Restore
```

Mailbox Component (MB)

The Mailbox Component (MB) is responsible for processing all Nucleus PLUS mailbox facilities. A Nucleus PLUS mailbox is a low overhead mechanism for inter-task communication. Each mailbox is capable of holding one message. A mailbox message consists of four 32-bit words. Tasks may suspend while waiting for a message from an empty mailbox. Conversely, tasks may suspend while trying to send to a mailbox that already contains a message. Mailboxes are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about mailboxes.

Mailbox Files

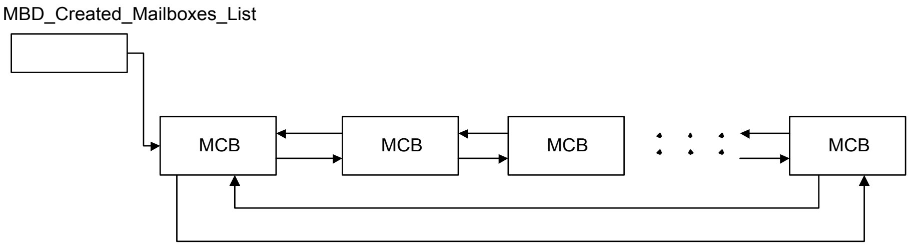
The Mailbox Component (MB) consists of nine files. Each source file of the Mailbox Component is defined below.

File	**Description**
`MB_DEFS.H`	This file contains constants and data structure definitions specific to the MB.
`MB_EXTR.H`	All external interfaces to the MB are defined in this file.
`MBD.C`	Global data structures for the MB are defined in this file.
`MBI.C`	This file contains the initialization function for the MB.
`MBF.C`	This file contains the information gathering functions for the MB.
`MBC.C`	This file contains all of the core functions of the MB. Functions that handle basic send-to-mailbox and receive-from-mailbox services are defined in this file.
`MBS.C`	This file contains supplemental functions of the MB. Functions contained in this file are typically used less frequently than the core functions
`MBCE.C`	This file contains the error checking function interfaces for the core functions defined in `MBC.C`.
`MBSE.C`	This file contains the error checking function interfaces for the supplemental functions defined in `MBS.C`.

Mailbox Data Structures

Created Mailbox List

Nucleus PLUS mailboxes may be created and deleted dynamically. The Mailbox Control Block (MCB) for each created mailbox is kept on a doubly linked, circular list. Newly created mailboxes are placed at the end of the list, while deleted mailboxes are completely removed from the list. The head pointer of this list is `MBD_Created_Mailboxes_List`.



Created Mailbox List Protection

Nucleus PLUS protects the integrity of the Created Mailboxes List from competing tasks and/or HISRs. This is done by using an internal protection structure called `MBD_List_Protect`. All mailbox creation and deletion is done under the protection of `MBD_List_Protect`.

Field Declarations

```
TC_TCB              *tc_tcb_pointer
UNSIGNED            tc_thread_waiting
```

Field Summary

Field	Description
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Mailboxes

The total number of currently created Nucleus PLUS mailboxes is contained in the variable `MBD_Total_Mailboxes`. The content of this variable corresponds to the number of MCBs on the created list. Manipulation of this variable is also done under the protection of `MBD_List_Protect`.

Mailbox Control Block

The Mailbox Control Block `MB_MCB` contains the mailbox message area (4 32-bit unsigned words) and other fields necessary for processing mailbox requests.

Field Declarations

```
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
```

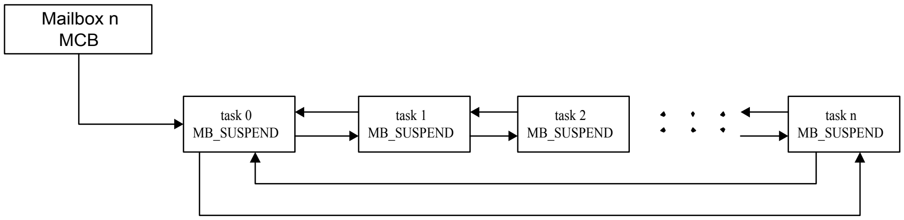
Field Summary

Field	**Description**
`mb_created`	This is the link node structure for mailboxes. It is linked into the created mailbox list, which is a doubly linked, circular list.
`mb_id`	This holds the internal mailbox identification of `0x4D424F58`, which is an equivalent to ASCII MBOX.
`mb_name`	This is the user-specified, 8 character name for the mailbox.
`mb_message_present`	A flag that indicates if a message is present in the mailbox.
`mb_fifo_suspend`	A flag that determines whether tasks suspend in FIFO or priority order.
`mb_padding`	This is used to align the mailbox structure on an even boundary. In some ports this field is not used.
`mb_tasks_waiting`	Indicates the number of tasks that are currently suspended on the mailbox.
`mb_message_area`	The storage area for the message.
`*mb_suspension_list`	The head of the mailbox suspension list. If no tasks are suspended, this pointer is `NULL`.

Mailbox Suspension Structure

Tasks can suspend on empty and full mailbox conditions. During the suspension process a `MB_SUSPEND` structure is built. This structure contains information about the task and the task's mailbox request at the time of suspension. This suspension structure is linked onto the MCB in a doubly linked, circular list and is allocated off of the suspending task's stack. There is one suspension block for every task suspended on the mailbox.

The order of the suspension block placement on the suspend list is determined at mailbox creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks for tasks of equal or higher priority.



Field Declarations

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

Field Summary

Field	Description
`mb_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked circular suspension list.
`*mb_mailbox`	A pointer to the mailbox structure.
`*mb_suspended_task`	A pointer to the Task Control Block of the suspended task.
`*mb_message_area`	A pointer indicating where the suspended tasks's message buffer is.
`mb_return_status`	The completion status of the task suspended on the mailbox.

Mailbox Functions

The following sections provide a brief description of the functions in the Mailbox Component (MB). Review of the actual source code is recommended for further information.

MBC_Create_Mailbox

Creates a mailbox and then places it on the list of created mailboxes.

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

Functions Called

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

MBC_Delete_Mailbox

This function deletes a mailbox and removes it from the list of created mailboxes. All tasks suspended on the mailbox are resumed. Note that this function does not free the memory associated with the mailbox control block. That is the responsibility of the application.

```
STATUS  MBC_Delete_Mailbox(NU_MAILBOX *mailbox_ptr)
```

Functions Called

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
```

MBC_Send_To_Mailbox

This function sends a 4-word message to the specified mailbox. If there are one or more tasks suspended on the mailbox for a message, the message is copied into the message area of the first task waiting and that task is resumed. If the mailbox is full, suspension of the calling task is possible.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

MBC_Receive_From_Mailbox

This function receives a message from the specified mailbox. If there is a message currently in the mailbox, the message is removed from the mailbox and placed in the caller’s area. Otherwise, if no message is present in the mailbox, suspension of the calling task is possible.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

MBC_Cleanup

This function is responsible for removing a suspension block from a mailbox. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  MBC_Cleanup(VOID *information)
```

Functions Called

```
CSC_Remove_From_List
```

MBCE_Create_Mailbox

This function performs error checking on the parameters supplied to the mailbox create function.

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

Functions Called

```
MBC_Create_Mailbox
```

MBCE_Delete_Mailbox

This function performs error checking on the parameters supplied to the actual delete mailbox function.

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

Functions Called

```
MBC_Delete_Mailbox
```

MBCE_Send_To_Mailbox

This function performs error checking on the parameters supplied to the send-to-mailbox function.

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

Functions Called

```
MBC_Sent_To_Mailbox
TCCE_Suspend_Error
```

MBCE_Receive_From_Mailbox

This function performs error checking on the parameters supplied to the receive message from mailbox function.

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

Functions Called

```
MBC_Receive_From_Mailbox
TCCE_Suspend_Error
```

MBF_Established_Mailboxes

Returns the current number of established mailboxes. Mailboxes previously deleted are no longer considered established.

```
UNSIGNED  MBF_Established_Mailboxes(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

MBF_Mailbox_Pointers

Builds a list of mailbox pointers, starting at the specified location. The number of mailbox pointers placed in the list is equivalent to the total number of mailboxes or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

MBF_Mailbox_Information

Returns information about the specified mailbox. However, if the supplied mailbox pointer is invalid, the function simply returns an error status.

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

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

MBI_Initialize

This function initializes the data structures that control the operation of the Mailbox Component. There are no mailboxes initially.

```
VOID  MBI_Initialize(VOID)
```

Functions Called

None

MBS_Reset_Mailbox

This function resets a mailbox back to the initial state. Any message in the mailbox is discarded. Also, all tasks suspended on the mailbox are resumed with the reset completion status.

```
STATUS  MBS_Reset_Mailbox(NU_MAILBOX *mailbox_ptr)
```

Functions Called

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

MBS_Broadcast_To_Mailbox

This function sends a message to all tasks currently waiting for a message from the mailbox. If no tasks are waiting, this service behaves like a normal send message routine.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

`MBSE_Reset_Mailbox`

This function performs error checking on the parameters supplied to the actual reset mailbox function.

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

Functions Called

```
MBS_Reset_Mailbox
```

`MBSE_Broadcast_To_ Mailbox`

This function performs error checking on the parameters supplied to the mailbox broadcast function.

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

Functions Called

```
MBS_Broadcast_To_Mailbox
TCCE_Suspend_Error
```

Queue Component (QU)

The Queue Component (QU) is responsible for processing all Nucleus PLUS queue facilities. A Nucleus PLUS queue is a mechanism for tasks to communicate between each other. Each queue is capable of holding multiple messages. A queue message consists of one or more 32-bit words. Tasks may suspend while waiting for a message from an empty queue. Conversely, tasks may suspend while trying to send to a queue that is in a full condition. Queues are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about queues.

Queue Files

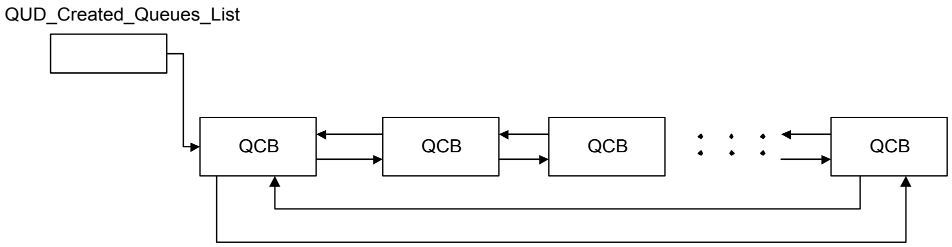
The Queue Component (QU) consists of nine files. Each source file of the Queue Component is defined below.

File	**Description**
`QU_DEFS.H`	This file contains constants and data structure definitions specific to the QU.
`QU_EXTR.H`	All external interfaces to the QU are defined in this file.
`QUD.C`	Global data structures for the QU are defined in this file.
`QUI.C`	This file contains the initialization function for the QU.
`QUF.C`	This file contains the information gathering functions for the QU.
`QUC.C`	This file contains all of the core functions of the QU. Functions that handle basic send-to-queue and receive-from-queue services are defined in this file.
`QUS.C`	This file contains supplemental functions of the QU. Functions contained in this file are typically used less frequently than the core functions.
`QUCE.C`	This file contains the error checking function interfaces for the core functions defined in *`QUC.C`*.
`QUSE.C`	This file contains the error checking function interfaces for the supplemental functions defined in *`QUS.C`*.

Queue Data Structures

Created Queue List

Nucleus PLUS queues may be created and deleted dynamically. The Queue Control Block (QCB) for each created queue is kept on a doubly linked, circular list. Newly created queues are placed at the end of the list, while deleted queues are completely removed from the list. The head pointer of this list is `QUD_Created_Queues_List`.



