



Nucleus® Kernel Guide

Release 2013.08

August 2013

**© 2007-2013 Mentor Graphics Corporation
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Nucleus Kernel Components	21
Build Configurations	21
Required Include File	22
Kernel Components	22
Nucleus Kernel Examples	23

Chapter 2

Initialization	25
Initialization Overview	25
Notes on RunLevel Initialization	27
Run-level Initialization APIs	28
NU_RunLevel_Current	29
NU_RunLevel_Status	30
NU_RunLevel_Complete_Wait	31
NU_RunLevel_Next_Start	32
NU_RunLevel_0_Init	33

Chapter 3

Device Manager	35
Device Manager Overview	35
Enabling the Device Manager Module	35
Middleware Device Manager APIs	35
DVC_Dev_ID_Open	36
DVC_Dev_Open	37
DVC_Dev_Close	38
DVC_Dev_Read	39
DVC_Dev_Write	41
DVC_Dev_Ioctl	43
Core Processing Functions	44
DVC_Dev_Register	45
DVC_Dev_Unregister	47
DVC_Reg_Change_Check (<i>Deprecated</i>)	48
DVC_Reg_Change_Notify	51
DVC_Dev_ID_Get	53
DVC_Dev_Labels_Get	54
Supplemental Processing Functions	54
DVS_Label_List_Contains	56
DVS_Label_Append	57
DVS_Label_Replace	58
DVS_Label_Copy	59
DVS_Label_Append_Instance	60

DVS_Label_Remove	61
Macros and Typedefs in dv_defs.h	61
IOCTL_0	62
DV_INVALID_DEV	62
DV_INVALID_SES	62
DV_COMPARE_LABELS	63
DV_GET_LABEL_COUNT	63
DV_CLEAR_LABEL	63
DV_DEV_ID	63
DV_SES_ID	64
DV_DEV_HANDLE	64
DV_DEV_LABEL	64
DV_LISTENER_HANDLE	64
DEV_REGISTER_CALLBACK	65
DEV_UNREGISTER_CALLBACK	65
DV_DRV_OPEN_FUNCTION	65
DV_DRV_CLOSE_FUNCTION	65
DV_DRV_READ_FUNCTION	66
DV_DRV_WRITE_FUNCTION	66
DV_DRV_IOCTL_FUNCTION	66
Data Structures in dv_defs.h	66
DV_DRV_FUNCTIONS	67
IOCTL0_STRUCT	67
DV_APP_REGISTRY_CHANGE	67
Error Codes in dv_defs.h	68
Chapter 4	
Memory Management Unit (MMU)	69
MMU Overview	69
MMU Modules	69
MMU Configuration	69
MMU Initialization	70
MMU Linker Control File	71
MMU and the C++ Service	73
Hardware Memory Regions	74
MMU and Sharing Permissions	75
MMU Module Creation	75
Memory Region Function Reference	76
NU_Attach_Memory_Region	78
NU_Change_Memory_Region_Options	80
NU_Create_Module_Memory_Regions	82
NU_Delete_Memory_Region	84
NU_Local_Change_Memory_Region_Options	86
NU_Local_Share_Memory_Region	88
NU_Memory_Region_Information	90
NU_Memory_Region_Phys_Base_Addr	92
NU_Memory_Region_Pointers	93
NU_Memory_Regions	95

Table of Contents

NU_Place_Memory_Region.....	97
NU_Share_Memory_Region	100
NU_Shared_Memory_Region_Pointers	103
NU_Shared_Memory_Regions.....	105
Memory Access, Sharing, and Caching Options	106
Module Support Function Reference.....	107
NU_Bind_Module_HISR	109
NU_Bind_Module_Task.....	111
NU_Change_Module_Options	113
NU_Clear_BSS.....	114
NU_Copy_DATA.....	115
NU_Create_Module	117
NU_Current_Module_Pointer.....	120
NU_Delete_Module	121
NU_Detach_Memory_Region	123
NU_Established_Modules	125
NU_Get_Default_Exception_Handler	126
NU_Get_Module_Memory_Pool	127
NU_Module_Established_HISRs.....	128
NU_Module_Established_Tasks.....	129
NU_Module_Exception_Handler	130
NU_Module_HISR_Pointers	131
NU_Module_Information.....	132
NU_Module_Memory_Region_Pointers	134
NU_Module_Memory_Regions	136
NU_Module_Pointers	138
NU_Module_Task_Pointers	140
NU_Read_Module_Options	141
NU_Register_Exception_Handler	142
NU_Set_Default_Exception_Handler.....	144
NU_Start_Module.....	145
NU_Stop_Module.....	147
NU_Unbind_Module_HISR.....	149
NU_Unbind_Module_Task.....	150
NU_Virtual_To_Physical	151
Module Support Macro Definitions.....	152
NU_NEXT_PAGE_BOUNDARY.....	153
NU_PAGE_EXPAND_SIZE	154
NU_PAGE_TRUNCATE_SIZE.....	155
NU_PREV_PAGE_BOUNDARY	156
Module Service Options	156
User-Defined Routines.....	157
module_cleanup	158
module_init.....	159
module_exception.....	161
Mode Switching Macros	161
NU_IS_SUPERVISOR_MODE.....	163
NU_SUPERV_USER_VARIABLES.....	164
NU_SUPERVISOR_MODE.....	165

NU_SUPERVISOR_MODE_ISR	166
NU_USER_MODE.....	167
NU_USER_MODE_ISR.....	168
Chapter 5	
Nucleus Plus.....	169
Nucleus Plus Overview	169
Task Control.....	169
Task States	169
Preemption	170
Relinquish	170
Time Slicing	170
Timer Interrupt	170
Dynamic Task Creation	170
Task Determinism.....	171
Stack Checking	171
Task Information.....	171
Priority	171
Task Control Services Function Reference	173
NU_Change_Preemption	174
NU_Change_Priority	175
NU_Change_Time_Slice	176
NU_Check_Stack	177
NU_Create_Task.....	178
NU_Create_Auto_Clean Task	181
NU_Current_Task_Pointer.....	184
NU_Delete_Task.....	185
NU_Established_Tasks.....	187
NU_Relinquish	188
NU_Reset_Task	189
NU_Resume_Task	191
NU_Sleep	192
NU_Suspend_Task	193
NU_Task_Information	194
NU_Task_Pointers	197
NU_Terminate_Task.....	199
Dynamic Memory	200
Dynamic Memory Suspension	200
Dynamic Memory Component Dynamic Creation.....	201
Dynamic Memory Determinism.....	201
Dynamic Memory Pool Information.....	201
Dynamic Memory Services Function Reference	201
NU_Add_Memory	203
NU_Allocate_Memory	205
NU_Allocate_Aligned_Memory.....	207
NU_Create_Memory_Pool.....	210
NU_Deallocate_Memory	212
NU_Delete_Memory_Pool.....	213

Table of Contents

NU_Established_Memory_Pools	215
NU_Memory_Pool_Information.	216
NU_Memory_Pool_Pointers.	218
NU_Reallocate_Aligned_Memory	220
NU_Reallocate_Memory	223
Dynamic Memory Example Source Code	225
Partition Memory	228
Partition Memory Suspension.	228
Partition Memory Dynamic Creation	228
Partition Determinism.	229
Partition Information.	229
Partition Memory Services Function Reference	229
NU_Allocate_Partition	230
NU_Create_Partition_Pool	232
NU_Deallocate_Partition	234
NU_Delete_Partition_Pool	235
NU_Established_Partition_Pools	237
NU_Partition_Pool_Information.	238
NU_Partition_Pool_Pointers	240
Partition Memory Example Source Code	241
Mailboxes.	244
Mailbox Suspension	244
Mailbox Broadcast	245
Mailbox Dynamic Creation.	245
Mailbox Determinism.	245
Mailbox Information.	245
Mailbox Services Function Reference	245
NU_Broadcast_To_Mailbox.	246
NU_Create_Mailbox.	248
NU_Delete_Mailbox.	250
NU_Established_Mailboxes	251
NU_Mailbox_Information	252
NU_Mailbox_Pointers	254
NU_Receive_From_Mailbox	256
NU_Reset_Mailbox	258
NU_Send_To_Mailbox.	259
Mailbox Example Source Code	261
Queues	264
Queue Message Size	264
Queue Suspension	265
Queue Broadcast	265
Queue Dynamic Creation	265
Queue Determinism	265
Queue Information	265
Queue Services Function Reference	265
NU_Broadcast_To_Queue	267
NU_Create_Queue	270
NU_Delete_Queue	273
NU_Established_Queues	274

NU_Queue_Information	275
NU_Queue_Pointers	278
NU_Receive_From_Queue	280
NU_Reset_Queue	283
NU_Send_To_Front_Of_Queue	284
NU_Send_To_Queue	287
Queue Example Source Code	289
Event Queue Manager	292
Event Queue Manager Function Reference	293
NU_EQM_Create	294
NU_EQM_Delete	295
NU_EQM_Post_Event	296
NU_EQM_Get_Event_Data	297
NU_EQM_Wait_Event	298
Pipes	299
Pipe Message Size	299
Pipe Suspension	299
Pipe Broadcast	299
Pipe Dynamic Creation	299
Pipe Determinism	300
Pipe Information	300
Pipe Services Function Reference	300
NU_Broadcast_To_Pipe	301
NU_Create_Pipe	304
NU_Delete_Pipe	307
NU_Established_Pipes	308
NU_Pipe_Information	309
NU_Pipe_Pointers	312
NU_Receive_From_Pipe	314
NU_Reset_Pipe	317
NU_Send_To_Front_Of_Pipe	318
NU_Send_To_Pipe	321
Pipe Example Source Code	323
Semaphores	327
Semaphore Suspension	328
Deadlock	328
Priority Inversion	328
Semaphore Dynamic Creation	329
Semaphore Determinism	329
Semaphore Information	329
Semaphore Services Function Reference	329
NU_Create_Semaphore	331
NU_Delete_Semaphore	333
NU_Established_Semaphores	335
NU_Obtain_Semaphore	336
NU_Release_Semaphore	338
NU_Reset_Semaphore	340
NU_Semaphore_Information	342
NU_Get_Semaphore_Owner	344

Table of Contents

NU_Semaphore_Pointers	346
Semaphore Example Source Code	347
Event Groups	351
Event Group Suspension	352
Event Group Dynamic Creation	352
Event Group Determinism	352
Event Group Information	352
Event Group Services Function Reference	352
NU_Create_Event_Group	354
NU_Delete_Event_Group	356
NU_Established_Event_Groups	357
NU_Event_Group_Information	358
NU_Event_Group_Pointers	360
NU_Retrieve_Events	362
NU_Set_Events	364
Event Group Example Source Code	365
Signals	368
Signal Handling Routine	368
Enable Signal Handling	369
Clearing Signals	369
Multiple Signals	370
Signal Determinism	370
Signal Services Function Reference	370
NU_Control_Signals	371
NU_Receive_Signals	372
NU_Register_Signal_Handler	373
NU_Send_Signals	375
Signal Example Source Code	376
Timers	379
Ticks	379
Margin of Error	379
Hardware Requirement	380
Continuous Clock	380
Task Timers	380
Application Timers	380
Re-Scheduling	380
Enable/Disable	380
Reset	381
Timer Dynamic Creation	381
Timer Determinism	381
Timer Information	381
Timer Services Function Reference	381
NU_Control_Timer	383
NU_Create_Timer	385
NU_Delete_Timer	387
NU_Established_Timers	389
NU_Get_Remaining_Time	390
NU_Pause_Timer	392
NU_Reset_Timer	393

NU_Resume_Timer	395
NU_Retrieve_Clock	396
NU_Retrieve_Clock64	397
NU_Set_Clock (<i>Deprecated</i>)	398
NU_Set_Clock64	399
NU_Ticks_To_Time	400
NU_Time_To_Ticks	401
NU_Get_Time_Stamp	402
NU_Timer_Information	403
NU_Timer_Pointers	405
Timer Example Source Code	406
Interrupts	408
Interrupt Protection	408
Low-Level ISR	408
High-Level ISR	409
HISR Information	409
Interrupt Latency	409
Application Interrupt Lockout	410
Direct Vector Access	410
Managed ISRs	410
Interrupt Services Function Reference	411
NU_Activate_HISR	413
NU_Control_Interrupts	415
NU_Create_HISR	417
NU_Current_HISR_Pointer	419
NU_Delete_HISR	420
NU_Established_HISRs	421
NU_HISR_Information	422
NU_HISR_Pointers	424
NU_Local_Control_Interrupts	426
NU_Register_LISR	427
System Diagnostics	429
Error Management	429
NU_ASSERT	430
NU_CHECK	431
Run Time Library	431
Enabling RTL Support	432
Memory Allocation	432
Reentrancy	432
RTL Functions	432
Nucleus Plus Examples	437
Kernel Demo	438
Chapter 6	
Nucleus Kernel User's Manual	441
Kernel User Overview	441
Nucleus Kernel Operation	443
Nucleus System Configuration	443

Table of Contents

Nucleus System Initialization	446
Application Initialization Entry	452
Nucleus Kernel Internals	453
Dynamic Memory	453
Partitioned Memory	454
Interrupts	455
Tasks	458
Timers	462
Semaphores	464
Event Groups	467
Queues	468
Event Queue Manager	469
Pipes	469
Mailboxes	471
Signals	472
 Chapter 7	
Nucleus Processes	475
Nucleus Processes Overview	475
Process IDs and Information	475
Process Initialization and Termination	476
Processes Data Structures	477
NU_LOAD_EXTENSION	477
NU_PROCESS_INFO	478
NU_PROCESS_EXCEPTION	479
Processes APIs	480
NU_Load	482
NU_Start	484
NU_Stop	485
NU_Unload	487
NU_Kill	488
NU_EXPORT_SYMBOL	490
NU_Symbol	491
NU_Symbol_Close	493
NU_Getpid	495
NU_Get_Exit_Code	496
NU_Established_Processes	497
NU_Processes_Information	498
NU_Process_Information	500
NU_Memory_Map	501
NU_Memory_Unmap	504
NU_Memory_Get_ID	505
NU_Memory_Share	507
Process_Exception_Handler	509
main	510
atexit	512
exit	514
abort	515

Shell Commands	515
load	517
tryload	519
start	521
stop	522
unload	523
kill	524
proclist	525
Chapter 8	
Nucleus Processes User Manual	527
Nucleus Processes User Overview	527
Operation	528
Initialization	528
Nucleus Processes Configuration	529
Use Cases	532
Dynamic Linking Techniques	533
Leveraging Open Source Libraries	533
Exporting Symbols	534
Symbol Exports Validation	534
Obtaining and Sharing Memory Resources	535
Internals	535
Processes States	536
Resource Management	536
Dynamic Memory Allocation	537
Process Memory Layout	538
Chapter 9	
Nucleus C++	541
C++ Overview	541
C++ Portability	541
C++ Programming Language Support	541
C++ Limitations	542
Chapter 10	
Nucleus Shell	543
Nucleus Shell Overview	543
Nucleus Shell Function Reference	543
NU_Register_Command	544
NU_Unregister_Command	545
NU_Create_Shell	546
NU_Delete_Shell	548
NU_Get_Shell_Serial_Session_ID	549
Chapter 11	
Nucleus Trace Service	551
Nucleus Trace Overview	551
Nucleus Trace Configuration and Initialization	552

Table of Contents

Trace Initialization	555
Trace Configuration	555
Setup for Kernel Tracing	555
Setup for Application Tracing	557
Setup for Networking Tracing	558
Setup for Storage Tracing	559
Setup for Power Management Services Tracing	560
Setup for Hot-Spot Tracing	560
Controlling the Amount of Trace Data	561
Trace Data Communication	562
Debug Interface for Trace Data Communication	562
Serial or Ethernet (TCP) for Trace Data Communication	563
FTP Comms Interface for Trace Data Communication	563
Using the built-in trace data communication task	563
Using the trace buffer flush API	564
Using application controlled methods to transmit trace data	564
Configuration for Serial Trace Communications	564
Configuration for Ethernet TCP Based Communications	565
Configuration for FTP Comms Trace Communications	565
Trace Function APIs	566
NU_Trace_Initialize	567
NU_Trace_Deinitialize	568
NU_Trace_Arm	569
NU_Trace_Disarm	571
NU_Trace_Mark_I32	572
NU_Trace_Mark_U32	573
NU_Trace_Mark_Float	574
NU_Trace_Mark_String	575
NU_Trace_Mark	576
NU_Trace_Comms_Start	578
NU_Trace_Comms_Stop	579
NU_Trace_Comms_Flush	580
NU_Trace_Comms_Transmit_N_Packets	581

Chapter 12

Dynamic Download	583
Dynamic Download Overview	583
Dynamic Download Limitations	583
Dynamic Download Terminology	584
Dynamic Download Application Development	584
Dynamic Download Function Reference	585
NU_Load_Module_From_File	586
NU_Get_Load_Address	588
NU_Kill_Module	589
NU_Resume_Module	590

Chapter 13

Power Management Services (PMS)	591
PMS Overview	591
Terminology	593
Saving Memory During Hibernate	593
Hibernate Run-level Usage	594
Idle Scheduler API Functions	594
NU_PM_Get_CPU_Counters	595
NU_PM_Start_Tick_Suppress	596
NU_PM_Stop_Tick_Suppress	597
DVFS API Functions	597
NU_PM_DVFS_Control_Transition	598
NU_PM_DVFS_Register	599
NU_PM_DVFS_Unregister	600
NU_PM_DVFS_Update_MPL	601
NU_PM_Get_Current_OP	603
NU_PM_Get_Freq_Count	604
NU_PM_Get_Freq_Info	605
NU_PM_Get_OP_Additional_Info	606
NU_PM_Get_OP_Specific_Info	607
NU_PM_Get_OP_Count	609
NU_PM_Get_OP_VF	610
NU_PM_Get_Voltage_Count	611
NU_PM_Get_Voltage_Info	612
NU_PM_Release_Min_OP	613
NU_PM_Request_Min_OP	614
NU_PM_Set_Current_OP	615
Peripheral State API Functions	615
NU_PM_Get_Power_State	617
NU_PM_Get_Power_State_Count	618
NU_PM_Min_Power_State_Release	619
NU_PM_Min_Power_State_Request	620
NU_PM_Set_Power_State	621
System State Service APIs	622
NU_PM_Emergency_State_Release	623
NU_PM_Emergency_State_Request	624
NU_PM_Get_System_State	625
NU_PM_Get_System_State_Count	626
NU_PM_Get_System_State_Map	627
NU_PM_Map_System_Power_State	628
NU_PM_Unmap_System_Power_State	629
NU_PM_System_Min_State_Release	630
NU_PM_System_Min_State_Request	631
NU_PM_Set_System_State	632
NU_PM_System_State_Init	633
Watchdog API Functions	633
NU_PM_Create_Watchdog	634
NU_PM_Delete_Watchdog	635

Table of Contents

NU_PM_Is_Watchdog_Active	636
NU_PM_Reset_Watchdog	637
NU_PM_Set_Watchdog_Notification	638
NU_PM_Is_Watchdog_Expired	639
Hibernate APIs	640
NU_PM_Set_Hibernate_Level	641
NU_PM_Hibernate_Boot	642
NU_PM_Get_Hibernate_Exit_OP	643
NU_PM_Set_Hibernate_Exit_OP	644
NU_PM_Hibernate_Level_Count	645
NVM APIs	646
NVM_Tgt_Open	647
NVM_Tgt_Close	648
NVM_Tgt_Read	649
NVM_Tgt_Write	650
NVM_Tgt_Reset	652
NVM_Tgt_Info	653
 Chapter 14	
Nucleus Power Services User Manual	655
Power Services User Overview	655
Operation	655
Configuration	655
Power Services Initialization	657
Use Cases	657
Using the Peripheral State (PS)	657
Setting/Clearing minimal PS	658
PS behavior when minimal PS requests are active	659
Creating and Using the System State (SS)	660
Setting/Clearing minimal SS	661
SS behavior when minimal SS requests are active	661
Using the Operating Point (OP)	663
Setting/Clearing Minimal OP	663
OP behavior when Minimal OP Requests are Active	664
Nucleus Power Services Internals	666
Tasks	666
Queues	668
Semaphores	668
Event Groups	669
Timers	669
Nucleus Power Services Hibernate Internals	669
Target Specific Hibernate Driver Functions	670
Hibernate_Tgt_Initialize	671
Hibernate_Tgt_Shutdown	672
Hibernate_Tgt_Standby_Enter	673
Hibernate_Tgt_Stanby_Exit	674
Hibernate_Tgt_Exit	675
Hibernate_Tgt_Get_Region_Count	676

Hibernate_Tgt_Pre_Save	677
Hibernate_Tgt_Get_Region_Info	678
nu_bsp_drvr_hibernate_<platform>_init	679
Target specific Non-Volatile Memory (NVM) Driver Functions	679
NVM_Tgt_Open	680
NVM_Tgt_Close	681
NVM_Tgt_Read	682
NVM_Tgt_Write	683
NVM_Tgt_Reset	684
NVM_Tgt_Info	685
Modifications Required in the <i>.platform</i> file	685
Nucleus Power Services Hibernate Low-level Driver Changes (BSP)	686
Run-level Init Function Changes	686
Hibernate Resume and Hibernate Restore Functions	689

Embedded Software and Hardware License Agreement

List of Figures

Figure 2-1. System Initialization Task Chronology	26
Figure 5-1. Kernel Demo Hardware Platform Output	439
Figure 6-1. Nucleus Kernel Components.	442
Figure 6-2. Nucleus Kernel Initialization at Power Reset	447
Figure 6-3. Platform Specific Nucleus Kernel Initialization	447
Figure 6-4. OS Specific Initialization	448
Figure 6-5. Nucleus Scheduler Initialization	449
Figure 6-6. Device Discovery Mechanism	451
Figure 6-7. Task States and Transitions.	459
Figure 8-1. Process State Transitions.	536
Figure 8-2. Memory Layout without Memory Regions Isolation Support	539
Figure 8-3. Memory Layout with Memory Regions Isolation Support Enabled.	540
Figure 11-1. Nucleus Software Trace High Level View	552
Figure 11-2. Nucleus Configuration Tree for Kernel Tracing.	556
Figure 11-3. Nucleus Configuration Tree for Middleware and Services Tracing.	559
Figure 11-4. Nucleus Configuration Tree for Hot-Spot Tracing.	561
Figure 13-1. PMS Components	592
Figure 14-1. PS APIs With and Without Minimal Requests Active	659
Figure 14-2. SS APIs With and Without Minimal Requests Active	662
Figure 14-3. OP APIs With and Without Minimal Requests Active.	665

List of Tables

Table 1-1. Nucleus Kernel Components	22
Table 2-1. Platform Run-levels	25
Table 4-1. MMU module configuration options	70
Table 4-2. Required Symbols	71
Table 4-3. Memory Access, Sharing, and Caching Options	106
Table 4-4. Module Service Options	156
Table 5-1. Task States	169
Table 5-2. Supported Functions in ctype.h	432
Table 5-3. Supported Constants/Macros/Types in limits.h	433
Table 5-4. Supported Functions in math.h	433
Table 5-5. Supported Constants/Macros/Types in stdarg.h	434
Table 5-6. Supported Functions in stdarg.h	434
Table 5-7. Supported Constants/Macros/Types in stddef.h	434
Table 5-8. Supported Functions in stddef.h	434
Table 5-9. Supported Constants/Macros/Types in stdio.h	435
Table 5-10. Supported Functions in stdio.h	435
Table 5-11. Supported Constants/Macros/Types in stdlib.h	435
Table 5-12. Supported Functions in stdlib.h	435
Table 5-13. Supported Functions in string.h	436
Table 5-14. Supported Constants/Macros/Types in wchar.h	436
Table 5-15. Supported Functions in wchar.h	436
Table 5-16. Supported Functions in time.h	437
Table 5-17. Kernel Demo Services	438
Table 6-1. Nucleus Kernel Core Options	444
Table 6-2. Nucleus RTL Allocation Functions	444
Table 6-3. RTL Component Options	445
Table 6-4. Device Manager Options	446
Table 6-5. Default Run-Level Settings	449
Table 6-6. Task States	458
Table 7-1. Process Exit Codes	476
Table 7-2. NU_LOAD_EXTENSION	478
Table 7-3. NU_PROCESS_INFO	478
Table 7-4. NU_PROCESS_EXCEPTION	479
Table 8-1. Core Configuration Options	529
Table 8-2. Linkload Configuration Options	530
Table 8-3. User Run-tim Configuration Options	531
Table 11-1. Nucleus Trace configuration options	553
Table 12-1. Nucleus Dynamic Download Terminology	584
Table 13-1. Terminology	593
Table 14-1. Core Power Services Configuration Options	656

List of Tables

Table 14-2. OS Tasks Created at Power Services Initialization	666
Table 14-3. Queues created at Power Services Initialization	668
Table 14-4. Semaphores Created at Power Services Initialization	668
Table 14-5. Event Groups Created at Power Services Initialization	669
Table 14-6. Timers Created at Power Services Initialization	669

Chapter 1

Nucleus Kernel Components

The Nucleus kernel components make up the Nucleus kernel package, one of the Nucleus RTOS packages. The kernel components are located at *os\kernel*. Depending on the embedded application, any of these components can be configured for use.

Note



Your source code is initially located in the `<install_root>\nucleus` directory. The source from this directory is copied into the project folder you specify.

This guide describes in detail each kernel component and its APIs. For more information about packages and components, see “Nucleus ReadyStart Configuration” in the *Nucleus ReadyStart Guide* or “Nucleus Source Code Configuration” in the *Nucleus Source Code Guide*.

Build Configurations

By default all components are enabled through the *.metadata* file located at the top level of each component’s installation, for example for Plus: *\os\kernel\plus*. When a component is enabled, it is included in the build. You can create a user configuration file to override the default configuration and exclude a component from the build.

For example, if you have two different applications of Nucleus, one requiring Device Manager, Dynamic Download, and Nucleus C++, and a second application requiring Multi-Task Management, Device Manager, and Power Management Services, you can create two configuration files, one for each application.

For more information, see “Creating a Custom Configuration” in the *Nucleus ReadyStart Guide* or in the *Nucleus Source Code Guide*.

Note




The option `auto_clear_cb` is set to true by default in the *.metadata* file at `<install_root>\nucleus\os\kernel\plus\core`. When true, the kernel will automatically clear all kernel object control blocks during the create call. When false, it becomes the application's responsibility to clear the kernel control blocks/structures before calling the respective kernel APIs such as those that create kernel objects.

Required Include File

To make all the kernel component APIs visible to an application, include the file `kernel\nu_kernel.h` in your application, as in the following statement:

```
#include "kernel/nu_kernel.h"
```

 **Warning** In your applications, use only interfaces, structures, macros, and so on, that are documented within this and other Nucleus guides. There is no guarantee of future support or compatibility for any interface that is not documented.

Kernel Components

Table 1-1 summarizes each Nucleus kernel component and links to additional detailed usage information located in this document.

Table 1-1. Nucleus Kernel Components

Kernel Module	Description	Previously Documented in
Device Manager	Device management APIs for middleware, core functions, and supplemental processing functions.	Nucleus Device Manager APIs
Memory Management Unit (MMU)	MMU based on designating memory areas defined by threads into modules with defined addressing, control, sharing, permissions, caching and initializing information.	Module Support for Nucleus PLUS Reference Manual
Nucleus Plus	Mechanisms for managing competing multiple tasks including mailboxes, queues, pipes, semaphores, event groups, signals, real time external and internal interrupts, and system memory usage.	Nucleus Plus Reference Manual
Nucleus C++	Nucleus C++ services for support of C++ usage in developing an embedded application.	Nucleus C++ Reference Manual
Nucleus Shell	Simple console interface for executing dynamically registered commands.	New
Nucleus Trace Service	Software tracing of Nucleus kernel and user applications.	New

Table 1-1. Nucleus Kernel Components (cont.)

Kernel Module	Description	Previously Documented in
Dynamic Download	Runtime service that enables dynamic loading, execution and unloading of applications to Nucleus.	Dynamic Download User's Guide and Reference Manual
Power Management Services (PMS)	Power management services for six different areas of an embedded system including idle scheduler, DVFS, peripheral state, system state, watchdog, and hibernate.	Power Management Services Reference Manual

Note

The documents have been reorganized in the 2012.3 release. The manuals from prior releases have been grouped by category and relocated to a new document for that category to reduce the total number of documents and to increase navigability between them. The *Nucleus Kernel Guide* is a new document that consists of reference manuals from prior releases that contain information pertaining to the kernel components. The third column of [Table 1-1](#) maps earlier reference manuals to the chapters in the *Nucleus Kernel Guide*.

Nucleus Kernel Examples

This guide offers information about a working [Kernel Demo](#) project that is included in your installation. This guide also provides examples for the following Nucleus kernel components. These examples can be used as a guide for creating your own projects.

- [Nucleus Plus Examples](#)

This section provides a high level overview of how the OS performs initialization/ booting process.

Initialization Overview

Because the OS can be re-configured for different scenarios, it is important to understand its initialization.

Components within the kernel are initialized within run-levels. A single run-level within the OS is a collection of one or more components that are initialized in no pre-determined order. Table 2-1 shows there are 32 available run-levels that can be configured.

Table 2-1. Platform Run-levels

0	None	Components in this run-level are not automatically initialized. They require manual initialization by applications using NU_RunLevel_0_Init API.
1	Device Manager	Initialize the device manager early because many other components have dependencies on it.
2-3	CPU Driver Reserved for BSP devices	Initialize platform devices if device discovery is disabled.
4	Power Management Syslogger	Power Management must have a task running to detect any drivers initialized in later run-levels
5	Buses (USB, SPI, I2C, SDIO, CAN, Serial)	Bus protocols must be initialized in an earlier run-level because they are used by many stacks.
6	File System	The file system is utilized by many other stacks and services.
7	Net Stack and USB class drivers	IP stack (IPv4 and IPv6) and base USB class drivers (function and host - hid, comm, ms).

Table 2-1. Platform Run-levels (cont.)

8	USB drivers	USB user drivers like comm-ethernet, comm-modem, hid-keyboard, and hid-mouse.
9	Networking Protocols and Servers	Networking protocols (IPSec, SNMP, etc) and servers (webserv, telnet/FTP servers, etc) are initialized after the networking stack.
10-11	Reserved for BSP drivers	Initialize platform devices when using device discovery task.
12	C++ services	C++ initialization
13-15	UI, Debug Agent, Trace Service, Profiler, Shell	User Interface (Inflexion, Grafix RS, Input Management, etc) and OS services such as debug agent, POSIX core, profiler (14-15), and shell.
16-20	Application reserved	
31	Application Initialize	Application initialization entry.

Figure 2-1 shows component processing within each run-level over time.

Figure 2-1. System Initialization Task Chronology

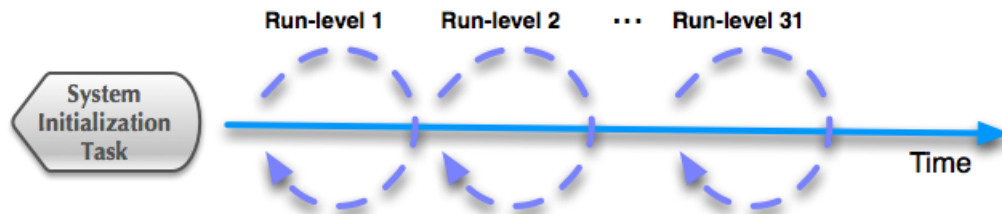


Figure 2-1 shows that all run-level initialization occurs within a single thread. This thread initializes all components at a given run-level *before* initializing components at the next run-level. The process starts at run-level 1 and continues through run-level 31. If a run-level has no components to initialize, processing immediately moves to the next run-level.

Note

Although the order of components within these run-levels is configurable, there are risks associated with changing the order of kernel components. These risks include the possibility of OS components not getting initialized or system failure and/or system crash.

The following rules are associated with the run-level initialization. You must understand and follow them for correct kernel initialization.

- Component initialization routines **cannot** block (sleep, suspend, and so on).
- Component initialization routines **cannot** have ordering dependencies on another component at the same run-level. There is no guaranteed order of initialization within a run-level.
- Component initialization routines can create threads that require blocking services (threads to wait for device registration, and so on).
- Run-level 0 components are **not** auto-initialized as part of the system initialization sequence.
- Components with no run-level designation are not placed in any run-level (that is, no *default* run-level for these components). This means no initialization is required or initialization occurs as part of the application's use of the component.
- Full system (kernel + application) functionality is only guaranteed when all run-levels have completed (that is all kernel services available). This excludes hot-swappable devices that are not present when run-level initialization completes.

Notes on RunLevel Initialization

The RunLevel Initialization and the Device Discovery thread have some important properties:

- The Device Discovery thread is running at an extremely higher priority than the RunLevel Initialization thread and is non-preemptable.
- The RunLevel thread is preemptable. This implies that any device registering in the system via the RunLevel thread will be discovered by the Device Manager instantly (and notified to the client via a callback mechanism), by preempting the RunLevel thread.

Therefore, by the time the RunLevel thread completes, all the devices in the system would get initialized and ready to use (except hot-swappable devices).

Completion of RunLevel initialization does not indicate initialization of all the components within the system. There may be threads created by the RunLevel initialization being suspended on events, etc. due to hot-swappable devices (USB, SD Card, Ethernet, and so on) in the system which were not present during RunLevel initialization. The middleware components provide

appropriate APIs for the availability of devices – for example, the FAT file system provides `NU_Storage_Device_Wait()` API to ensure that storage media has been inserted/mounted and ready to use. Similarly, the Networking stack provides `NETBOOT_Wait_For_Network_Up()` API for network up event.

Caution

The only issue in the RunLevel / Device Discovery method is: if any device initialization ends up with failure status during RunLevel Initialization, it cannot be discovered by the Device Manager.

Related Topics

[Initialization Overview](#)

Run-level Initialization APIs

This section lists the run-level initialization APIs, which can be found in the file:

`<install_root>\nucleus\os\services\init\src\rnl_api.c`

- [NU_RunLevel_Current](#)
- [NU_RunLevel_Status](#)
- [NU_RunLevel_Complete_Wait](#)
- [NU_RunLevel_Next_Start](#)
- [NU_RunLevel_0_Init](#)

NU_RunLevel_Current

This function returns the current run-level.

Usage

```
STATUS  NU_RunLevel_Current (INT *runlevel);
```

Arguments

- **runlevel**
Pointer to run-level returned.

Return Values

- **NU_SUCCESS**
Function completed successfully, it returned a valid run-level.
- **NU_NOT_PRESENT**
Run-level initialization has not started.

Example

```
INT      current_runlevel;  
STATUS  status;  
  
/* Get current runlevel */  
status = NU_RunLevel_Current (&current_runlevel);
```

Related Topics

[Run-level Initialization APIs](#)

[NU_RunLevel_Status](#)

NU_RunLevel_Status

This function retrieves the status of a given run-level.

Usage

```
STATUS  NU_RunLevel_Status(INT runlevel);
```

Arguments

- **runlevel**
Run-level to retrieve status for.

Return Values

- **NU_SUCCESS**
Run-level has completed initialization.
- **NU_RUNLEVEL_NOT_STARTED**
Run-level has not started initialization .
- **NU_RUNLEVEL_IN_PROGRESS**
Run-level is in process of initialization.
- **NU_NOT_PRESENT**
No run-level exists for the specified run-level.

Example

```
/* Check if runlevel 1 has completed */  
if (NU_RunLevel_Status(1) == NU_SUCCESS)  
{  
    /* Runlevel has completed */  
    ....  
}
```

Related Topics

[Run-level Initialization APIs](#)

[NU_RunLevel_Current](#)

NU_RunLevel_Complete_Wait

This function waits until the specified run-level has completed, meaning all components within that run-level have finished initialization.

Usage

```
STATUS NU_RunLevel_Complete_Wait(INT      runlevel,
                                UNSIGNED timeout);
```

Arguments

- **runlevel**
Run-level to wait upon.
- **timeout**
Timeout (ticks), NU_SUSPEND, or NU_NO_SUSPEND (immediate completion).

Return Values

- **NU_SUCCESS**
Function completed successfully, run-level completed.
- **NU_TIMEOUT**
Timeout occurred before run-level completed.
- **NU_NOT_PRESENT**
Specified run-level does not exist.

Example

```
STATUS status;

/* Wait up to 1 second for run-level 4 to complete */
status = NU_RunLevel_Complete_Wait(4, NU_PLUS_TICKS_PER_SEC);
```

Related Topics

[Run-level Initialization APIs](#)

[NU_RunLevel_Next_Start](#)

NU_RunLevel_Next_Start

This function starts the initialization of the “next” run-level, which is greater by 1 then the current run-level.