Created Queue List Protection

Nucleus PLUS protects the integrity of the Created Queues List from competing tasks and/or HISRs. This is done by using an internal protection structure called `QUD_List_Protect`. All queue creation and deletion is done under the protection of `QUD_List_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Queues

The total number of currently created Nucleus PLUS queues is contained in the variable `QUD_Total_Queues`. The contents of this variable corresponds to the number of QCBs on the created list. Manipulation of this variable is also done under the protection of `QUD_List_Protect`.

Queue Control Block

The Queue Control Block `QU_QCB` contains the queue message area (one or more 32-bit unsigned words) and other fields necessary for processing queue requests.

Field Declarations

```
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_messages
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
struct QU_SUSPEND_STRUCT        *qu_urgent_list
struct QU_SUSPEND_STRUCT        *qu_suspension_list
```

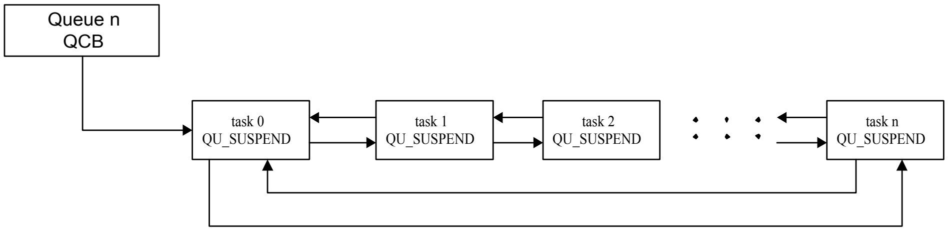
Field Summary

Field	Description
`qu_created`	This is the link node structure for queues. It is linked into the created queues list, which is a doubly linked, circular list.
`qu_id`	This holds the internal queue identification of 0x51554555, which is equivalent to ASCII QUEU.
`qu_name`	This is the user-specified, 8 character name for the queue.
`qu_fixed_size`	A flag that indicates if the size of the queue is fixed or variable.
`qu_fifo_suspend`	A flag that determines whether tasks suspend in fifo or priority order.
`qu_padding`	This is used to align the queue structure on an even boundary. In some ports this field is not used.
`qu_queue_size`	This is the total size of the queue.
`qu_messages`	A flag that indicates if there is a message present in the queue.
`qu_message_size`	Holds the size of the queue message.
`qu_available`	Tells how many bytes are available in the queue.
`qu_start`	Stores the beginning of the queue.
`qu_end`	Stores the end of the queue.
`qu_read`	This is the read pointer.
`qu_write`	This is the write pointer.
`qu_tasks_waiting`	Indicates the number of tasks that are currently suspended on the queue.
`*qu_urgent_list`	A pointer to the suspension list for urgent messages.
`*qu_suspension_list`	The head pointer of the queue suspension list. If no tasks are suspended, this pointer is `NULL`.

Queue Suspension Structure

Tasks can suspend on empty and full queue conditions. During the suspension process a `QU_SUSPEND` structure is built. This structure contains information about the task and the task's queue request at the time of suspension. This suspension structure is linked onto the QCB in a doubly linked, circular list and is allocated off of the suspending task's stack. There is one suspension block for every task suspended on the queue.

The order of the suspension block placement on the suspend list is determined at queue creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks for tasks of equal or higher priority.



Field Declarations

```
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
```

Field Summary

Field	Description
`qu_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked, circular suspension list.
`*qu_queue`	A pointer to the queue structure.
`*qu_suspended_task`	A pointer to the Task Control Block of the suspended task.
`qu_message_area`	A pointer indicating where the suspended task's message buffer is.
`qu_message_size`	Stores the size of the requested message
`qu_actual_size`	Stores the actual size of the message.
`qu_return_status`	The completion status of the task suspended on the queue.

Queue Functions

The following sections provide a brief description of the functions in the Queue Component (QU). Review of the actual source code is recommended for further information.

QUC_Create_Queue

This function creates a queue and then places it on the list of created queues.

```
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)
```

Functions Called

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

QUC_Delete_Queue

This function deletes a queue and removes it from the list of created queues. All tasks suspended on the queue are resumed. Note that this function does not free the memory associated with the queue. That is the responsibility of the application.

```
STATUS  QUC_Delete_Queue(NU_QUEUE *queue_ptr)
```

Functions Called

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
```

QUC_Send_To_Queue

This function sends a message to the specified queue. The caller determines the message length. If there are one or more tasks suspended on the queue for a message, the message is copied into the message area of the first waiting task. If the task's request is satisfied, it is resumed. Otherwise, if the queue cannot hold the message, suspension of the calling task is an option of the caller.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

QUC_Receive_From_Queue

This function receives a message from the specified queue. The caller specifies the size of the message. If there is a message currently in the queue, the message is removed from the queue and placed in the caller's area. Suspension is possible if the request cannot be satisfied.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

QUC_Cleanup

This function is responsible for removing a suspension block from a queue. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  QUC_Cleanup(VOID *information)
```

Functions Called

`CSC_Remove_From_List`

QUCE_Create_Queue

This function performs error checking on the parameters supplied to the queue create function.

```
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)
```

Functions Called

`QUC_Create_Queue`

QUCE_Delete_Queue

This function performs error checking on the parameter supplied to the queue delete function.

```
STATUS  QUCE_Delete_Queue(NU_QUEUE *queue_ptr)
```

Functions Called

`QUC_Delete_Queue`

QUCE_Send_To_Queue

This function performs error checking on the parameters supplied to the send message to queue function.

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

Functions Called

```
QUC_Send_To_Queue
TCCE_Suspend_Error
```

QUCE_Receive_From_Queue

This function performs error checking on the parameters supplied to the receive message from queue function.

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

Functions Called

```
QUC_Receive_From_Queue
TCCE_Suspend_Error
```

QUF_Established_Queues

This function returns the current number of established queues. Queues previously deleted are no longer considered established.

```
UNSIGNED  QUF_Established_Queues(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

QUF_Queue_Information

This function returns information about the specified queue. However, if the supplied queue pointer is invalid, the function simply returns an error status.

```
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)
```

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

QUF_Queue_Pointers

Builds a list of queue pointers, starting at the specified location. The number of queue pointers placed in the list is equivalent to the total number of queues or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

QUI_Initialize

This function initializes the data structures that control the operation of the Queue Management component. There are no queues initially.

```
VOID  QUI_Initialize(VOID)
```

Functions Called

None

QUS_Reset_Queue

This function resets the specified queue back to the original state. Any messages in the queue are discarded. Also, any tasks currently suspended on the queue are resumed with the reset status.

```
STATUS  QUS_Reset_Queue(NU_QUEUE *queue_ptr)
```

Functions Called

```
[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

This function sends a message to the front of the specified message queue. The caller determines the message length. If there are any tasks suspended on the queue for a message, the message is copied into the message area of the first waiting task and that task is resumed. If there is enough room in the queue, the message is copied in front of all other messages. If there is not enough room in the queue, suspension of the caller is possible.

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

Functions Called

```
CSC_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
[TCT_Check_Stack]
```

QUS_Broadcast_To_Queue

This function sends a message to all tasks waiting for a message from the specified queue. If there are no tasks waiting for a message the service performs like a standard send request.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

QUSE_Reset_Queue

This function performs error checking on the parameter supplied to the queue reset function.

```
STATUS  QUSE_Reset_Queue(NU_QUEUE *queue_ptr)
```

Functions Called

```
QUS_Reset_Queue
```

QUSE_Send_To_Front_Of_Queue

This function performs error checking on the parameters supplied to the send message to front of queue function.

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

Functions Called

```
QUS_Send_To_Front_Of_Queue
TCCE_Suspend_Error
```

QUSE_Broadcast_To_Queue

This function performs error checking on the parameters supplied to the broadcast message to queue function.

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

Functions Called

```
QUS_Broadcast_To_Queue
TCCE_Suspend_Error
```

Pipe Component (PI)

The Pipe Component (PI) is responsible for processing all Nucleus PLUS pipe facilities. A Nucleus PLUS pipe is a mechanism for tasks to communicate between each other. Each pipe is capable of holding multiple messages. A pipe message consists of one or more bytes. Tasks may suspend while waiting for a message from an empty pipe. Conversely, tasks may suspend while trying to send to a pipe that is in a full condition. Pipes are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about pipes.

Pipe Files

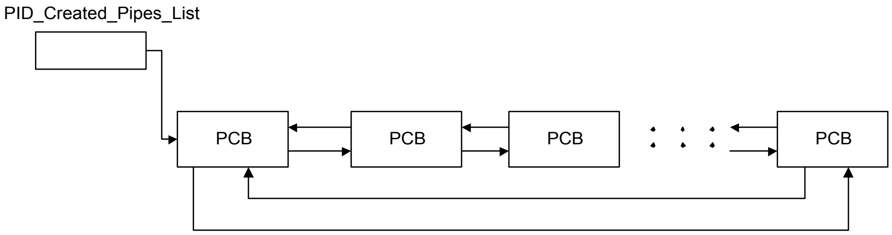
The Pipe Component (PI) consists of nine files. Each source file of the Pipe Component is defined below.

File	Description
`PI_DEFS.H`	This file contains constants and data structure definitions specific to the PI.
`PI_EXTR.H`	All external interfaces to the PI are defined in this file.
`PID.C`	Global data structures for the PI are defined in this file.
`PII.C`	This file contains the initialization function for the PI.
`PIF.C`	This file contains the information gathering functions for the PI.
`PIC.C`	This file contains all of the core functions of the PI. Functions that handle basic send-to-pipe and receive-from-pipe services are defined in this file.
`PIS.C`	This file contains supplemental functions of the PI. Functions contained in this file are typically used less frequently than the core functions.
`PICE.C`	This file contains the error checking function interfaces for the core functions defined in `PIC.C`.
`PISE.C`	This file contains the error checking function interfaces for the supplemental functions defined in `PIS.C`.

Pipe Data Structures

Created Pipe List

Nucleus PLUS pipes may be created and deleted dynamically. The Pipe Control Block (PCB) for each created pipe is kept on a doubly linked, circular list. Newly created pipes are placed at the end of the list, while deleted pipes are completely removed from the list. The head pointer of this list is `PID_Created_Pipes_List`.



Created Pipe List Protection

Nucleus PLUS protects the integrity of the Created Pipes List from competing tasks and/or HISRs. This is done by using an internal protection structure called `PID_List_Protect`. All pipe creation and deletion is done under the protection of `PID_List_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	Description
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Pipes

The total number of currently created Nucleus PLUS pipes is contained in the variable `PID_Total_Pipes`. The contents of this variable correspond to the number of PCBs on the created list. Manipulation of this variable is also done under the protection of `PID_List_Protect`.

Pipe Control Block

The Pipes Control Block `PI_PCB` contains the pipe message area (1 or more bytes) and other fields necessary for processing pipe requests.

Field Declarations

```
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
struct PI_SUSPEND_STRUCT *pi_urgent_list
struct PI_SUSPEND_STRUCT *pi_suspension_list
```

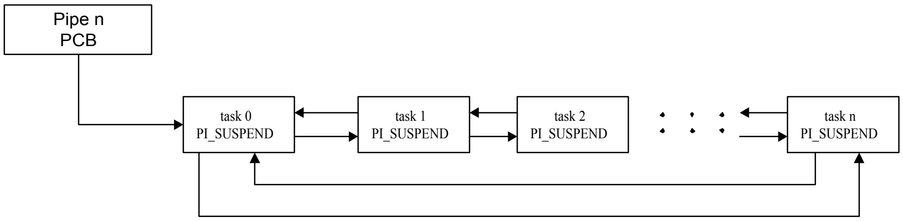
Field Summary

Field	Description
`pi_created`	This is the link node structure for pipes. It is linked into the created pipes list, which is a doubly linked, circular list.
`pi_id`	This holds the internal pipe identification of `0x50495045`, which is equivalent to ASCII PIPE.
`pi_name`	This is the user-specified, 8 character name for the pipe.
`pi_fixed_size`	A flag that indicates if the size of the pipe is fixed or variable.
`pi_fifo_suspend`	A flag that determines whether tasks suspend in fifo or priority order.
`pi_padding`	This is used to align the pipe structure on an even boundary. In some ports this field is not used.
`pi_pipe_size`	This is the total size of the pipe.
`pi_messages`	A flag that indicates if there is a message present in the pipe.
`pi_message_size`	Holds the size of the message.
`pi_available`	Tells how many bytes are available in the pipe.
`pi_start`	Stores the beginning of the pipe.
`pi_end`	Stores the end of the pipe.
`pi_read`	This is the read pointer.
`pi_write`	This is the write pointer.
`pi_tasks_waiting`	Indicates the number of tasks that are currently suspended on the pipe.
`*pi_urgent_list`	A pointer to the suspension list for urgent messages.
`*pi_suspension_list`	The head pointer of the pipe suspension list. If no tasks are suspended, this pointer is `NULL`.

Pipe Suspension Structure

Tasks can suspend on empty and full pipe conditions. During the suspension process a `PI_SUSPEND` structure is built. This structure contains information about the task and the task's pipe request at the time of suspension. This suspension structure is linked onto the PCB in a doubly linked, circular list and is allocated off of the suspending task's stack. There is one suspension block for every task suspended on the pipe.

The order of the suspension block placement on the suspend list is determined at pipe creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks for tasks of equal or higher priority.



Field Declarations

```
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
```

Field Summary

Field	Description
`pi_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked, circular suspension list.
`*pi_pipe`	A pointer to the pipe structure.
`*pi_suspended_task`	A pointer to the Task Control Block of the suspended task.
`pi_message_area`	A pointer indicating where the suspended task's message buffer is.
`pi_message_size`	Stores the size of the requested message
`pi_actual_size`	Stores the actual size of the message.
`pi_return_status`	The completion status of the task suspended on the pipe.

Pipe Functions

The following sections provide a brief description of the functions in the Pipe Component (PI). Review of the actual source code is recommended for further information.

PIC_Create_Pipe

Creates a pipe and then places it on the list of created pipes.

```
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)
```

Functions Called

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

PIC_Delete_Pipe

Deletes a pipe and removes it from the list of created pipes. All tasks suspended on the pipe are resumed. Note that this function does not free the memory associated with the pipe. That is the responsibility of the application.

```
STATUS  PIC_Delete_Pipe(NU_PIPE *pipe_ptr)
```

Functions Called

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
```

PIC_Send_To_Pipe

This function sends a message to the specified pipe. The caller determines the message length. If there are one or more tasks suspended on the pipe for a message, the message is copied into the message area of the first waiting task. If the task's request is satisfied, it is resumed. Otherwise, if the pipe cannot hold the message, suspension of the calling task is an option of the caller.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

PIC_Receive_From_Pipe

This function receives a message from the specified pipe. The caller specifies the size of the message. If there is a message currently in the pipe, themessage is removed from the pipe and placed in the caller's area. Suspension is possible if the request cannot be satisfied.

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

Functions Called

```
CSC_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
```

PIC_Cleanup

This function is responsible for removing a suspension block from a pipe. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  PIC_Cleanup(VOID *information)
```

Functions Called

`CSC_Remove_From_List`

PICE_Create_Pipe

This function performs error checking on the parameters supplied to the pipe create function.

```
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)
```

Functions Called

`PIC_Create_Pipe`

PICE_Delete_Pipe

This function performs error checking on the parameter supplied to the pipe delete function.

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

Functions Called

```
PIC_Delete_Pipe
```

PICE_Send_To_Pipe

This function performs error checking on the parameters supplied to the send message to pipe function.

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

Functions Called

```
PIC_Send_To_Pipe
TCCE_Suspend_Error
```

PICE_Receive_From_Pipe

This function performs error checking on the parameters supplied to the receive message from pipe function.

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

Functions Called

```
PIC_Receive_From_Pipe
TCCE_Suspend_Error
```

PIF_Established_Pipes

Returns the current number of established pipes. Pipes previously deleted are no longer considered established.

```
UNSIGNED  PIF_Established_Pipes(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

PIF_Pipe_Information

Returns information about the specified pipe. However, if the supplied pipe pointer is invalid, the function simply returns an error status.

```
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)
```

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

PIF_Pipe_Pointers

Builds a list of pipe pointers, starting at the specified location. The number of pipe pointers placed in the list is equivalent to the total number of pipes or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

PII_Initialize

This function initializes the data structures that control the operation of the Pipe Component. There are no pipes initially.

```
VOID  PII_Initialize(VOID)
```

Functions Called

None

PIS_Reset_Pipe

This function resets the specified pipe back to the original state. Any messages in the pipe are discarded. Also, any tasks currently suspended on the pipe are resumed with the reset status.

```
STATUS  PIS_Reset_Pipe(NU_PIPE *pipe_ptr)
```

Functions Called

```
[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

This function sends a message to the front of the specified message pipe. The caller determines the message length. If there are any tasks suspended on the pipe for a message, the message is copied into the message area of the first waiting task and that task is resumed. If there is enough room in the pipe, the message is copied in front of all other messages. If there is not enough room in the pipe, suspension of the caller is possible.

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

Functions Called

```
CSC_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

PIS_Broadcast_To_Pipe

This function sends a message to all tasks waiting for a message from the specified pipe. If there are no tasks waiting for a message the service performs like a standard send request.

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

Functions Called

```
CSC_Place_On_List
CSC_Priority_Place_On_List
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
TCC_Suspend_Task
```

PISE_Reset_Pipe

This function performs error checking on the parameter supplied to the pipe reset function.

```
STATUS  PISE_Reset_Pipe(NU_PIPE *pipe_ptr)
```

Functions Called

```
PIS_Reset_Pipe
```

PISE_Send_To_Front_Of_Pipe

This function performs error checking on the parameters supplied to the send message to front of pipe function.

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

Functions Called

```
PIS_Send_To_Front_Of_Pipe
TCCE_Suspend_Error
```

PISE_Broadcast_To_Pipe

This function performs error checking on the parameters supplied to the broadcast message to pipe function.

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

Functions Called

```
PIS_Broadcast_To_Pipe
TCCE_Suspend_Error
```

Semaphore Component (SM)

The Semaphore Component (SM) is responsible for processing all Nucleus PLUS semaphore facilities. A Nucleus PLUS semaphore is a mechanism to synchronize the execution of various tasks in an application. Nucleus PLUS provides counting semaphores that range in value from 0 to 4,294,967,294. Tasks may suspend while waiting for a non-zero semaphore value. Semaphores are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about semaphores.

Semaphore Files

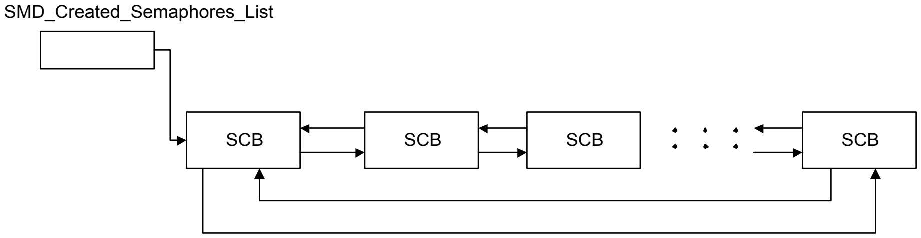
The Semaphore Component (SM) consists of nine files. Each source file of the Semaphore Component is defined below.

File	Description
`SM_DEFS.H`	This file contains constants and data structure definitions specific to the SM.
`SM_EXTR.H`	All external interfaces to the SM are defined in this file.
`SMD.C`	Global data structures for the SM are defined in this file.
`SMI.C`	This file contains the initialization function for the SM.
`SMF.C`	This file contains the information gathering functions for the SM.
`SMC.C`	This file contains all of the core functions of the SM. Functions that handle basic obtain-semaphore and release-semaphore services are defined in this file.
`SMS.C`	This file contains supplemental functions of the SM. Functions contained in this file are typically used less frequently than the core functions.
`SMCE.C`	This file contains the error checking function interfaces for the core functions defined in `SMC.C`.
`SMSE.C`	This file contains the error checking function interfaces for the supplemental functions defined in `SMS.C`.

Semaphore Data Structures

Created Semaphore List

Nucleus PLUS semaphores may be created and deleted dynamically. The Semaphore Control Block (SCB) for each created semaphore is kept on a doubly linked, circular list. Newly created semaphores are placed at the end of the list, while deleted semaphores are completely removed from the list. The head pointer of this list is `SMD_Created_Semaphores_List`.



Created Semaphore List Protection

Nucleus PLUS protects the integrity of the Created Semaphores List from competing tasks and/or HISRs. This is done by using an internal protection structure called `SMD_List_Protect`. All semaphore creation and deletion is done under the protection of `SMD_List_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	Description
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection

Total Semaphores

The total number of currently created Nucleus PLUS semaphores is contained in the variable `SMD_Total_Semaphores`. The contents of this variable correspond to the number of SCBs on the created list. Manipulation of this variable is also done under the protection of `SMD_List_Protect`.

Semaphore Control Block

The Semaphores Control Block `SM_SCB` contains the semaphore count and other fields necessary for processing semaphore requests.

Field Declarations

```
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
```

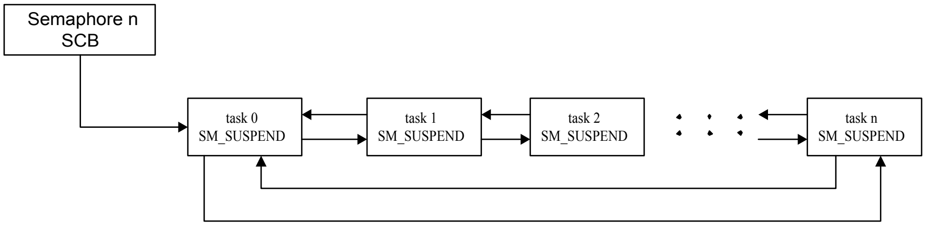
Field Summary

Field	Description
`sm_created`	This is the link node structure for semaphores. It is linked into the created semaphores list, which is a doubly linked, circular list.
`sm_id`	This holds the internal semaphore identification of `0x53454D41`, which is equivalent to ASCII SEMA.
`sm_name`	This is the user-specified, 8 character name for the semaphore.
`sm_semaphore_count`	Stores the current count of the semaphore.
`sm_fifo_suspend`	A flag that determines whether tasks suspend in fifo or priority order.
`sm_padding`	This is used to align the semaphore structure on an even boundary.
	In some ports this field is not used.
`sm_tasks_waiting`	Indicates the number of tasks that are currently suspended on the semaphore.
`*sm_suspension_list`	The head pointer of the semaphore suspension list. If no tasks are suspended, this pointer is `NULL`.

Semaphore Suspension Structure

Tasks can suspend on a semaphore whose current count is zero. During the suspension process a `SM_SUSPEND_STRUCT` structure is built. This structure contains information about the task and the task's semaphore request at the time of suspension. This suspension structure is linked onto the SCB in a doubly linked, circular list and is allocated from the suspending task's stack. There is one suspension block for every task suspended on the semaphore.

The order of the suspension block placement on the suspend list is determined at semaphore creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks for tasks of equal or higher priority.



Field Declarations

```
CS_NODE sm_suspend_link
SM_SCB *sm_semaphore
TC_TCB *sm_suspended_task
STATUS sm_return_status
```

Field Summary

Field	**Description**
`sm_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked, circular suspension list.
`*sm_semaphore`	A pointer to the semaphore structure.
`*sm_suspended_task`	A pointer to the Task Control Block of the suspended task.
`sm_return_status`	The completion status of the task suspended on the semaphore

Semaphore Functions

The following sections provide a brief description of the functions in the Semaphore Component (SM). Review of the actual source code is recommended for further information.

SMC_Create_Semaphore

This function creates a semaphore and places it on the list of created semaphores.

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

Functions Called

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

SMC_Delete_Semaphore

This function deletes a semaphore and removes it from the list of created semaphores. All tasks suspended on the semaphore are resumed. Note that this function does not free the memory associated with the semaphore control block. That is the responsibility of the application.