Usage

```
STATUS  NU_RunLevel_Next_Start (VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Function completed successfully, “next” run-level started.
- **NU_RUNLEVEL_IN_PROGRESS**
The current run-level is still being executed.
- **NU_NOT_PRESENT**
No run-level is “next”.

Description

This is only be applicable if run-levels exist that are not configured to "automatically" start. This implies that a run-level "limit" will exist within the system that allows only the 1st N run-levels to be automatically run.

Example

```
STATUS status;  
  
/* Start next runlevel */  
status = NU_RunLevel_Next_Start();
```

Related Topics

[Run-level Initialization APIs](#)

[NU_RunLevel_Complete_Wait](#)

NU_RunLevel_0_Init

This function initializes manually any run-level 0 component or all run-level 0 components from an application.

Usage

```
STATUS  NU_RunLevel_0_Init(const CHAR *compregpath);
```

Arguments

- **compregpath**
Pointer to component registry path. If NULL, it initializes all components in run-level 0.

Return Values

- **NU_SUCCESS**
Function completed successfully, run-level 0 component(s) initialization completed.
- **NU_RUNLEVEL_IN_PROGRESS**
Run-level 0 initialization has already started.
- **NU_NOT_PRESENT**
The specified component is not a run-level 0 component.

Example

```
STATUS status;  
  
/* Initialize the networking stack (ie nu.os.net.stack) */  
status = NU_RunLevel_0_Init("/nu/os/net/stack");  
  
/* Initialize all remaining runlevel 0 components */  
status = NU_RunLevel_0_Init(NU_NULL);
```

Related Topics

[Run-level Initialization APIs](#)

Device Manager Overview

This chapter provides information about the Nucleus Device Manager APIs for middleware, core functions, and supplemental processing functions.

The chapter also describes macros, type definitions, data structures and error codes pertaining to device management.

Enabling the Device Manager Module

You must include the following header files in your code to enable the device manager module:

```
#include "nucleus.h"  
#include "kernel/nu_kernel.h"
```

Middleware Device Manager APIs

- [DVC_Dev_ID_Open](#)
- [DVC_Dev_Open](#)
- [DVC_Dev_Close](#)
- [DVC_Dev_Read](#)
- [DVC_Dev_Write](#)
- [DVC_Dev_Ioctl](#)

DVC_Dev_ID_Open

This function opens a session on a device based on device ID and a set of labels.

Usage

```
STATUS DVC_Dev_ID_Open ( DV_DEV_ID      dev_id,  
                        DV_DEV_LABEL  dev_label_list[],  
                        INT           dev_label_cnt,  
                        DV_DEV_HANDLE *dev_handle_ptr );
```

Arguments

- **dev_id**
The device id of the device you want to open.
- **dev_label_list[]**
The mode used to open the device. This is the way to open the device in its low-level mode for IOCTL specific communication.
- **dev_label_cnt**
The number of labels in the label list.
- **dev_handle_ptr**
Pointer to where the device handle is returned.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_DEV_NOT_REGISTERED**
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- **DV_NO_AVAILABLE_SESSION**
All of the device's session registry is full.
- **Driver's OPEN function**
Driver's OPEN function returned status.

Related Topics

[Middleware Device Manager APIs](#)

DVC_Dev_Open

This function opens a session on a device based on device name.

Usage

```
STATUS DVC_Dev_Open ( DV_DEV_LABEL  *dev_name_ptr,  
                     DV_DEV_HANDLE *dev_handle_ptr);
```

Arguments

- `dev_name_ptr`
Pointer to the label name of the device to open
- `dev_handle_ptr`
Pointer to where the device handle is returned

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Invalid input parameters.
- `DV_DEV_NOT_REGISTERED`
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- `DV_NO_AVAILABLE_SESSION`
All the device's session registry is full.
- Driver's OPEN function
Driver's OPEN function returned status.
- [DVC_Dev_ID_Get](#)
The `DVC_Dev_Open` function calls `DVC_Dev_ID_Get` and returns its status.

Related Topics

[Middleware Device Manager APIs](#)

DVC_Dev_Close

This function closes a session on a device.

Usage

```
STATUS DVC_Dev_Close ( DV_DEV_HANDLE dev_handle );
```

Arguments

- `dev_handle`
Device handle of the device to close.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Invalid input parameters.
- `DV_DEV_NOT_REGISTERED`
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- `DV_SESSION_NOT_OPEN`
The session we are trying to close is not open.
- Driver's CLOSE function
Driver's CLOSE function returned status.

Related Topics

[Middleware Device Manager APIs](#)

DVC_Dev_Read

This function interfaces to the actual device driver's read function.

Usage

```
STATUS DVC_Dev_Read ( DV_DEV_HANDLE dev_handle,  
                      VOID           *buffer_ptr,  
                      UINT32         numbyte,  
                      OFFSET_T       byte_offset,  
                      UINT32         *bytes_read_ptr);
```

Arguments

- **dev_handle**
Device handle of the device to read.
- **buffer_ptr**
Pointer to the input buffer.
- **numbyte**
The size, in bytes, of the input buffer.
- **byte_offset**
The offset, in bytes, from 0. It can be 0.
- **bytes_read_ptr**
Pointer to where the number of bytes read is returned. It can be NU_NULL.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_DEV_NOT_REGISTERED**
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- **DV_SESSION_NOT_OPEN**
The session we are trying to close is not open.
- **Driver's READ function**
Driver's READ function returned status.

Example

```
#if (ESAL_TS_64BIT_SUPPORT == NU_TRUE)  
    typedef UINT64 OFFSET_T;  
#else
```

```
typedef UINT32 OFFSET_T;  
#endif /* (ESAL_TS_64BIT_SUPPORT == NU_TRUE) */
```

Related Topics

[Middleware Device Manager APIs](#)

DVC_Dev_Write

This function interfaces to the actual device driver's write function.

Usage

```
STATUS DVC_Dev_Write ( DV_DEV_HANDLE dev_handle,  
                      VOID          *buffer_ptr,  
                      UINT32        numbyte,  
                      OFFSET_T      byte_offset,  
                      UINT32        *bytes_written_ptr);
```

Arguments

- **dev_handle**
Device handle of the device to read.
- **buffer_ptr**
Pointer to the input buffer.
- **numbyte**
The size, in bytes, of the input buffer.
- **byte_offset**
The offset, in bytes, from 0. It can be 0.
- **bytes_written_ptr**
Pointer to where the number of bytes written is returned. It can be NU_NULL.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_DEV_NOT_REGISTERED**
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- **DV_SESSION_NOT_OPEN**
The session we are trying to close is not open.
- **Driver's WRITE function**
Driver's WRITE function returned status.

Example

```
#if (ESAL_TS_64BIT_SUPPORT == NU_TRUE)  
    typedef UINT64 OFFSET_T;  
#else
```

```
typedef UINT32 OFFSET_T;  
#endif /* (ESAL_TS_64BIT_SUPPORT == NU_TRUE) */
```

Related Topics

[Middleware Device Manager APIs](#)

DVC_Dev_Ioctl

This function interfaces to the actual device driver's ioctl function.

Usage

```
STATUS DVC_Dev_Ioctl ( DV_DEV_HANDLE dev_handle,  
                      INT             ioctl_num,  
                      VOID            *ioctl_data,  
                      INT             ioctl_data_len );
```

Arguments

- **dev_handle**
Device handle of the device to control.
- **ioctl_num**
Ioctl command.
- **ioctl_data**
Optional ioctl data.
- **ioctl_data_len**
Size, in bytes, of the ioctl data.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_DEV_NOT_REGISTERED**
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).
- **DV_SESSION_NOT_OPEN**
The session we are trying to close is not open.
- **Driver's IOCTL function.**
Driver's IOCTL function returned status.

Related Topics

[Middleware Device Manager APIs](#)

Core Processing Functions

- [DVC_Dev_Register](#)
- [DVC_Dev_Unregister](#)
- [DVC_Reg_Change_Check](#) (Deprecated)
- [DVC_Reg_Change_Notify](#)
- [DVC_Dev_ID_Get](#)
- [DVC_Dev_Labels_Get](#)

DVC_Dev_Register

This function registers a device with the Device Manager. It is called by the driver.

Usage

```
DV_DEV_ID DVC_Dev_Register (VOID*           instance_handle,  
                             DV_DEV_LABEL    dev_label_list[],  
                             INT             dev_label_cnt,  
                             DV_DRV_FUNCTIONS *drv_functions_ptr,  
                             DV_DEV_ID      *dev_id_ptr);
```

Arguments

- **instance handle**
Pointer to the 1:1 information needed by the driver. This is also the same pointer that will be passed as a parameter to the open function.
- **dev_label_list[]**
A list of labels that can be attached to a device and is not unique. This label is typically used to search and can be used to open with the combination of a dev_id. This is typically used for internal systems, but can be useful for applications.
- **dev_label_cnt**
The number of labels passed in the dev_label_list.
- **drv_functions_ptr**
Structure of pointers to the driver's function set that will be called by the associated DM API.
- **dev_id_ptr**
Pointer to where the device ID will be returned.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_NO_AVAILABLE_DEV_ID**
Registry is full.
- **NU_Set_Events**
NU_Set_Events function returned status.

Example

```
/* Application level structure for DVC_Dev_Register() */  
typedef struct _dv_drv_functions_struct  
{
```

```
    DV_DRV_OPEN_FUNCTION drv_open_ptr;
    DV_DRV_CLOSE_FUNCTION drv_close_ptr;
    DV_DRV_READ_FUNCTION drv_read_ptr;
    DV_DRV_WRITE_FUNCTION drv_write_ptr;
    DV_DRV_IOCTL_FUNCTION drv_ioctl_ptr;
} DV_DRV_FUNCTIONS;

/* Typedefs for driver open, close, read, write and ioctl functions. */
typedef STATUS (*DV_DRV_OPEN_FUNCTION) (VOID* instance_handle,
    DV_DEV_LABEL label_list[],
    INT label_cnt, VOID* *session_handle_ptr);
typedef STATUS (*DV_DRV_CLOSE_FUNCTION) (VOID* session_handle);
typedef STATUS (*DV_DRV_READ_FUNCTION) (VOID* session_handle,
    VOID* buffer,
    UINT32 numbyte, OFFSET_T byte_offset,
    UINT32 *bytes_read_ptr);
typedef STATUS (*DV_DRV_WRITE_FUNCTION) (VOID* session_handle,
    const VOID* buffer,
    UINT32 numbyte, OFFSET_T byte_offset,
    UINT32 *bytes_written_ptr);
typedef STATUS (*DV_DRV_IOCTL_FUNCTION) (VOID* session_handle,
    INT ioctl_num, VOID* ioctl_data, INT ioctl_data_len);
```

Related Topics

[Core Processing Functions](#)

DVC_Dev_Unregister

This function removes a device from the Device Manager registry. It is called by the driver, for example when a USB device is unplugged.

Usage

```
STATUS DVC_Dev_Unregister (DV_DEV_ID dev_id,  
                          VOID*      *instance_handle_ptr);
```

Arguments

- **dev_id**
The device id from the registry table.
- **instance_handle_ptr**
Pointer to the return value for the instance handle.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Invalid input parameters.
- **DV_DEV_NOT_REGISTERED**
This device has not been registered.

Related Topics

[Core Processing Functions](#)

DVC_Reg_Change_Check (*Deprecated*)

This function allows the user to know when the device registry changes and, if requested, will suspend until a change occurs.

Note



DVC_Reg_Change_Check has been deprecated, it is recommended to use DVC_Reg_Change_Notify instead of this API. For further details see the definition of the DVC_Reg_Change_Notify function.

Usage

```
STATUS DVC_Reg_Change_Check (DV_APP_REGISTRY_CHANGE *app_struct_ptr,  
                             UNSIGNED suspend);
```

Arguments

- `app_struct_ptr`
A structure containing pointers to lists and counts where the device registry changes can be returned.
- `suspend`
Flag to suspend if no registry changes are found. Choices are `NU_SUSPEND`, `NU_NO_SUSPEND` or a timeout value in system ticks. These are defined in *nucleus.h*.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Invalid input parameters.
- `DV_INVALID_REG_PARAMS`
Error with registered device id parameters.
- `DV_INVALID_UNREG_PARAMS`
Error with unregistered device id parameters.
- `DV_REG_LIST_TOO_SMALL`
More registered device ids were found than could be returned.
- `DV_UNREG_LIST_TOO_SMALL`
More unregistered device ids were found than could be returned.
- `DV_DEV_LIST_TOO_SMALL`
More device ids were found than could be returned.

- [DMC_Allocate_Memory](#)
returned status
- [NU_Create_Semaphore](#)
NU_Create_Semaphore returned status
- [NU_Obtain_Semaphore](#)
NU_Obtain_Semaphore returned status

Example

```
/* Application level structure for DVM_Reg_Change_Check() */
typedef struct _dv_app_registry_change_struct
{

    /* Maximum number of device ids that can be returned in each of the
    known, reg and unreg lists. */
    INT max_id_cnt;

    /* Total number of registry changes discovered. */
    INT registry_changes;

    /* Pointer to a list of labels that qualify the attributes of the
    device being searched for. This can be an empty list or NULL as long
    as dev_label_cnt is also 0. */
    DVM_DEV_LABEL *dev_label_list_ptr;

    /* Number of labels in the label list. If dev_label_list_ptr is NULL
    this must be 0. */
    INT dev_label_cnt;

    /* Pointer to a list of device ids that are already known. This can
    be an empty list. On return, the list will contain the total number
    of device ids found. */
    DVM_DEV_ID *known_id_list_ptr;

    /* Pointer to a count of how many device ids that are being passed
    in the known id list. The value it points to can be 0. On return, the
    value it points to will contain the total number of device ids
    found. */
    INT *known_id_cnt_ptr;

    /* Pointer to an empty list that, on return, will contain the number
    of registered device ids found. This can be NULL as long as
    reg_id_cnt_ptr is also NULL. */
    DVM_DEV_ID *reg_id_list_ptr;
```

```
    /* Pointer to a count that, on return, will contain the number of
    registered device ids found. This can be NULL as long as
    reg_id_list_ptr is also NULL. */
    INT *reg_id_cnt_ptr;

    /* Pointer to an empty list that, on return, will contain the number
    of unregistered device ids found. This can be NULL as long as
    unreg_id_cnt_ptr is also NULL. */
    DVM_DEV_ID *unreg_id_list_ptr;

    /* Pointer to a count that, on return, will contain the number of
    unregistered device ids found. This can be NULL as long as
    unreg_id_list_ptr is also NULL. */
    INT *unreg_id_cnt_ptr;
} DV_APP_REGISTRY_CHANGE;
```

Related Topics

[Core Processing Functions](#)

DVC_Reg_Change_Notify

This API behaves differently when dynamic device discovery is enabled and otherwise. If dynamic device discovery is enabled, it registers a device listener with device manager. Once device manager finds the device of interest, it notifies device registration or un-registration asynchronously.

If dynamic device discovery is disabled, then this function searches for devices in the device registry immediately and lets the caller know if devices of interest are available in the device registry. Otherwise, it returns an error.

Usage

```
STATUS DVC_Reg_Change_Notify(DV_DEV_LABEL dev_label_list[],
                             INT dev_label_cnt,
                             DEV_REGISTER_CALLBACK register_cb,
                             DEV_UNREGISTER_CALLBACK unregister_cb,
                             VOID *context,
                             DV_LISTENER_HANDLE *listener_id)
```

Arguments

- `dev_label_list[]`
List of labels against which the device manager should listen for a device. Upon every device registration and un-registration, the device manager will compare the labels to identify device listeners.
- `dev_label_cnt`
Number of labels in 'dev_label_list'.
- `register_cb`
Callback function to be called in the event of device registration.
- `unregister_cb`
Callback function to be called in the event of device un-registration.
- `context`
Pointer to user supplied 'context'.
- `listener_id`
Pointer to DV_LISTENER_HANDLE containing the listener ID when the function returns.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.

- **DV_MAX_LISTENER_LIMIT_EXCEEDED**
There is no room for creating a new device listener (only returned when dynamic device discovery is enabled).
- **DV_DEV_NOT_REGISTERED**
A device with labels is not registered with the device manager (only returned if dynamic device discovery is disabled).

Description

- By default, dynamic device discovery is enabled, however, you can disable dynamic discovery by setting the default value of the `discovery_task_enable` option to 'false' in the device manager metadata.
- Once dynamic discovery is disabled it is considered the responsibility of the user to resolve race conditions in component run-level.
- In order to save the system from deadlocks, calling self suspension services from device registration and un-registration callback functions is not allowed. For further details about self suspension services, refer to the [Nucleus Plus](#) chapter.

Related Topics

[Core Processing Functions](#)

DVC_Dev_ID_Get

This function searches the device registry for the device ID(s) that match the ones passed in labels list.

Usage

```
STATUS DVC_Dev_ID_Get (DV_DEV_LABEL dev_label_list[],
                      INT dev_label_cnt,
                      DV_DEV_ID dev_id_list[],
                      INT *dev_id_cnt_ptr);
```

Arguments

- `dev_label_list[]`
This parameter is a list of labels to search for a device.
- `dev_label_cnt`
Number of device labels. If 0, then return all device IDs.
- `dev_id_list[]`
The list to save the device id's in.
- `dev_id_cnt_ptr`
Pointer to variable that contains the maximum IDs to return. It is also the place where we return the number of IDs found.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.
- `DV_DEV_LIST_TOO_SMALL`
More device ids were found than could be returned.

Related Topics

[Core Processing Functions](#)

DVC_Dev_Labels_Get

This function returns the labels for the device.

Usage

```
STATUS DVC_Dev_Labels_Get (DV_DEV_ID    dev_id,  
                          DV_DEV_LABEL dev_label_list[],  
                          INT          *dev_label_cnt_ptr );
```

Arguments

- **dev_id**
Device id of the device to search for.
- **ev_label_list[]**
The list to save the labels in.
- **dev_label_cnt_ptr**
Pointer to variable that contains the maximum labels to return and it is also the place where we return the number of labels found.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **DV_INVALID_INPUT_PARAMS**
Any of the input parameters is invalid.
- **DV_DEV_LIST_TOO_SMALL**
More device ids were found than could be returned.
- **DV_DEV_NOT_REGISTERED**
This device has not been registered.

Related Topics

[Core Processing Functions](#)

Supplemental Processing Functions

- [DVS_Label_List_Contains](#)
- [DVS_Label_Append](#)
- [DVS_Label_Replace](#)

- [DVS_Label_Copy](#)
- [DVS_Label_Append_Instance](#)
- [DVS_Label_Remove](#)

DVS_Label_List_Contains

This function checks if a label exists in a label list.

Usage

```
STATUS DVS_Label_List_Contains (DV_DEV_LABEL label_list[],  
                               INT          label_cnt,  
                               DV_DEV_LABEL search_label );
```

Arguments

- `label_list[]`
A list of labels to search
- `label_cnt`
The number of labels in the search list
- `search_label`
The label we are searching for

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_LABEL_NOT_FOUND`
The label searched for was not found.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.

Related Topics

[Supplemental Processing Functions](#)

DVS_Label_Append

This function appends labels to the end of a label list.

Usage

```
STATUS DVS_Label_Append (DV_DEV_LABEL new_label_list[],
                        INT new_label_max,
                        DV_DEV_LABEL old_label_list[],
                        INT old_label_cnt,
                        DV_DEV_LABEL app_label_list[],
                        INT app_label_cnt );
```

Arguments

- `new_label_list[]`
The combined label list.
- `new_label_max`
The maximum number of labels the new list can contain.
- `old_label_list[]`
The original list of labels.
- `old_label_cnt`
The number of labels in the original list.
- `app_label_list[]`
The updated list.
- `app_label_cnt`
The number of labels in the append list.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_LABEL_LIST_TOO_SMALL`
The updated list is larger than the return list can handle.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.

Related Topics

[Supplemental Processing Functions](#)

DVS_Label_Replace

This function replaces a label with a new label.

Usage

```
STATUS DVS_Label_Replace (DV_DEV_LABEL label_list[],  
                          INT label_cnt,  
                          DV_DEV_LABEL search_label,  
                          DV_DEV_LABEL new_label );
```

Arguments

- `label_list[]`
List of labels to search.
- `label_cnt`
Number of labels in the search list.
- `search_label`
Label we are searching for.
- `new_label`
Label that will replace the searched for label.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_LABEL_NOT_FOUND`
Did not find the label we were searching for.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.

Related Topics

[Supplemental Processing Functions](#)

DVS_Label_Copy

This function copies a label.

Usage

```
STATUS DVS_Label_Copy (DV_DEV_LABEL to_label_list[],  
                      INT to_label_max,  
                      DV_DEV_LABEL from_label_list[],  
                      INT from_label_cnt );
```

Arguments

- `to_label_list[]`
The destination address for the copy.
- `to_label_max`
The maximum number of labels that can be copied.
- `from_label_list[]`
The source address for the copy.
- `from_label_cnt`
The number of labels to be copied.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_INVALID_INPUT_PARAMS`
Any of the input parameters is invalid.
- `DV_LABEL_NOT_FOUND`
We did not find the label we were searching for.

Related Topics

[Supplemental Processing Functions](#)

DVS_Label_Append_Instance

This function appends the registry instance label to the end of a label list.

Usage

```
STATUS DVS_Label_Append_Instance (DV_DEV_LABEL new_label_list[],  
                                INT             new_label_max,  
                                DV_DEV_LABEL old_label_list[],  
                                INT             old_label_cnt,  
                                const CHAR     *key );
```

Arguments

- new_label_list[]
The combined label list.
- new_label_max
The maximum number of labels the new list can contain.
- old_label_list[]
The original list of labels.
- old_label_cnt
The number of labels in the original list.
- key
Pointer to the registry key path where the instance label can be found.

Return Values

- NU_SUCCESS
Function completed successfully.
- DV_INVALID_INPUT_PARAMS
Any of the input parameters is invalid.
- DV_LABEL_LIST_TOO_SMALL
The updated list is larger than the return list can handle.
- DV_UNEXPECTED_ERROR
The registry key path is invalid.

Related Topics

[Supplemental Processing Functions](#)

DVS_Label_Remove

This function removes a label from a label list.

Usage

```
STATUS DVS_Label_Remove (DV_DEV_LABEL label_list[],  
                        INT label_cnt,  
                        DV_DEV_LABEL remove_label );
```

Arguments

- `label_list[]`
A list of labels to search.
- `label_cnt`
The number of labels in the search list.
- `remove_label`
The label we want to remove.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `DV_LABEL_NOT_FOUND`
Did not find the label we were searching for.

Related Topics

[Supplemental Processing Functions](#)

Macros and Typedefs in `dv_defs.h`

- [IOCTL_0](#)
- [DV_INVALID_DEV](#)
- [DV_INVALID_SES](#)
- [DV_COMPARE_LABELS](#)
- [DV_CLEAR_LABEL](#)
- [DV_DEV_ID](#)
- [DV_SES_ID](#)
- [DV_DEV_HANDLE](#)
- [DV_DEV_LABEL](#)

- [DV_LISTENER_HANDLE](#)
- [DEV_REGISTER_CALLBACK](#)
- [DEV_UNREGISTER_CALLBACK](#)
- [DV_DRV_OPEN_FUNCTION](#)
- [DV_DRV_CLOSE_FUNCTION](#)
- [DV_DRV_READ_FUNCTION](#)
- [DV_DRV_WRITE_FUNCTION](#)
- [DV_DRV_IOCTL_FUNCTION](#)

IOCTL_0

This macro defines IOCTL0.

```
#define IOCTL0      0
#define DV_IOCTL_INVALID_LENGTH -1
#define DV_IOCTL_INVALID_MODE -2
#define DV_IOCTL_INVALID_CMD -3
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_INVALID_DEV

This macro defines the illegal dev id flag value.

```
#define DV_INVALID_DEV      ((DV_DEV_ID)-1)
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_INVALID_SES

This macro defines the illegal session id flag value.

```
#define DV_INVALID_SES      ((DV_SES_ID)-1)
#define DV_INVALID_HANDLE ((DV_DEV_HANDLE)-1)
#define DV_SES_CLOSED      0
#define DV_SES_LOCKED      1
#define DV_SES_OPEN        2
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_COMPARE_LABELS

This macro compares two labels.

```
#define DV_COMPARE_LABELS (label1,label2)  
(memcmp((VOID*)label1,(*VOID)label2,16) == 0)
```

DV_GET_LABEL_COUNT

This macro calculates the size of the label array.

```
#define DV_GET_LABEL_COUNT (label_array)  
((INT)(sizeof(label_array)/sizeof(DV_DEV_LABEL)))
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_CLEAR_LABEL

This macro clears a label.

```
#define DV_CLEAR_LABEL (label)((VOID)memset(label, 0, 16))
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_DEV_ID

This macro defines the device manager device ID.

```
typedef INT16 DV_DEV_ID;
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_SES_ID

This macro defines the device manager session ID.

```
typedef INT16 DV_SES_ID;
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_DEV_HANDLE

This macro defines the device manager device handle.

```
typedef INT32 DV_DEV_HANDLE;
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_DEV_LABEL

This macro defines the device label (GUID).

```
typedef struct _dv_dev_label_struct  
{  
  
    UINT8* data[16];  
  
} DV_DEV_LABEL;
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_LISTENER_HANDLE

This macro defines the device listener ID.

```
typedef UINT32 DV_LISTENER_HANDLE;
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DEV_REGISTER_CALLBACK

This macro defines the device registration callback.

```
typedef STATUS (*DEV_REGISTER_CALLBACK) (DV_DEV_ID, VOID *);
```

Related Topics

[Macros and Typedefs in `dv_defs.h`](#)

DEV_UNREGISTER_CALLBACK

This macro defines the device un-registration callback.

```
typedef STATUS (*DEV_UNREGISTER_CALLBACK) (DV_DEV_ID, VOID *);
```

Related Topics

[Macros and Typedefs in `dv_defs.h`](#)

DV_DRV_OPEN_FUNCTION

This macro defines the driver open function.

```
typedef STATUS (*DV_DRV_OPEN_FUNCTION) (VOID* instance_handle,  
DV_DEV_LABEL label_list[], INT label_cnt, VOID** session_handle_ptr);
```

Related Topics

[Macros and Typedefs in `dv_defs.h`](#)

DV_DRV_CLOSE_FUNCTION

This macro defines the driver close function.

```
typedef STATUS (*DV_DRV_CLOSE_FUNCTION) (VOID* session_handle);
```

Related Topics

[Macros and Typedefs in `dv_defs.h`](#)

DV_DRV_READ_FUNCTION

This macro defines the driver read function.

```
typedef STATUS (*DV_DRV_READ_FUNCTION) (VOID* session_handle, VOID*  
buffer, UINT32 numbyte, OFFSET_T byte_offset, UINT32* bytes_read_ptr);
```

Note



Changed from DV_DRV_READ_FUNCTION to *DV_DRV_READ_FUNCTION to match OPEN, CLOSE, and WRITE.

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_DRV_WRITE_FUNCTION

This macro defines the driver write function.

```
typedef STATUS (*DV_DRV_WRITE_FUNCTION) (VOID* session_handle, const VOID  
*buffer, UINT32 numbyte, OFFSET_T byte_offset, UINT32 *bytes_written_ptr);
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

DV_DRV_IOCTL_FUNCTION

This macro defines the driver IOCTL function.

```
typedef STATUS (*DV_DRV_IOCTL_FUNCTION) (VOID* session_handle, INT  
ioctl_num, VOID* ioctl_data, INT ioctl_data_len);
```

Related Topics

[Macros and Typedefs in dv_defs.h](#)

Data Structures in dv_defs.h

- [DV_DRV_FUNCTIONS](#)
- [IOCTL0_STRUCT](#)
- [DV_APP_REGISTRY_CHANGE](#)

DV_DRV_FUNCTIONS

```
/* Application level structure for DVC_Dev_Register() */
typedef struct _dv_drv_functions_struct
{
    DV_DRV_OPEN_FUNCTION drv_open_ptr;
    DV_DRV_CLOSE_FUNCTION drv_close_ptr;
    DV_DRV_READ_FUNCTION drv_read_ptr;
    DV_DRV_WRITE_FUNCTION drv_write_ptr;
    DV_DRV_IOCTL_FUNCTION drv_ioctl_ptr;
} DV_DRV_FUNCTIONS;
```

Related Topics

[Data Structures in dv_defs.h](#)

IOCTL0_STRUCT

```
/* Application level structure for IOCTL0 */
typedef struct _ioctl0_struct
{
    DV_DEV_LABEL label;
    INT base;

} IOCTL0_STRUCT;
```

Related Topics

[Data Structures in dv_defs.h](#)

DV_APP_REGISTRY_CHANGE

```
/* Application level structure for DVC_Reg_Change_Check() */
typedef struct _dv_app_registry_change_struct
{
    INT max_id_cnt;
    INT registry_changes;
    DV_DEV_LABEL *dev_label_list_ptr;
    INT dev_label_cnt;
    DV_DEV_ID *known_id_list_ptr;
    INT *known_id_cnt_ptr;
    DV_DEV_ID *reg_id_list_ptr;
    INT *reg_id_cnt_ptr;
    DV_DEV_ID *unreg_id_list_ptr;
    INT *unreg_id_cnt_ptr;
} DV_APP_REGISTRY_CHANGE;
```

Related Topics

[Data Structures in dv_defs.h](#)

Error Codes in dv_defs.h

#define	DV_INVALID_INPUT_PARAMS	-1
#define	DV_DEV_LIST_TOO_SMALL	-2
#define	DV_LABEL_LIST_TOO_SMALL	-3
#define	DV_REG_LIST_TOO_SMALL	-4
#define	DV_UNREG_LIST_TOO_SMALL	-5
#define	DV_INVALID_REG_PARAMS	-6
#define	DV_INVALID_UNREG_PARAMS	-7
#define	DV_NO_AVAILABLE_DEV_ID	-8
#define	DV_NO_AVAILABLE_SESSION	-9
#define	DV_DEV_NOT_REGISTERED	-10
#define	DV_SESSION_NOT_OPEN	-11
#define	DV_LABEL_NOT_FOUND	-12
#define	DV_UNEXPECTED_ERROR	-13

Related Topics

[Device Manager Overview](#)

Chapter 4

Memory Management Unit (MMU)

MMU Overview

Nucleus MMU binds one or more tasks, high-level interrupt service routines (HISRs), and signal handlers, collectively referred to as threads, into a single unit that can be manipulated by other components of Nucleus PLUS. This single unit is called a module. The threads that make up a module make use of one or more memory regions, which can contain either data (task stacks, pools, and the like) or executable code.

MMU creates a barrier between the kernel and the user application or between different applications in the system.

Nucleus MMU also allows for exception handling. The typical flow in the system involves branching to a low level handler when an exception vector is hit. The handler calls the MMU handler that in turn sets up the call to a module's exception handler.

Caution



In your applications, use only interfaces, structures, macros, and so on, that are documented within this and other Nucleus reference guides. There is no guarantee of future support or compatibility for any interface that is not documented.

MMU Modules

This section provides a functional description of the components that make up module support for Nucleus PLUS.

MMU Configuration

The MMU Module kernel component can be enabled by setting the following:

```
# Enable the core component of Modules
nu.os.kernel.module.enable=true
```

This component can be configured to support different modes:

Table 4-1. MMU module configuration options

Option	Description
1	Enables mode switching.
2	Enables mode switching and full use of the Module API.
3	Enables mode switching, full use of the Module API, and hardware supported memory protection using an MMU.
4	Enables mode switching, full use of the Module API, and hardware supported memory protection using an MPU.

The default configuration option is:

```
nu.os.kern.module.mode=3
```

To set the mode for switching and module API only:

```
# Set the mode for mode switching and module API only
nu.os.kern.module.mode=2
```

The MMU Module component also relies on architectural support that needs to be enabled (for example: enable `nu.os.arch.arm.module.v4v5` for an ARM926EJS based BSP).

```
# Set the MMU architecture to v4v5
nu.os.arch.arm.module.v4v5.enable=true
```

Architectures can have zero to many module components, but only one can be enabled at any given time. The correct module component in the architecture, if one exists, is dependent on the BSP being used. To get a better understanding of the appropriate architecture settings, utilize the *module.min_kernel.config* template via the UI configuration tool or via the `USER_CONFIG` setting from the command-line. This configuration file will enable the appropriate module component (if one is available) for the BSP being used.

NOTE: If building with RealView tools, you must enable the following metadata option:

```
# Enable data copy operations
nu.os.kern.plus.core.rom_to_ram_copy=true
```

MMU Initialization

Module Support is initialized early in the kernel initialization. This initialization is invoked from the `INC_Initialize` routine that is responsible for initializing the Nucleus thread component. `INC_Initialize` calls the `NU_MIC_INITIALIZE` macro, defined as the `MICT_Module_Initialize` function.

`MICT_Module_Initialize` is the main initialization function for the module library. First, it makes sure cache and the MMU are disabled. Next, the exception handlers for the module library are registered with the kernel's low level handler. Then the target-independent memory region component is initialized with a call to the `MRC_Initialize` routine. The `MRC_Initialize` routine in turn invokes the `MRT_Initialize` routine to perform target-specific memory region component initialization, such as creating the kernel memory regions.

After the memory region component is initialized, the `MICT_Module_Initialize` routine invokes the `MSC_Initialize` routine to perform initialization of the target-independent module support component. The `MSC_Initialize` routine in turn invokes the `MST_Initialize` routine to perform target-specific module support initialization, such as creating the kernel module.

After the module support component is initialized, `MICT_Module_Initialize` invokes the `MMCT_Initialize_MMU` routine that performs target specific MMU initialization and turns MMU on.

MMU Linker Control File

Setting up the linker control file for use with MMU requires consideration. Symbols are used to place memory protection around various sections within code. To do this correctly, you must consider many things including required symbols and alignment.

Although a sample linker control file is provided, it might need to have the start address modified for your platform. The start address in the example is `0x20000000`. Edit this value to enable the Module example build and execution.

Required Symbols

Specific symbols must be exposed within the linker control file to enable Nucleus to place the text and data sections for memory protection. These symbols are used to signify the start and end of each major code section. [Table 4-2](#) shows the list of required symbols.

Table 4-2. Required Symbols

Name	CS GNU	RVCT
Kernel text start	<code>_stext</code>	text
Kernel text end	<code>_etext</code>	-
Kernel data start	<code>_sdata</code>	data
Kernel data end	<code>_edata</code>	-
Kernel bss start	<code>_sbss</code>	bss
Kernel bss end	<code>_ebss</code>	-

Table 4-2. Required Symbols

Name	CS GNU	RVCT
RTL start	_srtl	srtl
RTL end	_ertl	ertl
Heap end	_end	-

Alignment

Memory protection is enforced by hardware. To correctly use this, all distinct memory sections should be aligned on a 4k boundary. Data and bss sections are usually identified as a single section and no alignment is required on the bss sections.

The following examples show how to perform alignment:

- CS GNU:

```
.mydata ALIGN(_my_prev_section, 0x1000):
```

- RVCT:

```
MYDATA ((ImageLimit(MYPREVSECTION) + (0x1000 - 1)) AND ~(0x1000 - 1))
```

Section Order

The following requirements define the order of section definition:

- The kernel bss section always follows the data section.
- The kernel bss section is always last.

RTL Usage

If you use RTL, note that some tools access globals to achieve reentrancy and track errors. To avoid exceptions from user mode applications using these calls, place certain symbols between the rtl start and end symbols.

Note



Some toolsets give warnings if a symbol is placed that isn't used in the current image.

CS GNU RTL setup

The following example shows how to setup RTL using CS GNU:

```
.rtl ALIGN(_esem_bss, 0x1000):  
{  
    _srtl = .;  
    *libc.a:lib_a-impure.o (.data)
```



```
*libc.a:lib_a-mallocr.o (.data)
*libc.a:lib_a-mallocr.o (.bss)
*libc.a:lib_a-gdtoa-gethex.o (COMMON)
*libc.a:lib_a-reent.o (COMMON)
} > ram

.heap (NOLOAD):
{
    end = .;
    _HEAP = .;
    _HEAP_START = end;
    *(.heap)
    _HEAP_END = _HEAP_START + 0x2000;

    . = ALIGN(4);
    _ertl = .;
} > ram
```

RVCT RTL Setup

The following example shows how to setup RTL using RVCT:

```
; RTL section start
srtl +0 empty 0x0
{
}

rtl +0
{
    strtok.o(+DATA)
    atexit.o(+DATA)
    rt_errno_addr.o(+BSS)
    rand.o(+BSS)
    atexit.o(+BSS)
    localtime.o(+BSS)
    asctime.o(+BSS)
}

; Default heap size
heap +0 empty 0x1000
{
}

; RTL section end
ertl +0 empty 0x0
{
}
```

MMU and the C++ Service

When you enable MMU, you must correctly place portions of the C++ Service and the supporting C++ tool set implementation in the image. Use linker command (.ld) files, specific to each tool set, to enable MMU.

To see examples of how to correctly place the C++ support into an MMU image file, view the appropriate linker command files for your tool set. These files are provided with the MMU service sample application, found at *os/samples/kernel/module*.

Hardware Memory Regions

Memory regions provide a means to define the physical address, size, and various attributes of a portion of the target processor's physical address space. This definition is then used to control access to the physical address range described by the memory region when it is used later.

See the hardware manual for target-specific possible limitations or mappings.

Physical Starting Address

Each memory region has a physical starting address that defines the beginning of the memory region in the processor's physical address space. For releases of Module Support for Nucleus PLUS that include Nucleus MMU support, the starting address of a memory region is typically forced to be an integer multiple of the smallest page size of the target hardware.

Memory Region Size

The size of a memory region is another important attribute of the memory region definition. A memory region consists of the physical addresses from its starting address to that address plus the size of the memory region minus one. This address range is important because (except via a sharing mechanism described later) a new memory region is not permitted to overlap an existing memory region.

Note



If the BSP map file virtually maps an address, that mapping will be maintained in region creation.

Memory Access Permissions

In addition to the physical address range that a memory region occupies, there are attributes that are associated with a memory region. The first set of attributes includes the memory access permissions associated with the memory region. These attributes specify whether the memory region may be read from, executed from, or written to. These attributes are typically only enforced when `NU_MMU_MODE` is enabled.

Cache Control Options

The second set of attributes associated with a memory region are cache control options. These attributes specify whether the cache is enabled for the memory region. The write policy and pre-fetch policy of the memory region can also be specified, if this is supported by the target hardware. These attributes may only have meaning when `NU_MMU_MODE` is enabled.

Related Topics

[Memory Access, Sharing, and Caching Options](#)

MMU and Sharing Permissions

The third set of attributes specifies the region's sharing permissions. Normally, memory regions are not permitted to overlap, but it is often necessary for one module to access a physical address range controlled by another module. Nucleus MMU provides a special case referred to as sharing, where a shared memory region is permitted to overlap another, sharable memory region. The sharing permission attributes allow the sharable memory region to limit the memory access permissions of the new memory regions shared from it. See the [Memory Access, Sharing, and Caching Options](#) section later in this document for more information.

Alter Permissions

Memory regions have a final set of attributes that specify who can alter the options of a region. In most cases a region's options may only be modified by an application running in supervisor mode. This set of permissions can be used to give further flexibility with which a region is handled.

Related Topics

[Memory Access, Sharing, and Caching Options](#)

MMU Module Creation

Once placed or shared, memory regions can be combined to form a module. The attached memory regions that form a module then provide a context in which one or more threads of control (tasks, HISRs, and so on) can execute. If `NU_MODULE_MODE` is enabled, modules even form protection domains where only the threads that belong to the module can access the memory regions it contains.

In addition to the discussion in this section, more information about modules and their interaction with memory regions can be found in the descriptions for the [NU_Attach_Memory_Region](#), [NU_Create_Module](#), and [NU_Start_Module](#) service routines.

Module Memory Regions

A module is created from a list of previously placed or shared memory regions. The memory regions that make up a module are typically provided as parameters to the `NU_Create_Module` service routine. The supplied memory regions must either be shared regions or regions which are not attached to any other module. Memory regions could be dynamically attached and detached from a module via the [NU_Attach_Memory_Region](#) and

[NU_Detach_Memory_Region](#) service routines, but this should be done with care to insure that detached memory regions are not in use by a thread of control when they are detached.

Module Initialization

After the module is created and the specified memory regions are attached, an optional user-supplied module initialization routine is executed to define the initial module environment. This initialization can include the creation of tasks, mailboxes, queues, pipes, semaphores, event groups, and other Nucleus PLUS objects used by the module. This routine can be invoked by [NU_Create_Module](#) at the time of module creation, or later by calling [NU_Start_Module](#). See the descriptions of these service routines and `module_init` for more information.

Note



Module start must be called from task context.

Module Cleanup

Before a module can be destroyed, some module-specific cleanup must be performed. If specified at module creation, a module cleanup routine is executed to free up all Nucleus PLUS resources allocated by the module. Note that all tasks, HISRs, partition pools, memory pools, and other PLUS kernel objects must be deleted before returning from the module cleanup routine.

Invoking the [NU_Stop_Module](#) service routine marks a module as stopped so that [NU_Delete_Module](#) can then delete it. If a user-supplied module cleanup routine is provided to the [NU_Create_Module](#) call, [NU_Stop_Module](#) will execute this routine before marking the module as stopped.

Note



Unless otherwise stated, the function does not perform any tasking changes.

Memory Region Function Reference

Memory region functions allow physical memory to be subdivided by Nucleus PLUS into separate regions for use with Nucleus PLUS modules. The following functions pertain to memory region management:

- [NU_Attach_Memory_Region](#)
- [NU_Change_Memory_Region_Options](#)
- [NU_Create_Module_Memory_Regions](#)
- [NU_Delete_Memory_Region](#)
- [NU_Local_Change_Memory_Region_Options](#)

- [NU_Local_Share_Memory_Region](#)
- [NU_Memory_Region_Information](#)
- [NU_Memory_Region_Phys_Base_Addr](#)
- [NU_Memory_Region_Pointers](#)
- [NU_Memory_Regions](#)
- [NU_Place_Memory_Region](#)
- [NU_Share_Memory_Region](#)
- [NU_Shared_Memory_Region_Pointers](#)
- [NU_Shared_Memory_Regions](#)

NU_Attach_Memory_Region

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Support services

Implemented by: MSC_Attach_Memory_Region

This service attaches the specified memory region to the specified module, if that module has hardware resources remaining to attach an additional memory region. This service routine must be called from Supervisor mode in Supervisor/User mode-switching kernels.

Usage

```
STATUS NU_Attach_Memory_Region (NU_MODULE      *module,  
                                NU_MEMORY_REGION *region,  
                                VOID             **virt_base_addr);
```

Arguments

- **module**
Pointer to the module control block to which to attach the specified memory region.
- **region**
Pointer to the memory region control block to attach the specified module.
- **virt_base_addr**
Pointer to a location in which to store the virtual (logical) address of the base of the memory region, in the context of the specified module.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE_POINTER**
The module pointer was NU_NULL.
- **NU_MODULE_INVALID_MODULE**
The module points to an invalid control block.
- **NU_MODULE_INVALID_REGION_POINTER**
The region pointer was NU_NULL.
- **NU_MODULE_INVALID_REGION**
The memory region control block does not refer to a valid memory region.
- **NU_MODULE_REGION_UNAVAILABLE**
Region is already attached to a module.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Not called from supervisor mode.

- **NU_MODULE_REGION_OVERLAPS**
Region overlaps with existing region.