```
STATUS  SMC_Delete_Semaphore(NU_SEMAPHORE *semaphore_ptr)
```

Functions Called

```
CSC_Remove_From_List
[HIC_Make_History_Entry]
TCC_Resume_Task
[TCT_Check_Stack]
TCT_Control_To_System
```

SMC_Obtain_Semaphore

This function obtains an instance of the semaphore. An instance corresponds to decrementing the counter by one. If the counter is greater than zero at the time of this call, this function can be completed immediately. Otherwise, suspension is possible.

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

Functions Called

```
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
```

SMC_Release_Semaphore

This function releases a previously obtained semaphore. If one or more tasks are waiting, the first task is given the released instance of the semaphore. Otherwise, the semaphore instance counter is simply incremented.

```
STATUS  SMC_Release_Semaphore(NU_SEMAPHORE *semaphore_ptr)
```

Functions Called

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

SMC_Cleanup

This function is responsible for removing a suspension block from a semaphore. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  SMC_Cleanup(VOID *information)
```

Functions Called

```
CSC_Remove_From_List
```

SMCE_Create_Semaphore

This function performs error checking on the parameters supplied to the create semaphore function.

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

Functions Called

```
SMC_Create_Semaphore
```

SMCE_Delete_Semaphore

This function performs error checking on the parameters supplied to the delete semaphore function.

```
STATUS  SMCE_Delete_Semaphore(NU_SEMAPHORE *semaphore_ptr)
```

Functions Called

```
SMC_Delete_Semaphore
```

SMCE_Obtain_Semaphore

This function performs error checking on the parameters supplied to the obtain semaphore function.

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

Functions Called

```
SMC_Obtain_Semaphore
TCCE_Suspend_Error
```

SMCE_Release_Semaphore

This function performs error checking on the parameters supplied to the release semaphore function.

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

Functions Called

```
SMC_Release_Semaphore
```

SMF_Established_Semaphores

This function returns the current number of established semaphores. Semaphores previously deleted are no longer considered established.

```
UNSIGNED SMF_Established_Semaphores(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

SMF_Semaphore_Pointers

Builds a list of semaphore pointers, starting at the specified location. The number of semaphore pointers placed in the list is equivalent to the total number of semaphores or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

SMF_Semaphore_Information

This function returns information about the specified semaphore. However, if the supplied semaphore pointer is invalid, the function simply returns an error status.

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

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

SMI_Initialize

This function initializes the data structures that control the operation of the Semaphore Component. There are no semaphores initially.

```
VOID  SMI_Initialize(VOID)
```

Functions Called

```
None
```

SMS_Reset_Semaphore

This function resets a semaphore back to the initial state. All tasks suspended on the semaphore are resumed with the reset completion status.

```
STATUS  SMS_Reset_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                            UNSIGNED initial_count)
```

Functions Called

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

SMSE_Reset_Semaphore

This function performs error checking on the parameters supplied to the reset semaphore function.

```
STATUS  SMSE_Reset_Semaphore(NU_SEMAPHORE *semaphore_ptr,
                              UNSIGNED initial_count)
```

Functions Called

`SMS_Reset_Semaphore`

Event Group Component (EV)

The Event Group Component (EV) is responsible for processing all Nucleus PLUS event group facilities. A Nucleus PLUS event is a mechanism to indicate that a certain system event has occurred. An event is represented by a single bit in an event group. This bit is called an event flag. There are 32 event flags in each event group. Tasks may suspend while waiting for a particular set of event flags. Event groups are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about events.

Event Group Files

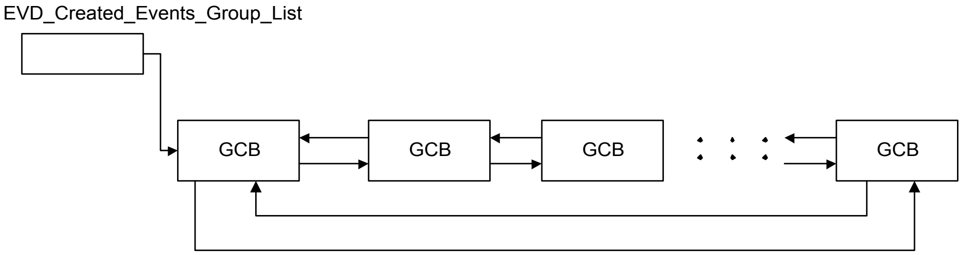
The Event Group Component (EV) consists of seven files. Each source file of the Event Group Component is defined below.

File	**Description**
`EV_DEFS.H`	This file contains constants and data structure definitions specific to the EV.
`EV_EXTR.H`	All external interfaces to the EV are defined in this file.
`EVD.C`	Global data structures for the EV are defined in this file.
`EVI.C`	This file contains the initialization function for the EV.
`EVF.C`	This file contains the information gathering functions for the EV.
`EVC.C`	This file contains all of the core functions of the EV. Functions that handle basic set-event and retrieve-event services are defined in this file.
`EVCE.C`	This file contains the error checking function interfaces for the core functions defined in `EVC.C`.

Event Group Data Structures

Created Event Group List

Nucleus PLUS events may be created and deleted dynamically. The Event Group Control Block (GCB) for each created event group is kept on a doubly linked, circular list. Newly created event groups are placed at the end of the list, while deleted event groups are completely removed from the list. The head pointer of this list is `EVD_Created_Events_Group_List`.



Created Event Group List Protection

Nucleus PLUS protects the integrity of the Created Events Group List from competing tasks and/or HISRs. This is done by using an internal protection structure called `EVD_List_Protect`. All event group creation and deletion is done under the protection of `EVD_List_Protect`.

```
TC_TCB       *tc_tcb_pointer
UNSIGNED     tc_thread_waiting
```

Functions Called

Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for protection

Total Event Groups

The total number of currently created Nucleus PLUS event groups is contained in the variable `EVD_Total_Event_Groups`. The contents of this variable correspond to the number of GCBs on the created list. Manipulation of this variable is also done under the protection of `EVD_List_Protect`.

Event Group Control Block

The Event Group Control Block `EV_GCB` contains the current event flags and other fields necessary for processing event requests.

Field Declarations

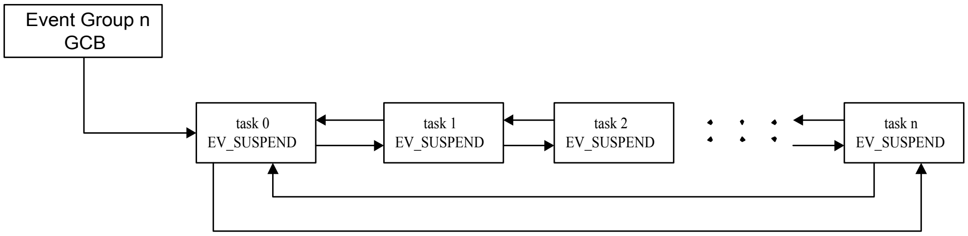
```
CS_NODE                          ev_created
UNSIGNED                         ev_id
CHAR                             ev_name[NU_MAX_NAME]
UNSIGNED                         ev_current_events
UNSIGNED                         ev_tasks_waiting
struct EV_SUSPEND_STRUCT         *ev_suspension_list
```

Field Summary

Field	**Description**
`ev_created`	This is the link node structure for events. It is linked into the created events group list, which is a doubly linked, circular list.
`ev_id`	This holds the internal event group identification of `0x45564E54`, which is equivalent to ASCII EVNT.
`ev_name`	This is the user-specified, 8 character name for the event group.
`ev_current_events`	Contains the current event flags.
`ev_tasks_waiting`	Indicates the number of tasks that are currently suspended on an event group.
`*ev_suspension_list`	The head pointer of the event group suspension list. If no tasks are suspended, this pointer is `NULL`.

Event Group Suspension Structure

Tasks can suspend when an event group does not match the user specified combination of event flags. During the suspension process the `EV_SUSPEND_STRUCT` structure is built. This structure contains information about the task and the task's event group request at the time of suspension. This suspension structure is linked onto the GCB in a doubly linked, circular list and is allocated off of the suspending task's stack. There is one suspension block for every task suspended on the event group.



Field Declarations

```
CS_NODE             ev_suspend_link
EV_GCB              *ev_event_group
UNSIGNED            ev_requested_events
DATA_ELEMENT        ev_operation
DATA_ELEMENT        ev_padding[PAD_1]
TC_TCB              *ev_suspended_task
STATUS              ev_return_status
UNSIGNED            ev_actual_events
```

Field Summary

Field	Description
`ev_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly-linked circular suspension list
`*em_event_group`	A pointer to the event group structure.
`ev_requested_events`	The event group that has been requested.
`ev_operation`	The type of operation that is requested on the event group. This is typically some sort of AND/OR combination.
`ev_padding`	This is used to align the suspend event group structure on an even boundary. In some ports this field is not used.
`*ev_suspended_task`	A pointer to the Task Control Block of the suspended task.
`ev_return_status`	The completion status of the task suspended on the event group.
`ev_actual_events`	The set of actual event flags returned by the request.

Event Group Functions

The following sections provide a brief description of the functions in the Event Group Component (EV). Review of the actual source code is recommended for further information.

EVC_Create_Event_Group

Creates an event group and then places it on the list of created event groups.

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

Functions Called

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

EVC_Delete_Event_Group

Deletes an event group and removes it from the list of created event groups. All tasks suspended on the event group are resumed. Note that this function does not free the memory associated with the event group control block. That is the responsibility of the application.

```
STATUS  EVC_Delete_Event_Group(NU_EVENT_GROUP *event_group_ptr)
```

Functions Called

```
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
```

EVC_Set_Events

Sets event flags within the specified event flag group. Event flags may be **AND**ed or **OR**ed against the current events of the group. Tasks suspended on a group are resumed when the requested event is satisfied.

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

Functions Called

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

EVC_Retrieve_Events

Retrieves various combinations of event flags from the specified event group. If the group does not contain the necessary flags, suspension of the calling task is possible.

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

Functions Called

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

EVC_Cleanup

This function is responsible for removing a suspension block from an event group. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  EVC_Cleanup(VOID *information)
```

Functions Called

```
CSC_Remove_From_List
```

EVCE_Create_Event_Group

This function performs error checking on the parameters supplied to the create event group function.

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

Functions Called

`EVC_Create_Event_Group`

EVCE_Delete_Event_Group

This function performs error checking on the parameters supplied to the delete event group function.

```
STATUS  EVCE_Delete_Event_Group(NU_EVENT_GROUP *event_group_ptr)
```

Functions Called

`EVC_Delete_Event_Group`

EVCE_Set_Events

This function performs error checking on the parameters supplied to the set events group function.

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

Functions Called

`EVC_Set_Events`

EVCE_Retrieve_Events

This function performs error checking on the parameter supplied to the retrieve events function.

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

Functions Called

```
EVC_Retrieve_Events
TCCE_Suspend_Error
```

EVF_Established_Event_Groups

Returns the current number of established event groups. Event groups previously deleted are no longer considered established.

```
UNSIGNED  EVF_Established_Event_Groups(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

EVF_Event_Group_Pointers

Builds a list of event group pointers, starting at the specified location. The number of event group pointers placed in the list is equivalent to the total number of event groups or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

EVF_Event_Group_Information

Returns information about the specified event group. However, if the supplied event group pointer is invalid, the function simply returns an error status.

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

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

EVI_Initialize

This function initializes the data structures that control the operation of the Event Group Component. There are no event groups initially.

```
VOID  EVI_Initialize(VOID)
```

Functions Called

`None`

Partition Memory Component (PM)

The Partition Memory Component (PM) is responsible for processing all Nucleus PLUS partition memory facilities. A Nucleus PLUS partition memory pool contains a specific number of fixed-size memory partitions. Tasks may suspend while waiting for a memory partition from an empty pool. Partition pools are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about partition memory pools.

Partition Memory Files

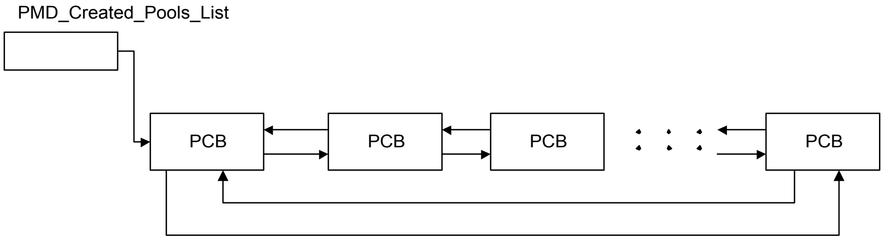
The Partition Memory Component (PM) consists of seven files. Each source file of the Partition Memory Component is defined below.

File	**Description**
`PM_DEFS.H`	This file contains constants and data structure definitions specific to the PM.
`PM_EXTR.H`	All external interfaces to the PM are defined in this file.
`PMD.C`	Global data structures for the PM are defined in this file.
`PMI.C`	This file contains the initialization function for the PM.
`PMF.C`	This file contains the information gathering functions for the PM.
`PMC.C`	This file contains all of the core functions of the PM. Functions that handle basic allocate-memory and deallocate-memory services are defined in this file.
`PMCE.C`	This file contains the error checking function interfaces for the core functions defined in `PMC.C`.

Partition Memory Data Structures

Created Partition Memory List

Nucleus PLUS partition pools may be created and deleted dynamically. The Partition Memory Control Block (PCB) for each created partition memory pool is kept on a doubly linked, circular list. Newly created partition memory pools are placed at the end of the list, while deleted partition memory pools are completely removed from the list. The head pointer of this list is `PMD_Created_Pools_List`.



Created Partition Memory List Protection

Nucleus PLUS protects the integrity of the Created Partition Memory List from competing tasks and/or HISRs. This is done by using an internal protection structure called `PMD_List_Protect`. All partition memory creation and deletion is done under the protection of `PMD_List_Protect`.

Field Declarations

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

Field Summary

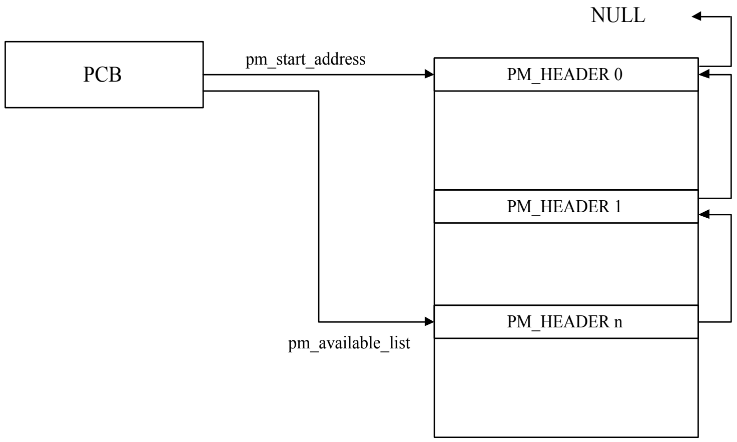
Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Partition Pools

The total number of currently created Nucleus PLUS partition memory pools is contained in the variable `PMD_Total_Pools`. The content of this variable corresponds to the number of PCBs on the created list. Manipulation of this variable is also done under the protection of `PMD_List_Protect`.

Available Partitions List

The Available Partitions List is a singly linked `NULL` terminated list, which contains the available partitions. The PCB contains pointers to the starting address of the list as well as the next available partition in the list. Allocated partitions are removed from the front of the list and deallocated partitions are place at the front of the list. Each partition has a header block that links the partitions together.ports this field is not used.



Partition Pool Control Block

The Partition Memory Pool Control Block `PM_PCB` contains the starting address of the current memory pool and other fields necessary for processing partition pool requests.

Field Declarations

```
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
```

Field Summary

Field	**Description**
`pm_created`	This is the link node structure for partition memory pools. It is linked into the created partition pools list, which is a doubly linked, circular list.
`pm_id`	This holds the internal partition memory pool identification of `0x50415254`, which is equivalent to ASCII PART.
`pm_name`	This is the user-specified, 8 character name for the partition memory pool.
`*pm_start_address`	This is the starting address of the current partition memory pool.
`pm_pool_size`	Holds the size of the partition memory pool.
`pm_partition_size`	This is the size of the current memory pool partition.
`pm_available`	This is the number of partitions available for use in the current memory pool.
`pm_allocated`	Holds the number of allocated partitions.
`*pm_available_list`	This is the list of available partitions of the current memory pool.
`pm_fifo_suspend`	A flag that determines whether tasks suspend in fifo or priority order.
`pm_padding`	This is used to align the partition memory pool structure on an even boundary. In some ports this field is not used.
`pm_tasks_waiting`	Indicates the number of tasks that are currently suspended on a partition memory pool.
`*pm_suspension_list`	The head pointer of the partition memory pool suspension list. If no tasks are suspended, this pointer is `NULL`.

Partition Memory Pool Header Structure

The partition header structure `PM_HEADER` is placed at the beginning of each available partition. Each header contains a pointer to the next available partition, except for the last partition, which points to a null terminator. Each partition header also contains a pointer to its PCB (Partition Memory Pool Control Block).

Field Declarations

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

Field Summary

Field	**Description**
`*pm_next_available`	A pointer to the next partition in the available list.
`pm_partition_pool`	A pointer to this partition's PCB.

Partition Memory Pool Suspension Structure

Tasks can suspend on empty and full partition memory pool conditions. During the suspension process a `PM_SUSPEND` structure is built. This structure contains information about the task and the task's partition pool request at the time of suspension. This suspension structure is linked to the PCB in a doubly linked, circular list and is allocated from the suspending task's stack. There is one suspension block for every task suspended on the partition memory pool.

The suspension block's position on the suspend list is determined at partition pool creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks with tasks of equal or higher priority.

Partition Memory Pool Header Structure

The partition header structure `PM_HEADER` is placed at the beginning of each available partition. Each header contains a pointer to the next available partition, except for the last partition, which points to a null terminator. Each partition header also contains a pointer to its PCB (Partition Memory Pool Control Block).

Field Declarations

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

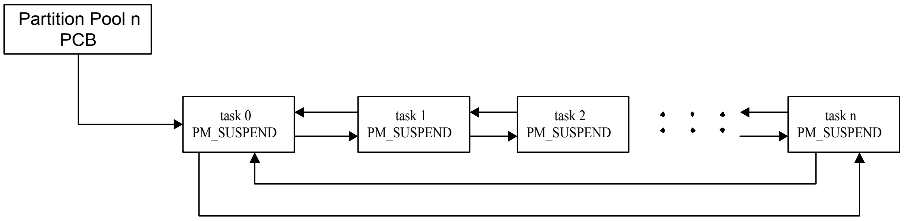
Field Summary

Field	**Description**
`*pm_next_available`	A pointer to the next partition in the available list.
`pm_partition_pool`	A pointer to this partition's PCB.

Partition Memory Pool Suspension Structure

Tasks can suspend on empty and full partition memory pool conditions. During the suspension process a `PM_SUSPEND` structure is built. This structure contains information about the task and the task’s partition pool request at the time of suspension. This suspension structure is linked to the PCB in a doubly linked, circular list and is allocated from the suspending task’s stack. There is one suspension block for every task suspended on the partition memory pool.

The suspension block's position on the suspend list is determined at partition pool creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks with tasks of equal or higher priority.



Field Declarations

```
CS_NODE     pm_suspend_link
PM_PCB     *pm_partiton_pool
TC_TCB     *pm_suspended_task
VOID       *pm_return_status
```

Field Summary

Field	Description
`pm_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked, circular suspension list.
`*pm_partiton_pool`	A pointer to the partition memory pool structure.
`*pm_suspended_task`	A pointer to the Task Control Block of the suspended task.
`*pm_return_pointer`	The return memory address that has been requested.
`pm_return_status`	The completion status of the task suspended on the partition pool.

Partition Memory Functions

The following sections provide a brief description of the functions in the Partition Memory Component (PM). Review of the actual source code is recommended for further information.

PMC_Create_Partition_Pool

Creates a memory partition pool and then places it on the list of created partition pools.

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

Functions Called

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

PMC_Delete_Partition_Pool

This function deletes a memory partition pool and removes it from the list of created partition pools. All tasks suspended on the partition pool are resumed with the appropriate error status. Note that this function does not free any memory associated with either the pool area or the pool control block. That is the responsibility of the application.

```
STATUS  PMC_Delete_Partition_Pool(NU_PARTITION_POOL *pool_ptr)
```

Functions Called

```
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

This function allocates a memory partition from the specified memory partition pool. If a memory partition is currently available, this function is completed immediately. Otherwise, if there are no partitions currently available, suspension is possible.

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

Functions Called

```
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

This function deallocates a previously allocated partition. If there is a task waiting for a partition, the partition is simply given to the waiting task and the waiting task is resumed. Otherwise, the partition is returned to the partition pool.

```
STATUS  PMC_Deallocate_Partition(VOID *partition)
```

Functions Called

```
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

This function is responsible for removing a suspension block from a partition pool. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  PMC_Cleanup(VOID *information)
```

Functions Called

```
CSC_Remove_From_List
```

PMCE_Create_Partition_Pool

This function performs error checking on the parameters supplied to the create partition pool function.

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

Functions Called

`PMC_Create_Partition_Pool`

PMCE_Delete_Partition_Pool

This function performs error checking on the parameters supplied to the delete partition pool function.

```
STATUS  PMCE_Delete_Partition_Pool(NU_PARTITION_POOL *pool_ptr)
```

Functions Called

`PMC_Delete_Partition_Pool`

PMCE_Allocate_Partition

This function performs error checking on the parameters supplied to the allocate partition function.

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

Functions Called

```
PMC_Allocate_Partition
TCCE_Suspend_Error
```

PMCE_Deallocate_Partition

This function performs error checking on the parameters supplied to the deallocate partition function.

```
STATUS  PMCE_Deallocate_Partition(VOID *partition)
```

Functions Called

```
PMC_Deallocate_Partition
```

PMF_Established_Partition_Pools

This function returns the current number of established partition pools. Pools previously deleted are no longer considered established.

```
UNSIGNED  PMF_Established_Partition_Pools(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

PMF_Partition_Pool_Pointers

Builds a list of pool pointers, starting at the specified location. The number of pool pointers placed in the list is equivalent to the total number of pools or the maximum number of pointers specified in the call.

```
UNSIGNED  PMF_Partition_Pool_Pointers(NU_PARTITION_POOL *pointer_list,
                                       UNSIGNED maximum_pointers)
```

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

PMF_Partition_Pool_Information

This function returns information about the specified partition pool. However, if the supplied partition pool pointer is invalid, the function simply returns an error status.

```
STATUSPMF_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)
```

Functions Called

```
[TCT_Check_Stack]
TCT_System_Protect
TCT_Unprotect
```

PMI_Initialize

This function initializes the data structures that control the operation of the Partition Memory component. There are no partition pools initially.

```
VOID  PMI_Initialize(VOID)
```

Functions Called

```
None
```

Dynamic Memory Component (DM)

The Dynamic Memory Component (DM) is responsible for processing all Nucleus PLUS dynamic memory facilities. A Nucleus PLUS dynamic memory pool contains a user-specified number of bytes. The memory location of the pool is determined by the application. Tasks may suspend while waiting for enough dynamic memory to become available. Dynamic pools are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about dynamic memory pools.