Example

```
STATUS status;  
/* No virtual addressing in Nucleus MMU yet pass null. */  
status = NU_Attach_Memory_Region(module, new_region, NU_NULL);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Detach_Memory_Region](#)

NU_Change_Memory_Region_Options

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Memory Region Services

Implemented by: MRC_Change_Memory_Region_Options

This service changes the access, caching, and sharing options of an existing memory region. Some options cannot be changed by this service, depending on the particular target implementation. A memory region's sharing and caching options cannot be changed once there are other memory regions shared from it.

Usage

```
STATUS NU_Change_Memory_Region_Options (NU_Memory_Region *region,  
                                         UNSIGNED          option_mask,  
                                         UNSIGNED          region_options);
```

Arguments

- **region**
Pointer to the memory region control block for which to change options.
- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be changed; changed options is defined as (original options AND (NOT options_mask)) or (region_options AND options_mask)
- **region_options**
Specifies the new memory region options to set or clear.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_MODULE_INVALID_REGION_POINTER**
The memory region pointer is NU_NULL.
- **NU_MODULE_INVALID_REGION**
The memory region pointer does not point to a valid memory region control block.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Not called from supervisor mode.
- **NU_MODULE_INVALID_REGION_OPTIONS**
One or more of the supplied access, sharing, caching, or target-specific options to be changed is invalid or not supported.

Example

```
NU_Memory_Region Region;
```



```
STATUS status;  
VOID Base;  
/* Set MEM options to NU_MEM_CODE, without changing other options */  
status =  
NU_Change_Memory_Region_Option(&Region, NU_MEM_MASK, NU_MEM_CODE);
```

Related Topics

Memory Region Function Reference	NU_Memory_Region_Information
NU_Place_Memory_Region	NU_Share_Memory_Region
NU_Memory_Region_Pointers	

NU_Create_Module_Memory_Regions

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Memory Region Services

Implemented by: MRC_Create_Module_Memory_Regions

This function creates shares for all the kernel memory regions. When the kernel regions have been created, it creates the basic module regions, text, and data. If text_end is NU_NULL, no module regions are created. If data_end is NU_NULL, the data region will not be created.

Usage

```
STATUS NU_Create_Module_Memory_Regions (
    NU_MEMORY_POOL      *pool,
    VOID                *text_start,
    VOID                *text_end,
    UNSIGNED             text_options,
    VOID                *data_start,
    VOID                *data_end,
    UNSIGNED             data_options,
    const CHAR          *name,
    NU_MEMORY_REGION    **module_regions);
```

Arguments

- pool
Pointer to memory pool from which the region structures will be allocated.
- text_start
Starting address of module's executable region.
- text_end
Ending address of module's executable region.
- text_options
Additional non-default options for the executable region.
- data_start
Starting address of module's data region.
- data_end
Ending address of module's data region.
- data_options
Additional non-default options for the data region.
- name
Pointer to name of regions and will be prefixed with additional characters so that the regions can be more clearly distinguished.

- `module_regions`
The regions will be returned in this arrayed list.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_POOL`
The pointer is null or the control block is invalid
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called from supervisor mode

Example

```
STATUS      status;
NU_MEMORY_REGION *module_regions[NU_TARGET_MODULE_REGIONS];

/* Setup the kernel and basic module regions */
status = NU_Create_Module_Memory_Regions(&System_Memory,
    __MYTEXT_START, __MYTEXT_END, NU_NULL,
    __MYDATA_START, __MYBSS_END, NU_SHARE_READ,
    "MYMOD", module_regions);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Place_Memory_Region](#)

[NU_Share_Memory_Region](#)

[NU_Memory_Region_Information](#)

NU_Delete_Memory_Region

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Memory Region Services

Implemented by: MRC_Delete_Memory_Region

This service deletes a previously placed or shared memory region. Prior to deleting the memory region, it must be detached from any owning module, and any other memory regions must not share it.

Usage

```
STATUS NU_Delete_Memory_Region(NU_MEMORY_REGION *region);
```

Arguments

- **region**
Pointer to the user-supplied memory region control block of the memory region to delete.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_REGION_POINTER**
The memory region pointer is NU_NULL.
- **NU_MODULE_INVALID_REGION**
The memory region pointer does not point to a valid memory region control block.
- **NU_MODULE_INVALID_REGION_DELETE**
The memory region is still attached to a module.
- **NU_MODULE_REGION_IS_SHARED**
The memory region is still shared by other memory regions.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
The service routine must be executed from Supervisor Mode (Supervisor/User mode switching targets only).

Example

```
STATUS status;  
/* Detach region from the owning module */  
/* Delete memory regions sharing the one to be deleted */  
/* It is now safe to delete the memory region */  
status = NU_Delete_Memory_Region(&Region)
```

Related Topics

[Memory Region Function Reference](#)

[NU_Place_Memory_Region](#)

[NU_Share_Memory_Region](#)

[NU_Detach_Memory_Region](#)

NU_Local_Change_Memory_Region_Options

Allowed from: User tasks and HISRs

Category: Memory Region Services

Implemented by: MRS_Local_Change_Memory_Region_Options

This service checks to see if the region a user is attempting to modify can be changed. If it can, it calls the `NU_Change_Memory_Region_Options` routine. The memory region must be owned by the current module and the region must have `NU_ALTER_USER_MODIFY` set in the region options.

Usage

```
STATUS NU_Local_Change_Memory_Region_Options (
                                         NU_MEMORY_REGION *region,
                                         UNSIGNED          option_mask,
                                         UNSIGNED          region_options);
```

Arguments

- **region**
Pointer to the memory region control block for which to change options.
- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be changed.
- **region_options**
Specifies the new memory region options to set or clear.

Return Values

- **NU_SUCCESS**
Function completed successfully
- **NU_MODULE_INVALID_REGION_POINTER**
The pointer region was `NU_NULL`.
- **NU_MODULE_INVALID_REGION**
The memory region control block does not refer to a valid memory region.
- **NU_MODULE_INVALID_REGION_OPTIONS**
The region does not allow the requested permissions.
- **NU_MODULE_INVALID_MODULE_REGION**
The region is not a member of the current module.

Example

See similar example for [NU_Change_Memory_Region_Options](#).

Related Topics

[Memory Region Function Reference](#)

[NU_Change_Memory_Region_Options](#)

NU_Local_Share_Memory_Region

Allowed from: User tasks and HISRs

Category: Memory Region Services

Implemented by: MRS_Local_Share_Memory_Region

This service checks to see if the region a user is attempting to modify can be shared, if it can then it calls the routines NU_Share_Memory_Region and NU_Attach_Memory_Region. The memory region must be owned by the current module or we must be in supervisor mode and the region must have NU_ALTER_USER_SHARE set in the region options.

Usage

```
STATUS NU_Local_Share_Memory_Region(  
    NU_MODULE          *module_ptr,  
    NU_MEMORY_REGION   *original_region_ptr,  
    NU_MEMORY_REGION   *new_region_ptr,  
    CHAR               *name,  
    UNSIGNED            offset,  
    UNSIGNED            size,  
    UNSIGNED            region_options,  
    VOID               **new_phys_base,  
    UNSIGNED            *actual_size);
```

Arguments

- **module_ptr**
Pointer to module that new region will be added.
- **original_region_ptr**
Pointer to original memory region control block that permits sharing.
- **new_region_ptr**
Pointer to the user-supplied memory region control block of the memory region to be created.
- **name**
Pointer to an 8-character name for the memory region. The name does not need to be null-terminated. If this pointer is NU_NULL, then no name is assigned to the memory region.
- **offset**
Specifies the requested offset within the original memory region where the new region should begin.
- **size**
Specifies the requested size of the new memory region.
- **region_options**
Specifies the requested memory region options for the new memory region.

- `new_phys_base`
Pointer to the actual physical base address of the newly created memory region.
- `actual_size`
Pointer to the actual allocated size of the new memory region.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_REGION_POINTER`
The pointer region was `NU_NULL`.
- `NU_MODULE_INVALID_REGION`
The memory region control block does not refer to a valid memory region.
- `NU_MODULE_INVALID_REGION_OPTIONS`
The region does not allow the requested permissions.
- `NU_MODULE_INVALID_MODULE_REGION`
The region is not a member of the current module.

Example

See similar example in [NU_Share_Memory_Region](#).

Related Topics

[Memory Region Function Reference](#)

[NU_Share_Memory_Region](#)

NU_Memory_Region_Information

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Region Services

Implemented by: MRF_Memory_Region_Information

This service provides various information about the specified memory region.

Usage

```
STATUS NU_Memory_Region_Information(NU_MEMORY_REGION *region,  
                                     NU_MODULE          **owner,  
                                     VOID                **phys_base_addr,  
                                     UNSIGNED             *size,  
                                     VOID                **virt_base_addr,  
                                     UNSIGNED             *region_options);
```

Arguments

- **region**
Pointer to the user-supplied memory region control block.
- **owner**
Pointer to a location where the module control block pointer of the owner of the specified memory region will be stored. After successful completion of this service, if *owner is NU_NULL, then no module is currently attached to the memory region.
- **phys_base_addr**
Pointer to a location where the actual, placed physical base address of the memory region will be stored.
- **size**
Pointer to a location where the actual, placed size of the memory region will be stored.
- **virt_base_addr**
Pointer to a location where the virtual base address of the memory region will be stored.
- **region_options**
Pointer to a location where the access, sharing, and caching options currently applied to the memory region will be stored.

For any values that are not required, NU_NULL can be supplied, instead of a pointer for storing that particular value.

Note



For most targets, the virtual base address returned by this function is also the physical base address of the memory region, as address translation is not currently implemented by Memory Region and Module Support services.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_MODULE_INVALID_REGION_POINTER**
Indicates the memory region pointer is **NU_NULL**.
- **NU_MODULE_INVALID_REGION**
Indicates the memory region pointer does not point to a valid memory region control block.

Example

```
NU_MEMORY_REGION Region;  
VOID base_addr;  
UNSIGNED size;  
STATUS status;  
  
/* Place or share the memory region */  
  
/* Retrieve physical base address and size from memory region */  
status = NU_Memory_Region_Information(&Region, NU_NULL, &base_addr,  
                                     &size, NU_NULL, NU_NULL);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Share_Memory_Region](#)

[NU_Place_Memory_Region](#)

NU_Memory_Region_Phys_Base_Addr

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Module Support Services

Implemented by: MSF_Memory_Region_Phys_Base_Addr

This service returns the physical base address of the supplied memory region pointer. This service returns NU_NULL if the pointer does not point to a valid memory region control block.

Usage

```
VOID* NU_Memory_Region_Phys_Base_Addr (NU_MEMORY_REGION *region);
```

Arguments

- region

Pointer to the memory region control block of which to retrieve the base address.

Return Values

- The physical base address of the supplied memory region pointer.

Example

```
NU_Memory_Region Region;  
VOID base_addr;  
  
/* Place or share the memory region... */  
  
base_addr = NU_Memory_Region_Phys_Base_Addr(&Region);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Delete_Module](#)

[NU_Module_Pointers](#)

[NU_Create_Module](#)

NU_Memory_Region_Pointers

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Region Services

Implemented by: MRF_Memory_Region_Pointers

This service builds a list of pointers to all memory regions in the system that shares the specified memory region access, sharing, and/or caching options.

Usage

```
STATUS NU_Memory_Region_Pointers(INT          option_mask,
                                INT          region_options,
                                INT          match_all,
                                UNSIGNED     maximum_pointers,
                                NU_MEMORY_REGION *regions_array[],
                                UNSIGNED     *region_count);
```

Arguments

- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as actual options AND options_mask.
- **region_options**
Specifies the memory region options to match; desired options is defined as region_options AND options_mask. The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the pointer list if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, then a memory region is added to the pointer list if any of the options specified in desired options are present in compared options.
- **maximum_pointers**
Specifies the maximum size of the list.
- **regions_array**
Pointer to the location used for building the list of pointers to matched memory regions.
- **region_count**
Pointer to the location where the actual number of pointers placed in the list is written.

Return Values

- **NU_SUCCESS**
Function completed successfully

- **NU_MODULE_INVALID_POINTER**
The region array pointer is **NU_NULL** or the region count pointer is **NU_NULL**.
- **NU_MODULE_INVALID_SIZE**
The maximum pointers count is not greater than zero.

Description

This list of pointers can contain memory regions that exactly match a set of options, or memory regions that match at least one of a set of options. While the pointers in this list are unique, they are not guaranteed to be in a particular order.

Example

```
#define MAX_REGIONS 20
NU_MEMORY_REGION Region[MAX_REGIONS];
UNSIGNED exact;
UNSIGNED partial;
STATUS status;

/* Consider only the sharing options, and exactly match
   NU_SHARE_READ and NU_SHARE_EXEC selected, NU_SHARE_WRITE *NOT*
   selected. */

#define SHARE_MASK (NU_SHARE_READ|NU_SHARE_WRITE |NU_SHARE_EXEC)
#define SHARE_OPTIONS (NU_SHARE_READ|NU_SHARE_EXEC)
status = NU_Memory_Region_Pointers(SHARE_MASK, SHARE_OPTIONS,
                                   NU_TRUE, MAX_REGIONS,
                                   &Region[0], &exact);

/* Consider only the NU_CACHE_NO_DATA and NU_CACHE_NO_CODE
   options. Match either NU_CACHE_NO_CODE selected or
   NU_CACHE_NO_DATA not selected, or both. */

#define CACHE_MASK (NU_CACHE_NO_DATA|NU_CACHE_NO_CODE)
#define CACHE_OPTIONS (NU_CACHE_NO_DATA|NU_CACHE_NO_CODE)
status = NU_Memory_Region_Pointers(CACHE_MASK, CACHE_OPTIONS,
                                   NU_FALSE, MAX_REGIONS,
                                   &Regions[0], &partial);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Memory_Region_Information](#)

[NU_Place_Memory_Region](#)

[NU_Memory_Regions](#)

NU_Memory_Regions

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Region Services

Implemented by: MRF_Memory_Regions

This service returns the number of memory regions in the system that share the specified memory region access, sharing, and/or caching options. This count can include either memory regions that exactly match a set of options, or memory regions that match at least one of a set of options.

Usage

```
UNSIGNED NU_Memory_Regions (INT    option_mask,  
                             INT    region_options,  
                             INT    match_all);
```

Arguments

- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as (actual options and option_mask).
- **region_options**
Specifies the memory region options to match; desired options is defined as (region_options and option_mask). The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the count if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, then a memory region is added to the count if any of the options specified in desired options are present in compared options.

Return Values

- The number of memory regions in the system that share the specified memory region access, sharing, and/or caching options.

Example

```
UNSIGNED exact;  
UNSIGNED partial;  
  
/* Consider only the sharing options, and exactly match  
   NU_SHARE_READ and NU_SHARE_EXEC selected, NU_SHARE_WRITE *NOT*  
   selected. */  
  
#define SHARE_MASK (NU_SHARE_READ|NU_SHARE_WRITE|NU_SHARE_EXEC)  
#define SHARE_OPTIONS (NU_SHARE_READ|NU_SHARE_EXEC)  
exact = NU_Memory_Regions(SHARE_MASK, SHARE_OPTIONS, NU_TRUE);
```

```
/* Consider only the NU_CACHE_NO_DATA and NU_CACHE_NO_CODE
options. Match either NU_CACHE_NO_CODE selected or
NU_CACHE_NO_DATA not selected, or both. */

#define CACHE_MASK (NU_CACHE_NO_DATA|NU_CACHE_NO_CODE)
#define CACHE_OPTIONS (NU_CACHE_NO_DATA|NU_CACHE_NO_CODE)
partial = NU_Memory_Regions(CACHE_MASK, CACHE_OPTIONS, NU_FALSE);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Memory_Region_Information](#)

[NU_Place_Memory_Region](#)

[NU_Memory_Region_Pointers](#)

NU_Place_Memory_Region

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Memory Region Services

Implemented by: MRC_Place_Memory_Region

This service is used to create a memory region with a specified type, providing a mechanism to map physical address space to a new memory region.

Usage

```
STATUS NU_Place_Memory_Region(NU_MEMORY_REGION *region,  
                              CHAR *name,  
                              VOID *phys_base_addr,  
                              UNSIGNED size,  
                              UNSIGNED region_options,  
                              VOID **aligned_base_addr,  
                              UNSIGNED *actual_size);
```

Arguments

- **region**
Pointer to the user-supplied module control block. All subsequent requests made to this module require this pointer.
- **name**
Pointer to an 8-character name for the memory region. The name does not have to be null-terminated. If this pointer is NU_NULL, then no name is assigned to the memory region.
- **phys_base_addr**
The physical base address of the memory region.
- **size**
Specifies the requested size of the memory region.
- **region_options**
Specifies the requested access, sharing, and caching options for the memory region.
- **aligned_base_addr**
Pointer to the actual placed physical base pointer. Upon successful completion of this service, the actual base address of the placed memory region will be in *aligned_base_addr, as described above.
- **actual_size**
Pointer to the actual placed size of the memory region. Upon successful completion of this service, the actual size of the placed memory region will be in *actual_size, as described above.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_MODULE_INVALID_REGION_POINTER**
Indicates the memory region control block pointer is **NU_NULL** or the memory region pointer points to a memory region control block pointer that is already in use.
- **NU_MODULE_REGION_OVERLAPS**
Indicates the region requested overlaps with an existing region.
- **NU_MODULE_INVALID_REGION_SIZE**
Indicates the requested memory region size was invalid.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Indicates the service routine must be executed from Supervisor Mode (Supervisor/User mode switching targets only Tasking Changes).
- **NU_MODULE_REGION_UNAVAILABLE**
Region control block is already in use.

Description

There are four principal uses of **NU_Place_Memory_Region**. The first use is to create a memory region for executable code already existing in ROM or FLASH. The second is to create a memory region for data (initialized or uninitialized) already existing in RAM. The third is to create a memory region for stack space, dynamic memory pools, or partition pools from memory allocated from the system memory pool. Finally, the fourth is to create a memory region for memory-mapped I/O devices.

For memory regions that overlay portions of the address space (the first, second, and fourth cases) that are known at compile time, use the command file of the target linker to insure that code and static data sections begin on **NU_TARGET_MIN_PAGE_SIZE** aligned boundaries. For these memory regions, specify a size that is rounded up to the next multiple of **NU_TARGET_MIN_PAGE_SIZE**. This technique can also be used to statically declare stack and heap space that is properly aligned and sized at link time.

For compatibility with memory allocators that do not allocate on **NU_TARGET_MIN_PAGE_SIZE** boundaries (the third case), the placed memory region always fits within the address range specified by the supplied physical base address and that same physical base address plus the supplied size minus one, inclusive. If the supplied address range is not properly aligned, the placed memory region will be smaller than the supplied size.

Specifically, the actual aligned base address of a placed memory region is the same as the physical base address supplied if that physical base address is aligned on a **NU_TARGET_MIN_PAGE_SIZE** boundary. If it is not, then the actual aligned base address of

a placed memory region is the next highest address past the supplied physical base address that is `NU_TARGET_MIN_PAGE_SIZE` aligned.

Similarly, if the supplied region end plus one (supplied physical base plus supplied size) is aligned on a `NU_TARGET_MIN_PAGE_SIZE` boundary, then the actual region end is the supplied physical base plus the supplied size minus one. Otherwise, the actual region end is the next lower address that is `NU_TARGET_MIN_PAGE_SIZE` aligned, minus one.

Memory regions placed with this service are not permitted to overlap an existing memory region. Only the resulting placed memory region address ranges, not necessarily the supplied address ranges, are used to determine memory region overlap, due to the address range adjustments described on the previous page.

Example

```
NU_MEMORY_REGION Region;
VOID actual_base;
UNSIGNED actual_size;
STATUS status;
#define ROM_OPTIONS (NU_MEM_EXEC | NU_MEM_READ)

/* Create a 64 KB code memory region */
status = NU_Place_Memory_Region(&Region, "ROM", 0xFFFF0000,
                                65536, ROM_OPTIONS, &actual_base,
                                &actual_size);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Memory_Region_Information](#)

[NU_Memory_Region_Pointers](#)

[NU_Memory_Regions](#)

NU_Share_Memory_Region

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Memory Region Services

Implemented by: MRC_Share_Memory_Region

This service creates a new memory region that is a copy of an original memory region that permits sharing.

Usage

```
STATUS NU_Share_Memory_Region(NU_MEMORY_REGION *original_region,  
                              NU_MEMORY_REGION *new_region,  
                              CHAR *name,  
                              UNSIGNED offset,  
                              UNSIGNED size,  
                              UNSIGNED region_options,  
                              VOID **new_phys_base,  
                              UNSIGNED *actual_size);
```

Arguments

- **original_region**
Pointer to the original memory region control block that permits sharing.
- **new_region**
Pointer to the user-supplied memory region control block of the memory region to be created. Note: all subsequent requests made to this memory region require this pointer.
- **name**
Pointer to an 8-character name for the memory region. The name does not have to be null-terminated. If this pointer is NU_NULL, then no name is assigned to the memory region.
- **offset**
Specifies the requested offset within the original memory region where the new memory region should begin.
- **size**
Specifies the requested size of the new memory region.
- **region_options**
Specifies the requested memory access options for the new memory region. Sharing, caching, and target-specific options cannot be specified. A particular NU_MEM option is only permitted if the original shared memory region specified the corresponding NU_SHARE option.
- **new_phys_base**
Pointer to the actual physical base address of the new memory region. Upon successful completion of this service, *new_phys_base will contain the aligned physical base address of the new memory region.

- `actual_size`
Pointer to the actual allocated size of the new memory region. Upon successful completion of this service, `*actual_size` will contain the actual size allocated.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.
- `NU_MODULE_INVALID_REGION_POINTER`
Indicates one or more of the memory region control block pointers is `NU_NULL`.
- `NU_MODULE_INVALID_REGION`
Indicates the original memory region pointer does not point to a valid memory region control block.
- `NU_MODULE_INVALID_REGION_SIZE`
Indicates the requested size of the new memory region was invalid.
- `NU_MODULE_INVALID_REGION_OPTIONS`
Indicates that the options are not allowed by the original region.
- `NU_MODULE_INVALID_REGION_ADDRESS`
Indicates that the start address is beyond the end address.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called from supervisor mode.

Description

The new memory region may be a sub-region of the original shared memory region. This new region spans from the base of the original memory region plus an offset to this new base plus the specified size. The sub-region specified must fall completely within the original memory region.

This new region will have none of its other `NU_SHARE` options enabled, and it will “inherit” the caching and target-specific options of the original memory region. The `NU_MEM` options for the new memory region are specified by `region_options`. These memory access options must be a subset of the corresponding `NU_SHARE` options specified by the original shared memory region.

The new memory region actually created is only guaranteed to be no larger than the requested size, and is likely to be smaller. Similarly, this memory region is only guaranteed to begin at or after the requested offset. The actual offset and size of the new region depends on the granularity of memory regions on the target. This granularity depends on the capabilities of the

target hardware. See the target specific appendices for details about available allocation granularities.

By definition, memory regions created with this service overlap an existing memory region. Due to limitations of the target hardware, it may not be possible, or even meaningful, to attach overlapping memory regions to the same module.

Example

```
NU_MEMORY_REGION Original;
NU_MEMORY_REGION New_Region;
VOID actual_base;
UNSIGNED actual_size;
STATUS status;

/* Place original memory region with sharing options */

#define SHARED_OPTIONS (NU_MEM_EXEC | NU_MEM_READ)

/* Share a 64 KB memory region from start of original region */
status = NU_Share_Memory_Region(&Original, &New_Region, "SHARE",
                                0x0, 65536, SHARED_OPTIONS,
                                &actual_base, &actual_size);
```

Related Topics

[Memory Region Function Reference](#)

[NU_Shared_Memory_Regions](#)

[NU_Memory_Region_Information](#)

[NU_Shared_Memory_Region_Pointers](#)

[NU_Place_Memory_Region](#)

NU_Shared_Memory_Region_Pointers

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Region Services

Implemented by: MRF_Shared_Memory_Region_Pointers

This service builds a list of pointers to all memory regions shared from the specified memory region that have the specified memory region access, sharing, and/or caching options.

Usage

```
STATUS NU_Shared_Memory_Region_Pointers (
    NU_MEMORY_REGION *region,
    UNSIGNED          option_mask,
    UNSIGNED          region_options,
    INT               match_all,
    UNSIGNED          maximum_pointers,
    NU_MEMORY_REGION *regions_array[],
    UNSIGNED          *region_count);
```

Arguments

- **region**
 Supplied memory region control block, which permits sharing.
- **option_mask**
 Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as (actual options and options_mask).
- **region_options**
 Specifies the memory region options to match; desired options is defined as (region_options and option_mask). The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
 Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the count if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, a memory region is added to the count if any of the options specified in desired options are present in compared options.
- **maximum_pointers**
 Specifies the maximum size of the list.
- **regions_array**
 Pointer to the location used for building the list of pointers to shared memory regions.
- **region_count**
 Pointer to the location where the actual number of pointers placed in the list will be written.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_MODULE_INVALID_REGION_POINTER**
Indicates the shared memory region control block pointer is **NU_NULL**.
- **NU_MODULE_INVALID_REGION**
Indicates the shared memory region pointer does not point to a valid memory region control block.
- **NU_MODULE_INVALID_POINTER**
Indicates the region array pointer is **NU_NULL**, or the region count pointer is **NU_NULL**.
- **NU_MODULE_INVALID_SIZE**
Indicates the maximum pointers count is not greater than zero.

Description

This list of pointers can contain memory regions that exactly match a set of options, or memory regions that match at least one of a set of options. While the pointers in this list are unique, they are not guaranteed to be in a particular order.

Example

See similar example for [NU_Memory_Region_Pointers](#).

Related Topics

[Memory Region Function Reference](#)

[NU_Shared_Memory_Regions](#)

[NU_Share_Memory_Region](#)

NU_Shared_Memory_Regions

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Region Services

Implemented by: MRF_Shared_Memory_Regions

This service returns the number of memory regions shared from the specified memory region that have the specified memory region access, sharing, and/or caching options.

Usage

```
UNSIGNED NU_Shared_Memory_Regions(NU_MEMORY_REGION *region,  
                                   UNSIGNED           option_mask,  
                                   UNSIGNED           region_options,  
                                   INT                match_all);
```

Arguments

- **region**
Pointer to the user supplied memory region control block, which permits sharing.
- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as (actual options and options_mask).
- **region_options**
Specifies the memory region options to match; desired options is defined as (region_options and option_mask). The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the count if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, then a memory region is added to the count if any of the options specified in desired options are present in compared options.

Return Values

- Returns the number of memory regions shared from the specified memory region that match the specified memory region option criteria.

Description

This count can include either memory regions that exactly match a set of options, or memory regions that match at least one of a set of options.

Example

See similar example for [NU_Memory_Regions](#).

Related Topics

[Memory Region Function Reference](#)

[NU_Shared_Memory_Region_Pointers](#)

[NU_Share_Memory_Region](#)

Memory Access, Sharing, and Caching Options

Nucleus PLUS memory regions can be created with many combinations of attributes. Depending on the actual hardware memory protection support, some of these combinations may not be possible. Also, some combinations, such as (NU_MEM_DATA|NU_MEM_CODE), are not recommended for code executing in User mode.

Table 4-3. Memory Access, Sharing, and Caching Options

Option	Meaning
NU_MEM_READ	Specifies a memory region that can be read from.
NU_MEM_WRITE	Specifies a memory region that can be written to.
NU_MEM_DATA	(NU_MEM_READ NU_MEM_WRITE)
NU_MEM_EXEC	Specifies a memory region that can be executed.
NU_MEM_CODE	(NU_MEM_READ NU_MEM_EXEC)
NU_MEM_ACCESS	Specifies only the access rights options of private memory regions; used with various memory region enumeration functions. (NU_MEM_READ NU_MEM_WRITE NU_MEM_EXEC)
NU_SHARE_READ	Specifies a memory region that can possibly be read from by modules other than its owner.
NU_SHARE_WRITE	Specifies a memory region that can possibly be written to by modules other than its owner.
NU_SHARE_DATA	(NU_SHARE_READ NU_SHARE_WRITE)
NU_SHARE_EXEC	Specifies a memory region that can possibly be executed by modules other than its owner.
NU_SHARE_CODE	(NU_SHARE_READ NU_SHARE_EXEC)
NU_SHARE_ACCESS	Specifies only the access rights options of shared memory regions; used with various memory region enumeration functions. (NU_SHARE_READ NU_SHARE_WRITE NU_SHARE_EXEC)
NU_CACHE_NO_DATA	Specifies a memory region with the data cache disabled; otherwise, the data cache is enabled.

Table 4-3. Memory Access, Sharing, and Caching Options (cont.)

Option	Meaning
NU_CACHE_NO_CODE	Specifies a memory region with the instruction cache disabled; otherwise, the instruction cache is enabled.
NU_CACHE_WRITE_THROUGH	Specifies a memory region with a write-through data cache policy; otherwise, a write-back policy is used.
NU_CACHE_NO_PREFETCH	Specifies a memory region with pre-fetching (speculative access) disabled. The default is to allow pre-fetching.
NU_CACHE_NO_COHERENT	Specifies a memory region with multiple master cache coherency disabled. The default is to enable coherency.
NU_CACHE_WRITE_ALLOCATE	This attribute allows better use of write-back cache with L2 caches.
NU_ALTER_USER_MEMORY	Specifies a memory region's options can be altered by an application running in user mode.
NU_ALTER_USER_SHARE	Specifies that a memory region can be shared by an application in user mode.

Module Support Function Reference

Note



Unless otherwise stated, the function does not perform any tasking changes.

Module Support services allow memory regions to be combined into a single logical unit, called a module in Nucleus PLUS. The following services pertain to module management:

- [NU_Bind_Module_HISR](#)
- [NU_Bind_Module_Task](#)
- [NU_Change_Module_Options](#)
- [NU_Clear_BSS](#)
- [NU_Copy_DATA](#)
- [NU_Create_Module](#)
- [NU_Current_Module_Pointer](#)
- [NU_Delete_Module](#)
- [NU_Detach_Memory_Region](#)
- [NU_Established_Modules](#)

- [NU_Get_Default_Exception_Handler](#)
- [NU_Get_Module_Memory_Pool](#)
- [NU_Module_Established_HISRs](#)
- [NU_Module_Established_Tasks](#)
- [NU_Module_Exception_Handler](#)
- [NU_Module_HISR_Pointers](#)
- [NU_Module_Information](#)
- [NU_Module_Memory_Region_Pointers](#)
- [NU_Module_Memory_Regions](#)
- [NU_Module_Pointers](#)
- [NU_Module_Task_Pointers](#)
- [NU_Read_Module_Options](#)
- [NU_Register_Exception_Handler](#)
- [NU_Set_Default_Exception_Handler](#)
- [NU_Start_Module](#)
- [NU_Stop_Module](#)
- [NU_Unbind_Module_HISR](#)
- [NU_Unbind_Module_Task](#)
- [NU_Virtual_To_Physical](#)

NU_Bind_Module_HISR

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Support services

Implemented by: MSC_Bind_Module_HISR

This routine writes the module pointer to the HISR control block if the HISR has not yet been bound to a module. It also places the HISR on a list of HISRs in the module and increments the number of HISRs in the module.

Usage

```
STATUS NU_Bind_Module_HISR(NU_MODULE *module,  
                           NU_HISR *hisr_ptr);
```

Arguments

- **module**
Pointer to module control block.
- **hisr_ptr**
Pointer to HISR control block.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE**
A valid module pointer does not exist.
- **NU_MODULE_MODULE_IS_STOPPED**
Module is terminated.
- **NU_MODULE_HISR_ALREADY_BOUND**
HISR is bound to another module.
- **NU_INVALID_HISR**
Bad HISR control block passed in.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Cannot execute from user mode.

Example

```
status = NU_Bind_Module_HISR(&msd_kernel_module, new_hisr);
```

Related Topics

[Module Support Function Reference](#) [NU_Unbind_Module_HISR](#)
[NU_Bind_Module_Task](#)

NU_Bind_Module_Task

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Support services

Implemented by: MSC_Bind_Module_Task

This routine writes the module pointer to the task control block if the task has not yet been bound to a module. It also places the task on a list of tasks in the module and increments the number of tasks in the module

Usage

```
STATUS NU_Bind_Module_Task (NU_MODULE *module,  
                           NU_TASK *task_ptr);
```

Arguments

- **module**
Pointer to module control block.
- **task_ptr**
Pointer to task control block.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE**
A valid module pointer does not exist.
- **NU_MODULE_MODULE_IS_STOPPED**
Module is terminated.
- **NU_MODULE_TASK_ALREADY_BOUND**
Task is bound to another module.
- **NU_INVALID_TASK**
Bad task control block passed in.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Cannot execute from user mode.

Example

```
status = NU_Bind_Module_Task(&msd_kernel_module, new_task);
```

Related Topics

[Module Support Function
Reference](#)

[NU_Unbind_Module_Task](#)

[NU_Bind_Module_HISR](#)

NU_Change_Module_Options

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Support Services

Implemented by: MSS_Change_Module_Options

This service changes the module options of an existing module. By changing the module options, the affected module will either be given or lose the ability to call additional Nucleus PLUS calls. For more information, see the [Module Service Options](#) section.

Usage

```
STATUS NU_Change_Module_Options (NU_MODULE *module_ptr,  
                                UNSIGNED option_mask,  
                                UNSIGNED module_options);
```

Arguments

- **module_ptr**
Pointer to the module control block for which to change options.
- **option_mask**
Specifies an “AND mask” indicating which module options are to be changed.
- **module_options**
Specifies the new module options to set or clear.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE_POINTER**
The module pointer is NU_NULL.
- **NU_MODULE_INVALID_MODULE**
The module pointer points to an invalid module control block.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
The function was called in user mode.

Example

```
/* Set module up for access to NU_Activate_HISR and NU_Register_LISR */  
status = NU_Change_Module_Options(module, NU_MODULE_OPTIONS_MASK,  
                                (NU_MOD_COMMON | (NU_MOD_VECTOR)));
```

Related Topics

[Module Support Function Reference](#)

[NU_Read_Module_Options](#)

NU_Clear_BSS

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Utility Services

Implemented by: MUC_Clear_BSS

This service provides a way for an application module to clear the memory of its .bss section.

Usage

```
STATUS NU_Clear_BSS(VOID *bss_start,  
                   VOID *bss_end);
```

Arguments

- **bss_start**
Pointer to the start of the application module's .bss section.
- **bss_end**
Pointer to the byte immediately after the last byte of the application module's .bss section.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_POINTER**
One or more of the .bss pointers is NU_NULL.
- **NU_MODULE_INVALID_SIZE**
Indicates that bss_end points to an address less than that pointed to by bss_start.

Example

```
/* Linker-defined .bss section symbols */  
extern char __MODULE_BSS_START;  
extern char __MODULE_BSS_END;  
/* Clear the module's .bss section */  
status = NU_Clear_BSS(__MODULE_BSS_START, __MODULE_BSS_END);
```

Related Topics

[Module Support Function Reference](#) [NU_Copy_DATA](#)

NU_Copy_DATA

Allowed from: Application_Initialize, HISR, LISR, signal handler, and task

Category: Module Utility Services

Implemented by: MUC_Copy_DATA

This service provides a way for an application module to copy the initialized values of its data section from ROM.

Note

Many targets place ROM data differently. Check tool and hardware documentation for your setup to verify ROM start.

Usage

```
STATUS NU_Copy_DATA(VOID *data_start,  
                    VOID *data_end,  
                    VOID *rom_start);
```

Arguments

- **data_start**
Pointer to the start of the application module's .data section.
- **data_end**
Pointer to the byte immediately after the last byte of the application module's .data section.
- **rom_start**
Pointer to the start of the application module's initialized values to be copied into the module's .data section.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_POINTER**
One or more of the .data or ROM pointers is NU_NULL.
- **NU_MODULE_INVALID_SIZE**
data_end points to an address less than that pointed to by data_start.

Example

```
/* Linker-defined .data section symbols */  
extern char __MODULE_DATA_START;  
extern char __MODULE_DATA_END;  
extern char __MODULE_ROMDATA_START;  
/* Initialize the module's .data section */  
status = NU_Copy_DATA(__MODULE_DATA_START, __MODULE_DATA_END,  
                      __MODULE_ROMDATA_START);
```

Related Topics

[Module Support Function Reference](#)

[NU_Clear_BSS](#)

NU_Create_Module

Allowed from: Application_Initialize, Supervisor mode routine or task

Tasking changes: While this service routine does not directly cause tasking changes, such changes could occur as a result of executing the user-supplied module_init routine, since NU_Create_Module may call NU_Start_Module.

Category: Module Support Services

Implemented by: MSC_Create_Module

This service creates a module from an array of previously placed memory regions.

Usage

```
STATUS NU_Create_Module(NU_MODULE          *module,
                        CHAR                 *name,
                        NU_MODULE_START_FUNC module_init,
                        NU_MODULE_STOP_FUNC  module_cleanup,
                        NU_MEMORY_REGION     *pool_region,
                        NU_MEMORY_REGION     *regions_array[],
                        UNSIGNED              regions,
                        OPTION                auto_start);
```

Arguments

- **module**
Pointer to the user-supplied module control block. Note: all subsequent requests made to this module require this pointer.
- **name**
Pointer to an 8-character name for the module. The name does not have to be null-terminated. If name is NU_NULL, then no module name is specified.
- **module_init**
Specifies the initialization routine of the module. If memory protection or address translation is available, this must be the physical address of the routine.
- **module_cleanup**
Specifies the cleanup routine of the module. If memory protection or address translation is available, this must be the physical address of the routine.
- **pool_region**
Pointer to a memory region control block that indicates the memory region to be used by the module for dynamic memory needs. This parameter can be NU_NULL.

If NU_NULL is used, a memory pool will be allocated from the system's memory pool. A region will be created around that pool and a pointer to the pool control block will be passed on to the module initialization routine.

- **regions_array**
An array of pointers to memory region control blocks that will be associated with the created module.
- **regions**
Indicates the number of entries in the regions Array parameter.
- **auto_start**
Valid options for this parameter are **NU_START** and **NU_NO_START**. **NU_START** causes the **module_init** routine to be executed after the module is created. **NU_NO_START** leaves the module uninitialized. Modules created with **NU_NO_START** can be initialized later by calling **NU_Start_Module**.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE_START**
auto_start option is invalid.
- **NU_MODULE_INVALID_MODULE_ENTRY**
Indicates either the **module_init** **module_cleanup** pointer is invalid.
- **NU_MODULE_INVALID_MODULE_POINTER**
Indicates that pointer to the module control block was **NU_NULL**.
- **NU_MODULE_INVALID_MODULE**
Indicates that the module control block is currently in use.
- **NU_MODULE_INVALID_POINTER**
regions_array is **NU_NULL**.
- **NU_MODULE_INVALID_REGION_COUNT**
Number of regions specified is 0.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Indicates Modules was not created from Supervisor Mode.
- **NU_MODULE_INVALID_REGION_POOL**
Indicates supplied pool region pointer was not present in the supplied memory regions array.
- **other status values**
Indicates a status value other than **NU_SUCCESS** was returned by the module init routine.

Description

An array of memory region control block pointers is supplied, and regions indicate the number of elements in this array. The supplied memory regions must either be shared regions or regions which are not attached to any other module.

After the module is created and the specified memory regions are attached and if the value of `auto_start` is `NU_START`, the user-supplied routine specified by `module_init` is executed to define the initial module environment. This includes creation of tasks, mailboxes, queues, pipes, semaphores, event groups, and other Nucleus PLUS objects used by the module.

The routine specified by `module_init` is provided with the module control block pointer `module` and the `pool_region` memory region pointer. The memory region indicated by this pointer must be present in the `regions_array` list, or this pointer can be `NU_NULL` if the module does not allocate dynamic memory from a specific memory region.

The `pool_region` pointer is intended to be used to specify to the module a memory region in which dynamic memory pools or partition pools can be created. This use of the `pool_region` parameter is not enforced, merely recommended.

The module control block pointer can be passed to Module Support service routines to obtain information on the other memory regions available to the module.

Example