Dynamic Memory Files

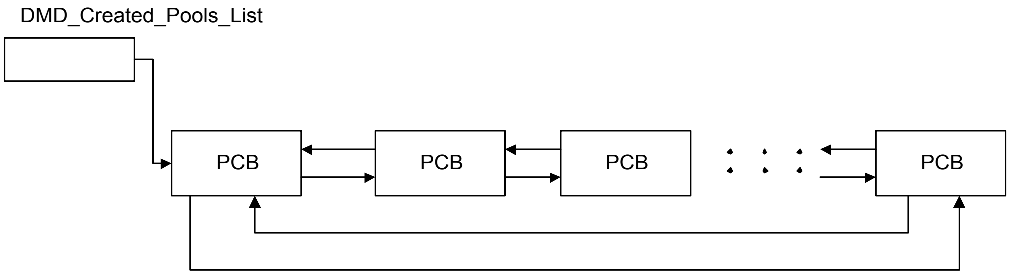
The Dynamic Memory Component (DM) consists of seven files. Each source file of the Dynamic Memory Component is defined below.

Field	**Description**
`DM_DEFS.H`	This file contains constants and data structure definitions specific to the DM
`DM_EXTR.H`	All external interfaces to the DM are defined in this file
`DMD.C`	Gloabal data structures for the DM are defined in this file.
`DMI.C`	This file contains the initialization function for the DM
`DMF.C`	This file contains the information gathering functions for the DM
`DMC.C`	This file contains all of the core functions of the DM. Functions that handle basic allocate-memory and deallocat-memory services are defined in this file.
`DMCE.C`	This file contains the error checking function interfaces for the core functions defined in `DMC.C`.

Dynamic Memory Data Structures

Created Dynamic Memory List

Nucleus PLUS dynamic memory pools may be created and deleted dynamically. The Dynamic Memory Control Block (PCB) for each created dynamic memory pool is kept on a doubly linked, circular list. Newly created dynamic memory pools are placed at the end of the list, while deleted dynamic memory pools are completely removed from the list. The head pointer of this list is `DMD_Created_Pools_List`.



Created Dynamic Memory List Protection

Nucleus PLUS protects the integrity of the Created Dynamic Memory List from competing tasks and/or HISRs. This is done by using an internal protection structure called `DMD_List_Protect`. All dynamic memory creation and deletion is done under the protection of `DMD_List_Protect`.

Field Declarations

```
TC_TCB *tc_tcb_pointer
UNSIGNED tc_thread_waiting
```

Field Summary

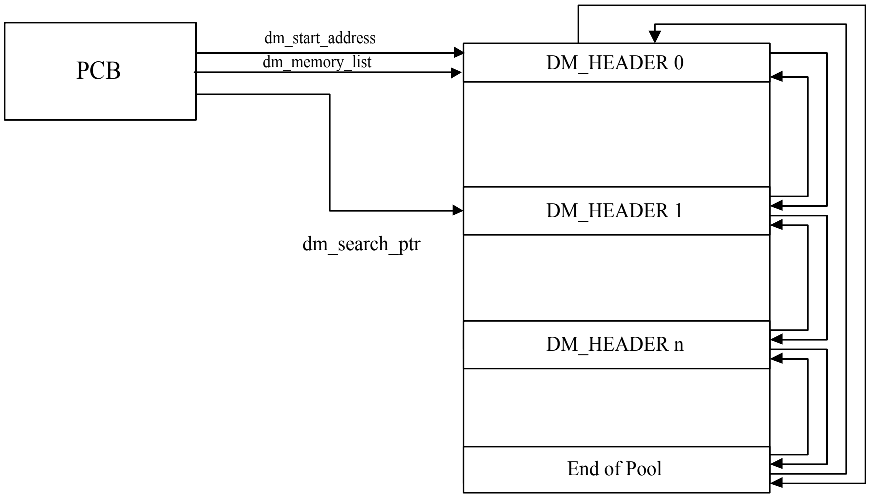
Field	**Description**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Dynamic Pools

The total number of currently created Nucleus PLUS dynamic memory pools is contained in the variable `DMD_Total_Pools`. The content of this variable corresponds to the number of PCBs on the created list. Manipulation of this variable is also done under the protection of `DMD_List_Protect`.

Available Memory List

The Available Memory List is a doubly linked, `NULL` terminated, circular list, which contains the available dynamic memory blocks. The PCB contains pointers to the starting address of the list as well as the next available block in the list. A search pointer is also contained in the PCB. It linearly searches for and accumulates available memory blocks in order to fill memory requests. Allocated blocks are removed from the front of the list and deallocated blocks are placed back in the list at the point where they came from. Each block includes a header that links the various blocks together.



Dynamic Pool Control Block

The Dynamic Memory Pool Control Block `DM_PCB` contains the starting address of the current memory pool and other fields necessary for processing dynamic memory pool requests.

Field Declarations

```
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
```

Field Summary

Field	**Description**
`dm_created`	This is the link node structure for dynamic memory pools. It is linked into the created dynamic pools list, which is a doubly linked, circular list.
`dm_protect`	A pointer to the protection structure for the dynamic memory pool.
`dm_id`	This holds the internal dynamic memory pool identification of 0x44594E41, which is equivalent to ASCII DYNA.
`dm_name`	This is the user-specified, 8 character name for the dynamic memory pool.
`*dm_start_address`	This is the starting address of the current dynamic memory pool.
`dm_pool_size`	Holds the size of the dynamic memory pool.
`dm_min_allocation`	The minimum number of bytes to be allocated in a block.
`dm_available`	This is the total number of bytes available for use in the current memory pool.
`*dm_memory_list`	A list of the memory blocks in the current memory pool.
`*dm_search_ptr`	The search pointer used for locating a dynamic memory pool header.
`dm_fifo_suspend`	A flag that determines whether tasks suspend in fifo or priority order.
`dm_padding`	This is used to align the dynamic memory pool structure on an even boundary. In some ports this field is not used.
`dm_tasks_waiting`	Indicates the number of tasks that are currently suspended on a dynamic memory pool.
`*dm_suspension_list`	The head pointer of the dynamic memory pool suspension list. If no tasks are suspended, this pointer is `NULL`.

Dynamic Memory Pool Header Structure

The dynamic header structure `DM_HEADER` is placed at the beginning of each available memory block. Each header contains pointers to both the next available memory block and the previous available memory block. The last block's next pointer points to a null terminator. Each dynamic memory header also contains a pointer to its PCB.

Field Declarations

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

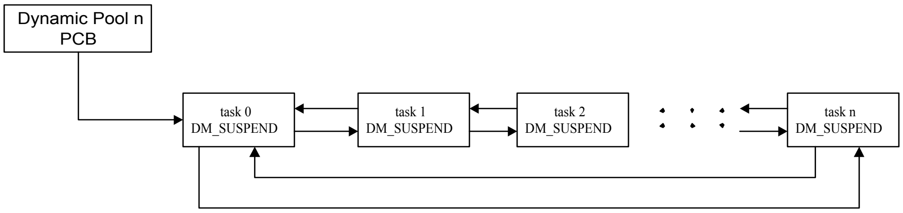
Field Summary

Field	**Description**
`*dm_next_memory`	A pointer to the next memory block in the available list.
`*dm_previous_memory`	A pointer to the previous memory block in the available list.
`dm_memory_free`	A flag that indicates if the current memory block is free.
`dm_memory_pool`	A pointer to the PCB, which this memory block belongs to.

Dynamic Memory Pool Suspension Structure

Tasks can suspend on empty and full dynamic memory pool conditions. During the suspension process a `DM_SUSPEND_STRUCT` structure is built. This structure contains information about the task and the task's dynamic pool request at the time of suspension. This suspension structure is linked onto the PCB in a doubly linked, circular list and is allocated off of the suspending task's stack. There is one suspension block for every task suspended on the dynamic memory pool.

The order of the suspension block placement on the suspend list is determined at dynamic pool creation. If a FIFO suspension was selected, the suspension block is added to the end of the list. Otherwise, if priority suspension was selected, the suspension block is placed after suspension blocks for tasks of equal or higher priority.



Field Declarations

```
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
```

Field Summary

Field	**Description**
`dm_suspend_link`	A link node structure for linking with other suspended blocks. It is used in a doubly linked, circular suspension list.
`*dm_memory_pool`	A pointer to the dynamic memory pool structure.
`dm_request_size`	Contains the size of the requested memory block.
`*dm_suspended_task`	A pointer to the Task Control Block of the suspended task.
`*dm_return_pointer`	The return memory address that has been requested.
`dm_return_status`	The completion status of the task suspended on the dynamic pool.

Dynamic Memory Functions

The following sections provide a brief description of the functions in the Dynamic Memory Component (DM). Review of the actual source code is recommended for further information.

DMC_Create_Memory_Pool

Creates a dynamic memory pool and then places it on the list of created dynamic memory pools. If the list does not exist, then this pool becomes the first item in the dynamic memory pools list.

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

Functions Called

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

DMC_Delete_Memory_Pool

This function deletes a dynamic memory pool and removes it from the list of created memory pools. All tasks suspended on the memory pool are resumed with the appropriate error status. Note that this function does not free any memory associated with either the pool area or the pool control block. That is the responsibility of the application.

```
STATUS DMC_Delete_Memory_Pool(NU_MEMORY_POOL *pool_ptr)
```

Functions Called

```
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

This function allocates memory from the specified dynamic memory pool. If enough dynamic memory is currently available, this function is completed immediately. Otherwise, task suspension is possible.

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

Functions Called

```
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

This function deallocates a previously allocated dynamic memory block. The deallocated dynamic memory block is merged with any adjacent neighbors. This insures that there are no consecutive blocks of free memory in the pool, which makes the search easier. If there is a task waiting for dynamic memory, a determination of whether or not the request can now be satisfied is made after the deallocation is complete.

```
STATUS  DMC_Deallocate_Memory(VOID *memory)
```

Functions Called

```
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

This function is responsible for removing a suspension block from a memory pool. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  DMC_Cleanup(VOID *information)
```

Functions Called

```
CSC_Remove_From_List
```

DMCE_Create_Memory_Pool

This function performs error checking on the parameters supplied to create the dynamic memory pool function.

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

Functions Called

```
DMC_Create_Memory_Pool
```

DMCE_Delete_Memory_Pool

This function performs error checking on the parameters supplied to the delete dynamic memory pool function.

```
STATUS  DMCE_Delete_Memory_Pool(NU_MEMORY_POOL *pool_ptr)
```

Functions Called

```
DMC_Delete_Memory_Pool
```

DMC_Allocate_Memory

This function allocates memory from the specified dynamic memory pool. If enough dynamic memory is currently available, this function is completed immediately. Otherwise, task suspension is possible.

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

Functions Called

```
CSC_Place_on_list
[HIC_Make_History_Entry]
TCC_Suspend_Task
TCC_Task_Priority
[TCT_Check_Stack]
TCT_Currtne_Thread
TCT_Protect
TCT_Set_Suspend_Protect
TCT_System_Protect
TCT_Unprotect
TCT_Unprotect_Specific
```

DMC_Deallocate_Memory

This function deallocates a previously allocated dynamic memory block. The deallocated dynamic memory block is merged with any adjacent neighbors. This insures that there are no consecutive blocks of free memory in the pool, which makes the search easier. If there is a task waiting for dynamic memory, a determination of whether or not the request can now be satisfied is made after the deallocation is complete.

```
STATUS  DMC_Deallocate_Memory(VOID *memory)
```

Functions Called

```
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

This function is responsible for removing a suspension block from a memory pool. It is not called unless a timeout or a task terminate is in progress. Note that protection (the same as at suspension time) is already in effect.

```
VOID  DMC_Cleanup(VOID *information)
```

Functions Called

`CSC_Remove_From_List`

DMCE_Create_Memory_Pool

This function performs error checking on the parameters supplied to create the dynamic memory pool function.

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

Functions Called

`DMC_Create_Memory_Pool`

DMCE_Delete_Memory_Pool

This function performs error checking on the parameters supplied to the delete dynamic memory pool function.

```
STATUS  DMCE_Delete_Memory_Pool(NU_MEMORY_POOL *pool_ptr)
```

Functions Called

`DMC_Delete_Memory_Pool`

DMCE_Allocate_Memory

This function performs error checking on the parameters supplied to the allocate memory function.

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

Functions Called

```
DMC_Allocate_Memory
TCCE_Suspend_Error
```

DMCE_Deallocate_Memory

This function performs error checking on the parameters supplied to the deallocate memory function.

```
STATUS  DMCE_Deallocate_Memory(VOID *memory)
```

Functions Called

```
DMC_Deallocate_Memory
```

DMF_Established_Memory_Pools

Returns the current number of established memory pools. Pools previously deleted are no longer considered established.

```
UNSIGNED DMF_Established_Memory_Pools(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

DMF_Memory_Pool_Pointers

Builds a list of pool pointers, starting at the specified location. The number of pool pointers placed in the list is equivalent to the total number of pools or the maximum number of pointers specified in the call.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

DMF_Memory_Pool_Information

Returns information about the specified memory pool. However, if the supplied memory pool pointer is invalid, the function simply returns an error status.

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

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

`DMI_Initialize`

This function initializes the data structures that control the operation of the Dynamic Memory component. There are no dynamic memory pools initially.

```
VOID  DMI_Initialize(VOID)
```

Functions Called

`None`

Input/Output Driver Component (IO)

The Input/Output Driver Component (IO) is responsible for processing all Nucleus PLUS input/output facilities. A Nucleus PLUS IO Driver Component provides a standard I/O driver interface for initialization, assign, release, input, output, status and terminate requests. This interface is implemented with a common control structure. This enables applications to deal with a variety of peripherals in a similar, if not the same manner. Tasks may suspend while waiting for a peripheral to become available. I/O drivers are dynamically created and deleted by the user. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about input/output drivers.

Input/Output Driver Files

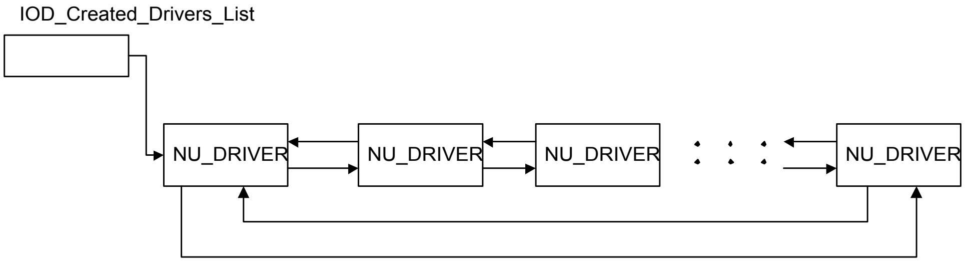
The Input/Output Driver Component (IO) consists of seven files. Each source file of the Input/Output Driver Component is defined below.

File	**Description**
`IO_DEFS.H`	This file contains constants and data structure definitions specific to the IO.
`IO_EXTR.H`	All external interfaces to the IO are defined in this file.
`IOD.C`	Global data structures for the IO are defined in this file.
`IOI.C`	This file contains the initialization function for the IO.
`IOF.C`	This file contains the information gathering functions for the IO.
`IOC.C`	This file contains all of the core functions of the IO. Functions that handle basic input and ouput services are defined in this file.
`IOCE.C`	This file contains the error checking function interfaces for the core functions defined in `IOC.C`.

Input/Output Data Structures

Created Input/Output List

Nucleus PLUS input/output drivers may be created and deleted dynamically. The Input/Output Control Block `(NU_DRIVER)` for each created input/output driver is kept on a doubly linked, circular list. Newly created input/output drivers are placed at the end of the list, while deleted input/output drivers are completely removed from the list. The head pointer of this list is `IOD_Created_Drivers_List`.



Input/Output Driver Control Block

The Input/Output Driver Control Block `(NU_DRIVER)` contains the entry function of the current driver and other fields necessary for processing input/output driver requests.

Field Declarations

```
UNSIGNED words      [NU_DRIVER_SIZE]
CHAR                nu_driver_name[NU_MAX_NAME]
VOID                *nu_info_ptr
UNSIGNED            nu_driver_id
VOID                (*nu_driver_entry)(struct NU_DRIVER_STRUCT*,
                     NU_DRIVER_REQUEST *)
```

Field Summary

Field	**Description**
`words`	This is the link node structure for I/O drivers. It is linked into the created I/O driver's list, which is a doubly linked, circular list.
`nu_driver_name`	This is the user-specified, 8-character name for the I/O driver.
`*nu_info_ptr`	A pointer to the users structure.
`nu_driver_id`	This holds the internal I/O driver identification of `0x494F4452`, which is an equivalent to ASCII IODR.
`(*nu_driver_entry)`	This is the I/0 driver's entry function.

Created Input/Output List Protection

Nucleus PLUS protects the integrity of the Created Input/Output List from competing tasks and/or HISRs. This is done by using an internal protection structure called `IOD_List_Protect`. All input/output creation and deletion is done under the protection of `IOD_List_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	Description
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Input/Output Drivers

The total number of currently created Nucleus PLUS input/output drivers is contained in the variable `IOD_Total_Drivers`. The contents of this variable correspond to the number of `NU_DRIVER`s on the created list. Manipulation of this variable is also done under the protection of `IOD_List_Protect`.

Input/Output Driver Request Structure

The input/output driver request structure `NU_DRIVER_REQUEST` is responsible for passing necessary information to and from a created I/O driver. The type of information in the request is specified by the `nu_function` field in the request structure. Of course, the exact interpretation of this structure depends on the specific driver.

Field Declarations

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

Field Summary

Field	**Description**
`nu_function`	This is the I/O request function code. It can have one of 7 values depending on which request is desired:
	`NU_INITIALIZE      1`
	`NU_ASSIGN          2`
	`NU_RELEASE         3`
	`NU_INPUT           4`
	`NU_OUTPUT          5`
	`NU_STATUS          6`
	`NU_TERMINATE       7`
`nu_timeout`	Holds the timeout on request.
`nu_status`	Contains the status of the request.
`nu_supplemental`	Contains user supplied supplemental information.
`*nu_supplemental_ptr`	A pointer to the driver specific supplemental information [optional].
`nu_request_info`	A union of the structures that are used for driver requests. These requests include initialization, assign, release, input, output, status, and terminate.

Input/Output Driver Initialization Requests

I/O driver's initialization requests are made using the initialization structure `NU_INITIALIZE_STRUCT`. This structure contains information about the driver's base address and the driver's interrupt vector. This request is designated with an `NU_INITIALIZE` value in the nu_function field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

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

Field Summary

Field	**Description**
`*nu_io_address`	A pointer to the base I/O address of the driver.
`nu_logical_units`	Contains the number of logical units in the driver.
`*nu_memory`	A generic memory pointer.
`nu_vector`	Contains the interrupt vector number of the driver.

Input/Output Driver Assignment Requests

I/O driver assignment requests are made using the `NU_ASSIGN_STRUCT` structure. This request is designated with an `NU_ASSIGN` value in the `nu_function` field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

```
UNSIGNED    nu_logical_unit
INT         nu_assign_info
```

Field Summary

Field	**Description**
`nu_logical_unit`	Contains the I/O driver's logical unit number.
`nu_assign_info`	This variable is used for additional I/O driver assign information.

Input/Output Driver Release Requests

I/O driver release requests are made using the `NU_RELEASE_STRUCT` structure. This request is designated with an `NU_RELEASE` value in the `nu_function` field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

```
UNSIGNED    nu_logical_unit
INT         nu_release_info
```

Field Summary

Field	**Description**
`nu_logical_unit`	Contains the I/O driver's logical unit number.
`nu_assign_info`	This variable is used for additional I/O driver release information.

Input/Output Driver Input Requests

I/O driver inputs are made using the `NU_INPUT_STRUCT` structure. This structure contains information about data sent to the driver for processing. This request is designated with an `NU_INPUT` value in the nu_function field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

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

Field Summary

Field	**Description**
`nu_logical_unit`	Contains the I/O driver's logical unit number.
`nu_offset`	An I/O offset used as an offset from the device base I/O address or an offset into the input buffer.
`nu_request_size`	The requested size of the I/O driver input data.
`nu_actual_size`	The actual size of the I/O driver input data.
`*nu_buffer_ptr`	A pointer to the I/O driver input data buffer.

Input/Output Driver Output Requests

I/O driver output requests are made using the `NU_OUTPUT_STRUCT` structure. This structure contains information about data received from the driver. This request is designated with an `NU_OUTPUT` value in the `nu_function` field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

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

Field Summary

Field	**Description**
`nu_logical_unit`	Contains the I/O driver's logical unit number.
`nu_offset`	An I/O offset used as an offset from the device base I/O address or an offset into the output buffer.
`nu_request_size`	The requested size of the I/O driver output data.
`nu_actual_size`	The actual size of the I/O driver output data.
`*nu_buffer_ptr`	A pointer to the I/O driver output buffer.

Input/Output Driver Status Requests

I/O driver status requests are made using the `NU_STATUS_STRUCT` structure. This request is designated with an `NU_STATUS` value in the `nu_function` field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

```
UNSIGNED    nu_logical_unit
VOID        *nu_extra_status
```

Field Summary

Field	**Description**
`nu_logical_unit`	Contains the I/O driver's logical unit number.
`*nu_extra_status`	A pointer to additional status information.

Input/Output Driver Terminate Requests

I/O driver terminate requests are made using the `NU_TERMINATE_STRUCT` structure. This request is designated with an `NU_TERMINATE` value in the nu_function field of the `NU_DRIVER_REQUEST` structure.

Field Declarations

```
UNSIGNED nu_logical_unit
```

Field Summary

Field	Description
`nu_logical_unit`	Contains the I/O driver's logical unit number.

Input/Output Driver Functions

The following sections provide a brief description of the functions in the Input/Output Driver Component (IO). Review of the actual source code is recommended for further information.

IOC_Create_Driver

Creates an I/O driver and places it on the list of created I/O drivers. Note that this function does not actually invoke the driver.

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

Functions Called

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

IOC_Delete_Driver

This function deletes an I/O driver and removes it from the list of created drivers. Note that this function does not actually invoke the driver.

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

Functions Called

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

IOC_Request_Driver

This function sends a user request to the specified I/O driver.

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

Functions Called

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

IOC_Resume_Driver

Resumes a task previously suspended inside an I/O driver. Typically, this function is called from within an I/O driver.

```
STATUS  IOC_Resume_Driver(NU_TASK *task)
```

Functions Called

```
[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

This function suspends a task inside an I/O driver. It is the responsibility of the I/O driver to keep track of tasks waiting inside an I/O driver.

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

Functions Called

```
[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
```

IOCE_Create_Driver

This function performs error checking on the parameters supplied to the I/O driver create function.

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

Functions Called

```
IOC_Create_Driver
```

IOCE_Delete_Driver

This function performs error checking on the parameters supplied to the I/O driver delete function.

```
STATUS  IOCE_Delete_Driver(NU_DRIVER *driver)
```

Functions Called

```
IOC_Delete_Driver
```

IOCE_Request_Driver

This function performs error checking on the parameters supplied to the I/O driver request function.

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

Functions Called

```
IOC_Request_Driver
```

IOCE_Resume_Driver

This function performs error checking on the parameters supplied to the I/O driver resume function.