```
#define NUM_REGIONS 2
NU_MEMORY_REGION Regions[NUM_REGIONS];
NU_MODULE Module;
STATUS Module_Init(NU_MODULE *module, VOID *base)
return NU_SUCCESS;
/* Regions in "Regions" are placed, and one is a code memory
   region, and the other is a data memory region... */
STATUS status;
/* Module has no cleanup routine or dynamic memory pool region */
status = NU_Create_Module(&Module, "any name", Module_Init,
                        NU_NULL, NU_NULL, Regions, NUM_REGIONS,
                        NU_START);
```

Related Topics

Module Support Function Reference	NU_Attach_Memory_Region
NU_Delete_Module	NU_Start_Module
NU_Stop_Module	module_init

NU_Current_Module_Pointer

Allowed from: LISR, HISR, Signal Handler, Task

Category: Module Support Services

Implemented by: MSC_Current_Module_Pointer

This service returns the currently active module pointer. Once Application_Initialize is complete, a pointer to a module control block is always returned. If no user threads are active, a pointer to the Kernel Module is returned.

Usage

```
NU_MODULE *NU_Current_Module_Pointer(VOID);
```

Return Values

- This service returns a pointer to the currently active module control block.

Example

```
NU_MODULE *module;  
  
/* Obtain the currently active module pointer */  
module = NU_Current_Module_Pointer();
```

Related Topics

[Module Support Function Reference](#)

[NU_Module_Information](#)

[NU_Module_Pointers](#)

[NU_Established_Modules](#)

NU_Delete_Module

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Module Support Services

Implemented by: MSC_Delete_Module

This service deletes a previously created module. Prior to deleting the module, the module must have been previously been stopped with NU_Stop_Module. All memory regions attached to the module must also be detached before calling this service routine.

Usage

```
STATUS NU_Delete_Module(NU_MODULE *module);
```

Arguments

- module
Pointer to the user-supplied module control block of the module to delete.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_MODULE_INVALID_MODULE_POINTER
Indicates the module pointer is NU_NULL.
- NU_MODULE_INVALID_MODULE
Indicates the module pointer does not point to a valid module control block.
- NU_MODULE_MODULE_NOT_TERMINATED
Indicates the specified module is still active (has not been terminated with the NU_Stop_Module service routine).
- NU_MODULE_INVALID_MODULE_DELETE
Indicates the module still has attached memory regions.
- NU_MODULE_NOT_IN_SUPERVISOR_MODE
Indicates the service routine must be executed from supervisor mode (supervisor/user mode switching targets only).

Example

```
STATUS status;  
  
/* Execute the modules cleanup routine */  
status = NU_Stop_Module(&Module);  
  
/* ...Detach regions owned by the module... */  
  
/* It is now safe to delete the module */  
status = NU_Delete_Module(&Module);
```

Related Topics

Module Support Function Reference	module_init
NU_Attach_Memory_Region	NU_Create_Module
NU_Detach_Memory_Region	NU_Stop_Module
module_cleanup	

NU_Detach_Memory_Region

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Module Support Services

Implemented by: MSC_Detach_Memory_Region

This service detaches the specified memory region from the specified module.

Usage

```
STATUS NU_Detach_Memory_Region(NU_MODULE      *module,  
                               NU_MEMORY_REGION *region);
```

Arguments

- **module**
Pointer to the user supplied module control block.
- **region**
Pointer to the memory region control block of the memory region to detach.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE_POINTER**
Indicates the module pointer or the memory region pointer is NU_NULL.
- **NU_MODULE_INVALID_REGION_POINTER**
Indicates the region pointer was NU_NULL.
- **NU_MODULE_INVALID_REGION**
Indicates that the memory region control block does not refer to a valid memory region
- **NU_MODULE_INVALID_REGION_DELETE**
Indicates that the memory region could not be detached from the specified module. For instance, it is not possible to detach any memory regions that are attached to the Nucleus PLUS kernel “module.”
- **NU_MODULE_INVALID_MODULE**
Indicates the module pointer does not point to a valid module control block, or the memory region pointer does not point to a valid memory region control block.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Indicates the service routine must be executed from Supervisor Mode (Supervisor/User mode switching targets only).

Description

All memory pools, partition pools, and other Nucleus PLUS objects created in the specified memory region must be deleted before detaching the region from the module.

Any Nucleus PLUS kernel objects contained in a memory region must be deleted before that memory region is detached from the module. Any memory region containing executable code in use by the module (for example, by tasks, HISRs, or signal handlers of the module) must not be detached.

Example

```
STATUS status;  
  
/* ...Ensure that memory region is no longer used... */  
  
/* Detach region owned by the module */  
status = NU_Detach_Memory_Region(&Module, &Region);
```

Related Topics

[Module Support Function Reference](#)

[NU_Create_Module](#)

[NU_Module_Memory_Regions](#)

[NU_Module_Memory_Region_Pointers](#)

[NU_Attach_Memory_Region](#)

NU_Established_Modules

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Module Support Services

Implemented by: MSF_Established_Modules

This service returns the number of established modules. All created modules are considered established. Deleted modules are no longer considered established.

Usage

```
UNSIGNED NU_Established_Modules(VOID);
```

Return Values

- The number of established modules.

Example

```
UNSIGNED modules;  
modules = NU_Established_Module();
```

Related Topics

[Module Support Function Reference](#)

[NU_Delete_Module](#)

[NU_Module_Pointers](#)

[NU_Create_Module](#)

NU_Get_Default_Exception_Handler

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Exception-Handler Services

Implemented by: EHF_Get_Default_Exception_Handler

This function returns the exception handler that is currently used in the system for assignment to new modules.

Usage

```
STATUS NU_Get_Default_Exception_Handler(  
    VOID(**exception_ptr)(NU_MODULE *, NU_EXCEPTION *));
```

Arguments

- `exception_ptr`
Pointer to default exception handler to be filled in.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called in supervisor mode.

Example

```
status = NU_Get_Default_Exception_Handler(default_handler);
```

Related Topics

[Module Support Function Reference](#)

[NU_Set_Default_Exception_Handler](#)

NU_Get_Module_Memory_Pool

This service returns a pointer to the memory pool specific to the current module.

Usage

```
STATUS NU_Get_Module_Memory_Pool (NU_MEMORY_POOL **memory_pool_ptr);
```

Arguments

- `memory_pool_ptr`
Return location of the pool

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_POINTER`
Memory pool return pointer is null.

Related Topics

[Module Support Function Reference](#)

[NU_Module_Pointers](#)

[NU_Module_Memory_Regions](#)

[NU_Module_Memory_Region_Pointers](#)

NU_Module_Established_HISRs

Allowed from: HISR, Signal Handler, Task

Category: Module Support Services

Implemented by: TCFM_Module_Established_HISRs

This function returns the current number of established HISRs in a specified module. HISRs previously deleted are no longer considered established.

Usage

```
UNSIGNED NU_Module_Established_HISRs(NU_MODULE *module_ptr);
```

Arguments

- `module_ptr`
Pointer to the module for which to get then number of HISRs.

Return Values

- Number of HISRs in the specified module.

Example

```
UNSIGNED hisr_count;  
hisr_count = NU_Module_Established_HISRs(my_module);
```

Related Topics

[Module Support Function Reference](#)

NU_Module_Established_Tasks

Allowed from: HISR, Signal Handler, Task

Category: Module Support Services

Implemented by: TCFM_Module_Established_Tasks

This function returns the current number of established tasks in a specified module. Tasks previously deleted are no longer considered established.

Usage

```
UNSIGNED NU_Module_Established_Tasks(NU_MODULE *module_ptr);
```

Arguments

- `module_ptr`
Pointer to module for which to get then number of tasks.

Return Values

- Number of tasks in the specified module.

Example

```
UNSIGNED task_count;  
task_count = NU_Module_Established_Tasks(my_module);
```

Related Topics

[Module Support Function Reference](#)

NU_Module_Exception_Handler

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Exception-Handler Services

Implemented by: EHFM_Module_Exception_Handler

This function retrieves the exception handler for the specified module.

Usage

```
STATUS NU_Module_Exception_Handler(  
    NU_MODULE *module_ptr,  
    VOID(**exception_ptr) (NU_MODULE *,NU_EXCEPTION *));
```

Arguments

- `module_ptr`
Pointer to module from which to retrieve the handler
- `exception_ptr`
Return pointer of the handler for the passed in module.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called from supervisor mode.

Example

```
status = NU_Module_Exception_Handler(my_module, my_handler);
```

Related Topics

[Module Support Function Reference](#)

[NU_Register_Exception_Handler](#)

[NU_Get_Default_Exception_Handler](#)

NU_Module_HISR_Pointers

Allowed from: HISR, Signal Handler, Task

Category: Module Support Services

Implemented by: TCFM_Module_HISR_Pointers

This function builds a list of HISR pointers for the module, 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.

Usage

```
UNSIGNED NU_Module_HISR_Pointers(NU_MODULE *module_ptr,  
                                NU_HISR **pointer_list,  
                                UNSIGNED maximum_pointers);
```

Arguments

- `module_ptr`
Pointer to module.
- `pointer_list`
Pointer to the list area.
- `maximum_pointers`
Maximum number of pointers in `pointer_list`.

Return Values

- Number of HISRs placed in list

Example

```
#define MAX_HISR_PTRS 5  
  
/* Create an array of HISR pointers. */  
NU_HISR* hisrs[MAX_HISR_PTRS];  
UNSIGNED number_of_hisrs;  
UNSIGNED num_ret;  
  
/* Get a list of HISRs within the current module. */  
num_ret = NU_Module_HISR_Pointers(NU_Current_Module_Pointer(),  
                                hisrs, MAX_HISR_PTRS);
```

Related Topics

[Module Support Function Reference](#)

NU_Module_Information

Allowed from: Application_Initialize, Supervisor mode routine or task

Category: Module Support Services

Implemented by: MSF_Module_Information

This service provides various information about the specified module.

Usage

```
STATUS NU_Module_Information (NU_MODULE          *module,  
                             NU_MODULE_START_FUNC *module_init,  
                             NU_MODULE_STOP_FUNC  *module_cleanup);
```

Arguments

- **module**
Pointer to the user-supplied module control block.
- **module_init**
Pointer to a location where the address of the module initialization routine will be stored.
- **module_cleanup**
Pointer to a location where the address of the module cleanup routine is stored.

For any values that are not required, NU_NULL can be supplied, instead of a pointer for storing that particular value.

Return Values

- **NU_SUCCESS**
Function completed successfully
- **NU_MODULE_INVALID_MODULE_POINTER**
Indicates that the module pointer is NU_NULL.
- **NU_MODULE_INVALID_MODULE**
Indicates that the module pointer does not point to a valid module control block.
- **NU_MODULE_NOT_IN_SUPERVISOR_MODE**
Not called from supervisor mode.

Example

```
NU_MODULE Module;  
VOID cleanup;  
STATUS status;  
/* Obtain the address of the module's cleanup routine */  
status = NU_Module_Information(&Module, NU_NULL, &cleanup);
```

Related Topics

[Module Support Function Reference](#)

[NU_Create_Module](#)

NU_Module_Memory_Region_Pointers

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Module Support Services

Implemented by: MRFM_Module_Memory_Region_Pointers

This service builds a list of pointers to all memory regions attached to the specified module that share the specified memory region access, sharing, and/or caching options.

Usage

```
STATUS NU_Module_Memory_Region_Pointers (
    NU_MODULE          *module,
    UNSIGNED            option_mask,
    UNSIGNED            region_options,
    INT                 match_all,
    UNSIGNED            maximum_pointers,
    NU_MEMORY_REGION    *regions_array[],
    UNSIGNED            *region_count);
```

Arguments

- **module**
Pointer to the specified module control block.
- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as (actual options and option_mask).
- **region_options**
Specifies the memory region options to match; desired options is defined as (region_options AND option_mask). The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the pointer list if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, then a memory region is added to the pointer list if any of the options specified in desired options are present in compared options.
- **maximum_pointers**
Specifies the maximum size of the list.
- **regions_array**
Pointer to the location used for building the list of pointers to matched memory regions.
- **region_count**
Pointer to the location where the actual number of pointers placed in the list will be written.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_MODULE_POINTER**
Indicates the module pointer, region array pointer, or region count pointer is **NU_NULL**.
- **NU_MODULE_INVALID_MODULE**
Indicates the module pointer does not point to a valid module control block.
- **NU_MODULE_INVALID_POINTER**
Indicates region array pointer, or region count pointer is **NU_NULL**.
- **NU_MODULE_INVALID_SIZE**
Indicates the maximum pointers count is not greater than zero.

Description

This list of pointers can contain memory regions that exactly match a set of options, or memory regions that match at least one of a set of options. While the pointers in this list are unique, they are not guaranteed to be in a particular order.

Example

See similar example for [NU_Memory_Region_Pointers](#).

Related Topics

Module Support Function Reference	NU_Create_Module
NU_Detach_Memory_Region	NU_Memory_Region_Pointers
NU_Module_Memory_Regions	NU_Attach_Memory_Region

NU_Module_Memory_Regions

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Module Support Services

Implemented by: MRFM_Module_Memory_Regions

This service returns the number of memory regions attached to the specified module that share the specified memory region access, sharing, and/or caching options. This count can include either memory regions that exactly match a set of options, or memory regions that match at least one of a set of options.

Usage

```
UNSIGNED NU_Module_Memory_Regions (NU_MODULE    *module,  
                                   UNSIGNED       option_mask,  
                                   UNSIGNED       region_options,  
                                   INT            match_all);
```

Arguments

- **module**
Pointer to the specified module control block.
- **option_mask**
Specifies an “AND mask” indicating which memory region options are to be compared; compared options is defined as actual options and options_mask.
- **region_options**
Specifies the memory region options to match; desired options is defined as region_options and option_mask. The interpretation of this parameter depends on the value of the match_all parameter.
- **match_all**
Specifies how desired options and compared options are matched. If match_all is NU_TRUE, then a memory region is only added to the count if all of the options in compared options exactly match the options in desired options. If match_all is NU_FALSE, then a memory region is added to the count if any of the options specified in desired options are present in compared options.

Return Values

- Returns the number of memory regions attached to the specified module that share the specified memory region access, sharing, and/or caching options.

Example

See similar example for [NU_Memory_Regions](#).

Related Topics

[Module Support Function Reference](#)

[NU_Create_Module](#)

[NU_Detach_Memory_Region](#)

[NU_Memory_Regions](#)

[NU_Module_Memory_Region_Pointers](#)

[NU_Attach_Memory_Region](#)

NU_Module_Pointers

Allowed from: Application_Initialize, HISR, task

Category: Module Support Services

Implemented by: MSF_Module_Pointers

This service builds a sequential list of pointers to all established modules in the system.

Note



Modules that have been deleted are no longer considered established.

Usage

```
STATUS NU_Module_Pointers (UNSIGNED    maximum_pointers,  
                           NU_MODULE    *modules_array[],  
                           UNSIGNED     *module_count);
```

Arguments

- **maximum_pointers**
Specifies the maximum size of the list.
- **modules_array**
Pointer to the location used for building the list of pointers to established modules.
- **module_count**
Pointer to the location where the actual number of pointers placed in the list will be written.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_MODULE_INVALID_POINTER**
Indicates the module pointer array or the module count pointer is NU_NULL.
- **NU_MODULE_INVALID_MODULE_COUNT**
Indicates the size of the module pointer array is invalid.

Example

```
#define MAX_MODULES 10  
  
NU_MODULE *Module[MAX_MODULES];  
UNSIGNED modules;  
STATUS status;  
  
status = NU_Module_Pointers(MAX_MODULES, &Module[0], &modules);
```

Related Topics

[Module Support Function Reference](#)

[NU_Established_Modules](#)

[NU_Delete_Module](#)

[NU_Create_Module](#)

NU_Module_Task_Pointers

Allowed from: HISR, Signal Handler, Task

Category: Module Support Services

Implemented by: TCFM_Module_Task_Pointers

Usage

```
UNSIGNED NU_Module_Task_Pointers( NU_MODULE *module_ptr,  
                                  NU_TASK   **pointer_list,  
                                  UNSIGNED   maximum_pointers);
```

Arguments

- `module_ptr`
Pointer to module.
- `pointer_list`
Pointer to the list area.
- `maximum_pointers`
Maximum number of pointers.

Return Values

- Number of tasks placed in `pointer_list`.

Example

```
#define MAX_TASK_PTRS 5  
  
/* Create an array of task pointers. */  
NU_TASK* tasks[MAX_TASK_PTRS];  
UNSIGNED number_of_tasks;  
UNSIGNED num_ret;  
  
/* Get a list of tasks within the current module. */  
num_ret = NU_Module_Task_Pointers(NU_Current_Module_Pointer(),  
                                  tasks, MAX_TASK_PTRS);
```

Related Topics

[Module Support Function Reference](#)

NU_Read_Module_Options

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Module Support Services

Implemented by: MSS_Read_Module_Options

This service reads the module options of an existing module. This routine must be called from Supervisor mode.

Usage

```
UNSIGNED NU_Read_Module_Options(NU_MODULE *module_ptr);
```

Arguments

- `module_ptr`
Pointer to the module control block to read.

Return Values

- The options currently used by the module control block.

Example

```
UNSIGNED options;  
/* Read the options for my module */  
options = NU_Read_Options(my_module);
```

Related Topics

[Module Support Function Reference](#)

[NU_Change_Module_Options](#)

NU_Register_Exception_Handler

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Exception-Handler Services

Implemented by: EHC_Register_Exception_Handler

This function registers the desired exception handler with the specified module. It also returns the old exception handler. If NULL is passed as the new exception the function will not return success as modules requires some type of exception-handler.

Usage

```
STATUS NU_Register_Exception_Handler(  
    NU_MODULE *module_ptr,  
    VOID(*exception_entry)(NU_MODULE *,NU_EXCEPTION *),  
    VOID(**old_exception)(NU_MODULE *,NU_EXCEPTION *));
```

Arguments

- `module_ptr`
Pointer to module to update.
- `exception_entry`
Pointer to new exception-handler.
- `old_exception`
Pointer to previous exception-handler.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_MODULE_POINTER`
Module pointer is NULL.
- `NU_MODULE_INVALID_MODULE`
Module control block invalid.
- `NU_MODULE_INVALID_EXCEPTION_POINTER`
Exception pointer is NULL.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called from supervisor mode.

Example

```
status = NU_Register_Exception_Handler(my_module, my_handler,  
                                     save_handler);
```

Related Topics

[Module Support Function Reference](#)

[NU_Module_Exception_Handler](#)

NU_Set_Default_Exception_Handler

Allowed from: Application_Initialize, Supervisor-mode routine or task

Category: Exception-Handler Services

Implemented by: EHC_Set_Default_Exception_Handler

This function registers the desired exception handler with the system for assignment to new modules. If NULL is passed as the new exception the function will not return success.

Usage

```
STATUS NU_Set_Default_Exception_Handler(  
    VOID(*exception_entry)(NU_MODULE *, NU_EXCEPTION *),  
    VOID(**old_exception)(NU_MODULE *, NU_EXCEPTION *));
```

Arguments

- `exception_entry`
Pointer to new exception-handler.
- `old_exception`
Pointer to previous exception-handler.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_EXCEPTION_POINTER`
Exception pointer is NULL.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Not called from supervisor mode.

Example

```
status = NU_Set_Default_Exception_Handler(new_global_handler,  
    old_global_handler);
```

Related Topics

[Module Support Function Reference](#)

[NU_Get_Default_Exception_Handler](#)

NU_Start_Module

Allowed from: Application_Initialize, Supervisor mode routines or tasks

Tasking changes: While this service routine does not directly cause tasking changes, such changes could occur as a result of executing the user-supplied module initialization routine.

Category: Module Support Services

Implemented by: MSC_Start_Module

This service executes the module initialization routine of the given module, as specified to NU_Create_Module by the module_init parameter. A module cannot be started again without intervening calls to NU_Stop_Module, NU_Delete_Module, and then NU_Create_Module.

Usage

```
STATUS NU_Start_Module(NU_MODULE *module);
```

Arguments

- module
Pointer to the specified module control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_MODULE_INVALID_MODULE_POINTER
Indicates the module pointer is NU_NULL.
- NU_MODULE_INVALID_MODULE_START
Indicates the module initialization routine associated with the module is invalid.
- NU_MODULE_INVALID_MODULE
Indicates the module pointer does not point to a valid module control block.
- NU_MODULE_MODULE_ALREADY_STARTED
Indicates the module has already been started, and cannot be restarted.
- NU_MODULE_NOT_IN_SUPERVISOR_MODE
Indicates the service routine must be executed from Supervisor Mode (Supervisor/User mode switching targets only).
- other status values
Indicates a status value other than NU_SUCCESS was returned by the module_init routine.

Example

```
NU_MODULE Module;  
/* ...Module created but not started... */  
STATUS status;
```

NU_Start_Module

```
status = NU_Start_Module(&Module);  
status = NU_Stop_Module(&Module);
```

Related Topics

[Module Support Function Reference](#)

[NU_Delete_Module](#)

[NU_Stop_Module](#)

[NU_Create_Module](#)

NU_Stop_Module

Allowed from: Application_Initialize, supervisor mode routine or task

Tasking changes: While this service routine does not directly cause tasking changes, such changes could occur as a result of executing the user-supplied module_cleanup routine.

Category: Module Support Services

Implemented by: MSC_Stop_Module

This service executes the module cleanup routine of the given module, as specified to NU_Create_Module by the module_cleanup parameter.

Usage

```
STATUS NU_Stop_Module(NU_MODULE *module);
```

Arguments

- module
Pointer to the specified module control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_MODULE_INVALID_MODULE_POINTER
Indicates the module pointer is NU_NULL.
- NU_MODULE_MODULE_ALREADY_STOPPED
Indicates an attempt was made to stop an already-stopped module.
- NU_MODULE_NOT_IN_SUPERVISOR_MODE
Indicates this service routine must be called from Supervisor mode.
- NU_MODULE_INVALID_MODULE
Indicates the module pointer does not point to a valid module control block.
- other status values
Indicates a status value other than NU_SUCCESS was returned by the module cleanup routine.

Description

That cleanup routine is executed (in User mode, if applicable) to free up all Nucleus PLUS resources allocated by the module. Note that all tasks, HISRs, partition pools, memory pools, and other Nucleus PLUS kernel objects must be deleted before returning from the module cleanup routine. If a module cleanup routine was not specified at module creation, this service routine simply marks the module as stopped so that it can be deleted.

Example

See example for [NU_Delete_Module](#).

Related Topics

[Module Support Function Reference](#)

[NU_Delete_Module](#)

[NU_Start_Module](#)

[NU_Create_Module](#)

NU_Unbind_Module_HISR

Allowed from: Application_Initialize, supervisor mode routine or task

Category: Module Support Services

Implemented by: MSC_Unbind_Module_HISR

This routine clears the module pointer to the HISR control block. The HISR is also removed from the list of HISRs in the module and decrements the number of HISRs in the module.

Usage

```
STATUS NU_Unbind_Module_HISR(NU_HISR *hisr_ptr);
```

Arguments

- `hisr_ptr`
Pointer to HISR control block.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_MODULE_POINTER`
HISR module pointer is NULL.
- `NU_MODULE_INVALID_MODULE`
A valid module pointer does not exist in the HISR passed in.
- `NU_INVALID_HISR`
Bad HISR control block passed in.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Cannot execute from user mode.

Example

```
status = NU_Unbind_Module_HISR(hisr);
```

Related Topics

[Module Support Function Reference](#)

[NU_Bind_Module_HISR](#)

NU_Unbind_Module_Task

Allowed from: Application_Initialize, supervisor mode routine or task

Category: Module Support Services

Implemented by: MSC_Unbind_Module_Task

This routine clears the module pointer to the task control block. The task is also removed from the list of tasks in the module and decrements the number of tasks in the module.

Usage

```
STATUS NU_Unbind_Module_Task(NU_TASK *task_ptr);
```

Arguments

- `task_ptr`
Pointer to task control block.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_MODULE_INVALID_MODULE_POINTER`
Task module pointer is NULL.
- `NU_MODULE_INVALID_MODULE`
A valid module pointer does not exist in the task passed in.
- `NU_INVALID_TASK`
Bad task control block passed in.
- `NU_MODULE_NOT_IN_SUPERVISOR_MODE`
Cannot execute from user mode.

Example

```
status = NU_Unbind_Module_Task(task);
```

Related Topics

[Module Support Function Reference](#)

[NU_Unbind_Module_HISR](#)

[NU_Bind_Module_Task](#)

NU_Virtual_To_Physical

Allowed from: Application_Initialize, HISR, signal handler, task

Category: Memory Support Services

Implemented by: MST_Virtual_To_Physical

This service obtains the physical address of the specified virtual address in the context of the specified module.

Note



For most targets, the physical address returned by this function is the same as the virtual address supplied, as address translation is not currently implemented by Memory Region and Module Support services. This function should be used if future compatibility with address translation support is required.

Usage

```
STATUS NU_Virtual_To_Physical (NU_MODULE *module,  
                              VOID      *virtual_addr,  
                              VOID      **physical_addr);
```

Arguments

- **module**
Pointer to the specified module control block.
- **virtual_addr**
Address of the virtual memory location of which to obtain the physical address in the context of the specified module.
- **physical_addr**
Pointer to location in which to store the obtained physical address.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POINTER**
Indicates the module pointer or the physical address pointer is **NU_NULL**.
- **NU_NOT_PRESENT**
Indicates the module pointer does not point to a valid module control block.

Example

```
NU_MODULE Module;  
VOID *phys_addr;  
STATUS status;  
  
status = NU_Virtual_To_Physical(&Module, 0x1000, &phys_addr);
```

Related Topics

[Module Support Function Reference](#)

[NU_Place_Memory_Region](#)

[NU_Attach_Memory_Region](#)

Module Support Macro Definitions

Each Module Support macro for Nucleus PLUS is described in this section. The macros are described in alphabetical order.

- [NU_NEXT_PAGE_BOUNDARY](#)
- [NU_PAGE_EXPAND_SIZE](#)
- [NU_PAGE_TRUNCATE_SIZE](#)
- [NU_PREV_PAGE_BOUNDARY](#)

NU_NEXT_PAGE_BOUNDARY

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Memory Region Services

Implemented by: A macro definition in *numodule.h*

This macro returns the supplied address rounded up to the next minimum page boundary of the specific target, unless the supplied address is already an integer multiple of that minimum page size.

Macro

Usage

```
NU_NEXT_PAGE_BOUNDARY(address)
```

Parameters

- address

Address to round up to the next page boundary.

Example

```
/* next_page will be 0x00002000 if target page size is 4 Kbytes */  
unsigned next_page = NU_NEXT_PAGE_BOUNDARY(0x00001234);
```

Related Topics

[Module Support Macro Definitions](#) [NU_PREV_PAGE_BOUNDARY](#)

NU_PAGE_EXPAND_SIZE

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Memory Region Services

Implemented by: A macro definition in *numodule.h*

This macro returns the supplied size rounded up to the next multiple of the minimum page size of the specific target, unless the supplied size is already an integer multiple of that minimum page size.

Macro

Usage

```
NU_PAGE_EXPAND_SIZE(size)
```

Parameters

- size
Size to expand to the next multiple of the page size.

Example

```
/* mem_size will be 0x00002000 if target page size is 4 Kbytes */  
unsigned mem_size = NU_PAGE_EXPAND_SIZE(0x00001234);
```

Related Topics

[Module Support Macro Definitions](#) [NU_PAGE_TRUNCATE_SIZE](#)

NU_PAGE_TRUNCATE_SIZE

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Memory Region Services

Implemented by: A macro definition in *numodule.h*

This macro returns the supplied size rounded down to the next multiple of the minimum page size of the specific target, unless the supplied size is already an integer multiple of that minimum page size.

Macro

Usage

```
NU_PAGE_TRUNCATE_SIZE(size)
```

Parameters

- size
Size to truncate to the previous multiple of the page size.

Example

See similar example for [NU_PAGE_EXPAND_SIZE](#).

Related Topics

[Module Support Macro Definitions](#)

[NU_PAGE_EXPAND_SIZE](#)

NU_PREV_PAGE_BOUNDARY

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Memory Region Services

Implemented by: A macro definition in *numodule.h*

This macro returns the supplied address rounded down to the next minimum page boundary of the specific target, unless the supplied address is already an integer multiple of that minimum page size.

Macro

Usage

```
NU_PREV_PAGE_BOUNDARY(address)
```

Parameters

- address
Address to round down to the next page size.

Example

See similar example for [NU_NEXT_PAGE_BOUNDARY](#).

Related Topics

[Module Support Macro Definitions](#)

[NU_NEXT_PAGE_BOUNDARY](#)

Module Service Options

There are several services that a module may not need to access. Modules have the ability to set options to prevent certain calls from taking place. Options can be set through supplemental services provided. If the options are not set and a module tries to call one of the services the module's exception handler will be called to handle the error. The following table lists the option and which services are allowed to run with the option set:

Table 4-4. Module Service Options

Option	Services Allowed
NU_MOD_COMMON	NU_Activate_HISR NU_Check_Stack
NU_MOD_INTERRUPT	NU_Control_Interrupts NU_Local_Control_Interrupts
NU_MOD_SYSTEM_INT	NU_Restore_Interrupts

Table 4-4. Module Service Options (cont.)

Option	Services Allowed
NU_MOD_ALL_INTS	NU_MOD_INTERRUPT NU_MOD_SYSTEM_INT
NU_MOD_PROTECT	NU_Protect NU_Unprotect NU_Unprotect_Specific
NU_MOD_VECTOR	NU_Register_LISR
NU_MOD_CLOCK	NU_Set_Clock

Related Topics

[NU_Change_Module_Options](#)[NU_Read_Module_Options](#)[module_exception](#)

User-Defined Routines

Module Support for Nucleus PLUS makes use of two user-defined routines, `module_init` and `module_cleanup`. Implementations of these routines must typically be supplied by the application developer for each module created. The actual implementation and name of each of these routines is module-specific. The remainder of this section describes the required interface and behavior of these two user-defined routines.

- [module_cleanup](#)
- [module_init](#)

module_cleanup

Allowed from: Within NU_Stop_Module

Tasking changes: The module_cleanup service actually represents a pointer to a routine implemented by the application programmer.

Category: User-defined services

Implemented by: Possibly, if other Nucleus PLUS routines that cause tasking changes are called.

This service is actually a placeholder for one or more user-supplied functions. No service named module_cleanup actually exists in Nucleus PLUS.

Usage

```
STATUS module_cleanup(NU_MODULE *module);
```

Arguments

- module
Pointer to the module control block for which the module cleanup routine is being called.

Return Values

- NU_SUCCESS
Indicates that the cleanup routine successfully returned resources used by the module to Nucleus PLUS.
- other status
Indicates that the cleanup routine encountered an error while attempting to return resources used by the module to Nucleus PLUS.

Description

The user-supplied routine is invoked by NU_Stop_Module, if it was specified when the module was created with NU_Create_Module. The purpose of these user-supplied functions is to allow module-specific resources of the specified module to be disposed of before a module is actually deleted by NU_Delete_Module.

Example

See example for [NU_Create_Module](#).

Related Topics

User-Defined Routines	NU_Delete_Module
NU_Start_Module	NU_Stop_Module
NU_Create_Module	

module_init

Allowed from: Within `NU_Create_Module` or `NU_Start_Module`

Tasking changes: Possibly, if other Nucleus PLUS routines that cause tasking changes are called.

Category: User-defined services

Implemented by: The `module_init` service actually represents a pointer to a routine implemented by the application programmer.

This service is actually a placeholder for one or more user-supplied functions. No service named `module_init` actually exists in Nucleus PLUS.

Usage

```
STATUS module_init(NU_MODULE      *module,  
                  NU_MEMORY_POOL *mem_pool_ptr);
```

Arguments

- `module`
Pointer to the module control block for which the module initialization routine is being called.
- `mem_pool_ptr`
Pointer to the memory pool attached to the module intended to be used for dynamic memory needs.

Return Values

- `NU_SUCCESS`
Indicates that the initialization routine successfully received the resources required by the module from Nucleus PLUS.
- other status
Indicates that the initialization routine encountered an error while requesting resources used by the module from Nucleus PLUS.

Description

The user-supplied routine is invoked by `NU_Start_Module` (or `NU_Create_Module`, if the `NU_START` option was supplied), if it was specified when the module was created with `NU_Create_Module`. The purpose of these user-supplied functions is to allow module-specific resources of the specified module to be reserved after a module is created by `NU_Create_Module`.

Note



If `NU_START` is passed during module creation, the module is created regardless of the return value of the user initialization. If the initialization routine returns an error, the module will not be automatically deleted.

Example

See example for [NU_Create_Module](#).

Related Topics

User-Defined Routines	NU_Delete_Module
NU_Start_Module	NU_Stop_Module
NU_Create_Module	

module_exception

Allowed from: Exception safe memory

Category: User-defined services

Implemented by: The module_exception service actually represents a pointer to a routine implemented by the application programmer.

This service is actually a placeholder for one or more user-supplied functions. No service named module_exception actually exists in Nucleus PLUS or Modules Support.

Usage

```
VOID module_exception(NU_MODULE      *module,  
                     NU_EXCEPTION *exception);
```

Arguments

- **module**
Pointer to the module control block for which the module initialization routine is being called.
- **exception**
Pointer to control block with information on the exception such as the thread pointer and error type.

Description

The user-supplied routine is invoked by MMU based exceptions. The purpose of these user-supplied functions is to allow module-specific exception-handlers to be more versatile.

Example

See the file */os/arch/<architecture>/<sub architecture>/ehct.c*.

Related Topics

[User-Defined Routines](#) [NU_Set_Default_Exception_Handler](#)
[NU_Get_Default_Exception_Handler](#) [NU_Module_Exception_Handler](#)
[NU_Register_Exception_Handler](#)

Mode Switching Macros

Module Support also defines a number of macros useful in combination with the Module Support services. Each Module Support macro for Nucleus PLUS is described in this section.

- [NU_IS_SUPERVISOR_MODE](#)
- [NU_SUPERV_USER_VARIABLES](#)
- [NU_SUPERVISOR_MODE](#)

- `NU_USER_MODE`
- `NU_SUPERVISOR_MODE_ISR`
- `NU_USER_MODE_ISR`

NU_IS_SUPERVISOR_MODE

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro returns non-zero if the target is currently in Supervisor mode, and zero (0) if the target is currently in User mode. Depending on the particular implementation of this macro, the routine in which it is called must add `NU_SUPERV_USER_VARIABLES` to the local variables declared at the beginning of the calling routine.

Macro

Usage

```
NU_IS_SUPERVISOR_MODE ( )
```

Example

See example for [NU_SUPERV_USER_VARIABLES](#).

Related Topics

[Mode Switching Macros](#)

[NU_SUPERVISOR_MODE](#)

[NU_USER_MODE](#)

[NU_SUPERV_USER_VARIABLES](#)

NU_SUPERV_USER_VARIABLES

Allowed from: Application_Initialize, HISR, LISR, signal handler, task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro defines target-specific variables needed by the other Supervisor/User mode switching macros. This macro must be placed at the end of the variable list within the block (scope) where other Supervisor/User mode switching macros are used.

Macro

Usage

NU_SUPERV_USER_VARIABLES

Example

```
VOID f(VOID)
{
    /* Declare variables used by Supervisor/User macros */
    NU_SUPERV_USER_VARIABLES
    NU_SUPERVISOR_MODE();      /* Switch to Supervisor mode */
    if(NU_IS_SUPERVISOR_MODE())
    {
        /* ...Do something in Supervisor mode... */
    }
    NU_USER_MODE();           /* Revert to User mode */
}
```

Related Topics

[Mode Switching Macros](#)

[NU_SUPERVISOR_MODE](#)

[NU_USER_MODE](#)

[NU_IS_SUPERVISOR_MODE](#)

NU_SUPERVISOR_MODE

Allowed from: Application_Initialize, permitted HISR, permitted signal handler, permitted task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro invokes a target-specific mechanism to place the target processor into Supervisor mode, if the processor is currently in User mode. Use of this macro is restricted to "permitted object code". What object code is considered "permitted" is specific to a particular target release, but usually includes object code that is linked into the same section as the Nucleus PLUS kernel, and excluded object code in other sections of the executable file.

Macro

Usage

```
NU_SUPERVISOR_MODE ( )
```

Example

See example for [NU_SUPERV_USER_VARIABLES](#).

Related Topics

[Mode Switching Macros](#)

[NU_USER_MODE](#)

[NU_SUPERV_USER_VARIABLES](#)

[NU_IS_SUPERVISOR_MODE](#)

NU_SUPERVISOR_MODE_ISR

Allowed from: Application_Initialize, permitted HISR, permitted signal handler, permitted task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro invokes checks to see if the currently executing service routine is in a nested interrupt, such a case arises in LISRs. If not in a nested interrupt the macro then calls the macro NU_SUPERVISOR_MODE. This routine is useful with mode switching in drivers.

Macro

Usage

```
NU_SUPERVISOR_MODE_ISR()
```

Example

```
VOID f(VOID)
{
    /* Declare variables used by Supervisor/User macros */
    NU_SUPERV_USER_VARIABLES
    NU_SUPERVISOR_MODE_ISR();    /* Switch to Supervisor mode */
    if(NU_IS_SUPERVISOR_MODE())
    {
        /* ...Do something in Supervisor mode...          */
    }
    NU_USER_MODE_ISR();          /* Revert to User mode    */
}
```

Related Topics

[Mode Switching Macros](#)

[NU_SUPERVISOR_MODE](#)

[NU_USER_MODE](#)

[NU_SUPERV_USER_VARIABLES](#)

[NU_IS_SUPERVISOR_MODE](#)

NU_USER_MODE

Allowed from: Application_Initialize, permitted HISR, permitted signal handler, permitted task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro invokes a target-specific mechanism to place the target processor into User mode, if the processor is currently in Supervisor mode. Use of this macro is restricted to “permitted object code.” What object code is considered “permitted” is specific to a particular target release, but usually includes object code that is linked into the same section as the Nucleus PLUS kernel, and excluded object code in other sections of the executable file.

Macro

Usage

```
NU_USER_MODE ( )
```

Example

See example for [NU_SUPERV_USER_VARIABLES](#).

Related Topics

[Mode Switching Macros](#)

[NU_SUPERVISOR_MODE](#)

[NU_SUPERV_USER_VARIABLES](#)

[NU_IS_SUPERVISOR_MODE](#)

NU_USER_MODE_ISR

Allowed from: Application_Initialize, permitted HISR, permitted signal handler, permitted task

Category: Supervisor/User Services

Implemented by: A macro definition in *nucleus.h* (or *su_extr.h* if MMU support is present)

This macro invokes checks to see if the currently executing service routine is in a nested interrupt, such a case arises in LISRs. If not in a nested interrupt the macro then calls the macro NU_USER_MODE. This routine is useful with mode switching in drivers.

Macro

Usage

```
NU_USER_MODE_ISR ( )
```

Example

See example for [NU_SUPERVISOR_MODE_ISR](#).

Related Topics

[Mode Switching Macros](#)

[NU_SUPERVISOR_MODE](#)

[NU_SUPERV_USER_VARIABLES](#)

[NU_SUPERVISOR_MODE_ISR](#)

[NU_IS_SUPERVISOR_MODE](#)

Nucleus Plus Overview

This chapter describes the Nucleus Plus component of the Nucleus kernel. The Nucleus Plus component provides a framework for managing competing multiple tasks in a real time embedded application. It provides many mechanisms for control including mailboxes, queues, pipes, semaphores, event groups and signals. It also provides a management framework for real time external and internal interrupts (including timer interrupts) and system memory usage through dynamic and partition memory management.

To see examples of some of the services provided by the Nucleus Plus component, refer to [Nucleus Plus Examples](#).

Task Control

The Task Control services manage the execution of multiple, competing tasks in a real-time application. A task is a semi-independent program segment with a dedicated purpose. Most modern real-time applications require multiple tasks. Additionally, these tasks often have varying degrees of importance.

Task States

Each task is always in one of five states: executing, ready, suspended, terminated, or finished. The following list describes each of the task states:

Table 5-1. Task States

State	Meaning
executing	Task is currently running.
ready	Task is ready, but another task is currently running.
suspended	Task is dormant while waiting for the completion of a service request. When the request is complete, the task is moved to the ready state.
terminated	Task was killed. Once in this state, the task cannot execute again until it is reset.
finished	Task finished its processing and returned from initial entry routine. Once in this state, the task cannot execute again until it is reset.

Preemption

Preemption is the act of suspending a lower priority task when a higher priority task becomes ready. For example, suppose a task with a priority of 100 is executing. If an interrupt occurs that readies a task with a priority of 20, the task with priority 20 is executed before the interrupted task is resumed. Preemption also occurs when a lower priority task calls a Nucleus PLUS service that makes a higher priority task ready.

Preemption may be disabled on an individual task basis. When preemption is disabled, no other task is allowed to run until the executing task suspends, relinquishes control, or enables preemption. A task that suspends or relinquishes control with preemption disabled has preemption disabled when it is resumed.

A task is created with preemption either enabled or disabled. Preemption may also be enabled and disabled during task execution.

Relinquish

A mechanism is provided to share the processor with other ready tasks at the same priority level in a round-robin fashion. When a task requests this service, all other ready tasks at the same priority are executed before the originally executing task is resumed.

Time Slicing

Time slicing provides another mechanism to share the processor with tasks having the same priority. A time slice corresponds to the maximum number of timer ticks (timer interrupts) that can occur before all other ready tasks at the same priority level are given a chance to execute. A time slice behaves like an unsolicited task relinquish. Note that disabling preemption also disables time slicing.

Timer Interrupt

Nucleus PLUS requires a periodic interrupt in order to provide time-oriented services such as time-slicing, service call timeouts, and application timers. TMCT_Timer_Interrupt is the generic timer interrupt handler. Initialization and interrupt vector assignments are target specific.

Dynamic Task Creation

Nucleus PLUS tasks are created and deleted dynamically. There is no preset limit on the number of tasks an application may have. Each task requires a control block and a stack. The memory for each element is supplied by the application.

Task Determinism

Processing time associated with suspending and resuming tasks is a constant. It is not affected by the number of application tasks. Additionally, the method in which tasks execute is not only predictable, but also guaranteed. Higher priority, ready tasks execute before lower priority, ready tasks. Ready tasks of the same priority execute in the order they became ready.

Stack Checking

Application tasks may check the amount of memory left on the current stack. This function also keeps track of maximum stack usage. Stack checking may also be enabled inside Nucleus PLUS services through a conditional compilation option.

Task Information

Application tasks may obtain a list of active tasks. Detailed information about each task can also be obtained. This information includes the task name, current state, scheduled count, priority, and stack parameters.

Priority

A task's priority is defined during task creation. Additionally, dynamic modification of a task's priority is supported. A task that has a numerical priority of 0 has a higher priority than a task with a numerical priority of 255. Nucleus PLUS executes higher priority tasks before lower priority tasks. Tasks having the same priority are executed in the order in which they became ready for execution.

Note



Care must be taken when assigning priorities to application tasks. If care is not taken, the priorities can cause task starvation and excessive system overhead.

A task may only execute if it is the highest priority, ready task. Therefore, if a task or tasks at a certain priority are always ready, all tasks of a lower priority never execute. This situation is called starvation. There are several cures for this. First, higher priority tasks should suspend to allow lower priority tasks to execute.

Tasks that run at or near continuously should have a relatively low priority. Another technique to combat starvation is to gradually raise the priority of the starving task.

A substantial amount of additional overhead may be incurred if task priorities are used improperly. Consider a system of three tasks named A, B, and C. Each task has similar processing that consists of waiting for a message and/or sending a message in an infinite loop. Task A waits for a message from an Interrupt Service Routine (ISR) and, then, sends a message to task B. Task B waits for a message from task A and, then, sends a message to task C. Task C waits for a message from task B and, then, increments a counter. After this simple system starts (regardless of priority), all tasks execute briefly, and then suspend waiting for a message.

If all of the tasks have the same priority, the following set of events take place after the ISR sends a message to task A:

- Task A is resumed
- Task A sends a message to task B, making task B ready
- Task A suspends waiting for another message
- Task B is resumed
- Task B sends a message to task C, making task C ready
- Task B suspends waiting for another message
- Task C is resumed
- Task C increments a counter
- Task C suspends waiting for another message

Now assume that task A is lower priority than task B and task B is lower priority than task C. The following events take place after the ISR sends a message to task A:

- Task A is resumed
- Task A sends a message to task B, making task B ready
- Task A relinquishes to higher priority task B
- Task B is resumed
- Task B sends a message to task C, making task C ready
- Task B relinquishes to higher priority task C
- Task C is resumed
- Task C increments a counter
- Task C suspends waiting for another message
- Task B is resumed again
- Task B suspends waiting for another message

- Task A is resumed again
- Task A suspends waiting for another message

The application work performed in both of the previous examples is the same, that is, two tasks sent messages and three tasks received messages. However, the amount of system overhead in resuming and suspending tasks doubled. Also notice the delay incurred in task A between sending a message and waiting for another message in the last example.

Obviously the previous example systems are useful only to show how priorities can affect system overhead. Different priorities are necessary for real-time applications to respond to external events and to allocate processing time to relatively more important tasks. However, in order to reduce unnecessary system overhead, the number of different priorities in an application should be minimized.

Task Control Services Function Reference

The following function reference contains all Nucleus PLUS task control services. The following functions are contained in this reference:

- [NU_Change_Preemption](#)
- [NU_Change_Priority](#)
- [NU_Change_Time_Slice](#)
- [NU_Check_Stack](#)
- [NU_Create_Task](#)
- [NU_Create_Auto_Clean Task](#)
- [NU_Current_Task_Pointer](#)
- [NU_Delete_Task](#)
- [NU_Established_Tasks](#)
- [NU_Relinquish](#)
- [NU_Reset_Task](#)
- [NU_Resume_Task](#)
- [NU_Sleep](#)
- [NU_Suspend_Task](#)
- [NU_Task_Information](#)
- [NU_Task_Pointers](#)
- [NU_Terminate_Task](#)

NU_Change_Preemption

Allowed from: Task

Category: Task Control Services

Tasking changes: Yes

This service changes the preemption posture of the currently executing task.

Usage

```
OPTION NU_Change_Preemption (OPTION preempt);
```

Arguments

- `preempt`
Valid options for this parameter are `NU_PREEMPT` and `NU_NO_PREEMPT`.
`NU_PREEMPT` indicates that the task is preemptable, while `NU_NO_PREEMPT` indicates that the task is not preemptable. Time slicing is disabled if the task is not preemptable.

Return Values

- The previous preemption posture (either `NU_NO_PREEMPT` or `NU_PREEMPT`) is returned.

Description

If the `preempt` parameter contains `NU_NO_PREEMPT`, preemption of the calling task is disabled. Otherwise, if the `preempt` parameter contains `NU_PREEMPT`, preemption of the calling task is enabled.

Note



Disabling preemption also disables any time-slice associated with the calling task.

Example

```
OPTION    old_preempt;  
/* Disable preemption of the current task. */  
old_preempt = NU_Change_Preemption(NU_NO_PREEMPT);  
.  
.  
.  
/* Restore previous preemption posture. */  
NU_Change_Preemption(old_preempt);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Change_Time_Slice](#)

[NU_Create_Task](#)

[NU_Change_Priority](#)

NU_Change_Priority

Allowed from: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking changes: Yes

This service changes the priority of the specified task to the value contained in new_priority.

Usage

```
OPTION NU_Change_Priority (NU_TASK *task,  
                           OPTION new_priority);
```

Arguments

- **task**
Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.
- **new_priority**
Specifies a priority value between 0 and 255. The lower the numeric value, the higher the task's priority.

Return Values

- This service returns the previous priority to the caller.

Description

Priorities are numerical values ranging from 0 to 255. Lower numerical values indicate greater task priority.

Example

```
NU_TASK    Task;  
OPTION     old_priority;  
.  
.  
.  
/* Change the priority of the task control block "Task" to priority 10.  
   Assume "Task" has previously been created with the Nucleus PLUS  
   NU_Create_Task service call. */  
old_priority = NU_Change_Priority(&Task, 10);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Change_Time_Slice](#)

[NU_Create_Task](#)

[NU_Change_Preemption](#)

NU_Change_Time_Slice

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service changes the time slice of the specified task to the value contained in `time_slice`. If `time_slice` contains a value of 0, time slicing for the task is disabled.

Usage

```
UNSIGNED NU_Change_Time_Slice (NU_TASK  *task,  
                               UNSIGNED  time_slice);
```

Arguments

- `task`
Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.
- `time_slice`
Indicates the maximum amount of timer ticks that can expire while executing this task. A value of 0 in this field disables time slicing for this task.

Return Values

- This service returns the previous time slice value to the caller.

Example

```
NU_TASK      Task;  
UNSIGNED     old_time_slice;  
.  
.  
.  
/* Change the time slice of the task control block "Task" to 35 timer  
   ticks.  
   Assume "Task" has previously been created with the Nucleus PLUS  
   NU_Create_Task service call.*/  
old_time_slice = NU_Change_Time_Slice(&Task, 35);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Change_Priority](#)

[NU_Create_Task](#)

[NU_Change_Preemption](#)

NU_Check_Stack

Allowed From: HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service examines the stack usage of the caller.

Usage

```
UNSIGNED NU_Check_Stack (VOID);
```

Note



Stack checking functionality is conditionally compiled.

Return Values

- This service returns the number of bytes currently available on the caller's stack.

Description

If the remaining amount of space is less than that required to save the caller's context, a stack overflow condition is present and control will not return to the caller. If a stack overflow condition is not present, the service returns the number of free bytes remaining in the stack. Additionally, this service keeps track of the minimum amount of available stack space.

Example

```
UNSIGNED remaining;

/* Check the current stack for an overflow condition. Store the number of
   free stack bytes in "remaining". */
remaining = NU_Check_Stack( );
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Create_HISR](#)

[NU_Create_Task](#)

NU_Create_Task

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: Yes

This service creates an application task.

Usage

```
STATUS NU_Create_Task (NU_TASK  *task,  
                      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);
```

Arguments

- **task**
Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.
- **name**
Pointer to a seven-character name for the task. The name must be null-terminated.
- **task_entry**
Specifies the entry function of the task.
- **argc**
An UNSIGNED data element that may be used to pass initial information to the task.
- **argv**
A pointer that may be used to pass information to the task.
- **stack_address**
Designates the starting memory location of the task's stack.
- **stack_size**
Specifies the number of bytes in the stack.
- **priority**
Specifies a priority value between 0 and 255. The lower the numeric value, the higher the task's priority.

- **time_slice**
 Indicates the maximum amount of timer ticks that can expire while executing this task. A value of 0 in this field disables time slicing for this task.
- **preempt**
 Valid options for this parameter are **NU_PREEMPT** and **NU_NO_PREEMPT**. **NU_PREEMPT** indicates that the task is preemptable, while **NU_NO_PREEMPT** indicates that the task is not preemptable. Time slicing is disabled if the task is not preemptable.
- **auto_start**
 Valid options for this parameter are **NU_START** and **NU_NO_START**. **NU_START** places the task in a ready state after it is created. Tasks created with **NU_NO_START** must be resumed later.

Return Values

- **NU_SUCCESS**
 Function completed successfully.
- **NU_INVALID_TASK**
 The task control block pointer is NULL.
- **NU_INVALID_ENTRY**
 The task's entry function pointer is NULL.
- **NU_INVALID_MEMORY**
 The memory area specified by the `stack_address` is NULL.
- **NU_INVALID_SIZE**
 The specified stack size is not large enough.
- **NU_INVALID_PREEMPT**
 The `preempt` parameter is invalid.
- **NU_INVALID_START**
 The `auto_start` parameter is invalid.
- **NU_INVALID_PRIORITY**
 Priority exceeds the maximum configured priorities.

Example

```
/* Assume task control block "Task" is defined as global data structure.
   This is one of several ways to allocate a control block. */

NU_TASK    Task;
STATUS     status;    /* Task creation status */
.
.
```

```
/* Create a task whose entry point is the function "task_entry" and that
   has a 2000-byte stack pointed to by "stack_ptr".
   Note the following additional parameters:
       argc and argv (0, NULL)
       priority is 200
       15 timer-tick time slice
       preemptable
       automatic start */

status = NU_Create_Task(&Task, "any name", task_entry, 0, NULL, stack_ptr,
                        2000, 200, 15, NU_PREEMPT, NU_START);

/* At this point status indicates if the service was successful. */
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Established_Tasks](#)

[NU_Reset_Task](#)

[NU_Task_Information](#)

[NU_Task_Pointers](#)

[Kernel Demo](#)

[NU_Delete_Task](#)

NU_Create_Auto_Clean Task

Tasking Changes: Yes

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

This service creates an application task that automatically manages basic task resource allocation and de-allocation (of task control block and stack memory). Resources will be allocated at the time of the service call and de-allocated when the task is terminated or the task entry function exits.

Usage

```
STATUS TCCT_Create_Auto_Clean_Task(
    NU_TASK      **task_ptr,
    CHAR         *name,
    VOID         (*task_entry)(UNSIGNED, VOID *),
    UNSIGNED     argc,
    VOID         *argv,
    NU_MEMORY_POOL *pool_ptr,
    UNSIGNED     stack_size,
    OPTION       priority,
    UNSIGNED     time_slice,
    OPTION       preempt,
    OPTION       auto_start);
```

Arguments

- **task_ptr**
Return pointer that will be updated to contain a pointer to the created task control block. All subsequent requests made to this task require the pointer to the control block. NU_NULL can be passed if no return pointer is desired.
- **name**
Pointer to a seven-character name for the task. The name must be null-terminated.
- **task_entry**
Specifies the entry function of the task.
- **argc**
An UNSIGNED data element that is used to pass initial information to the task.
- **argv**
A pointer that is used to pass information to the task.
- **pool_ptr**
Nucleus memory pool that will be used for resource allocations. Resources will be de-allocated back to this pool when the task finishes or is terminated.

- **stack_size**
Specifies the number of bytes in the stack.
- **priority**
Specifies a priority value between 0 and 255. The lower the numeric value, the higher the task's priority.
- **time_slice**
Indicates the maximum amount of timer ticks that can expire while executing this task. A value of 0 in this field disables time slicing for this task.
- **preempt**
Valid options for this parameter are **NU_PREEMPT** and **NU_NO_PREEMPT**.
NU_PREEMPT indicates that the task is preemptable, while **NU_NO_PREEMPT** indicates that the task is not preemptable. Time slicing is disabled if the task is not preemptable.
- **auto_start**
Valid options for this parameter are **NU_START** and **NU_NO_START**. **NU_START** places the task in a ready state after it is created. Tasks created with **NU_NO_START** must be resumed later.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_ENTRY**
The task's entry function pointer is **NULL**.
- **NU_INVALID_SIZE**
The specified stack size is not large enough.
- **NU_INVALID_PREEMPT**
The preempt parameter is invalid.
- **NU_INVALID_START**
The auto_start parameter is invalid.

Example

```
NU_TASK *          task_ptr;
NU_MEMORY_POOL *   mem_pool;
NU_MEMORY_POOL *   uncached_mem_pool;
STATUS             status;

/* Retrieve a pointer to the system memory resources. */
status = NU_System_Memory_Get(&mem_pool, &uncached_mem_pool);
if (status == NU_SUCCESS)
{
```

```
/* Create an auto-clean task whose entry point is the function
   "task_entry" and has a 2000-byte stack. Memory resources will be
   allocated from the cached system memory pool.

   Note the following additional parameters:
       argc and argv (0, NULL)
       priority is 200
       15 timer-tick time slice
       preemptable
       automatic start */

status = NU_Create_Auto_Clean_Task(&task_ptr, "any name", task_entry,
                                   0, NULL, mem_pool, 2000, 200, 15,
                                   NU_PREEMPT, NU_START);

/* At this point status indicates if the service was successful. */
}
```

Related Topics

Task Control Services Function Reference	NU_Established_Tasks
NU_Reset_Task	NU_Task_Information
NU_Task_Pointers	NU_Delete_Task
NU_Create_Task	

NU_Current_Task_Pointer

Allowed From: LISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service returns the currently active task pointer. If no task is currently active, an NU_NULL is returned. If a HISR is the active thread, and a task that could resume after the HISR completes has been interrupted, the Return Value is still NU_NULL.

Usage

```
NU_TASK *NU_Current_Task_Pointer (VOID);
```

Return Values

- This service returns a pointer to the currently active task control block.

Example

```
NU_TASK *task_ptr;  
  
/* Obtain the currently active task's pointer. */  
task_ptr = NU_Current_Task_Pointer( );
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Task_Information](#)

[NU_Task_Pointers](#)

[Kernel Demo](#)

[NU_Established_Tasks](#)

NU_Delete_Task

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service deletes a previously created application task. The parameter task identifies the task to delete. Note that the specified task must be either in a finished or terminated state prior to calling this service. Additionally, the application must prevent the use of this task during and after deletion.

Usage

```
STATUS NU_Delete_Task (NU_TASK *task);
```

Arguments

- task
Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TASK
The task pointer is invalid.
- NU_INVALID_DELETE
The task is not in a finished or terminated state.

Example

```
NU_TASK    Task;
STATUS     status
.
.
/* Delete the task control block "Task". Assume "Task" has
   previously been created with the Nucleus PLUS NU_Create_Task
   service call. */
status = NU_Delete_Task(&Task);
/* At this point, status indicates whether the service request was
   successful. */
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Established_Tasks](#)

[NU_Reset_Task](#)

[NU_Task_Information](#)

[NU_Task_Pointers](#)

[NU_Create_Task](#)

NU_Established_Tasks

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service returns the number of established application tasks. All created tasks are considered established. Deleted tasks are no longer considered established.

Usage

```
UNSIGNED NU_Established_Tasks (VOID);
```

Return Values

- This service call returns the number of established application tasks.

Example

```
UNSIGNED  total_tasks;

/* Obtain the total number of tasks. */
total_tasks = NU_Established_Tasks( );
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Delete_Task](#)

[NU_Reset_Task](#)

[NU_Task_Information](#)

[NU_Task_Pointers](#)

[NU_Create_Task](#)

NU_Relinquish

Allowed From: Task

Category: Task Control Services

Tasking Changes: Yes

This service allows all other ready tasks of the same priority a chance to execute before the calling task runs again.

Usage

```
VOID NU_Relinquish (VOID);
```

Example

```
/* Allow other tasks that are ready at the same priority to execute before  
   the calling task resumes.  */  
NU_Relinquish( );
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Resume_Task](#)

[NU_Sleep](#)

[NU_Suspend_Task](#)

[NU_Task_Information](#)

[NU_Terminate_Task](#)

[NU_Reset_Task](#)

NU_Reset_Task

Allowed From: HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service resets a previously terminated or finished task.

Usage

```
STATUS NU_Reset_Task (NU_TASK  *task,  
                     UNSIGNED  argc,  
                     VOID      *argv);
```

Note



This service does not resume the task after it is reset. `NU_Resume_Task` must be called to actually start the task again. The parameters of this service are further defined as follows.

Arguments

- `task`
Pointer to the task control block.
- `argc`
An UNSIGNED data element that may be used to pass information to the task.
- `argv`
A pointer that may be used to pass information to the task.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_TASK`
The task pointer is invalid.
- `NU_NOT_TERMINATED`
The specified task is not in a terminated or finished state. Only tasks in a terminated or finished state can be reset.

Example

```
NU_TASK    Task;  
STATUS     status  
.  
.  
.
```

```
/* Reset the previously terminated task control block "Task". Pass the
   task values of 0 and NULL for argc and argv. Assume "Task" has
   previously been created with the Nucleus PLUS NU_Create_Task service
   call. */
status = NU_Reset_Task(&Task, 0, NULL);
```

Related Topics

Task Control Services Function Reference	NU_Delete_Task
NU_Resume_Task	NU_Suspend_Task
NU_Task_Information	NU_Terminate_Task
NU_Create_Task	

NU_Resume_Task

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: Yes

This service resumes a task that was previously suspended by the NU_Suspend_Task service. Additionally, this service initiates a task that was previously reset or created without an automatic start.

Usage

```
STATUS NU_Resume_Task (NU_TASK *task);
```

Arguments

- task
Pointer to the user-supplied task control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TASK
The task pointer is invalid.
- NU_INVALID_RESUME
The specified task is not in an unconditionally suspended state.

Example

```
NU_TASK  Task;  
STATUS   status  
.  
.  
.  
/* Resume the task control block "Task". Assume "Task" has previously been  
   created with the Nucleus PLUS NU_Create_Task service call.  */  
status =  NU_Resume_Task(&Task);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Reset_Task](#)

[NU_Suspend_Task](#)

[NU_Task_Information](#)

[NU_Create_Task](#)

NU_Sleep

Allowed From: Task

Category: Task Control Services

Tasking Changes: Yes

This service suspends the calling task for the specified number of timer ticks.

Usage

```
VOID NU_Sleep (UNSIGNED ticks);
```

Arguments

- ticks
Number of timer ticks that the task will be suspended.

Example

```
/* Sleep for 20 timer ticks */  
NU_Sleep(20);
```

Related Topics

[Task Control Services Function Reference](#)

[Kernel Demo](#)

[NU_Relinquish](#)

NU_Suspend_Task

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: Yes

This task unconditionally suspends the task specified by the pointer task. If the task is already in a suspended state, this service insures that the task stays suspended even after its original cause for suspension is lifted. NU_Resume_Task must be used to resume a task suspended in this manner.

Usage

```
STATUS NU_Suspend_Task (NU_TASK *task);
```

Arguments

- task
Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TASK
The task pointer is invalid.
- NU_INVALID_SUSPEND
The specified task has a status of NU_FINISHED or NU_TERMINATED.

Example

```
NU_TASK    Task;  
STATUS     status;  
.  
.  
  
/* Unconditionally suspend the task control block "Task". Assume "Task"  
   has previously been created with Nucleus PLUS NU_Create_Task service  
   call. */  
status = NU_Suspend_Task(&Task);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Resume_Task](#)

[NU_Terminate_Task](#)

[NU_Reset_Task](#)

NU_Task_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service returns various information about the specified task.

Usage

```
STATUS NU_Task_Information (NU_TASK      *task,  
                           CHAR          *name,  
                           DATA_ELEMENT *task_status,  
                           UNSIGNED      *scheduled_count,  
                           OPTION        *priority,  
                           OPTION        *preempt,  
                           UNSIGNED      *time_slice,  
                           VOID          **stack_base,  
                           UNSIGNED      *stack_size,  
                           UNSIGNED      *minimum_stack);
```

Arguments

- task
Pointer to the task.
- name
Pointer to an eight-character destination area for the task's name. This includes space for the null terminator.
- task_status
Pointer to a variable to hold the current status of the task.
- scheduled_count
Pointer to a variable to hold the number of times the task has been scheduled.
- priority
Pointer to a variable to hold the task's priority.
- preempt
Pointer to a variable to hold the task's preempt option. NU_PREEMPT indicates the task is preemptable, while NU_NO_PREEMPT indicates the task is not preemptable.
- time_slice
Pointer to a variable to hold the task's time slice value. A value of 0 indicates that time slicing for this task is disabled.
- stack_base
Pointer to a memory pointer to hold the starting address of the task's stack.

- `stack_size`
 Pointer to a variable to hold the total number of bytes in the task's stack.
- `minimum_stack`
 Pointer to a variable to hold the minimum amount of bytes left in the task's stack.

Task Status

The following table summarizes the possible values for the `task_status` parameter.

- `NU_READY`
 Ready to execute.
- `NU_PURE_SUSPEND`
 Unconditionally suspended.
- `NU_FINISHED`
 Returned from the entry function.
- `NU_TERMINATED`
 Terminated.
- `NU_SLEEP_SUSPEND`
 Sleeping.
- `NU_MAILBOX_SUSPEND`
 Suspended on a mailbox.
- `NU_QUEUE_SUSPEND`
 Suspended on a queue.
- `NU_PIPE_SUSPEND`
 Suspended on a pipe.
- `NU_EVENT_SUSPEND`
 Suspended on an event-flag group.
- `NU_SEMAPHORE_SUSPEND`
 Suspended on a semaphore.
- `NU_MEMORY_SUSPEND`
 Suspended on a dynamic-memory pool.
- `NU_PARTITION_SUSPEND`
 Suspended on a memory-partition pool.
- `NU_DRIVER_SUSPEND`
 Suspended from an I/O driver request.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_INVALID_TASK**
Indicates the task pointer is invalid.

Example

```
NU_TASK      Task;
CHAR         task_name[8];
DATA_ELEMENT task_status;
UNSIGNED     scheduled_count;
OPTION       priority;
OPTION       preempt;
UNSIGNED     time_slice;
VOID         *stack_base;
UNSIGNED     stack_size;
UNSIGNED     minimum_stack;
STATUS       status;
.
.
.
/* Obtain information about the task control block "Task". Assume "Task"
   has previously been created with the Nucleus PLUS NU_Create_Task
   service call. */
status = NU_Task_Information(&task, task_name, &task_status,
                           &scheduled_count, &priority, &preempt,
                           &time_slice, &stack_base,
                           &stack_size, &minimum_stack);
/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

Task Control Services Function Reference	NU_Delete_Task
NU_Established_Tasks	NU_Reset_Task
NU_Task_Pointers	NU_Create_Task

NU_Task_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: No

This service builds a sequential list of pointers to all established tasks in the system.

Usage

```
UNSIGNED NU_Task_Pointers (NU_TASK  **pointer_list,  
                           UNSIGNED maximum_pointers);
```

Note



Tasks that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Arguments

- `pointer_list`
Pointer to an array of `NU_TASK` pointers. This array will be filled with pointers to established tasks in the system.
- `maximum_pointers`
The maximum number of `NU_TASK` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of `NU_TASK` pointers placed into the array.

Example

```
/* Define an array capable of holding 20 task pointers. */  
NU_TASK  *Pointer_Array[20];  
UNSIGNED  number;  
  
/* Obtain a list of currently active task pointers (Max of 20). */  
number = NU_Task_Pointers(&Pointer_Array[0],20);  
  
/* At this point, number contains the actual number of pointers in the  
   list. */
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Delete_Task](#)

[NU_Established_Tasks](#)

[NU_Reset_Task](#)

[NU_Create_Task](#)

NU_Terminate_Task

Allowed From: HISR, Signal Handler, Task

Category: Task Control Services

Tasking Changes: Yes

This service terminates the task specified by the task parameter.

Note



A terminated task cannot execute again until it is reset.

Note



When calling this function from a signal handler, the task whose signal handler is executing cannot be terminated.

Usage

```
STATUS NU_Terminate_Task (NU_TASK *task);
```

Arguments

- **task**
 Pointer to the user-supplied task control block. All subsequent requests made to this task require this pointer.

Return Values

- **NU_SUCCESS**
 Function completed successfully.
- **NU_INVALID_TASK**
 Indicates the task pointer is invalid.

Example

```
NU_TASK    Task;
STATUS     status;
.
.
.
/* Terminate the task control block "Task". Assume "Task" has previously
   been created with the Nucleus PLUS NU_Create_Task service call. */
status = NU_Terminate_Task(&Task);
```

Related Topics

[Task Control Services Function Reference](#)

[NU_Resume_Task](#)

[NU_Suspend_Task](#)

[NU_Task_Information](#)

[NU_Reset_Task](#)

Dynamic Memory

A dynamic memory pool contains a user-specified number of bytes. The memory location of the pool is determined by the application. Variable-length allocation and deallocation services are provided for the dynamic memory pool. Allocations are performed in a first-fit manner, for example, the first available memory that satisfies the request is allocated. If the allocated block is significantly larger than the request, the unused memory is returned to the dynamic memory pool.

Each allocation from a memory pool requires some additional overhead to allow for its pointer structure. This overhead is consumed out of the memory pool from which the allocation is requested.

Dynamic Memory Suspension

The allocate dynamic memory service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to allocate dynamic memory from a pool that does not currently have enough available memory may suspend. Resumption of the task is possible when enough previously allocated memory is returned to the pool.

Multiple tasks may suspend on a single dynamic memory pool. Tasks are suspended in either FIFO or priority order, depending on how the dynamic memory pool was created. If the dynamic memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the dynamic memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Dynamic Memory Component Dynamic Creation

Nucleus PLUS dynamic memory pools are created and deleted dynamically. There is no preset limit on the number of dynamic memory pools an application may have. Each dynamic memory pool requires a control block and a pointer to the actual dynamic memory area. The memory for both the control block and the memory area is supplied by the application.

Dynamic Memory Determinism

Allocating memory from a dynamic memory pool is inherently undeterministic. This is largely due to possible memory fragmentation within the pool. The first-fit algorithm is basically a linear search, and as a result the worst-case performance depends on the amount of fragmentation.

However, memory deallocation is constant. Processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the dynamic memory pool.

Dynamic Memory Pool Information

Application tasks may obtain a list of active dynamic memory pools. Detailed information about each dynamic memory pool is also available. This information includes the dynamic memory pool name, starting pool address, total size, free bytes, number of tasks suspended, and the identity of the first suspended task.

Dynamic Memory Services Function Reference

The following function reference contains all functions related to the Nucleus PLUS dynamic memory component. The following functions are contained in this reference:

- [NU_Add_Memory](#)
- [NU_Allocate_Memory](#)
- [NU_Allocate_Aligned_Memory](#)
- [NU_Create_Memory_Pool](#)
- [NU_Deallocate_Memory](#)
- [NU_Delete_Memory_Pool](#)
- [NU_Established_Memory_Pools](#)
- [NU_Memory_Pool_Information](#)
- [NU_Memory_Pool_Pointers](#)
- [NU_Reallocate_Aligned_Memory](#)

- [NU_Reallocate_Memory](#)

NU_Add_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service adds a block of memory to the specified dynamic memory pool.

Usage

```
STATUS NU_Add_Memory (NU_MEMORY_POOL *pool_ptr,  
                     VOID *memory_start_address,  
                     UNSIGNED memory_size);
```

Arguments

- `pool_ptr`
Pointer to the dynamic memory pool control block.
- `memory_start_address`
Pointer to the new memory to be added to the memory pool.
- `memory_size`
Specifies the number of bytes of memory being added to the memory pool.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_POOL`
The dynamic memory pool control block is invalid.
- `NU_INVALID_POINTER`
The start address is invalid.
- `NU_INVALID_SIZE`
Indicates an invalid size request.
- `NU_NOT_ALIGNED`
Indicates the start address is not aligned.

Example

```
NU_MEMORY_POOL Pool;  
VOID *memory_ptr;  
STATUS status  
.  
.  
.
```

```
/* Add a 300-byte block of memory to the memory pool control block
"Pool". Assume "Pool" has previously been created with
the Nucleus PLUS NU_Create_Memory_Pool service call and that memory_ptr
contains the start address of the memory to add. */
status = NU_Add_Memory(&Pool, memory_ptr, 300);
/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Memory_Pool_Information](#)

[NU_Create_Memory_Pool](#)

NU_Allocate_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service allocates a block of memory from the specified dynamic memory pool.

Usage

```
STATUS NU_Allocate_Memory (NU_MEMORY_POOL *pool,
                           VOID **return_pointer,
                           UNSIGNED size,
                           UNSIGNED suspend);
```

Arguments

- **pool**
 Pointer to the dynamic memory pool.
- **return_pointer**
 Pointer to the caller's memory pointer. On a successful request, the address of the allocated block is placed in the caller's memory pointer.
- **size**
 Specifies the number of bytes to allocate from the dynamic memory pool. A value of 0 will return an error if error-checking is enabled.
- **suspend**
 Specifies whether or not to suspend the calling task if the requested amount of memory is not available. The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the requested memory is available.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the memory is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
 Function completed successfully.

- **NU_INVALID_POOL**
The dynamic memory pool is invalid.
- **NU_INVALID_POINTER**
The return pointer is NULL.
- **NU_INVALID_SIZE**
Indicates an invalid size request.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_NO_MEMORY**
Indicates the memory request could not be immediately satisfied.
- **NU_TIMEOUT**
Indicates the requested memory is still unavailable even after suspending for the specified timeout value.
- **NU_POOL_DELETED**
Dynamic memory pool was deleted while the task was suspended.

Example

```
NU_MEMORY_POOL Pool;
VOID           memory_ptr;
STATUS        status
.
.
.
/* Allocate a 300-byte block of memory with the memory pool control block
"Pool". If the requested memory is unavailable, suspend the calling
task unconditionally. Assume "Pool" has previously been created with
the Nucleus PLUS NU_Create_Memory_Pool service call. */
status = NU_Allocate_Memory(&Pool, &memory_ptr, 300, NU_SUSPEND);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Deallocate_Memory](#)

[NU_Memory_Pool_Information](#)

[Kernel Demo](#)

[NU_Allocate_Aligned_Memory](#)

NU_Allocate_Aligned_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service allocates an aligned block of memory from the specified dynamic memory pool.

Usage

```
STATUS NU_Allocate_Aligned_Memory (NU_MEMORY_POOL *memory,
                                   VOID **return_pointer,
                                   UNSIGNED size,
                                   UNSIGNED alignment,
                                   UNSIGNED suspend);
```

Arguments

- **memory**
 Pointer to the dynamic memory pool
- **return_pointer**
 Pointer to the caller's memory pointer. On a successful request, the address of the allocated block is placed in the caller's memory pointer.
- **size**
 Specifies the number of bytes to allocate from the dynamic memory pool.
- **alignment**
 Specifies the required alignment of the starting address of the requested memory block. This specified alignment must be an integer multiple of the minimum granularity of the dynamic memory pool. On most targets, this is size of (UNSIGNED).
- **suspend**
 Specifies whether or not to suspend the calling task if the requested amount of memory is not available. The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the requested memory is available.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the memory is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POOL**
The dynamic memory pool is invalid.
- **NU_INVALID_POINTER**
The return pointer is **NU_NULL**.
- **NU_INVALID_SIZE**
Indicates an invalid size request.
- **NU_INVALID_SUSPEND**
Indicates suspend was attempted from a non-task thread.
- **NU_INVALID_OPERATION**
Indicates an invalid alignment request.
- **NU_NO_MEMORY**
Indicates the memory request could not be immediately satisfied.
- **NU_TIMEOUT**
Indicates the requested memory is still unavailable even after suspending for the specified timeout value.
- **NU_POOL_DELETED**
Indicates the dynamic memory pool was deleted while the task was suspended.

Example

```
NU_MEMORY_POOL  Pool;
VOID            memory_ptr;
STATUS          status
.
.
.
/* Allocate a 300-byte block of memory that is aligned on a 1024 byte
   boundary. This allocates with the memory pool control block "Pool".
   If the requested memory is unavailable, suspend the calling task
   unconditionally. Assume "Pool" has previously been created with the
   Nucleus PLUS NU_Create_Memory_Pool service call. */
status =  NU_Allocate_Aligned_Memory(&Pool, &memory_ptr, 300, 1024,
                                     NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. */
```


Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Deallocate_Memory](#)

[NU_Memory_Pool_Information](#)

[NU_Reallocate_Aligned_Memory](#)

[NU_Allocate_Memory](#)

NU_Create_Memory_Pool

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service creates a dynamic memory pool inside a memory area specified by the caller.

Usage

```
STATUS NU_Create_Memory_Pool (NU_MEMORY_POOL *pool,  
                              CHAR             *name,  
                              VOID             *start_address,  
                              UNSIGNED         pool_size,  
                              UNSIGNED         min_allocation,  
                              OPTION          suspend_type);
```

Arguments

- **pool**
Pointer to the user-supplied memory pool control block. All subsequent requests made to the memory pool require this pointer.
- **name**
Pointer to a seven-character name for the memory pool. The name must be null-terminated.
- **start_address**
Specifies the starting address for the memory pool.
- **pool_size**
Specifies the number of bytes in the memory pool.
- **min_allocation**
Specifies the minimum number of bytes in each allocation from this memory pool.
- **suspend_type**
Specifies how tasks suspend on the memory pool. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent FIFO and priority-order task suspension, respectively.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POOL**
The memory pool control block pointer is NULL or is already in use.
- **NU_INVALID_MEMORY**
The memory area specified by the start address is invalid.
- **NU_INVALID_SIZE**
The pool size and/or the minimum allocation size are invalid.
- **NU_INVALID_SUSPEND**
The suspend_type parameter is invalid.
- **NU_NOT_ALIGNED**
The pointer is not aligned on a four byte boundary to the parameter start_address.

Example

```
/* Assume dynamic memory control block "Pool" is defined as a global data
   structure. This is one of several ways to allocate a control block. */

NU_MEMORY_POOL    Pool;
.
.
/* Assume status is defined locally. */

STATUS status; /* Memory Pool creation status */

/* Create a dynamic memory pool of 4000-bytes starting at the absolute
   address of 0xA000. Minimum allocation size is 30 bytes. Tasks suspend
   on the pool in order of priority. */

status = NU_Create_Memory_Pool(&Pool, "any name", (VOID *) 0xA000,
                               4000, 30, NU_PRIORITY);

/* At this point status indicates if the service was successful. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Established_Memory_Pools](#)

[NU_Memory_Pool_Information](#)

[NU_Memory_Pool_Pointers](#)

[NU_Delete_Memory_Pool](#)

NU_Deallocate_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service returns the memory block pointed to by memory back to the associated dynamic memory pool.

Usage

```
STATUS NU_Deallocate_Memory (VOID *memory);
```

Arguments

- **memory**
Pointer to a memory block previously allocated with NU_Allocate_Memory.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POINTER**
The memory block pointer is NULL, is not currently allocated, or is invalid.

Example

```
STATUS status;  
  
/* Deallocate the memory block pointed to by "memory". */  
status = NU_Deallocate_Memory(memory);  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Memory_Pool_Information](#)

[NU_Reallocate_Memory](#)

[NU_Reallocate_Aligned_Memory](#)

[NU_Allocate_Memory](#)

NU_Delete_Memory_Pool

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service deletes a previously created dynamic memory pool. The parameter pool identifies the dynamic memory pool to delete. Tasks suspended on this dynamic memory pool are resumed with the appropriate error status. The application must prevent the use of this dynamic memory pool during and after deletion.

Usage

```
STATUS NU_Delete_Memory_Pool (NU_MEMORY_POOL *pool);
```

Arguments

- pool
Pointer to the user-supplied memory pool control block that has been previously created with NU_Create_Memory_Pool.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_POOL
The dynamic memory pool pointer is invalid.

Example

```
NU_MEMORY_POOL Pool;  
STATUS status  
.  
.  
/* Delete the memory pool control block "Pool". Assume "Pool" has  
   previously been created with the Nucleus PLUS NU_Create_Memory_Pool  
   service call. */  
status = NU_Delete_Memory_Pool(&Pool);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Established_Memory_Pools](#)

[NU_Memory_Pool_Information](#)

[NU_Memory_Pool_Pointers](#)

[NU_Create_Memory_Pool](#)

NU_Established_Memory_Pools

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service returns the number of established dynamic memory pools. All created dynamic memory pools are considered established. Deleted dynamic-memory pools are no longer considered established.

Usage

```
UNSIGNED NU_Established_Memory_Pools (VOID);
```

Return Values

- This service call returns the number of created memory pools in the system.

Example

```
UNSIGNED total_memory_pools;

/* Obtain the total number of dynamic memory pools. */
total_memory_pools = NU_Established_Memory_Pools();
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Delete_Memory_Pool](#)

[NU_Memory_Pool_Information](#)

[NU_Memory_Pool_Pointers](#)

[NU_Create_Memory_Pool](#)

NU_Memory_Pool_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service returns information about the specified dynamic memory pool.

Usage

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

Arguments

- pool
Pointer to the dynamic memory pool.
- name
Pointer to an eight-character destination area for the dynamic memory pool's name. This includes space for the null terminator.
- start_address
Pointer to a memory pointer for holding the starting address of the pool.
- pool_size
Pointer to a variable for holding the number of bytes in dynamic memory pool.
- min_allocation
Pointer to a variable for holding the minimum number of bytes for each allocation from this pool.
- available
Pointer to a variable for holding the number of available bytes in the pool.
- suspend_type
Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
- tasks_waiting
Pointer to a variable for holding the number of tasks waiting on the dynamic memory pool.

- `first_task`
 Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Values

- `NU_SUCCESS`
 Function completed successfully.
- `NU_INVALID_POOL`
 Indicates the dynamic memory pool pointer is invalid.