```
STATUS  IOCE_Resume_Driver(NU_TASK *task)
```

Functions Called

```
IOC_Resume_Driver
TCCE_Validate_Resume
```

IOCE_Suspend_Driver

This function performs error checking on the parameters supplied to the I/O driver suspend function.

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

Functions Called

```
IOC_Suspend_Driver
TCCE_Suspend_Error
```

IOF_Established_Drivers

Returns the current number of established I/O drivers. I/O drivers previously deleted are no longer considered established.

```
UNSIGNED  IOF_Established_Drivers(VOID)
```

Functions Called

```
[TCT_Check_Stack]
```

IOF_Driver_Pointers

Builds a list of driver pointers, starting at the specified location. The number of driver pointers placed in the list is equivalent to the total number of drivers or the maximum number of pointers specified in the call.

```
UNSIGNED  IOF_Driver_Pointers(NU_DRIVER **pointer_list,
                              UNSIGNED maximum_pointers)
```

Functions Called

```
[TCT_Check_Stack]
TCT_Protect
TCT_Unprotect
```

IOI_Initialize

This function initializes the data structures that control the operation of the I/O driver component. There are no I/O drivers initially.

```
VOID  IOI_Initialize(VOID)
```

Functions Called

```
None
```

History Component (HI)

The History Component (HI) is responsible for processing all Nucleus PLUS History facilities. The Nucleus PLUS History Component maintains a circular log of various system activities. Application tasks and HISRs can make entries into the history log. Each entry in the history log contains information about the particular Nucleus PLUS service call and the caller. Please see Chapter 14 of the *Nucleus PLUS Reference Manual* for more detailed information about the History log.

History Files

The History Component (HI) consists of five files. Each source file of the History Component is defined below.

File	**Description**
`HI_DEFS.H`	This file contains constants and data structure definitions specific to the HI.
`HI_EXTR.H`	All external interfaces to the HI are defined in this file.
`HIC.C`	This file contains all of the core functions of the HI. Functions that handle basic enable-history-saving and disable-history-saving services are defined in this file.
`HID.C`	Global data structures for the HI are defined in this file.
`HII.C`	This file contains the initialization function for the HI.

History Data Structures

History Enable

Nucleus PLUS History entries may be made dynamically. The History Enable flag indicates whether or not history saving is enabled. If this value is `NU_FALSE`, history saving is disabled. Otherwise, history saving is enabled and an appropriate entry will be made in the history log as required.

Write Index

The index of the next entry into the Nucleus PLUS History table is contained in the variable `HID_Write_Index`. The contents of this variable correspond to the location of the index of the next available entry in the History table. Manipulation of this variable is also done under the protection of `HID_History_Protect`.

Read Index

The index of the oldest entry into the Nucleus PLUS History table is contained in the variable `HID_Read_Index`. The contents of this variable correspond to the location of the index of the oldest entry in the History table. Manipulation of this variable is also done under the protection of `HID_History_Protect`.

History Table Protection

Nucleus PLUS protects the integrity of the History Table from competing tasks and/or HISRs. This is done by using an internal protection structure called `HID_History_Protect`. All History enabling and disabling is done under the protection of `HID_History_Protect`.

Field Declarations

```
TC_TCB      *tc_tcb_pointer
UNSIGNED    tc_thread_waiting
```

Field Summary

Field	**Declarations**
`tc_tcb_pointer`	Identifies the thread that currently has the protection.
`tc_thread_waiting`	A flag indicating that one or more threads are waiting for the protection.

Total Entries

The total number of entries in the Nucleus PLUS History Table is contained in the variable `HID_Entry_Count`. The contents of this variable correspond to the number of valid entries in the History table. Manipulation of this variable is also done under the protection of `HID_History_Protect`.

History Table Structure

The History Table Structure `HI_HISTORY_ENTRY` contains the starting index of the current history entry and other fields necessary for processing History requests.

Field Declarations

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

Field Summary

Field	**Description**
`hi_id`	This is the index in the table for History entries. It is a simple array consisting only of `HI_HISTORY_ENTRY` structures.
`hi_caller`	The entity that made the entry into the History log. This can be a task, a HISR or the initialization process.
`hi_param1`	The first parameter for storing logged history information
`hi_param2`	The second parameter for storing logged history information.
`hi_param3`	The third parameter for storing logged history information.
`hi_time`	The current system time in clock ticks.
`*hi_thread`	A pointer to the calling thread.

History Functions

The following sections provide a brief description of the functions in the History Component (HI). Review of the actual source code is recommended for further information.

HIC_Disable_History_Saving

This function disables the history saving function.

```
VOID  HIC_Disable_History_Saving(VOID)
```

Functions Called

```
TCT_Protect
TCT_Unprotect
```

HIC_Enable_History_Saving

This function enables the history saving function.

```
VOID  HIC_Enable_History_Saving(VOID)
```

Functions Called

```
TCT_Protect
TCT_Unprotect
```

HIC_Make_History_Entry_Service

This function makes an application entry in the history table.

```
VOID  HIC_Make_History_Entry_Service(UNSIGNED param1, UNSIGNED param2,
                                     UNSIGNED param3)
```

Functions Called

```
HIC_Make_History_Entry
```

HIC_Make_History_Entry

This function makes an entry in the next available location in the history table, (if history saving is enabled).

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

Functions Called

```
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

This function retrieves the next oldest entry in the history table. If no more entries are available, an error status is returned.

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

Functions Called

```
TCT_Protect
TCT_Unprotect
```

HII_Initialize

This function initializes the data structures that control the operation of the History component.

```
VOID  HII_Initialize(VOID)
```

Functions Called

```
None
```

Error Component (ER)

The Error Component (ER) is responsible for processing all Nucleus PLUS System Errors. The Nucleus PLUS Error Component is a common error handling routine that handles fatal system-error conditions. System processing is transferred to this component when a fatal error occurs. The routine then creates an appropriate ASCII error message. This serves to inform the user about the type of error. The system is then trapped by an infinite loop. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about Error Management.

Error Files

The Error Component (ER) consists of four files. Each source file of the Error Component is defined below.

File	Description
`ER_EXTR.H`	All external interfaces to the ER are defined in this file.
`ERC.C`	This file contains the core function of the ER. The function that handles the basic system error service is defined in this file.
`ERD.C`	Global data structures for the ER are defined in this file.
`ERI.C`	This file contains the initialization function for the ER.

Error Data Structures

Error Codes

Nucleus PLUS errors are detected through the use of error codes. When the system determines an error condition exists, it determines the type of error through the use of an error code. The error code value is placed in the variable `ERD_Error_Code`. Nucleus PLUS error codes are listed below.

Code	Constant	Description
`1`	`NU_ERROR_CREATING_TIMER_HISR`	An error occurred creating the timer HISR.
`2`	`NU_ERROR_CREATING_TIMER_TASK`	An error occurred creating the timer task.
`3`	`NU_STACK_OVERFLOW`	A task or HISR stack overflow occurred.
`4`	`NU_UNHANDLED_INTERRUPT`	An interrupt occurred prior to a LISRregistration.

Error String

Nucleus PLUS reports errors in the form of an ASCII string. This string is built and stored in the variable `ERD_Error_String`. The contents of this variable correspond to the ASCII version of the error code that was reported by the system when the error occurred. This string is only produced if the conditional compilation flag `NU_ERROR_STRING` was used to compile `ERD.C`, `ERI.C`, and `ERC.C` inclusively.

Error Functions

The following sections provide a brief description of the functions in the Error Component (ER). Review of the actual source code is recommended for further information.

ERC_System_Error

This function processes system errors detected by various system components. Typically an error of this type is considered fatal.

```
VOID  ERC_System_Error(INT error_code)
```

Functions Called

`None`

`ERI_Initialize`

This function initializes the data structures of the Error Management Component.

```
VOID  ERI_Initialize(VOID)
```

Functions Called

`None`

License Component (LI)

The License Component (LI) is responsible for processing all Nucleus PLUS License facilities. The Nucleus PLUS License Component is a common License handling routine that stores and reports information about the customer license and the customer's serial number. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about License Management.

License Files

The License Component (LI) consists of two files. Each source file of the License Component is defined below.

File	**Description**
`LIC.C`	This file contains the core function of the LI. The function that handles the basic system license reporting service is defined in this file.
`LID.C`	Global data structures for the LI are defined in this file.

License Data Structures

License String

Nucleus PLUS reports Licenses in the form of an ASCII string. This string is stored in the variable `LID_License_String`. The contents of this variable include customer license information and the customer's serial number.

License Functions

The following sections provide a brief description of the functions in the License Component (LI). Review of the actual source code is recommended for further information.

`LIC_License_Information`

This function returns a pointer to the license information string. The information string identifies the customer and product line Nucleus PLUS is licensed for.

```
CHAR  *LIC_License_Information(VOID)
```

Functions Called

`None`

Release Component (RL)

The Release Component (RL) is responsible for processing all Nucleus PLUS Release facilities. The Nucleus PLUS Release Component is a routine that is dedicated to storing and reporting release information. This information includes the current version and release number of the Nucleus PLUS software. Please see Chapter 3 of the *Nucleus PLUS Reference Manual* for more detailed information about Release Management.

Release Files

The Release Component (RL) consists of two files. Each source file of the Release Component is defined below.

File	Description
`RLC.C`	This file contains the core function of the RL. The function that handles the basic system release reporting service is defined in this file.
`RLD.C`	Global data structures for the RL are defined in this file.

Release Data Structures

Release String

Nucleus PLUS reports Releases in the form of an ASCII string. This string is stored in the variable `RLD_Release_String`. This variable contains a description of the current release of the Nucleus PLUS software.

Special String

Nucleus PLUS reports miscellaneous information in the form of an ASCII string. This string is stored in the variable `RLD_Special_String`. This variable contains information about the origins of the Nucleus PLUS system.

Release Functions

The following sections provide a brief description of the functions in the Release Component (RL). Review of the actual source code is recommended for further information.

RLC_Release_Information

This function returns a pointer to the release information string. The information string identifies the current version of Nucleus PLUS.

```
CHAR  *RLC_Release_Information(VOID)
```

Functions Called

`None`

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®

A

Nucleus PLUS Constants

This appendix contains all Nucleus PLUS constants referenced in Chapter 4 (Nucleus PLUS Services) of the Reference Manual. The constants are first listed alphabetically and then by value.

Nucleus PLUS Constants (Alphabetical)

Name	Decimal Value	Hex Value
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_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

Nucleus PLUS Constants (Value)

Name	Decimal Value	Hex Value
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_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

Accelerated Technology
Embedded Systems Division of
Mentor Graphics

B

Fatal System Errors

This appendix contains all Nucleus PLUS fatal system error constants. If a fatal system error occurs, one of these constants is passed to the fatal error handling function, `ERC_System_Error`.

If the system error is `NU_STACK_OVERFLOW`, the currently executing thread's stack is too small. The current thread can be identified by examination of the global variable `TCD_Current_Thread`. This contains the pointer to the current thread's control block.

If the system error is `NU_UNHANDLED_INTERRUPT`, an interrupt was received that does not have an associated LISR. The interrupt vector number that caused the system error is stored in the global variable `TCD_Unhandled_Interrupt`.

Nucleus PLUS Fatal System Errors

Name	Decimal Value	Hex Value
`NU_ERROR_CREATING_TIMER_HISR`	1	1
`NU_ERROR_CREATING_TIMER_TASK`	2	2
`NU_STACK_OVERFLOW`	3	3
`NU_UNHANDLED_INTERRUPT`	4	4

C

I/O Driver Structure Requests

This appendix contains all standard Nucleus PLUS I/O driver constants and request structures. Chapters 3 and 5 of the *Nucleus PLUS Reference Manual* discuss usage of I/O drivers.

Nucleus PLUS I/O Driver Constants

Name	Decimal Value	Hex Value
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
NU_TERMINATE	7	7

Nucleus PLUS I/O Driver C Structures

```
/* 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;
```

Accelerated Technology®
Embedded Systems Division of
Mentor Graphics®