Example

```

NU_MEMORY_POOL  Pool;
CHAR            pool_name[8];
VOID            *start_address;
UNSIGNED        pool_size;
UNSIGNED        min_allocation;
UNSIGNED        available;
OPTION          suspend_type;
UNSIGNED        tasks_suspended;
NU_TASK         *first_task;
STATUS          status
.
.
.
/* Obtain information about the memory pool control block "Pool". Assume
   "Pool" has previously been created with the Nucleus PLUS
   NU_Create_Memory_Pool service call. */
status = NU_Memory_Pool_Information(&Pool, pool_name,
    &start_address, &pool_size,
    &min_allocation,
    &available, &suspend_type,
    &tasks_suspended, &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Delete_Memory_Pool](#)

[NU_Established_Memory_Pools](#)

[NU_Memory_Pool_Pointers](#)

[NU_Create_Memory_Pool](#)

NU_Memory_Pool_Pointers


Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service builds a sequential list of pointers to all established dynamic memory pools in the system.

Note

 Dynamic memory pools that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Memory_Pool_Pointers (NU_MEMORY_POOL    **pointer_list,  
                                UNSIGNED              maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_MEMORY_POOL` pointers. This array will be filled with pointers of established memory pools in the system.
- `maximum_pointers`
The maximum number of `NU_MEMORY_POOL` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created memory pools in the system.

Example

```
/* Define an array capable of holding 20 dynamic memory pool pointers. */  
NU_MEMORY_POOL *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active dynamic-memory pool pointers (Maximum  
   of 20). */  
number = NU_Memory_Pool_Pointers(&Pointer_Array[0],20);  
  
/* At this point, number contains the actual number of pointers in the  
   list. */
```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Delete_Memory_Pool](#)

[NU_Established_Memory_Pools](#)

[NU_Memory_Pool_Information](#)

[NU_Create_Memory_Pool](#)

NU_Reallocate_Aligned_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service reallocates an aligned block of memory in the specified dynamic memory pool.

Usage

```
STATUS NU_Reallocate_Aligned_Memory (NU_MEMORY_POOL *memory,  
                                     VOID **return_pointer,  
                                     UNSIGNED size,  
                                     UNSIGNED alignment,  
                                     UNSIGNED suspend);
```

Arguments

- **memory**
Pointer to the dynamic memory pool
- **return_pointer**
Pointer to the caller's memory pointer. On a successful request, the address of the allocated block is placed in the caller's memory pointer.
- **size**
Specifies the number of bytes to allocate from the dynamic memory pool.
- **alignment**
Specifies the required alignment of the starting address of the requested memory block. This specified alignment must be an integer multiple of the minimum granularity of the dynamic memory pool. On most targets, this is size of (UNSIGNED).
- **suspend**
Specifies whether or not to suspend the calling task if the requested amount of memory is not available. The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the requested memory is available.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the requested memory is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POOL**
The dynamic memory pool is invalid.
- **NU_INVALID_POINTER**
The return pointer is **NU_NULL**.
- **NU_INVALID_SIZE**
Indicates an invalid size request.
- **NU_INVALID_SUSPEND**
Indicates suspend was attempted from a non-task thread.
- **NU_INVALID_OPERATION**
Indicates an invalid alignment request.
- **NU_NO_MEMORY**
The memory request could not be immediately satisfied.
- **NU_TIMEOUT**
The requested memory is still unavailable even after suspending for the specified timeout value.
- **NU_POOL_DELETED**
The dynamic memory pool was deleted while the task was suspended.

Example

```

NU_MEMORY_POOL Pool;
VOID memory_ptr;
STATUS status
.
.
.
/* Reallocate a 300-byte block of memory that is aligned on a 1024 byte
boundary. This allocates with the memory pool control block "Pool".
If the requested memory is unavailable, suspend the calling task
unconditionally. Assume "Pool" has previously been created with the
Nucleus PLUS NU_Create_Memory_Pool service call, and "memory_ptr" was
previously allocated with either NU_Allocate_Memory, or
NU_Allocate_Aligned_Memory. */

status = NU_Reallocate_Aligned_Memory(&Pool, &memory_ptr, 300, 1024,
NU_SUSPEND);

/* At this point, status indicates whether the service request was
successful. */

```

Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Reallocate_Memory](#)

[NU_Memory_Pool_Information](#)

[NU_Deallocate_Memory](#)

[NU_Allocate_Aligned_Memory](#)

[NU_Allocate_Memory](#)

NU_Reallocate_Memory

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service allows for reallocation of a memory pointer. It will grow or shrink the memory to a new size. If the current memory increases in size and cannot fit at the current location a new memory block will be allocated and the contents of the current memory block will be copied to the new location and then the current memory block will be freed. If new requested size cannot be satisfied an error will be returned and the memory pointer will remain unchanged.

The service also allows if a passed in pointer is NULL to do a new allocation and if size is zero to deallocate the memory pointer.

Usage

```
STATUS NU_Reallocate_Memory(NU_MEMORY_POOL  *pool_ptr,  
                           VOID              **memory_ptr,  
                           UNSIGNED          size,  
                           UNSIGNED          suspend);
```

Arguments

- **pool_ptr**
Pointer a memory pool to allocate from. Note: if memory_ptr represents a valid allocation and the pool it was allocated from does not match an error will be returned.
- **memory_ptr**
Pointer to a memory block previously allocated with NU_Allocate_Memory or NU_Reallocate_Memory. This will contain the current pointer and in a successful request, the address of the allocated block is placed in the memory pointer. Passing null directly will result in an error but, if the memory pointer points to null a new allocation will occur from the passed in pool.
- **size**
Specifies the number of bytes for the memory to be changed to. A value of zero will result in deallocation if the memory_ptr is valid.
- **suspend**
Specifies whether or not to suspend the calling task if the requested amount of memory is not available. The following is a summary of the possible values for the *suspend* parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied.

This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the requested memory is available.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the memory is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POOL**
The dynamic memory pool is invalid or does not match the previous allocation.
- **NU_INVALID_POINTER**
The memory block pointer is NULL, is not currently allocated, or is invalid.
- **NU_INVALID_SIZE**
Indicates an invalid size request.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_NO_MEMORY**
The memory request could not be immediately satisfied.
- **NU_TIMEOUT**
The requested memory is still unavailable even after suspending for the specified timeout value.
- **NU_POOL_DELETED**
Dynamic memory pool was deleted while the task was suspended.

Example

```
NU_MEMORY_POOL Pool;
VOID *memory_ptr;
STATUS status

/* Reallocate a 300-byte block of memory. If the requested memory is
   unavailable, suspend the calling task unconditionally. Assume "Pool"
   has previously been created with the Nucleus PLUS
   NU_Create_Memory_Pool service call and that memory_ptr has previously
   be allocated from NU_Allocate_Memory. */

status = NU_Reallocate_Memory(&Pool, &memory_ptr, 300, NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. */
```


Related Topics

[Dynamic Memory Services Function Reference](#)

[NU_Deallocate_Memory](#)

[NU_Allocate_Memory](#)

Dynamic Memory Example Source Code

The following program demonstrates how the Nucleus PLUS dynamic memory pool component could be used to implement a memory allocation scheme similar to that of the ANSI C malloc and free. A single dynamic memory pool is created out of which all memory requests are allocated. The memory pool is created in the function `memory_init`, and is deleted in `memory_deinit`. All memory can then be allocated through the function calls `memory_allocate`, and `memory_startup_allocate`. The two separate calls are used because, in this example, during a running program we would like for tasks to be suspended when a memory request cannot be immediately satisfied. The function `memory_allocate` could be used during a running program to request memory. When a request cannot be satisfied the calling task would be suspended. However, suspension cannot be requested in the startup function `Application_Initialize`, so a separate function `startup_memory_allocate` is used which does not request suspension when memory requests cannot be immediately satisfied.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single `NU_MEMORY_POOL` control block is created. This memory pool control block will be later passed to the `NU_Create_Memory_Pool` service call, which will set up the memory pool for use.

```
NU_MEMORY_POOL memory_pool;
```

In this example, the functions `memory_init`, and `memory_deinit` will be used to initialize and de-initialize the memory pool which is to be used. Specific to Nucleus PLUS, the function `memory_init` will be used to create the memory pool out of which all memory will be allocated. The function `memory_deinit` will be used to delete the dynamic memory pool. Similarly, all memory allocation requests would be made through the `memory_allocate` and `memory_startup_allocate` service calls. Finally, all memory deallocations are made through the `memory_free` function.

```
VOID *memory_allocate(UNSIGNED alloc_size);  
VOID *memory_startup_allocate(UNSIGNED alloc_size);  
VOID memory_free(VOID *memory_ptr);  
VOID memory_init(VOID *start_addr, UNSIGNED size);  
VOID memory_deinit();
```

The function `memory_init` is used to create the dynamic memory pool, `memory_pool`, out of which all memory will be allocated. The function is passed the starting address, and the size of

the pool to create. These parameters are then passed to the `NU_Create_Memory_Pool` call to create the memory pool, and associate it with the `memory_pool` control block.

```
VOID memory_init(VOID *start_addr, UNSIGNED size)
{
```

Make the call to `NU_Create_Memory_Pool` to create the dynamic memory pool, and associate the memory pool with the `memory_pool` control block. The `memory_pool` pool will be created such that the minimum allocation request that will be satisfied is a request for 128 bytes of memory. Also, tasks that choose to suspend when a request cannot be satisfied will be resumed in priority order, as indicated by the `NU_PRIORITY` parameter.

```
    if (NU_Create_Memory_Pool(&memory_pool, "mempool", start_addr,
                             size, 128, NU_PRIORITY) == NU_SUCCESS)
    {
        /* The memory pool was successfully created. */
    }
    else
    {
        /* There was an error creating the memory pool. */
    }
}
```

Use `NU_Delete_Memory_Pool` to delete the memory pool. The only parameter needed by this call is a pointer to the `NU_MEMORY_POOL` control block. Note that any memory allocations that were not deallocated will remain allocated.

```
VOID memory_deinit()
{
    if (NU_Delete_Memory_Pool(&memory_pool) == NU_SUCCESS)
    {
        /* The memory pool was successfully deleted. */
    }
    else
    {
        /* There was an error deleting the memory pool. */
    }
}
```

The `memory_allocate` function is used to allocate any required memory. The only parameter necessary for this function call is the size, in bytes, of the allocation request. The function will then attempt to allocate the memory with a call to `NU_Allocate_Memory`. If the request is successful (as indicated by the `NU_Allocate_Memory` service call returning `NU_SUCCESS`) then a pointer to the allocated memory is returned to the calling function. Otherwise `NU_NULL` is returned.

```
VOID *memory_allocate(UNSIGNED alloc_size)
{
```

The void pointer, `temp_ptr` will be used to return the allocated memory to the calling function. It will be passed as a parameter to the `NU_Allocate_Memory` service call. If the call is successful, then `temp_ptr` will contain a valid pointer to the newly allocated memory.

VOID *temp_ptr; The NU_Allocate_Memory service call will request the memory allocation out of the memory_pool dynamic memory pool. If the request can be satisfied, then temp_ptr will contain a pointer to the newly allocated memory, and NU_SUCCESS will be returned. If the request cannot be immediately satisfied, then the calling task will be suspended, as indicated by the NU_SUSPEND parameter. Note that this call should only be used from a task, and not from Application_Initialize because suspension cannot be requested from the Application_Initialize function.

```

    if (NU_Allocate_Memory(&memory_pool, &temp_ptr, alloc_size,
                          NU_SUSPEND) == NU_SUCCESS)
    {
        return temp_ptr;
    }
    else
    {
        return NU_NULL;
    }
}

```

Similar to memory_allocate, the function memory_startup_allocate will use the NU_Allocate_Memory service call to request the memory allocation out of the memory_pool dynamic memory pool. However, if the request cannot be immediately satisfied, the function memory_startup_allocate will not suspend, as indicated by the NU_NO_SUSPEND parameter in NU_Allocate_Memory. Therefore, this function would be used to allocate memory from the Application_Initialize function.

```

VOID *memory_startup_allocate(UNSIGNED alloc_size)
{
    VOID *temp_ptr;

```

Use NU_Allocate_Memory to request the allocation out of the memory_pool dynamic memory pool.

```

    if (NU_Allocate_Memory(&memory_pool, &temp_ptr, alloc_size,
                          NU_NO_SUSPEND) == NU_SUCCESS)
    {
        return temp_ptr;
    }
    else
    {
        /* an error occurred allocating memory. */
    }
}

```

The memory_free function is used to deallocate any previously allocated memory. It does this with a call to NU_Deallocate_Memory.

```

VOID memory_free(VOID *memory_ptr)
{

```

Use NU_Deallocate_Memory to return the memory allocation to the memory_pool dynamic memory pool.

```

    if (NU_Deallocate_Memory(memory_ptr) == NU_SUCCESS)

```

```
    {  
        /* Memory successfully deallocated. */  
    }  
    else  
    {  
        /* An error occurred deallocating memory. */  
    }  
}
```

Partition Memory

Partition memory is ideal for tasks that require deterministic handling.

A partition memory pool contains a specific number of fixed size memory partitions. The memory location of the pool, the number of bytes in the pool, and the number of bytes in each partition are determined by the application. Individual partitions are allocated and deallocated from the partition memory pool.

Allocation from a memory pool requires some additional overhead to allow for its pointer structure.

Partition Memory Suspension

The allocate partition service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to allocate a partition from an empty pool can suspend. Resumption of that task is possible when a partition is returned to the pool.

Multiple tasks may suspend on a single partition memory pool. Tasks are suspended in either FIFO or priority order, depending on how the partition memory pool was created. If the partition memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the partition memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Partition Memory Dynamic Creation

Nucleus PLUS partition memory pools are created and deleted dynamically. There is no preset limit on the number of partition memory pools an application may have. Each partition memory pool requires a control block and a pointer to the memory area for the partition. The memory for both the control block and the partition area is supplied by the application.

Partition Determinism

Since searching is completely avoided, processing required for allocating and deallocating partitions is fast and constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the partition memory pool.

Partition Information

Application tasks may obtain a list of active partition memory pools. Detailed information about each partition memory pool is also available. This information includes the partition memory pool name, starting pool address, total partitions, partition size, remaining partitions, number of tasks suspended, and the identity of the first suspended task.

Partition Memory Services Function Reference

The following function reference contains all functions related to the Nucleus PLUS partition memory component. The following functions are contained in this reference:

- [NU_Allocate_Partition](#)
- [NU_Create_Partition_Pool](#)
- [NU_Deallocate_Partition](#)
- [NU_Delete_Partition_Pool](#)
- [NU_Established_Partition_Pools](#)
- [NU_Partition_Pool_Information](#)
- [NU_Partition_Pool_Pointers](#)

NU_Allocate_Partition

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service allocates a memory partition from the specified memory partition pool. Note that the size of the memory partition is defined when the memory partition pool is created.

Usage

```
STATUS NU_Allocate_Partition (NU_PARTITION_POOL *pool,  
                             VOID **return_pointer,  
                             UNSIGNED suspend);
```

Arguments

- pool
Pointer to the memory partition pool.
- return_pointer
Pointer to the caller's memory pointer. On a successful request, the address of the allocated memory partition is placed in the caller's memory pointer.
- suspend
Specifies whether to suspend the calling task if there are no memory partitions available. The following table summarizes the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until a memory partition is available.

timeout value

(1 – 4,294,967,293) The calling task is suspended until a memory partition is available, or until the specified number of ticks has expired.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_POOL
The memory partition pool pointer is invalid.

- **NU_INVALID_POINTER**
 The return pointer is NULL.
- **NU_INVALID_SUSPEND**
 A suspend was attempted from a non-task thread.
- **NU_NO_PARTITION**
 The memory partition request could not be immediately satisfied.
- **NU_TIMEOUT**
 No memory partition is available even after suspending for the specified timeout value.
- **NU_POOL_DELETED**
 Partition memory pool was deleted while the task was suspended.

Example

```

NU_PARTITION_POOL Pool;
VOID                *memory_ptr;
STATUS              status
.
/* Allocate a memory partition with the memory partition pool control
   block "Pool". If there are no partitions available, suspend the calling
   the task unconditionally. Assume "Pool" has previously been created with
   Nucleus PLUS NU_Create_Partition_Pool service call.*/

status = NU_Allocate_Partition(&Pool, &memory_ptr, NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. */

```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Deallocate_Partition](#)

[NU_Partition_Pool_Information](#)

[NU_Create_Partition_Pool](#)

NU_Create_Partition_Pool

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service creates a pool of fixed-size memory partitions inside a memory area specified by the caller.

Usage

```
STATUS NU_Create_Partition_Pool (NU_PARTITION_POOL *pool,  
                                CHAR *name,  
                                VOID *start_address,  
                                UNSIGNED pool_size,  
                                UNSIGNED partition_size,  
                                OPTION suspend_type);
```

Arguments

- **pool**
Pointer to the user-supplied partition pool control block. Subsequent requests made to this partition pool require this pointer.
- **name**
Pointer to a seven-character name for the partition pool. The name must be null-terminated.
- **start_address**
Specifies the starting address for the fixed-size memory partition pool.
- **pool_size**
Specifies the total number of bytes in the memory area.
- **partition_size**
Specifies the number of bytes for each partition in the pool. There is a small amount of memory “overhead” associated with each partition. This overhead is required by the two data pointers used.
- **suspend_type**
Specifies how tasks suspend on the partition pool. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent FIFO and priority-order task suspension, respectively.

Return Values

- **NU_SUCCESS**
Function completed successfully.

- **NU_INVALID_POOL**
 The partition pool control block pointer is NULL or is already in use.
- **NU_INVALID_MEMORY**
 The memory area specified by the start_ address is invalid.
- **NU_INVALID_SIZE**
 The partition size is either 0 or larger than the total partition memory area.
- **NU_INVALID_SUSPEND**
 The suspend_type parameter is invalid.
- **NU_NOT_ALIGNED**
 A pointer is not aligned on a four byte boundary to the parameter start_address.

Example

```
/* Assume partition memory control block "Pool" is defined as a global
   data structure. This is one of several ways to allocate a control
   block.  */

NU_PARTITION_POOL Pool;
.
.
/* Assume status is defined locally.  */

STATUS status; /* Partition Pool creation status */

/* Create a partition memory pool of 40-byte memory partitions, in a
   2000-byte memory area starting at the absolute address of 0xB000. Task
   suspend on the pool in FIFO order. */

status = NU_Create_Partition_Pool(&Pool, "any name",
                                (VOID *) 0xB000, 2000,
                                40, NU_FIFO);

/* At this point status indicates if the service was successful.  */
```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Established_Partition_Pools](#)

[NU_Partition_Pool_Information](#)

[NU_Partition_Pool_Pointers](#)

[NU_Delete_Partition_Pool](#)

NU_Deallocate_Partition

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service returns the memory partition pointed to by partition back to the associated pool.

Usage

```
STATUS NU_Deallocate_Partition (VOID *partition);
```

Arguments

- partition
Pointer to a memory partition previously allocated with NU_Allocate_Partition.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_POINTER
The memory partition pointer is NULL, is not currently allocated, or is invalid.

Example

```
STATUS status;  
  
/* Deallocate the memory partition pointed to by "partition". */  
status = NU_Deallocate_Partition(partition);  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Partition_Pool_Information](#)

[NU_Allocate_Partition](#)

NU_Delete_Partition_Pool

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: Yes

This service deletes a previously created memory partition pool. The parameter pool identifies the memory partition pool to delete. Tasks suspended on this memory partition pool are resumed with the appropriate error status. The application must prevent the use of this memory partition pool during and after deletion.

Usage

```
STATUS NU_Delete_Partition_Pool (NU_PARTITION_POOL *pool);
```

Arguments

- pool
Pointer to the user-supplied partition pool control block that has been previously created with NU_Create_Partition_Pool.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_POOL
The memory partition pool pointer is invalid.

Example

```
NU_PARTITION_POOL Pool;  
STATUS status  
.  
.  
/* Delete the partition pool control block "Pool". Assume "Pool" has  
   previously been created with the Nucleus PLUS NU_Create_Partition_Pool  
   service call.*/  
  
status = NU_Delete_Partition_Pool(&Pool);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Established_Partition_Pools](#)

[NU_Partition_Pool_Information](#)

[NU_Partition_Pool_Pointers](#)

[NU_Create_Partition_Pool](#)

NU_Established_Partition_Pools

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service returns the number of established memory partition pools. All created memory partition pools are considered established. Deleted memory partition pools are no longer considered established.

Usage

```
UNSIGNED NU_Established_Partition_Pools (VOID);
```

Return Values

- This service call returns the number of created partition pools in the system.

Example

```
UNSIGNED total_partition_pools;

/* Obtain the total number of memory partition pools. */
total_partition_pools = NU_Established_Partition_Pools();
```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Delete_Partition_Pool](#)

[NU_Partition_Pool_Information](#)

[NU_Partition_Pool_Pointers](#)

[NU_Create_Partition_Pool](#)

NU_Partition_Pool_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service returns various information about the specified partition memory pool.

Usage

```
STATUS NU_Partition_Pool_Information (
    NU_PARTITION_POOL *pool,
    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);
```

Arguments

- pool
Pointer to the partition pool.
- name
Pointer to an eight-character destination area for the partition pool's name. This includes space for the null terminator.
- start_address
Pointer to a memory pointer for holding the starting address of the pool.
- pool_size
Pointer to a variable for holding the total number of bytes in the partition pool.
- partition_size
Pointer to a variable for holding the number of bytes in each memory partition.
- available
Pointer to a variable for holding the number of available partitions in the pool.
- allocated
Pointer to a variable for holding the number of allocated pool partitions.
- suspend_type
Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.

- `tasks_waiting`
 Pointer to a variable for holding the number of tasks waiting on the partition pool.
- `first_task`
 Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Values

- `NU_SUCCESS`
 Function completed successfully.
- `NU_INVALID_POOL`
 Indicates the partition pool pointer is invalid.

Example

```

NU_PARTITION_POOL Pool;
CHAR               pool_name[8];
VOID               *start_address;
UNSIGNED           pool_size;
UNSIGNED           partition_size;
UNSIGNED           available;
UNSIGNED           allocated;
OPTION             suspend_type;
UNSIGNED           tasks_suspended;
NU_TASK            *first_task;
STATUS             status

.
.
/* Obtain information about the partition pool control block "Pool".
   Assume "Pool" has previously been created with the Nucleus PLUS
   NU_Create_Partition_Pool service call. */
status = NU_Partition_Pool_Information(&Pool, pool_name,
                                     &start_address, &pool_size,
                                     &partition_size, &available,
                                     &allocated, &suspend_type,
                                     &tasks_suspended,
                                     &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

Related Topics

[Partition Memory Services Function Reference](#)

[NU_Delete_Partition_Pool](#)

[NU_Established_Partition_Pools](#)

[NU_Partition_Pool_Pointers](#)

[NU_Create_Partition_Pool](#)

NU_Partition_Pool_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Memory Services

Tasking Changes: No

This service builds a sequential list of pointers to all established memory partition pools in the system.



Note

Memory partition pools that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Partition_Pool_Pointers (NU_PARTITION_POOL **pointer_list,  
                                     UNSIGNED          maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_PARTITION_POOL` pointers. This array will be filled with pointers of established partition pools in the system.
- `maximum_pointers`
The maximum number of `NU_PARTITION_POOL` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created memory pools in the system.

Example

```
/* Define an array capable of holding 20 memory partition pool pointers.  
*/  
NU_PARTITION_POOL*Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active memory partition pool pointers  
   (Maximum of 20). */  
number = NU_Partition_Pool_Pointers(&Pointer_Array[0], 20);  
  
/* number contains the actual number of pointers in the list. */
```


Related Topics

[Partition Memory Services Function Reference](#)

[NU_Delete_Partition_Pool](#)

[NU_Established_Partition_Pools](#)

[NU_Partition_Pool_Information](#)

[NU_Create_Partition_Pool](#)

Partition Memory Example Source Code

The following program demonstrates how the Nucleus PLUS partition memory pool component could be used to implement a memory allocation scheme similar to that of the ANSI C malloc and free. A single partition memory pool is created out of which all memory requests are allocated. The memory pool is created in the function `memory_init`, and is deleted in `memory_deinit`. All memory can then be allocated through the function calls `memory_allocate`, and `memory_startup_allocate`. The two separate calls are used because, in this example, during a running program we would like for tasks to be suspended when a memory request cannot be immediately satisfied. The function `memory_allocate` could be used during a running program to request memory. When a request cannot be satisfied the calling task would be suspended. However, suspension cannot be requested in the start-up function `Application_Initialize`, so a separate function `startup_memory_allocate` is used which does not request suspension when memory requests cannot be immediately satisfied.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single `NU_PARTITION_POOL` control block is created. This partition pool control block will be later passed to the `NU_Create_Partition_Pool` service call, which will set up the partition pool for use.

```
NU_SEMAPHORE semaphore_memory;  
NU_PARTITION_POOL memory_pool;
```

In this example, the functions `memory_init`, and `memory_deinit` will be used to initialize and de-initialize the partition memory pool which is to be used. Specific to Nucleus PLUS, the function `memory_init` will be used to create the partition pool out of which all memory will be allocated. The function `memory_deinit` will be used to delete the partition memory pool. Similarly, all memory allocation requests would be made through the `memory_allocate` and `memory_startup_allocate` service calls. Finally, all memory deallocations would be made through the `memory_free` function.

```
VOID *memory_allocate();  
VOID *memory_startup_allocate();  
VOID memory_free(VOID *memory_ptr);  
VOID memory_init(VOID *start_addr, UNSIGNED size,  
                 UNSIGNED partition_size);  
VOID memory_deinit();
```

The `memory_init` function is used to create the partition memory pool, `memory_pool`, out of which all memory will be allocated. The function is passed the starting address, the size of the pool to create, and the size of each partition to be allocated. These parameters are then passed to the `NU_Create_Partition_Pool` call to create the memory pool, and associate it with the `memory_pool` control block.

```
VOID memory_init(VOID *start_addr, UNSIGNED size, UNSIGNED partition_size)
{
```

Make the call to `NU_Create_Partition_Pool` to create the partition pool, and associate the memory pool with the `memory_pool` control block. As previously mentioned, the `memory_pool` partition pool will be created with the starting address, size, and partition size as specified in the function parameters. The partition pool will also be created such that tasks which choose to suspend when a request cannot be satisfied will be resumed in priority order, as indicated by the `NU_PRIORITY` parameter.

```
    if (NU_Create_Partition_Pool(&memory_pool, "mempool", start_addr,
                                size, partition_size, NU_PRIORITY)
        == NU_SUCCESS)
    {
        /* Partition pool successfully created. */
    }
    else
    {
        /* Error creating partition pool. */
    }
}
```

Use `NU_Delete_Partition_Pool` to delete the memory pool. The only parameter needed by this call is a pointer to the `NU_PARTITION_POOL` control block. Note that any memory allocations that were not deallocated will remain allocated.

```
VOID memory_deinit()
{
    if (NU_Delete_Partition_Pool(&memory_pool) == NU_SUCCESS)
    {
        /* Partition pool successfully deleted. */
    }
    else
    {
        /* Error deleting partition pool. */
    }
}
```

The `memory_allocate` function is used to allocate any required memory. Note that this function does not take any parameters, unlike its dynamic memory counterpart. Since all allocations are made in the size that was specified when the pool was created, the size parameter is not necessary.

The function will attempt to allocate the memory with a call to `NU_Allocate_Memory`. If the request is successful, as indicated by the `NU_Allocate_Memory` service call returning `NU_SUCCESS`, then a pointer to the allocated memory is returned to the calling function. Otherwise, `NU_NULL` is returned.

```
VOID *memory_allocate()
{
```

The void pointer, `temp_ptr` will be used to return the allocated memory to the calling function. It will be passed as a parameter to the `NU_Allocate_Partition` service call. If the call is successful, then `temp_ptr` will contain a valid pointer to the newly allocated memory.

```
VOID *temp_ptr;
```

The `NU_Allocate_Partition` service call will request the memory allocation out of the `memory_pool` partition memory pool. If the request can be satisfied, then `temp_ptr` will contain a pointer to the newly allocated memory, and `NU_SUCCESS` will be returned. If the request cannot be immediately satisfied, then the calling task will be suspended, as indicated by the `NU_SUSPEND` parameter. Note that this call should only be used from a task, and not from the `Application_Initialize` because suspension cannot be requested from the `Application_Initialize` function.

```
    if (NU_Allocate_Partition(&memory_pool, &temp_ptr, NU_SUSPEND)
        == NU_SUCCESS)
    {
        return temp_ptr;
    }
    else
    {
    }
}
```

Similar to `memory_allocate`, the function `memory_startup_allocate` will use the `NU_Allocate_Partition` service call to request the memory allocation out of the `memory_pool` partition memory pool. However, if the request cannot be immediately satisfied, the function `memory_startup_allocate` will not suspend, as indicated by the `NU_NO_SUSPEND` parameter in `NU_Allocate_Memory`. Therefore, this function would be used to allocate memory from the `Application_Initialize` function.

```
VOID *memory_startup_allocate()
{
    VOID *temp_ptr;
```

Use `NU_Allocate_Partition` to request the allocation out of the `memory_pool` partition memory pool.

```
    if (NU_Allocate_Partition(&memory_pool, &temp_ptr, NU_NO_SUSPEND)
        == NU_SUCCESS)
    {
        return temp_ptr;
    }
```

```
        else
        {
            /* Error in memory allocation. */
        }
    }
```

The `memory_free` function would be used to deallocate any previously allocated memory. It does this with a single call to `NU_Deallocate_Partition`.

```
VOID memory_free(VOID *memory_ptr)
{
```

Use `NU_Deallocate_Memory` to return the memory allocation to the `memory_pool` partition memory pool.

```
    if (NU_Deallocate_Partition(&memory_ptr) == NU_SUCCESS)
    {
    }
    else
    {
    }
}
```

Mailboxes

Mailboxes are a method of communication between tasks, intended for short messages.

Mailboxes provide a low-overhead mechanism to transmit simple messages. Each mailbox is capable of holding a single message the size of four 32-bit words. Messages are sent and received by value. A send message request copies the message into the mailbox, while a receive message request copies the message out of the mailbox.

Mailbox Suspension

Send and receive mailbox services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks can suspend on a mailbox for several reasons. A task attempting to receive a message from an empty mailbox can suspend. Also, a task attempting to send a message to a non-empty mailbox can suspend. A suspended task is resumed when the mailbox is able to satisfy that task's request. For example, suppose a task is suspended on a mailbox waiting to receive a message. When a message is sent to the mailbox, the suspended task is resumed.

Multiple tasks can suspend on a single mailbox. Tasks are suspended in either FIFO or priority order, depending on how the mailbox was created. If the mailbox supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the mailbox supports priority suspension, tasks are resumed from high priority to low priority.

Mailbox Broadcast

A mailbox message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the mailbox are given the broadcast message.

Mailbox Dynamic Creation

Nucleus PLUS mailboxes are created and deleted dynamically. There is no preset limit on the number of mailboxes an application may have. Each mailbox requires a control block. The memory for the control block is supplied by the application.

Mailbox Determinism

Processing time required for sending and receiving mailbox messages is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the mailbox.

Mailbox Information

Application tasks may obtain a list of active mailboxes. Detailed information about each mailbox can also be obtained. This information includes the mailbox name, suspension type, whether a message is present, and the first task waiting.

Mailbox Services Function Reference

The following function reference contains all functions related to Nucleus PLUS mailboxes. The following functions are contained in this reference:

- [NU_Broadcast_To_Mailbox](#)
- [NU_Create_Mailbox](#)
- [NU_Delete_Mailbox](#)
- [NU_Established_Mailboxes](#)
- [NU_Mailbox_Information](#)
- [NU_Mailbox_Pointers](#)
- [NU_Receive_From_Mailbox](#)
- [NU_Reset_Mailbox](#)
- [NU_Send_To_Mailbox](#)

NU_Broadcast_To_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service broadcasts a message to all tasks waiting for a message from the specified mailbox. If no tasks are waiting, the message is simply placed in the mailbox. Each message is equivalent in size to four UNSIGNED data elements.

Usage

```
STATUS NU_Broadcast_To_Mailbox (NU_MAILBOX *mailbox,  
                                VOID          *message,  
                                UNSIGNED      suspend);
```

Arguments

- mailbox
Pointer to the mailbox.
- message
Pointer to the broadcast message.
- suspend

Specifies whether or not to suspend the calling task if the mailbox already contains a message. The following are possible values for this argument:

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be copied into the mailbox.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the mailbox or until the specified number of ticks has expired.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox pointer is invalid.

- **NU_INVALID_POINTER**
 The message pointer is NULL.
- **NU_INVALID_SUSPEND**
 Indicates that suspend was attempted from a non-task thread.
- **NU_MAILBOX_FULL**
 The message could not be immediately placed in the mailbox because the mailbox already contains a message.
- **NU_TIMEOUT**
 The mailbox is still unable to accept the message even after suspending for the specified timeout value.
- **NU_MAILBOX_DELETED**
 Mailbox was deleted while the task was suspended.
- **NU_MAILBOX_RESET**
 Mailbox was reset while the task was suspended.

Example

```

NU_MAILBOX Mailbox;
UNSIGNED   message[4];
STATUS     status
.
.
.
/* Build a message to send to a mailbox.  The contents of  "message" are
   not significant  */

message[0]  =  0x00001111;
message[1]  =  0x22223333;
message[2]  =  0x44445555;
message[3]  =  0x66667777;

/* Send the message to the mailbox control block "Mailbox".  If the
   mailbox already contains a message, suspend for 20 timer ticks.
   Assume "Mailbox" has previously been created with the Nucleus
   PLUS NU_Create_Mailbox service call.  */
status =  NU_Broadcast_To_Mailbox(&Mailbox, &message[0], 20);

/* At this point, status indicates whether the service request was
   successful.  */

```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Receive_From_Mailbox](#)

[NU_Send_To_Mailbox](#)

[NU_Mailbox_Information](#)

NU_Create_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service creates a task communication mailbox. A mailbox is capable of holding a single message. Mailbox messages are equivalent in size to four UNSIGNED data elements.

Usage

```
STATUS NU_Create_Mailbox (NU_MAILBOX *mailbox,  
                        CHAR          *name,  
                        OPTION        suspend_type);
```

Arguments

- mailbox
Pointer to the user-supplied mailbox control block. All subsequent requests made to the mailbox require this pointer.
- name
Pointer to a seven-character name for the mailbox. The name must be null-terminated.
- suspend_type
Specifies how tasks suspend on the mailbox. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox control block pointer is NULL or is already in use.
- NU_INVALID_SUSPEND
The suspend_type parameter is invalid.

Example

```
/* Assume mailbox control block "Mailbox" is defined as a global data  
   structure. This is one of several ways to allocate a control block. */  
NU_MAILBOX Mailbox;  
.  
.  
/* Assume status is defined locally. */  
  
STATUS status; /* Mailbox creation status */
```



```
/* Create a mailbox that manages task suspension in a FIFO manner. */  
status = NU_Create_Mailbox(&Mailbox, "any name", NU_FIFO);  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Established_Mailboxes](#)

[NU_Mailbox_Information](#)

[NU_Mailbox_Pointers](#)

[NU_Delete_Mailbox](#)

NU_Delete_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service deletes a previously created mailbox. The parameter mailbox identifies the mailbox to delete. Tasks suspended on this mailbox are resumed with the appropriate error status. The application must prevent the use of this mailbox during and after deletion.

Usage

```
STATUS NU_Delete_Mailbox (NU_MAILBOX *mailbox);
```

Arguments

- mailbox
Pointer to the user-supplied mailbox control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox pointer is invalid.

Example

```
NU_MAILBOX Mailbox;  
STATUS      status  
.  
.  
.  
/* Delete the mailbox control block "Mailbox". Assume "Mailbox" has  
   previously been created with the Nucleus PLUS NU_Create_Mailbox service  
   call. */  
status =  NU_Delete_Mailbox(&Mailbox);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Established_Mailboxes](#)

[NU_Mailbox_Information](#)

[NU_Mailbox_Pointers](#)

[NU_Create_Mailbox](#)

NU_Established_Mailboxes

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns the number of established mailboxes. All created mailboxes are considered established. Deleted mailboxes are no longer considered established.

Usage

```
UNSIGNED NU_Established_Mailboxes (VOID);
```

Return Values

- This service call returns the number of created mailboxes in the system.

Example

```
UNSIGNED total_mailboxes;  
  
/* Obtain the total number of mailboxes. */  
total_mailboxes = NU_Established_Mailboxes();
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Delete_Mailbox](#)

[NU_Mailbox_Information](#)

[NU_Mailbox_Pointers](#)

[NU_Create_Mailbox](#)

NU_Mailbox_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns various information about the specified mailbox.

Usage

```
STATUS NU_Mailbox_Information (NU_MAILBOX *mailbox,  
                              CHAR *name,  
                              OPTION *suspend_type,  
                              DATA_ELEMENT *message_present,  
                              UNSIGNED *tasks_waiting,  
                              NU_TASK **first_task);
```

Arguments

- mailbox
Pointer to the user-supplied mailbox control block.
- name
Pointer to an eight-character destination area for the mailbox's name. This includes space for a null terminator.
- suspend_type
Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
- message_present
If a message is present in the mailbox, an NU_TRUE value is placed in the variable pointed to by this parameter. Otherwise, if the mailbox is empty, an NU_FALSE value is placed in the variable.
- tasks_waiting
Pointer to a variable for holding the number of tasks waiting on the mailbox.
- first_task
Pointer to a task pointer. The pointer of the first suspended task is place in the task pointer.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
Indicates the mailbox pointer is invalid.

Example

```

NU_MAILBOX    Mailbox;
CHAR          mailbox_name[8];
OPTION        suspend_type;
DATA_ELEMENT  message_present;
UNSIGNED      tasks_suspended;
NU_TASK       *first_task;
STATUS        status
.
.
.
/* Obtain information about the mailbox control block "Mailbox". Assume
   "Mailbox" has previously been created with the Nucleus PLUS
   NU_Create_Mailbox service call. */
status = NU_Mailbox_Information(&Mailbox, mailbox_name,
                               &suspend_type, &message_present,
                               &tasks_suspended, &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Delete_Mailbox](#)

[NU_Established_Mailboxes](#)

[NU_Mailbox_Pointers](#)

[NU_Create_Mailbox](#)

NU_Mailbox_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service builds a sequential list of pointers to all established mailboxes in the system.

Note



Mailboxes that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Mailbox_Pointers (NU_MAILBOX **pointer_list,  
                             UNSIGNED   maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of NU_MAILBOX pointers. This array will be filled with pointers of established mailboxes in the system.
- `maximum_pointers`
The maximum number of NU_MAILBOX pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created mailboxes in the system.

Example

```
/* Define an array capable of holding 20 mailbox pointers */  
NU_MAILBOX *Pointer_Array[20];  
UNSIGNED   number;  
  
/* Obtain a list of currently active mailbox pointers (Maximum of 20). */  
number = NU_Mailbox_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, the number contains the actual number of pointers in the  
list. */
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Established_Mailboxes](#)

[NU_Create_Mailbox](#)

[NU_Delete_Mailbox](#)

[NU_Mailbox_Information](#)

NU_Receive_From_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service retrieves a message from the specified mailbox. If the mailbox contains a message, it is immediately removed from the mailbox and is copied into the designated location. Mailbox messages are equivalent in size to four UNSIGNED data elements.

Usage

```
STATUS NU_Receive_From_Mailbox (NU_MAILBOX *mailbox,  
                                VOID          *message,  
                                UNSIGNED      suspend);
```

Arguments

- mailbox
Pointer to the user-supplied mailbox control block.
- message
Pointer to the message destination. The message destination must be at least the size of four UNSIGNED data elements.
- suspend
Specifies whether to suspend the calling task if the mailbox is empty.
The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until a message is available.

timeout value

(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox pointer is invalid.

- **NU_INVALID_POINTER**
 The message pointer is NULL.
- **NU_INVALID_SUSPEND**
 Indicates that suspend attempted from a non-task thread.
- **NU_MAILBOX_EMPTY**
 The mailbox is empty.
- **NU_TIMEOUT**
 The mailbox is still empty even after suspending for the specified timeout value.
- **NU_MAILBOX_DELETED**
 The mailbox was deleted while the task was suspended.
- **NU_MAILBOX_RESET**
 The mailbox was reset while the task was suspended.

Example

```

NU_MAILBOX mailbox;
UNSIGNED   message[4];
STATUS     status;
.
.
.
/* Receive a message from the mailbox control block "Mailbox". If the
   mailbox is empty, suspend for 20 timer ticks. Note: the order of
   multiple tasks suspending on the same mailbox is determined when the
   mailbox is created. Assume "Mailbox" has previously been created with
   the Nucleus PLUS NU_Create_Mailbox service call. */
status =  NU_Receive_From_Mailbox(&Mailbox,&message[0],20);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" contains the received mailbox
   message. */

```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Mailbox_Information](#)

[NU_Send_To_Mailbox](#)

[NU_Broadcast_To_Mailbox](#)

NU_Reset_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service discards a message currently in the mailbox specified by mailbox. All tasks suspended on the mailbox are resumed with the appropriate reset status.

Usage

```
STATUS NU_Reset_Mailbox (NU_MAILBOX *mailbox);
```

Arguments

- mailbox
Pointer to the user-supplied mailbox control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox pointer is invalid.

Example

```
NU_MAILBOX Mailbox;  
STATUS      status;  
.  
.  
.  
/* Reset the mailbox control block "Mailbox". Assume "Mailbox" has  
   previously been created with the Nucleus PLUS NU_Create_Mailbox service  
   call. */  
status = NU_Reset_Mailbox(&Mailbox);
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Mailbox_Information](#)

[NU_Receive_From_Mailbox](#)

[NU_Send_To_Mailbox](#)

[NU_Broadcast_To_Mailbox](#)

NU_Send_To_Mailbox

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service places a message into the specified mailbox. If the mailbox is empty, the message is copied immediately into the mailbox. Mailbox messages are equivalent to four UNSIGNED data elements in size. The parameters of this service are further defined as follows:

Usage

```
STATUS NU_Send_To_Mailbox (NU_MAILBOX *mailbox,  
                           VOID          *message,  
                           UNSIGNED      suspend);
```

Arguments

- mailbox
Pointer to the mailbox.
- message
Pointer to the message to send.
- suspend
Specifies whether to suspend the calling task if the mailbox already contains a message.
The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be sent.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_MAILBOX
The mailbox pointer is invalid.

- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_MAILBOX_FULL**
The mailbox is full.
- **NU_TIMEOUT**
The mailbox is still full even after suspending for the specified timeout value.
- **NU_MAILBOX_DELETED**
The mailbox was deleted while the task was suspended.
- **NU_MAILBOX_RESET**
The mailbox was reset while the task was suspended.

Example

```
NU_MAILBOX Mailbox;
UNSIGNED  message[4];
STATUS     status;
.
.
.
/* Build a 4 UNSIGNED-variable message to send. The contents of "message"
   have no significance. */
message[0] = 0x00001111;
message[1] = 0x00002222;
message[2] = 0x00003333;
message[3] = 0x00004444;

/* Send the message to the mailbox control block "Mailbox". Suspend the
   calling task until the message can be sent or until 25 timer ticks
   expire. Assume "Mailbox" has previously been created with the Nucleus
   PLUS NU_Create_Mailbox service call. */
status = NU_Send_To_Mailbox(&Mailbox, &message[0], 25);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" was sent to "Mailbox". */
```

Related Topics

[Mailbox Services Function Reference](#)

[NU_Mailbox_Information](#)

[NU_Receive_From_Mailbox](#)

[NU_Broadcast_To_Mailbox](#)

Mailbox Example Source Code

The following example will demonstrate the use of Nucleus PLUS mailboxes to communicate between tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Create structures for three tasks (NU_TASK). Also, create a mailbox structure (NU_MAILBOX). This mailbox will be used to communicate between the three tasks in the system.

```
NU_TASK task_recv_1;  
NU_TASK task_recv_2;  
NU_TASK task_send;  
NU_MAILBOX mailbox_comm;
```

Three void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_recv_1;  
VOID *stack_recv_2;  
VOID *stack_send;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call, which will associate these functions with each of their respective tasks.

```
VOID entry_recv_1(UNSIGNED argc, VOID *argv);  
VOID entry_recv_2(UNSIGNED argc, VOID *argv);  
VOID entry_send(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to set up memory for three task stacks in the system. Application_Initialize will also be used to create the mailbox, which will be used to communicate between the three tasks in the system.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the NU_Allocate_Memory call, we are allocating a 1024 byte block of memory out of the mem_pool dynamic memory pool, that is passed in to Application_Initialize. A pointer to the newly allocated memory is assigned to stack_recv_1, stack_recv_2, and stack_send respectively. The pointer to this memory allocation is passed to the NU_Create_Task call, which will use this memory as the task stack.

For this demonstration, note that `task_rcv_1` and `task_rcv_2` are given a higher priority (priority level of 7) than `task_send`. By doing this, you are ensuring that `task_rcv_1` and `task_rcv_2` will always run before `task_send`. The `task_send` will only run when both `task_rcv_1` and `task_rcv_2` are suspended.

```
NU_Allocate_Memory(mem_pool, &stack_rcv_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_1, "rcv_1", entry_rcv_1, 0, NU_NULL,
               stack_rcv_1, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(mem_pool, &stack_rcv_2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_2, "rcv_2", entry_rcv_2, 0, NU_NULL,
               stack_rcv_2, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(mem_pool, &stack_send, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,
               stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

Use `NU_Create_Mailbox` to create the `mailbox_comm` mailbox. This mailbox will be named “comm”, and tasks that choose to suspend on this mailbox will be resumed in FIFO order. Instead of specifying `NU_FIFO`, `NU_PRIORITY` could be specified instead, which would cause tasks to be resumed based upon their priority. For this example, the only tasks that will be suspending on this mailbox are of the same priority, so the results will be the same regardless of the suspension type specified.

```
NU_Create_Mailbox(&mailbox_comm, "comm", NU_FIFO);
}
```

The `entry_rcv_1` and `entry_rcv_2` functions serve as the entry point for the `task_rcv_1` and `task_rcv_2` tasks respectively. The tasks will continuously loop, issuing `NU_Receive_From_Mailbox` for each iteration of the loop. `NU_Receive_From_Mailbox` will suspend until there is a message placed into the mailbox (as indicated by `NU_SUSPEND`). Whenever a message is received, `NU_Receive_From_Mailbox` will exit with a Return Values of `NU_SUCCESS`. After the call has returned, `rcvmsg` will contain the message received. Therefore, there are two tasks that are continuously suspending on the same mailbox, both waiting for a message to be placed into the mailbox.

The PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `mailbox_comm` message box was created.

```
VOID entry_rcv_1(UNSIGNED argc, VOID *argv)
{
    UNSIGNED rcvmsg[4];

    while(1)
    {
        if (NU_Receive_From_Mailbox(&mailbox_comm, rcvmsg, NU_SUSPEND)
            == NU_SUCCESS)
        {
            /* rcvmsg contains the received message. */
        }
        else
        {

```

```

        /* an error has occurred. */
    }
}
}
VOID entry_rcv_2(UNSIGNED argc, VOID *argv)
{
    UNSIGNED rcvmsg[4];

    while(1)
    {
        if (NU_Receive_From_Mailbox(&mailbox_comm, rcvmsg, NU_SUSPEND)
            == NU_SUCCESS)
        {
            /* rcvmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}

```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_rcv_1` and `task_rcv_2` tasks are of a higher priority, and will always be given the first chance to run. Because of this, whenever `task_send` sends a message with the `mailbox_comm` message box, either `task_rcv_1` or `task_rcv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Mailbox` which will send a single message with the `mailbox_comm` mailbox. The second service call that is issued is `NU_Broadcast_To_Mailbox`, which will send the message to every task that is currently suspended on this mailbox. Note that in this example, whenever this task is running, there will always be two tasks (`task_rcv_1` and `task_rcv_2`) suspended on the `mailbox_comm` mailbox. The result is that the message that is sent with `NU_Send_To_Mailbox` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Mailbox` will be received by both suspended tasks.

```

VOID entry_send(UNSIGNED argc, VOID *argv)
{
    UNSIGNED sendmsg[4];

    while(1)
    {

```

Place decimal 1 in the first element of the four-element array, then issue `NU_Send_To_Mailbox` on the `mailbox_comm` message box. Since two tasks will always be suspended on this mailbox, and the mailbox was created with the `NU_FIFO` suspension flag, the first task that suspended on the mailbox will always receive this message.

```

        sendmsg[0]=1;
        if (NU_Send_To_Mailbox(&mailbox_comm, sendmsg, NU_SUSPEND)
            == NU_SUCCESS)
        {

```

```
        /* The message was successfully sent. */  
    }  
    else  
    {  
        /* An error occurred, or the message could not be sent. */  
    }  
}
```

Place a decimal 2 in the first element of the four-element array, then issue `NU_Broadcast_To_Mailbox` on the `mailbox_comm` message box. Because the priority of `task_rcv_1` and `task_rcv_2` is higher than this task, we are guaranteed that two tasks will always be suspended on this mailbox. Therefore, the result of the `NU_Broadcast_To_Mailbox` service is that both tasks will be sent the message.

```
    sendmsg[0]=2;  
    if (NU_Broadcast_To_Mailbox(&mailbox_comm, sendmsg, NU_SUSPEND)  
        == NU_SUCCESS)  
    {  
        /* The message was successfully sent. */  
    }  
    else  
    {  
        /* An error occurred, or the message could not be sent. */  
    }  
}  
}
```

Queues

Queueing is a method of communication between tasks intended for multiple messages of fixed or variable length.

Queues provide a mechanism to transmit multiple messages. Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue. Messages may be placed at the front of the queue or at the back of the queue.

Queue Message Size

A queue message consists of one or more 32-bit words. Both fixed and variable-length messages are supported. The type of message format is defined when the queue is created. Variable-length message queues require an additional 32-bit word of overhead for each message in the queue. Additionally, receive message requests on variable-length message queues specify the maximum message size, while the same requests on fixed-length message queues specify the exact message size.

Queue Suspension

Send and receive queue services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks may suspend on a queue for several reasons. A task attempting to receive a message from an empty queue can suspend. Additionally, a task attempting to send a message to a full queue can suspend. A suspended task is resumed when the queue is able to satisfy that task's request. For example, suppose a task is suspended on a queue waiting to receive a message. When a message is sent to the queue, the suspended task is resumed.

Multiple tasks may suspend on a single queue. Tasks are suspended in either FIFO or priority order, depending on how the queue was created. If the queue supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the queue supports priority suspension, tasks are resumed from high priority to low priority.

Queue Broadcast

A queue message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the queue are given the broadcast message.

Queue Dynamic Creation

Nucleus PLUS queues are created and deleted dynamically. There is no preset limit on the number of queues an application may have. Each queue requires a control block and a queue data area. The memory for each is supplied by the application.

Queue Determinism

Basic processing time required for sending and receiving queue messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the queue.

Queue Information

Application tasks may obtain a list of active queues. Detailed information about each queue can also be obtained. This information includes the queue name, message format, suspension type, number of messages present, and the first task waiting.

Queue Services Function Reference

The following function reference contains all functions related to Nucleus PLUS queues. The following functions are contained in this reference:

- [NU_Broadcast_To_Queue](#)
- [NU_Create_Queue](#)
- [NU_Delete_Queue](#)
- [NU_Established_Queues](#)
- [NU_Queue_Information](#)
- [NU_Queue_Pointers](#)
- [NU_Receive_From_Queue](#)
- [NU_Reset_Queue](#)
- [NU_Send_To_Front_Of_Queue](#)
- [NU_Send_To_Queue](#)

NU_Broadcast_To_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service broadcasts a message to all tasks waiting for a message from the specified queue. If no tasks are waiting, the message is simply placed at the end of the queue. Queues are capable of holding multiple messages. Queue messages are comprised of a fixed or variable number of UNSIGNED data elements, depending on how this queue was created. The parameters of this service are further defined as follows:

Usage

```
STATUS NU_Broadcast_To_Queue (NU_QUEUE *queue,
                             VOID      *message,
                             UNSIGNED  size,
                             UNSIGNED  suspend);
```

Arguments

- queue
 Pointer to the user-supplied queue control block.
- message
 Pointer to the broadcast message.
- size
 Specifies the number of UNSIGNED data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the message size supported by the queue. If the queue supports fixed-size messages, this parameter must be exactly the same as the message size supported by the queue.
- suspend
 Specifies whether to suspend the calling task if there is insufficient room in the queue to hold the message.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be copied into the queue.
 timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the queue or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The message size specified is not compatible with the size specified when the queue was created.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_QUEUE_FULL**
The message could not be immediately placed in the queue because there was not enough space available.
- **NU_TIMEOUT**
The queue is unable to accept the message even after suspending for the specified timeout value.
- **NU_QUEUE_DELETED**
The queue was deleted while the task was suspended.
- **NU_QUEUE_RESET**
The queue was reset while the task was suspended.

Example

```
NU_QUEUE Queue;
UNSIGNED message[4];
STATUS status
.
.
.
/* Build a message to send to a queue. The contents of "message" are not
   significant. */

message[0] = 0x00001111;
message[1] = 0x22223333;
message[3] = 0x44445555;
message[4] = 0x66667777;

/* Send the message to the queue control block "Queue". If the queue is
```

```
    full, suspend until the request can be satisfied. Assume "Queue" has
    previously been created with the Nucleus PLUS NU_Create_Queue service
    call.*/

    status = NU_Broadcast_To_Queue(&Queue, &message[0], 4, NU_SUSPEND);

    /* At this point, status indicates whether the service request was
       successful. */
```

Related Topics

[Queue Services Function Reference](#)

[NU_Receive_From_Queue](#)

[NU_Send_To_Front_Of_Queue](#)

[NU_Send_To_Queue](#)

[NU_Queue_Information](#)

NU_Create_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service creates a message queue. Queues are created to support management of either fixed or variable sized messages. Queue messages are comprised of one or more UNSIGNED data elements. The parameters of this service are further defined as follows:

Usage

```
STATUS NU_Create_Queue (NU_QUEUE *queue,  
                        CHAR      *name,  
                        VOID      *start_address,  
                        UNSIGNED queue_size,  
                        OPTION    message_type,  
                        UNSIGNED message_size,  
                        OPTION    suspend_type);
```

Arguments

- **queue**
Pointer to the user-supplied queue control block. Subsequent requests made to the queue require this pointer.
- **name**
Pointer to a seven-character name for the queue. The name must be null-terminated.
- **start_address**
Specifies the starting address for the queue. This address must be properly aligned for UNSIGNED data access.
- **queue_size**
Specifies the number of UNSIGNED elements in the queue.
- **message_type**
Specifies the type of messages managed by the queue. NU_FIXED_SIZE specifies that the queue's messages are fixed-size messages. A fixed-size message queue only uses the area of the queue that is evenly divisible by the message size. NU_VARIABLE_SIZE indicates that the queue manages variable-size messages. Each variable-size message requires an additional UNSIGNED data element of overhead inside the queue.
- **message_size**
If the queue supports fixed-size messages, this parameter specifies the exact size of each message. Otherwise, if the queue supports variable-size messages, this parameter indicates the maximum message size. All sizes are in terms of UNSIGNED data elements.

- **suspend_type**
Specifies how tasks suspend on the queue. Valid options for this parameter are **NU_FIFO** and **NU_PRIORITY**, which represent FIFO and priority-order task suspension, respectively.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue control block pointer is NULL or is already in use.
- **NU_INVALID_MEMORY**
The memory area specified by the **start_address** is invalid.
- **NU_INVALID_MESSAGE**
The **message_type** parameter is invalid.
- **NU_INVALID_SIZE**
Either the message size is greater than the queue size, or that the queue size or message size is 0.
- **NU_INVALID_SUSPEND**
Indicates the **suspend_type** parameter is invalid.
- **NU_NOT_ALIGNED**
A pointer is not aligned on a four byte boundary to the parameter **start_address**.

Example

```
/* Assume queue control block "Queue" is defined as a global data
   structure. This is one of several ways to allocate a control block */
NU_QUEUE Queue;
.
.
/* Assume status is defined locally. */

STATUS status;    /* Queue creation status */

/* Create a queue with a capacity of 1000 UNSIGNED elements starting at
   the address pointed to by the variable "start". Variable-length
   messages are supported, with a maximum message size of 20. Tasks
   suspend on this queue in FIFO order. */

status =  NU_Create_Queue(&Queue, "any name", start, 1000,
                        NU_VARIABLE_SIZE, 20,  NU_FIFO);

/* At this point status indicates if the service was successful. */
```

Related Topics

[Queue Services Function Reference](#)

[NU_Queue_Pointers](#)

[Kernel Demo](#)

[NU_Established_Queues](#)

[NU_Reset_Queue](#)

[NU_Delete_Queue](#)

NU_Delete_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service deletes a previously created message queue. The queue parameter identifies the message queue to delete. Tasks suspended on this queue are resumed with the appropriate error status. The application must prevent the use of this queue during and after deletion.

Usage

```
STATUS NU_Delete_Queue (NU_QUEUE *queue);
```

Arguments

- queue
Pointer to the user-supplied queue control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_QUEUE
The queue pointer is invalid.

Example

```
NU_QUEUE Queue;  
STATUS status  
.  
.  
.  
/* Delete the queue control block "Queue". Assume "Queue" has previously  
   been created with the Nucleus PLUS NU_Create_Queue service call. */  
status = NU_Delete_Queue(&Queue);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Queue Services Function Reference](#)

[NU_Established_Queues](#)

[NU_Queue_Information](#)

[NU_Queue_Pointers](#)

[NU_Reset_Queue](#)

[NU_Create_Queue](#)

NU_Established_Queues

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns the number of established queues. All created queues are considered established. Deleted queues are no longer considered established.

Usage

```
UNSIGNED NU_Established_Queues (VOID);
```

Return Values

- This service call returns the number of created queues in the system

Example

```
UNSIGNED total_queues;

/* Obtain the number of queues. */
total_queues = NU_Established_Queues();
```

Related Topics

[Queue Services Function Reference](#)

[NU_Queue_Information](#)

[NU_Reset_Queue](#)

[NU_Delete_Queue](#)

[NU_Queue_Pointers](#)

[NU_Create_Queue](#)

NU_Queue_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns various information about the specified message communication queue.

Usage

```
STATUS NU_Queue_Information (NU_QUEUE *queue,
                             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);
```

Arguments

- **queue**
 Pointer to the user-supplied queue control block.
- **name**
 Pointer to an eight-character destination area for the message-queue's name. This includes space for the null terminator.
- **start_address**
 Pointer to a memory pointer for holding the starting address of the queue.
- **queue_size**
 Pointer to a variable for holding the total number of UNSIGNED data elements in the queue.
- **available**
 Pointer to a variable for holding the number of available UNSIGNED data elements in the queue.
- **messages**
 Pointer to a variable for holding the number of messages currently in the queue.
- **message_type**
 Pointer to a variable for holding the type of messages supported by the queue. Valid message types are NU_FIXED_SIZE and NU_VARIABLE_SIZE.

- **message_size**
Pointer to a variable for holding the number of UNSIGNED data elements in each queue message. If the queue supports variable-length messages, this number is the maximum message size.
- **suspend_type**
Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
- **tasks_waiting**
Pointer to a variable for holding the number of tasks waiting on the queue.
- **first_task**
Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue pointer is invalid.

Example

```
NU_QUEUE Queue;
CHAR      queue_name[8];
VOID      *start_address;
UNSIGNED  size;
UNSIGNED  available;
UNSIGNED  messages;
OPTION    message_type;
UNSIGNED  message_size;
OPTION    suspend_type;
UNSIGNED  tasks_suspended;
NU_TASK   *first_task;
STATUS    status;

/* Obtain information about the message queue control block "Queue".
   Assume "Queue" has previously been created with the Nucleus PLUS
   NU_Create_Queue service call. */
status = NU_Queue_Information(&Queue, queue_name, &start_address,
                             &size, &available, &messages,
                             &message_type, &message_size,
                             &suspend_type, &tasks_suspended,
                             &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

[Queue Services Function Reference](#)

[NU_Established_Queue](#)

[NU_Reset_Queue](#)

[NU_Delete_Queue](#)

[NU_Queue_Pointers](#)

[NU_Create_Queue](#)

NU_Queue_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service builds a sequential list of pointers to all established message queues in the system.

Note



Queues that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Queue_Pointers (NU_QUEUE **pointer_list,  
                           UNSIGNED maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_QUEUE` pointers. This array will be filled with pointers of established queues in the system.
- `maximum_pointers`
The maximum number of `NU_QUEUE` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created queues in the system.

Example

```
/* Define an array capable of holding 20 queue pointers. */  
NU_QUEUE *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active queue pointers (Maximum of 20). */  
number = NU_Queue_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, number contains the actual number of pointers in the  
list. */
```

Related Topics

[Queue Services Function Reference](#)

[NU_Established_Queue](#)

[NU_Reset_Queue](#)

[NU_Delete_Queue](#)

[NU_Queue_Information](#)

[NU_Create_Queue](#)

NU_Receive_From_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service retrieves a message from the specified queue. If the queue contains one or more messages, the message in front is immediately removed from the queue and copied into the designated location. Queue messages are comprised of a fixed or variable number of UNSIGNED data elements, depending on the type of messages supported by the queue.

Usage

```
STATUS NU_Receive_From_Queue (NU_QUEUE *queue,  
                             VOID      *message,  
                             UNSIGNED  size,  
                             UNSIGNED  *actual_size,  
                             UNSIGNED  suspend);
```

Arguments

- queue
Pointer to the user-supplied queue control block.
- message
Pointer to the message destination. The message destination must be capable of holding “size” UNSIGNED data elements.
- size
Specifies the number of UNSIGNED data elements in the message. This number must correspond to the message size defined when the queue was created
- actual_size
Pointer to a variable to hold the actual number of UNSIGNED data elements in the received message.
- suspend
Specifies whether to suspend the calling task if the queue is empty.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until a message is available.
timeout value

(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue pointer is invalid.
- **NU_INVALID_POINTER**
The message destination pointer is NULL or the “actual_size” pointer is NULL.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_QUEUE_EMPTY**
The queue is empty.
- **NU_INVALID_SIZE**
The size parameter is different from the message size supported by the queue. Applies to queues defined with fixed and variable message sizes.
- **NU_TIMEOUT**
The queue is still empty even after suspending for the specified timeout value.
- **NU_QUEUE_DELETED**
The queue was deleted while the task was suspended.
- **NU_QUEUE_RESET**
The queue was reset while the task was suspended.

Example

```

NU_QUEUE    Queue;
UNSIGNED    message[4];
UNSIGNED    actual_size;
STATUS      status;
.
.
.
/* Receive a 4-UNSIGNED data element message from the queue control block
   "Queue". If the queue is empty, suspend until the request can be
   satisfied. Assume "Queue" has previously been created with the Nucleus
   PLUS NU_Create_Queue service call. */
status =  NU_Receive_From_Queue(&Queue, &message[0], 4,
                               &actual_size, NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" contains the received message. */

```

Related Topics

[Queue Services Function Reference](#)

[NU_Send_To_Front_Of_Queue](#)

[Kernel Demo](#)

[NU_Queue_Information](#)

[NU_Send_To_Queue](#)

[NU_Broadcast_To_Queue](#)

NU_Reset_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service discards all messages currently in the queue specified by queue.

Usage

```
STATUS NU_Reset_Queue (NU_QUEUE *queue);
```

Arguments

- queue
Pointer to the user-supplied queue control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_QUEUE
The queue pointer is invalid.

Example

```
NU_QUEUE Queue;
STATUS status
.
.
.
/* Reset the queue control block "Queue". Assume "Queue" has previously
   been created with the Nucleus PLUS NU_Create_Queue service call. */
status = NU_Reset_Queue(&Queue);
```

Related Topics

[Queue Services Function Reference](#)

[NU_Receive_From_Queue](#)

[NU_Send_To_Queue](#)

[NU_Queue_Information](#)

[NU_Send_To_Front_Of_Queue](#)

[NU_Broadcast_To_Queue](#)

NU_Send_To_Front_Of_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service places a message at the front of the specified queue. If there is enough space in the queue to hold the message, this service is processed immediately. Queue messages are comprised of a fixed or variable number of UNSIGNED data elements, depending on the types of messages supported by the queue.

Usage

```
STATUS NU_Send_To_Front_Of_Queue (NU_QUEUE *queue,  
                                  VOID      *message,  
                                  UNSIGNED  size,  
                                  UNSIGNED  suspend);
```

Arguments

- queue
Pointer to the user-supplied queue control block.
- message
Pointer to the message to send.
- size
Specifies the number of UNSIGNED data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the same as the message size supported by the queue.
- suspend
Specifies whether to suspend the calling task if the queue is full.

The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be sent.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The specified message size is incompatible with the message size supported by the queue.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_QUEUE_FULL**
The queue is full.
- **NU_TIMEOUT**
The queue is still full even after suspending for the specified timeout value.
- **NU_QUEUE_DELETED**
The queue was deleted while the task was suspended.
- **NU_QUEUE_RESET**
The queue was reset while the task was suspended.

Example

```

NU_QUEUE    Queue;
UNSIGNED    message[4];
STATUS      status;
.
.
.
/* Build a 4 UNSIGNED variable message to send. The contents of "message"
   have no significance. */
message[0]   = 0x00001111;
message[1]   = 0x00002222;
message[2]   = 0x00003333;
message[3]   = 0x00004444;

/* Send message to the queue control block "Queue". Suspend the calling
   task until the message can be sent. Assume "Queue" has previously been
   created with the Nucleus PLUS NU_Create_Queue service call. */

Status = NU_Send_To_Front_Of_Queue(&Queue, &message[0], 4, NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" was sent to "Queue". */

```

Related Topics

[Queue Services Function Reference](#)

[NU_Queue_Information](#)

[NU_Receive_From_Queue](#)

[NU_Send_To_Queue](#)

[NU_Broadcast_To_Queue](#)

NU_Send_To_Queue

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service places a message at the back of the specified queue. If there is enough space in the queue to hold the message, this service is processed immediately. Queue messages are comprised of a fixed or variable number of UNSIGNED data elements, depending on the type of messages supported by the queue.

Usage

```
STATUS NU_Send_To_Queue (NU_QUEUE *queue,  
                        VOID      *message,  
                        UNSIGNED size,  
                        UNSIGNED suspend);
```

Arguments

- **queue**
Pointer to the user-supplied queue control block.
- **message**
Pointer to the message to send.
- **size**
Specifies the number of UNSIGNED data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the message size supported by the queue. If the queue supports fixed-size messages, this parameter must be exactly the same as the message size supported by the queue.
- **suspend**
Specifies whether to suspend the calling task if the queue is full.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be sent.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_QUEUE**
The queue pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The message size is incompatible with the message size supported by the queue.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_QUEUE_FULL**
The queue is full.
- **NU_TIMEOUT**
The queue is still full even after suspending for the specified timeout value.
- **NU_QUEUE_DELETED**
The queue was deleted while the task was suspended.
- **NU_QUEUE_RESET**
The queue was reset while the task was suspended.

Example

```
NU_QUEUE Queue;
UNSIGNED message[4];
STATUS    status;
.
.
.
/* Build a 4 UNSIGNED variable message to send. The contents of "message"
   have no significance. */
message[0] = 0x00001111;
message[1] = 0x00002222;
message[2] = 0x00003333;
message[3] = 0x00004444;

/* Send the message to the queue control block "Queue". Suspend the
   calling task until the message can be sent. Assume "Queue" has
   previously been created with the Nucleus PLUS NU_Create_Queue service
   call. */
status = NU_Send_To_Queue(&Queue, &message[0], 4, NU_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" was sent to "Queue". */
```


Related Topics

[Queue Services Function Reference](#)[NU_Queue_Information](#)[NU_Receive_From_Queue](#)[NU_Send_To_Front_Of_Queue](#)[Kernel Demo](#)[NU_Broadcast_To_Queue](#)

Queue Example Source Code

In the previous chapter you looked at an example that demonstrated how to communicate between tasks with mailboxes. In this section, you will look at a very similar example, but using queues to communicate between several tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Four Nucleus PLUS structures are used in this example. Three NU_TASK structures are used, one for each task in the system. The NU_QUEUE structure is for the queue that will be used to communicate messages between the three tasks in the system.

```
NU_TASK task_recv_1;  
NU_TASK task_recv_2;  
NU_TASK task_send;  
NU_QUEUE queue_comm;
```

The three void pointers stack_recv_1, stack_recv_2, and stack_send will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_recv_1;  
VOID *stack_recv_2;  
VOID *stack_send;
```

Similar to these three void pointers, the data_queue pointer will be used to hold a pointer to the data area for the queue. It can either be used to deallocate the associated memory, or discarded if memory deallocation is not necessary.

```
VOID *data_queue;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.

```
VOID entry_recv_1(UNSIGNED argc, VOID *argv);  
VOID entry_recv_2(UNSIGNED argc, VOID *argv);  
VOID entry_send(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to set up memory for three task stacks, and the queue data area. Therefore, in Application_Initialize there are four separate calls to NU_Allocate_Memory. Application_Initialize is also used to create the queue and associate the allocated memory for its queue data area.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the NU_Allocate_Memory call, you are allocating a 1024 byte block of memory out of the mem_pool dynamic memory pool, that was passed in to Application_Initialize. A pointer to the newly allocated memory is assigned to the stack_rcv_1, stack_rcv_2, and stack_send respectively. The pointer to this memory allocation is passed to the NU_Create_Task call, which will use this memory as the task stack.

For this demonstration, note that task_rcv_1 and task_rcv_2 are given a higher priority (priority level of 7) than task_send. By doing this, you are ensuring that task_rcv_1 and task_rcv_2 will always run before task_send. The task_send will only run when both task_rcv_1 and task_rcv_2 are suspended.

```
NU_Allocate_Memory(mem_pool, &stack_rcv_1, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_rcv_1, "rcv_1", entry_rcv_1, 0, NU_NULL,  
              stack_rcv_1, 1024, 7, 0, NU_PREEMPT, NU_START);  
  
NU_Allocate_Memory(mem_pool, &stack_rcv_2, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_rcv_2, "rcv_2", entry_rcv_2, 0, NU_NULL,  
              stack_rcv_2, 1024, 7, 0, NU_PREEMPT, NU_START);  
  
NU_Allocate_Memory(mem_pool, &stack_send, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,  
              stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

First, allocate memory for the queue data area with a call to NU_Allocate_Memory. This call allocates 32768 bytes out of the mem_pool dynamic memory pool, and assigns a pointer to this memory to the data_queue void pointer. Then, call NU_Create_Queue to associate this memory to the queue_comm queue. The queue_comm queue is a queue with fixed sized messages (NU_FIXED_SIZE), and each message will be 32-bits in size. The queue is associated with the name "comm" and tasks that choose to suspend on this queue will be resumed in FIFO order.

```
NU_Allocate_Memory(mem_pool, &data_queue, 32768, NU_NO_SUSPEND);  
NU_Create_Queue(&queue_comm, "comm", data_queue, 8192,  
              NU_FIXED_SIZE, 1, NU_FIFO);  
}
```

The entry_rcv_1 and entry_rcv_2 functions serve as the entry point for the task_rcv_1 and task_rcv_2 tasks respectively. The tasks will continuously loop, issuing an NU_Receive_From_Queue call for each iteration of the loop. The NU_Receive_From_Queue will suspend until there is a message placed into the queue (as indicated by NU_SUSPEND). Whenever a message is received, NU_Receive_From_Queue will exit with a Return Values of

NU_SUCCESS. After the call has returned, `recvmsg` will contain the message received. Therefore, there are two tasks that are continuously suspending on the same queue, both waiting for a message to be placed into the queue. The Nucleus PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `queue_comm` queue was created.

```
VOID entry_recv_1(UNSIGNED argc, VOID *argv)
{
    UNSIGNED recvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Queue(&queue_comm, &recvmsg, 1, &actual_size,
                                   NU_SUSPEND) == NU_SUCCESS)
        {
            /* recvmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}

VOID entry_recv_2(UNSIGNED argc, VOID *argv)
{
    UNSIGNED recvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Queue(&queue_comm, &recvmsg, 1, &actual_size,
                                   NU_SUSPEND) == NU_SUCCESS)
        {
            /* recvmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}
```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_recv_1` and `task_recv_2` tasks are of a higher priority, and will always be given the first chance to run. Because of this, whenever `task_send` sends a message with `queue_comm`, either `task_recv_1` or `task_recv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Queue` which will send a single message with the `queue_comm` queue. The second service call that is issued is `NU_Broadcast_To_Queue`, which will send the message to every task that is currently suspended on this queue. Note that in this example, whenever this task is running, there will always be two tasks (`task_recv_1` and `task_recv_2`) suspended on the `queue_comm` queue. The

result is that the message that is sent with `NU_Send_To_Queue` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Queue` will be received by both suspended tasks.

```
VOID entry_send(UNSIGNED argc, VOID *argv)
{
    UNSIGNED sendmsg;

    while(1)
    {
```

Assign decimal 1 to `sendmsg`, then issue `NU_Send_To_Queue` on the `queue_comm` queue. Since two tasks will always be suspended on this queue, and the queue was created with the `NU_FIFO` suspension flag, the first task that suspended on the queue will always receive this message.

```
        sendmsg=1;
        if (NU_Send_To_Queue(&queue_comm, &sendmsg, 1, NU_SUSPEND)
            == NU_SUCCESS)
        {
            /* recvmmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
```

Assign decimal 2 to `sendmsg`, then issue `NU_Broadcast_To_Queue` on the `queue_comm` queue. Because the priority of `task_rcv_1` and `task_rcv_2` is higher priority than this task, you are guaranteed that two tasks will always be suspended on this queue. Therefore, the result of the `NU_Broadcast_To_Queue` service is that both tasks will be sent the message.

```
        sendmsg=2;
        if (NU_Broadcast_To_Queue(&queue_comm, &sendmsg, 1,
            NU_SUSPEND) == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}
```

Event Queue Manager

The event queue manager (EQM) facilitates communications between components via events that are dispatched globally and can also carry data.

EQM shares data through a global array of event data. This array is operated in circular fashion. Events are sent to the waiting components. When an event is received it is stored in the EQM circular buffer and a notification is sent to all the waiting components. The event will be sent to

only those components that are waiting for it. This architecture also handles the case when events will not be sent to components that are currently not in a waiting state and are busy processing any received event. This is handled with the provision of receiving the relevant event that has occurred after the last processed event.

Event Queue Manager Function Reference

The following function reference contains all functions related to Nucleus Event Queue Manager queues. The following functions are contained in this reference:

- [NU_EQM_Create](#)
- [NU_EQM_Delete](#)
- [NU_EQM_Post_Event](#)
- [NU_EQM_Get_Event_Data](#)
- [NU_EQM_Wait_Event](#)

NU_EQM_Create

This function creates a new instance of EQM and initializes the data structures that control the operation of the EQM.

Usage

```
STATUS NU_EQM_Create(EQM_EVENT_QUEUE *event_queue_ptr,  
                    UINT32          queue_size,  
                    UINT16          max_event_data_size,  
                    NU_MEMORY_POOL  *memory_pool_ptr);
```

Arguments

- `event_queue_ptr`
Event queue pointer.
- `queue_size`
Number of elements in the queue.
- `max_event_data_size`
Maximum size of event data.
- `memory_pool_ptr`
Memory pool pointer.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_EQM_INVALID_INPUT`
Invalid input.

Related Topics

[Event Queue Manager Function Reference](#)

[NU_EQM_Delete](#)

NU_EQM_Delete

This function deletes the instance of the Event Queue Manager.

Usage

```
STATUS NU_EQM_Delete (EQM_EVENT_QUEUE *event_queue_ptr);
```

Arguments

- `event_queue_ptr`
Event queue pointer.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_POINTER`
Invalid pointer.
- `NU_EQM_INVALID_INPUT`
Invalid event queue.

Related Topics

[Event Queue Manager Function Reference](#)

[NU_EQM_Create](#)

NU_EQM_Post_Event

This function posts the event in the data buffer and informs the waiting components about the arrival of the event.

Usage

```
STATUS NU_EQM_Post_Event(EQM_EVENT_QUEUE *event_queue_ptr,  
                        EQM_EVENT *event_ptr,  
                        UINT16 event_data_size,  
                        EQM_EVENT_ID *posted_event_id_ptr);
```

Arguments

- **event_queue_ptr**
Pointer to the event queue where event is to be posted.
- **event_ptr**
Pointer to the event containing the event type and associated data.
- **event_data_size**
Size of associated event data in bytes in posted_event_id_ptr. The pointer holds the posted event ID. NU_NULL is passed if it is not required.
- **posted_event_id_ptr**
Pointer to hold the posted event ID. NU_NULL is passed if it is not required.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POINTER**
Invalid pointer.
- **NU_EQM_INVALID_EVENT_SIZE**
Invalid event data size. Size is greater than the limit specified for the queue.

Related Topics

[Event Queue Manager Function Reference](#)

[NU_EQM_Get_Event_Data](#)

NU_EQM_Get_Event_Data

The components that are interested in reading data associated with the event call this function.

Usage

```
STATUS NU_EQM_Get_Event_Data(EQM_EVENT_QUEUE *event_queue_ptr,  
                             EQM_EVENT_ID   event_id,  
                             EQM_EVENT_HANDLE event_handle,  
                             EQM_EVENT      *event_ptr);
```

Arguments

- `event_queue_ptr`
Pointer to the event queue from where event data is to be extracted.
- `event_id`
ID of the event whose data is to be read.
- `event_handle`
Handle of the event whose data is to be read. Handle is used to avoid searching overhead.
- `event_ptr`
Points to the structure where event's data will be copied.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_EQM_EVENT_WITHOUT_DATA`
The event type doesn't have data.
- `NU_INVALID_POINTER`
Invalid pointer.
- `NU_EQM_INVALID_HANDLE`
Invalid handle.
- `NU_EQM_EVENT_EXPIRED`
Event is expired and it does not exist in the buffer any longer.

Related Topics

[Event Queue Manager Function Reference](#)

[NU_EQM_Post_Event](#)

NU_EQM_Wait_Event

The components that are interested in a particular event reading data associated with the event call this function.

Usage

```
STATUS NU_EQM_Wait_Event(  
    EQM_EVENT_QUEUE *event_queue_ptr,  
    UINT32           requested_events_mask,  
    UINT32           *recvd_event_type_ptr,  
    EQM_EVENT_ID     *recvd_event_id_ptr,  
    EQM_EVENT_HANDLE *recvd_event_handle_ptr);
```

Arguments

- **event_queue_ptr**
Pointer to the event queue.
- **requested_events_mask**
Mask that specifies the events to wait for.
- **recvd_event_type_ptr**
Points to the location where the type of the received event will be placed.
- **recvd_event_id_ptr**
Points to the location which has the event ID of the last processed event. The new event of relevant type will be searched from there onwards. For the first call the event_id must be '0'. A following '0' is used to indicate to skip the search and wait for the next received event of a relevant type. When the call returns it contains the ID of the received event.
- **recvd_event_handle_ptr**
Points to the location where a handle to the received event will be placed. This handle is required if data associated with the event is to be read.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_POINTER**
Invalid pointer.
- **NU_EQM_INVALID_INPUT**
Invalid input. Mask is 0.

Related Topics

[Event Queue Manager Function Reference](#)

[NU_EQM_Delete](#)

Pipes

Pipe is a method of communication between tasks similar to the queue.

Pipes provide a mechanism for transmitting multiple messages. Messages are sent and received by value. A send-message request copies the message into the pipe, while a receive-message request copies the message out of the pipe. Messages may be placed at the front of the pipe or at the back of the pipe.

Pipe Message Size

A pipe message consists of one or more bytes. Both fixed-length and variable-length messages are supported. The type of message format is defined when the pipe is created. Variable-length message pipes require an additional 32-bit word of overhead for each message in the pipe. Additionally, receive-message requests on variable-length message pipes specify the maximum message size, while the same request on fixed-length message pipes specify the exact message size.

Pipe Suspension

Send and receive pipe services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks may suspend on a pipe for several reasons. Tasks attempting to receive a message from an empty pipe can suspend. Also, a task attempting to send a message to a full pipe can suspend. A suspended task is resumed when the pipe is able to satisfy that task's request. For example, suppose a task is suspended on a pipe waiting to receive a message. When a message is sent to the pipe, the suspended task is resumed.

Multiple tasks may suspend on a single pipe. Tasks are suspended in either FIFO or priority order, depending on how the pipe was created. If the pipe supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the pipe supports priority suspension, tasks are resumed from high priority to low priority.

Pipe Broadcast

A pipe message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the pipe are given the broadcast message.

Pipe Dynamic Creation

Nucleus PLUS pipes are created and deleted dynamically. There is no preset limit on the number of pipes an application may have. Each pipe requires a control block and a pipe data area. The memory for each is supplied by the application.

Pipe Determinism

Basic processing time required for sending and receiving pipe messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the pipe.

Pipe Information

Application tasks may obtain a list of active pipes. Detailed information about each pipe can also be obtained. This information includes the pipe name, message format, suspension type, number of messages present, and the first task waiting.

Pipe Services Function Reference

The following function reference contains all functions related to Nucleus PLUS pipes. The following functions are contained in this reference:

- [NU_Broadcast_To_Pipe](#)
- [NU_Create_Pipe](#)
- [NU_Delete_Pipe](#)
- [NU_Established_Pipes](#)
- [NU_Pipe_Information](#)
- [NU_Pipe_Pointers](#)
- [NU_Receive_From_Pipe](#)
- [NU_Reset_Pipe](#)
- [NU_Send_To_Front_Of_Pipe](#)
- [NU_Send_To_Pipe](#)

NU_Broadcast_To_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service broadcasts a message to all tasks waiting for a message from the specified pipe. If no tasks are waiting, the message is simply placed at the end of the pipe. Pipes are capable of holding multiple messages. Pipe messages are comprised of a fixed or variable number of bytes, depending on how the pipe was created.

Usage

```
STATUS NU_Broadcast_To_Pipe (NU_PIPE  *pipe,
                             VOID      *message,
                             UNSIGNED  size,
                             UNSIGNED  suspend);
```

Arguments

- **pipe**
 Pointer to the user-supplied pipe control block.
- **message**
 Pointer to the broadcast message.
- **size**
 Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be exactly the same as the message size supported by the pipe.
- **suspend**
 Specifies whether to suspend the calling task if there is insufficient room in the pipe to hold the message.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be copied into the pipe.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the pipe or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_PIPE**
The pipe pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The message size specified is not compatible with the size specified when the pipe was created.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_PIPE_FULL**
The message could not be immediately placed in the pipe because there was not enough space available.
- **NU_TIMEOUT**
The pipe is unable to accept the message even after suspending for the specified timeout value.
- **NU_PIPE_DELETED**
The pipe was deleted while the task was suspended.
- **NU_PIPE_RESET**
The pipe was reset while the task was suspended.

Example

```
NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
STATUS       status
.
.
.
/* Build a 4-byte message to send to a pipe.  The contents of "message"
   are not significant. */

message[0]   = 0x01;
message[1]   = 0x23;
message[2]   = 0x45;
message[3]   = 0x67;

/* Send a message to the pipe control block "Pipe". Do not suspend even
   if the pipe does not have enough room for the message.  Assume "Pipe"
   has previously been created with the Nucleus PLUS NU_Create_Pipe
   service call. */
```

```
status = NU_Broadcast_To_Pipe(&Pipe,&message[0], 4, NU_NO_SUSPEND);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Pipe Services Function Reference](#)

[NU_Receive_From_Pipe](#)

[NU_Send_To_Front_Of_Pipe](#)

[NU_Send_To_Pipe](#)

[NU_Pipe_Information](#)

NU_Create_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service creates a message pipe. Pipes are created to support management of either fixed or variable sized messages.

Usage

```
STATUS NU_Create_Pipe (NU_PIPE    *pipe,  
                      CHAR        *name,  
                      VOID        *start_address,  
                      UNSIGNED     pipe_size,  
                      OPTION      message_type,  
                      UNSIGNED     message_size,  
                      OPTION      suspend_type);
```

Arguments

- **pipe**
Pointer to the user-supplied pipe control block. Subsequent requests made to the pipe require this pointer.
- **name**
Pointer to a seven-character name for the pipe. The name must be null-terminated.
- **start_address**
Specifies the starting address for the pipe.
- **pipe_size**
Specifies the total number of bytes in the pipe.
- **message_type**
Specifies the type of messages that are managed by the pipe. `NU_FIXED_SIZE` and `NU_VARIABLE_SIZE` are the only valid message types. Each variable-size message requires an additional `UNSIGNED` data type of overhead inside the pipe. Additional padding bytes may be necessary for a message in order to insure `UNSIGNED` alignment of the next variable-sized message. A fixed-size message pipe only uses the area of the pipe that is evenly divisible by the message size.
- **message_size**
If the pipe supports fixed-size messages, this parameter specifies the exact size of each message. Otherwise, if the pipe supports variable-size messages, this parameter indicates the maximum message size. All sizes are in terms of bytes.

- `suspend_type`
Specifies how tasks suspend on the pipe. Valid options for this parameter are `NU_FIFO` and `NU_PRIORITY`, which represent FIFO and priority-order task suspension, respectively.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_PIPE`
The pipe control block pointer is `NULL` or is already in use.
- `NU_INVALID_MEMORY`
The memory area specified by the `start_address` is invalid.
- `NU_INVALID_MESSAGE`
The `message_type` parameter is invalid.
- `NU_INVALID_SIZE`
Either the message size specified is larger than the pipe size, or that the message size or pipe size is 0.
- `NU_INVALID_SUSPEND`
Indicates the `suspend_type` parameter is invalid.
- `NU_NOT_ALIGNED`
A pointer is not aligned on a four byte boundary to the parameter `start_address`.

Example

```
/* Assume pipe control block "Pipe" is defined as a global data structure.
   This is one of several ways to allocate a control block. */

NU_PIPE Pipe;
.
.
/* Assume status is defined locally. */

STATUS status; /* Pipe creation status */

/* Create a pipe in a 1500-byte memory area starting at the address
   pointed to by the variable "start". Fixed-size, 20-byte messages are
   supported by this pipe. Tasks suspend on this pipe in order of their
   priority. */

status = NU_Create_Pipe(&Pipe, "any name", start, 1500,
                       NU_FIXED_SIZE, 20, NU_PRIORITY);

/* At this point status indicates if the service was successful. */
```

Related Topics

[Pipe Services Function Reference](#)

[NU_Pipe_Information](#)

[NU_Reset_Pipe](#)

[NU_Established_Pipes](#)

[NU_Pipe_Pointers](#)

[NU_Delete_Pipe](#)

NU_Delete_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service deletes a previously created message pipe. The pipe parameter identifies the message pipe to delete. Tasks suspended on this pipe are resumed with the appropriate error status. The application must prevent the use of this pipe during and after deletion.

Usage

```
STATUS NU_Delete_Pipe (NU_PIPE *pipe);
```

Arguments

- pipe
Pointer to the user-supplied pipe control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_PIPE
The pipe pointer is invalid.

Example

```
NU_PIPE Pipe;
STATUS status;
.
.
.
/* Delete the pipe control block "Pipe". Assume "Pipe" has previously
   been created with the Nucleus PLUS NU_Create_Pipe service call. */
status = NU_Delete_Pipe(&Pipe);

/* At this point, status indicates whether the service request was
   successful. */
```

Related Topics

[Pipe Services Function Reference](#)

[NU_Established_Pipes](#)

[NU_Pipe_Information](#)

[NU_Pipe_Pointers](#)

[NU_Reset_Pipe](#)

[NU_Create_Pipe](#)

NU_Established_Pipes

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns the number of established pipes. All created pipes are considered established. Deleted pipes are no longer considered established.

Usage

```
UNSIGNED NU_Established_Pipes (VOID);
```

Return Values

- This service call returns the number of created pipes in the system

Example

```
UNSIGNED total_pipes;  
  
/* Obtain the total number of pipes. */  
total_pipes = NU_Established_Pipes( );
```

See Also

, , ,

Related Topics

[Pipe Services Function Reference](#)

[NU_Pipe_Information](#)

[NU_Reset_Pipe](#)

[NU_Delete_Pipe](#)

[NU_Pipe_Pointers](#)

[NU_Create_Pipe](#)

NU_Pipe_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service returns various information about the specified message-communication pipe.

Usage

```
STATUS NU_Pipe_Information (NU_PIPE    *pipe,
                           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);
```

Arguments

- pipe
 Pointer to the user-supplied pipe control block.
- name
 Pointer to an eight-character destination area for the pipe's name. This includes space for the null terminator.
- start_address
 Pointer for holding the starting address of the pipe.
- pipe_size
 Pointer for holding the total number of bytes in the pipe.
- available
 Pointer for holding the number of available bytes in the pipe.
- messages
 Pointer to a variable for holding the number of messages currently in the pipe.
- message_type
 Pointer to a variable for holding the type of messages supported by the pipe. Valid message types are NU_FIXED_SIZE and NU_VARIABLE_SIZE.

- **message_size**
Pointer to a variable for holding the number of bytes in each message. If the pipe supports fixed-size messages, this is the exact size of each message. If the pipe supports variable-size messages, this is the maximum size of each message.
- **suspend_type**
Pointer to a variable for holding the task suspend type. Valid task suspend types are `NU_FIFO` and `NU_PRIORITY`.
- **tasks_waiting**
Pointer to a variable for holding the number of tasks waiting on the pipe.
- **first_task**
Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_PIPE**
The pipe pointer is invalid.

Example

```
NU_PIPE  Pipe;
CHAR     pipe_name[8];
VOID     *start_address;
UNSIGNED pipe_size;
UNSIGNED available;
UNSIGNED messages;
OPTION   message_type;
UNSIGNED message_size;
OPTION   suspend_type;
UNSIGNED tasks_suspended;
NU_TASK  *first_task;
STATUS   status
.
.
.
/* Obtain information about the message pipe control block "Pipe". Assume
   "Pipe" has previously been created with the Nucleus PLUS NU_Create_Pipe
   service call. */
status = NU_Pipe_Information(&Pipe, pipe_name, &start_address,
                           &pipe_size, &available, &messages,
                           &message_type, &message_size,
                           &suspend_type, &tasks_suspended,
                           &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

[Pipe Services Function Reference](#)

[NU_Established_Pipes](#)

[NU_Reset_Pipe](#)

[NU_Delete_Pipe](#)

[NU_Pipe_Pointers](#)

[NU_Create_Pipe](#)

NU_Pipe_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: No

This service builds a sequential list of pointers to all established message pipes in the system. Pipes that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Pipe_Pointers (NU_PIPE  **pointer_list,  
                           UNSIGNED maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_PIPE` pointers. This array will be filled with pointers of established pipes in the system.
- `maximum_pointers`
The maximum number of `NU_PIPE` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created pipes in the system.

Example

```
/* Define an array capable of holding 20 pipe pointers. */  
NU_PIPE  *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active pipe pointers (Maximum of 20). */  
number =  NU_Pipe_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, number contains the actual number of pointers in the  
   list.  */
```


Related Topics

[Pipe Services Function Reference](#)

[NU_Established_Pipes](#)

[NU_Reset_Pipe](#)

[NU_Delete_Pipe](#)

[NU_Pipe_Information](#)

[NU_Create_Pipe](#)

NU_Receive_From_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service retrieves a message from the specified pipe. If the pipe contains one or more messages, the message in front is immediately removed from the pipe and copied into the designated location. Pipe messages are comprised of a fixed or variable number of bytes, depending on the type of the messages supported by the pipe.

Usage

```
STATUS NU_Receive_From_Pipe (NU_PIPE  *pipe,  
                             VOID      *message,  
                             UNSIGNED  size,  
                             UNSIGNED  *actual_size,  
                             UNSIGNED  suspend);
```

Arguments

- pipe
Pointer to the pipe.
- message
Pointer to message destination. The message destination must be large enough to hold size bytes.
- size
Specifies the number of bytes in the message. This number must correspond to the message size defined when the pipe was created.
- actual_size
Pointer to a variable to hold the actual number of bytes in the received message.
- suspend
Specifies whether to suspend the calling task if the pipe is empty.

The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until a message is available.
timeout value

(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_PIPE**
The pipe pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL or the actual size pointer is NULL.
- **NU_INVALID_SIZE**
The size parameter is different from the message size supported by the pipe.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_PIPE_EMPTY**
The pipe is empty.
- **NU_TIMEOUT**
The pipe is still empty even after suspending for the specified timeout value.
- **NU_PIPE_DELETED**
Th pipe was deleted while the task was suspended.
- **NU_PIPE_RESET**
Th pipe was reset while the task was suspended.

Example

```

NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
UNSIGNED     actual_size;
STATUS       status;
.
.
.
/* Receive a 4-byte, fixed size message from the pipe control block
"Pipe".
   Do not suspend even if the pipe is empty. Assume "Pipe" has previously
   been created with the Nucleus PLUS NU_Create_Pipe service call. */
status = NU_Receive_From_Pipe(&Pipe,&message[0], 4, &actual_size,
                             NU_NO_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" contains the message and
   "actual_size" contains 4.* /

```

Related Topics

[Pipe Services Function Reference](#)

[NU_Pipe_Information](#)

[NU_Send_To_Front_Of_Pipe](#)

[NU_Send_To_Pipe](#)

[NU_Broadcast_To_Pipe](#)

NU_Reset_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service discards all messages currently in the pipe specified by pipe. All tasks suspended on the pipe are resumed with the appropriate reset status.

Usage

```
STATUS NU_Reset_Pipe (NU_PIPE *pipe);
```

Arguments

- pipe
Pointer to the user-supplied pipe control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_PIPE
The pipe pointer is invalid.

Example

```
NU_PIPE Pipe;  
STATUS status;  
.  
.  
.  
/* Reset the pipe control block "Pipe". Assume "Pipe" has previously been  
   created with the Nucleus PLUS NU_Create_Pipe service call. */  
status = NU_Reset_Pipe(&Pipe);
```

Related Topics

[Pipe Services Function Reference](#)

[NU_Pipe_Information](#)

[NU_Receive_From_Pipe](#)

[NU_Send_To_Front_Of_Pipe](#)

[NU_Send_To_Pipe](#)

[NU_Broadcast_To_Pipe](#)

NU_Send_To_Front_Of_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service places a message at the front of the specified pipe. If there is enough space in the pipe to hold the message, this service is processed immediately. Pipe messages are comprised of a fixed or variable number of bytes, depending on the type of messages supported by the pipe.

Usage

```
STATUS NU_Send_To_Front_Of_Pipe (NU_PIPE    *pipe,  
                                VOID        *message,  
                                UNSIGNED    size,  
                                UNSIGNED    suspend);
```

Arguments

- pipe
Pointer to the pipe.
- message
Pointer to the message to send.
- size
Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be exactly the same as the message size supported by the pipe.
- suspend
Specifies whether to suspend the calling task if the pipe is full.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be sent.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_PIPE**
The pipe pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The message size is incompatible with the message size supported by the pipe.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_PIPE_FULL**
The pipe is full.
- **NU_TIMEOUT**
The pipe is still full even after suspending for the specified timeout value.
- **NU_PIPE_DELETED**
Th pipe was deleted while the task was suspended.
- **NU_PIPE_RESET**
The pipe was reset while the task was suspended.

Example

```

NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
STATUS      status;
.
.
.
/* Build a 4-byte message to send.  The contents of "message" have no
   significance. */
message[0]   = 0x01;
message[1]   = 0x02;
message[2]   = 0x03;
message[3]   = 0x04;

/* Send a 4-byte, fixed size message to the pipe control block "Pipe".
   Do not suspend even if the pipe is full. Assume "Pipe" has previously
   been created with the Nucleus PLUS NU_Create_Pipe service call. */
status = NU_Send_To_Front_Of_Pipe(&Pipe, &message[0],
                                  4, NU_NO_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" was sent to "Pipe". */

```

Related Topics

[Pipe Services Function Reference](#)

[NU_Pipe_Information](#)

[NU_Broadcast_To_Pipe](#)

[NU_Receive_From_Pipe](#)

[NU_Send_To_Pipe](#)

NU_Send_To_Pipe

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Communication Services

Tasking Changes: Yes

This service places a message at the back of the specified pipe. If there is enough space in the pipe to hold the message, this service is processed immediately. Pipe messages are comprised of a fixed or variable number of bytes, depending on the type of messages supported by the pipe.

Usage

```
STATUS NU_Send_To_Pipe (NU_PIPE  *pipe,  
                        VOID      *message,  
                        UNSIGNED  size,  
                        UNSIGNED  suspend);
```

Arguments

- pipe
Pointer to the pipe.
- message
Pointer to the message to send.
- size
Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be the same as the message size supported by the pipe.
- suspend
Specifies whether to suspend the calling task if the pipe is full.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the message can be sent.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_PIPE**
The pipe pointer is invalid.
- **NU_INVALID_POINTER**
The message pointer is NULL.
- **NU_INVALID_SIZE**
The message size is incompatible with the message size supported by the pipe.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_PIPE_FULL**
The pipe is full.
- **NU_TIMEOUT**
The pipe is still full even after suspending for the specified timeout value.
- **NU_PIPE_DELETED**
The pipe was deleted while the task was suspended.
- **NU_PIPE_RESET**
The pipe was reset while the task was suspended.

Example

```
NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
STATUS       status;
.
.
.
/* Build a 4-byte message to send. The contents of "message" have no
   significance. */
message[0]   = 0x01;
message[1]   = 0x02;
message[2]   = 0x03;
message[3]   = 0x04;

/* Send a 4-byte message to the pipe control block "Pipe". Do not suspend
   even if the pipe is full. Assume "Pipe" has previously been created
   with the Nucleus PLUS NU_Create_Pipe service call. */
status = NU_Send_To_Pipe(&Pipe, &message[0], 4, NU_NO_SUSPEND);

/* At this point, status indicates whether the service request was
   successful. If successful, "message" was sent to "Pipe". */
```

Related Topics

[Pipe Services Function Reference](#)[NU_Pipe_Information](#)[NU_Receive_From_Pipe](#)[NU_Send_To_Front_Of_Pipe](#)[NU_Broadcast_To_Pipe](#)

Pipe Example Source Code

In previous sections we looked at examples that demonstrated how to communicate between tasks with mailboxes and queues. In this section we will look at a very similar example, but using pipes to communicate between several tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Four Nucleus PLUS structures are used in this example. Three NU_TASK structures are used, one for each task in the system. The NU_PIPE structure is for the pipe that will be used to communicate messages between the three tasks in the system. Dynamic memory will be used for allocation for the pipe data area and a stack for each of the three tasks.

```
NU_TASK task_recv_1;  
NU_TASK task_recv_2;  
NU_TASK task_send;  
NU_PIPE pipe_comm;
```

The three void pointers stack_recv_1, stack_recv_2, and stack_send will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the tasks stacks will never be deallocated.

```
VOID *stack_recv_1;  
VOID *stack_recv_2;  
VOID *stack_send;
```

Similar to these three void pointers, the data_pipe pointer will be used to hold a pointer to the data area for the pipe. It can either be used to deallocate the associated memory, or discarded if memory deallocation is not necessary.

```
VOID *data_pipe;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.

```
VOID entry_recv_1(UNSIGNED argc, VOID *argv);  
VOID entry_recv_2(UNSIGNED argc, VOID *argv);
```

```
VOID entry_send(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to set up memory for three task stacks, and the pipe data area. Therefore, in Application_Initialize there are four separate calls to NU_Allocate_Memory. Application_Initialize is also used to create the pipe and associate the allocated memory for its pipe data area.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the NU_Allocate_Memory call, you are allocating a 1024 byte block of memory out of the mem_pool dynamic memory pool. A pointer to the newly allocated memory is assigned to the stack_rcv_1, stack_rcv_2, and stack_rcv_3 respectively. the pointer to this memory allocation is passed to the NU_Create_Task call, which will use this memory as the task stack.

For this demonstration, note that task_rcv_1 and task_rcv_2 are given a higher priority (priority level of 7) than task_send. By doing this, you are ensuring that task_rcv_1 and task_rcv_2 will always run before task_send. The task_send will only run when both task_rcv_1 and task_rcv_2 are suspended.

```
NU_Allocate_Memory(mem_pool, &stack_rcv_1, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_rcv_1, "rcv_1", entry_rcv_1, 0, NU_NULL,  
              stack_rcv_1, 1024, 7, 0, NU_PREEMPT, NU_START);  
  
NU_Allocate_Memory(mem_pool, &stack_rcv_2, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_rcv_2, "rcv_2", entry_rcv_2, 0, NU_NULL,  
              stack_rcv_2, 1024, 7, 0, NU_PREEMPT, NU_START);  
  
NU_Allocate_Memory(mem_pool, &stack_send, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,  
              stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

Allocate memory for the pipe data area with a call to NU_Allocate_Memory. This call allocates 32768 bytes out of the mem_pool dynamic memory pool, and assigns a pointer to this memory to the data_pipe void pointer. Then call NU_Create_Pipe to associate this memory to the pipe_comm pipe. The pipe_comm pipe is a pipe with fixed sized messages (NU_FIXED_SIZE), and each message will be 8 bits in size. The pipe is associated with the name "comm" and tasks that choose to suspend on this pipe will resumed in FIFO order.

```
NU_Allocate_Memory(mem_pool, &data_pipe, 32768, NU_NO_SUSPEND);  
NU_Create_Pipe(&pipe_comm, "comm", data_pipe, 32768, NU_FIXED_SIZE,  
              1, NU_FIFO);  
}
```

The entry_rcv_1 and entry_rcv_2 functions serve as the entry point for the task_rcv_1 and task_rcv_2 tasks respectively. The tasks will continuously loop, issuing an NU_Receive_From_Pipe call for each iteration of the loop. The NU_Receive_From_Pipe will suspend until there is a message placed into the pipe (as indicated by NU_SUSPEND). Whenever a message is received, NU_Receive_From_Pipe will exit with a Return Values of

NU_SUCCESS. After the call has returned, recvmsg will contain the message received. Therefore, there are two tasks that are continuously suspending on the same pipe, both waiting for a message to be placed into the pipe.

The Nucleus PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `pipe_comm` pipe was created.

```
VOID entry_rcv_1(UNSIGNED argc, VOID *argv)
{
    CHAR rcvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Pipe(&pipe_comm, &rcvmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}

VOID entry_rcv_2(UNSIGNED argc, VOID *argv)
{
    CHAR rcvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Pipe(&pipe_comm, &rcvmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}
```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_rcv_1` and `task_rcv_2` tasks are of a higher priority, and will always be given first chance to run. Because of this, whenever `task_send` sends a message with `pipe_comm`, either `task_rcv_1` or `task_rcv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Pipe` which will send a single message with the `pipe_comm` pipe. The second service call that is issued is `NU_Broadcast_To_Pipe`, which will send the message to every task that is currently suspended on this pipe. Note that in this example, whenever this task is running, there will always be two tasks (`task_rcv_1` and `task_rcv_2`) suspended on the `pipe_comm` pipe. The result is that the message that is sent with `NU_Send_To_Pipe` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Pipe` will be received by both suspended tasks.

```
VOID entry_send(UNSIGNED argc, VOID *argv)
```

```
{  
    UNSIGNED sendmsg;  
  
    while(1)  
    {
```

Assign decimal 1 to sendmsg, then issue NU_Send_To_Pipe on the pipe_comm pipe. Since two tasks will always be suspended on this pipe, and the pipe was created with the NU_FIFO suspension flag, the first task that suspended on the pipe will always receive this message.

```
        sendmsg=1;  
        if (NU_Send_To_Pipe(&pipe_comm, &sendmsg, 1, NU_SUSPEND)  
            == NU_SUCCESS)  
        {  
        }  
        else  
        {  
        }  
    }
```

Assign decimal 2 to sendmsg, then issue NU_Broadcast_To_Pipe on the pipe_comm pipe. Because the priority of task_recv_1 and task_recv_2 are of a higher priority than this task, you are guaranteed that two tasks will always be suspended on this pipe. Therefore, the result of the NU_Broadcast_To_Pipe service is that both tasks will be sent the message.

```
        sendmsg=2;  
        if (NU_Broadcast_To_Pipe(&pipe_comm, &sendmsg, 1, NU_SUSPEND)  
            == NU_SUCCESS)  
        {  
        }  
        else  
        {  
        }  
    }  
}
```

Semaphores

Semaphores synchronize tasks and provide a mechanism to control execution of critical sections of an application. Nucleus PLUS provides counting semaphores that range in value from 0 to 4,294,967,294. The two basic operations on a semaphore are obtain and release. Obtain-semaphore requests decrement the semaphore, while release-semaphore requests increment the semaphore.

Resource allocation is the most common application of a semaphore. Additionally, semaphores created with an initial value can be used to indicate an event.

In addition to counting semaphores, Nucleus also provides Semaphores with Priority Inheritance features or Mutexes. These behave in a binary-semaphore like manner with possible counts of 0 or 1 only.

Mutexes use the same set of semaphore APIs extended to support Priority Inheritance and can be used to prevent priority inversion and avoid deadlock situations.

Semaphore Suspension

The obtain-semaphore service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to obtain a semaphore whose count is currently zero can suspend. Resumption of the task is possible when a release-semaphore request is made.

Multiple tasks may suspend trying to obtain a single semaphore. Tasks are suspended in either FIFO or priority order, depending on how the semaphore was created. If the semaphore supports FIFO suspension, tasks are resumed in the order in which they tried to obtain the semaphore. Otherwise, if the semaphore supports priority suspension, tasks are resumed from high priority to low priority.

Deadlock

A deadlock refers to a situation where two or more tasks are forever suspended attempting to obtain two or more semaphores. The simplest example of this situation is a system with two tasks and two semaphores. Suppose the first task has the second semaphore and the second task has the first semaphore. Now, suppose that the second task attempts to obtain the second semaphore and the first task attempts to obtain the first semaphore. Since each task has the semaphore that the other needs, the tasks could suspend on the semaphores forever.

Prevention is the preferred way to deal with deadlocks. This technique imposes rules on how semaphores are used by the application. For example, if tasks are not allowed to possess more than one semaphore at a time, deadlocks are prevented. Alternatively, deadlocks may be prevented if tasks obtain multiple semaphores in the same order.

Semaphores that support Priority Inheritance features/Mutexes can be used to prevent deadlock situations. The optional timeout on obtain-semaphore suspension can also be used to recover from a deadlock situation.

Priority Inversion

Priority inversion occurs when a higher priority task is suspended on a semaphore that a lower priority task has. This situation is unavoidable if different priority tasks share the same protected resources. In such situations, a limited and predictable amount of time in priority inversion is acceptable.

However, if the low priority task is preempted by a middle priority task during a priority inversion situation, the amount of time in priority inversion is no longer deterministic. Such a

situation can be avoided by insuring that all tasks using the same counting semaphore have the same priority, at least while they own the semaphore.

Alternatively, Priority Inheritance semaphores/Mutexes can be used to prevent priority inversion situations. These allow a lower priority task to inherit the priority of a higher priority task blocked on a priority inheritance mutex owned by the low priority task.

Semaphore Dynamic Creation

Nucleus PLUS semaphores are created and deleted dynamically. There is no preset limit on the number of semaphores an application may have. Each semaphore requires a control block. The memory for the control block is supplied by the application. Counting semaphores can be created with any initial count. In contrast, Priority Inheritance semaphores must be created with an initial count of 1.

Semaphore Determinism

Processing time required for obtaining and releasing semaphores is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the semaphore.

Semaphore Information

Application tasks can obtain a list of active semaphores. Detailed information about each semaphore is also available. This information includes the semaphore name, current count, suspension type, number of tasks waiting, and the first task waiting.

In the case of Priority Inheritance semaphores/mutexes, information about the owner can be obtained by using the **NU_Get_Semaphore_Owner** API.

Semaphore Services Function Reference

The following function reference contains all functions related to Nucleus PLUS semaphores. The following functions are contained in this reference:

- [NU_Create_Semaphore](#)
- [NU_Delete_Semaphore](#)
- [NU_Established_Semaphores](#)
- [NU_Obtain_Semaphore](#)
- [NU_Release_Semaphore](#)
- [NU_Reset_Semaphore](#)

- [NU_Semaphore_Information](#)
- [NU_Get_Semaphore_Owner](#)
- [NU_Semaphore_Pointers](#)

NU_Create_Semaphore

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service creates a counting or Priority Inheritance semaphore. Semaphore values can range from zero through 4,294,967,294 for counting semaphores. Priority Inheritance semaphores must be created with an initial count of 1.

Usage

```
STATUS NU_Create_Semaphore (NU_SEMAPHORE *semaphore,  
                           CHAR           *name,  
                           UNSIGNED      initial_count,  
                           OPTION        suspend_type);
```

Arguments

- semaphore
Pointer to the user-supplied semaphore control block.

Note



Subsequent requests made to the semaphore require this pointer.

- name
Pointer to a seven-character name for the semaphore. The name must be null-terminated.
- initial_count
Specifies the initial count of the semaphore.
Priority Inheritance semaphores must be created with an initial count of 1.
- suspend_type
Specifies how tasks suspend on the semaphore. Valid options for this parameter for counting semaphores are NU_FIFO and NU_PRIORITY, that represent FIFO and priority-order task suspension, respectively.

Priority Inheritance semaphores or mutexes must be created with a suspend type of NU_PRIORITY_INHERIT. Tasks suspending on Priority Inheritance semaphores will suspend and resume in priority order.

Return Values

- NU_SUCCESS
Function completed successfully.

- **NU_INVALID_SEMAPHORE**
The semaphore control block pointer is NULL or is already in use.
- **NU_INVALID_SUSPEND**
The suspend_type parameter is invalid.
- **NU_INVALID_COUNT**
A Priority Inheritance Semaphore is created with an initial count not equal to 1.

Example

```
/* Assume semaphore control block "Semaphore" is defined as global data
   structure. This is one of several ways to allocate a control block. */

NU_SEMAPHORE Semaphore;
.
.
/* Assume status is defined locally.  */

STATUS  status; /* Semaphore creation status  */

/* Create a semaphore with an initial count of 1 and priority order task
   suspension.  */

status =  NU_Create_Semaphore(&Semaphore, "any name", 1, NU_PRIORITY);

/* status indicates if the service was successful.  */
```

Related Topics

Semaphore Services Function Reference	NU_Established_Semaphores
NU_Semaphore_Information	NU_Semaphore_Pointers
Kernel Demo	NU_Delete_Semaphore

NU_Delete_Semaphore

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service deletes a previously created semaphore. The parameter semaphore identifies the semaphore to delete. Tasks suspended on this semaphore are resumed with the appropriate error status. A Priority Inheritance semaphore can only be deleted if nobody owns the semaphore or if it is being deleted by the task owning the semaphore. The application must prevent the use of this semaphore during and after deletion.

Usage

```
STATUS NU_Delete_Semaphore (NU_SEMAPHORE *semaphore);
```

Arguments

- semaphore
Pointer to the user-supplied semaphore control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid.
- NU_SEMAPHORE_INVALID_OWNER
An owned priority inheritance semaphore is being deleted by a task other than its owner.

Example

```
NU_SEMAPHORE Semaphore;  
STATUS      status;  
.  
.  
/* Delete the semaphore control block "Semaphore". Assume "Semaphore" has  
   previously been created with the Nucleus PLUS NU_Create_Semaphore  
   service call. */  
status = NU_Delete_Semaphore(&Semaphore);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Semaphore_Information](#)

[NU_Create_Semaphore](#)

[NU_Established_Semaphores](#)

[NU_Semaphore_Pointers](#)

NU_Established_Semaphores

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns the number of established semaphores. All created semaphores are considered established. Deleted semaphores are no longer considered established.

Usage

```
UNSIGNED NU_Established_Semaphores (VOID);
```

Return Values

- This service call returns the number of created pipes in the system.

Example

```
UNSIGNED total_semaphores;  
  
/* Obtain the total number of semaphores. */  
total_semaphores = NU_Established_Semaphores();
```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Delete_Semaphore](#)

[NU_Semaphore_Information](#)

[NU_Semaphore_Pointers](#)

[NU_Create_Semaphore](#)

NU_Obtain_Semaphore

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service obtains an instance of the specified semaphore. Since “instances” are implemented with an internal counter, obtaining a semaphore translates into decrementing the semaphore’s internal counter by one. If the semaphore counter is zero before this call, the service cannot be immediately satisfied. Priority Inheritance semaphores can only be obtained once. Any successive attempts to obtain the semaphore will return an error. A Priority Inheritance semaphore can be obtained only when its count is 1 i.e. when it is not owned.

Usage

```
STATUS NU_Obtain_Semaphore (NU_SEMAPHORE *semaphore,  
                           UNSIGNED      suspend);
```

Arguments

- semaphore
Pointer to the user-supplied semaphore control block.
- suspend
Specifies whether or not to suspend the calling task if the semaphore cannot be obtained (is currently zero).

The following is a summary of the possible values for the suspend parameter.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the semaphore is released.

timeout value

(1 – 4,294,967,293). The calling task is suspended until the semaphore is obtained, or until the specified number of ticks has expired.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid.

- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_SEMAPHORE_DELETED**
The semaphore was deleted while the task was suspended.
- **NU_SEMAPHORE_RESET**
The semaphore was reset while the task was suspended.
- **NU_TIMEOUT**
The semaphore is still unavailable even after suspending for the specified timeout value.
- **NU_UNAVAILABLE**
The semaphore is unavailable.
- **NU_SEMAPHORE_ALREADY_OWNED**
A priority inheritance semaphore is trying to be obtained by a task that already owns this semaphore.

Example

```
NU_SEMAPHORE Semaphore;  
STATUS          status;  
.  
.  
.  
/* Obtain an instance of the semaphore control block "Semaphore". If the  
   semaphore is unavailable, suspend for a maximum of 20 timer ticks.  
   Note: the order of multiple tasks suspending on the same semaphore is  
   determined when the semaphore is created. Assume "Semaphore" has  
   previously been created with the Nucleus PLUS NU_Create_Semaphore  
   service call. */  
status = NU_Obtain_Semaphore(&Semaphore, 20);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Semaphore_Information](#)

[Kernel Demo](#)

[NU_Release_Semaphore](#)

NU_Release_Semaphore

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service releases an instance of the semaphore specified by the parameter semaphore. If there are any tasks waiting to obtain the same semaphore, the first task waiting is given this instance of the semaphore. Otherwise, if there are no tasks waiting for this semaphore, the internal semaphore counter is incremented by one.

Priority Inheritance semaphores can only be released once, by the task owning the semaphore. Any successive attempts to release the semaphore will return an error.

Usage

```
STATUS NU_Release_Semaphore (NU_SEMAPHORE *semaphore);
```

Arguments

- semaphore
Pointer to the user-supplied semaphore control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid.
- NU_SEMAPHORE_COUNT_ROLLOVER
The semaphore count is about to rollover.
- NU_SEMAPHORE_INVALID_OWNER
The semaphore is already released or that the semaphore is being released from a task other than its owner.

Example

```
NU_SEMAPHORE Semaphore;  
STATUS      status;  
.  
.  
.  
/* Release an instance of the semaphore control block "Semaphore". If  
   other tasks are waiting to obtain the same semaphore, this service  
   results in a transfer of this instance of the semaphore to the first  
   task waiting. Assume "Semaphore" has previously been created with the  
   Nucleus PLUS NU_Create_Semaphore service call. */
```

```
status = NU_Release_Semaphore(&Semaphore);
```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Semaphore_Information](#)

[Kernel Demo](#)

[NU_Obtain_Semaphore](#)

NU_Reset_Semaphore

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service resets the semaphore specified by semaphore to the value of initial_count. All tasks suspended on the semaphore are resumed with the appropriate reset status. A Priority Inheritance semaphore must be reset with its initial count value equal to 1.

Usage

```
STATUS NU_Reset_Semaphore (NU_SEMAPHORE *semaphore,  
                           UNSIGNED      initial_count);
```

Arguments

- semaphore
Pointer to the user-supplied semaphore control block.
- initial_count
Specifies the initial count of the semaphore.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid.
- NU_INVALID_COUNT
The Priority Inheritance semaphore has been reset with a count not equal to 1.

Example

```
NU_SEMAPHORE Semaphore;  
STATUS      status  
.  
.  
.  
/* Reset the semaphore control block "Semaphore". The initial count is set  
   to 1. Assume "Semaphore" has previously been created with the Nucleus  
   PLUS NU_Create_Semaphore service call. */  
status = NU_Reset_Semaphore(&Semaphore, 1);
```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Release_Semaphore](#)

[NU_Semaphore_Information](#)

[NU_Obtain_Semaphore](#)

NU_Semaphore_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns various information about the specified task synchronization semaphore. The parameters of this service are further defined as follows:

Usage

```
STATUS NU_Semaphore_Information (NU_SEMAPHORE *semaphore,  
                                CHAR *name,  
                                UNSIGNED *current_count,  
                                OPTION *suspend_type,  
                                UNSIGNED *tasks_waiting,  
                                NU_TASK **first_task);
```

Arguments

- semaphore
Pointer to the synchronization semaphore.
- name
Pointer to an eight-character destination area for the semaphore's name. This includes space for the null terminator.
- current_count
Pointer to a variable to hold the current instance count of the semaphore.
- suspend_type
Pointer to a variable that holds the task's suspend type. Valid task suspend types are NU_FIFO, NU_PRIORITY, and NU_PRIORITY_INHERIT.
- tasks_waiting
Pointer to a variable to hold the number of tasks waiting on the semaphore.
- first_task
Pointer to a task pointer. The pointer of the first suspended task is placed in the task pointer.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid.

Example

```

NU_SEMAPHORE Semaphore;
CHAR          semaphore_name[8];
UNSIGNED      current_count;
OPTION        suspend_type;
UNSIGNED      tasks_suspended;
NU_TASK       *first_task;
STATUS        status;
.
.
.
/* Obtain information about the semaphore control block "Semaphore".
   Assume "Semaphore" has previously been created with the Nucleus PLUS
   NU_Create_Semaphore service call. */
status = NU_Semaphore_Information(&Semaphore, semaphore_name,
                                   &current_count, &suspend_type,
                                   &tasks_suspended, &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

Related Topics

[Semaphore Services Function Reference](#)

[NU_Delete_Semaphore](#)

[NU_Established_Semaphores](#)

[NU_Semaphore_Pointers](#)

[NU_Create_Semaphore](#)

NU_Get_Semaphore_Owner

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns a pointer to the task pointer of the task owning the Priority Inheritance semaphore/Mutex. This API will return an error if the semaphore is not a priority inheritance type.

The parameters of this service are further defined as follows:

Usage

```
STATUS NU_Get_Semaphore_Owner (NU_SEMAPHORE *semaphore_ptr,  
                               NU_TASK      **task);
```

Arguments

- semaphore_ptr
Pointer to the Priority Inheritance semaphore.
- task
Pointer to task pointer owning the semaphore.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_SEMAPHORE
The semaphore pointer is invalid or is not a priority inheritance type semaphore.

Example

```
NU_SEMAPHORE Semaphore;  
NU_TASK *owner_task;  
STATUS status;  
.  
/* Obtain owner task information about the semaphore control block */  
status = NU_Get_Semaphore_Owner(&Semaphore, &owner_task);  
/* If status is NU_SUCCESS, the other information is accurate. */  
/* If Priority Inheritance semaphore is not owned, owner_task will be  
NU_NULL. */
```


Related Topics

[Semaphore Services Function Reference](#)

[NU_Established_Semaphores](#)

[NU_Create_Semaphore](#)

[NU_Delete_Semaphore](#)

[NU_Semaphore_Information](#)

NU_Semaphore_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service builds a sequential list of pointers to all established semaphores in the system.



Note

Semaphores that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Semaphore_Pointers (NU_SEMAPHORE **pointer_list,  
                                UNSIGNED      maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_SEMAPHORE` pointers. This array will be filled with pointers of established semaphores in the system.
- `maximum_pointers`
The maximum number of `NU_SEMAPHORE` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created semaphores in the system.

Example

```
/* Define an array capable of holding 20 semaphore pointers. */  
NU_SEMAPHORE *Pointer_Array[20];  
UNSIGNED      number;  
  
/* Obtain a list of currently active semaphore pointers (Maximum of 20).  
*/  
number = NU_Semaphore_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, number contains the actual number of pointers in the  
list. */
```

Related Topics

[Semaphore Services Function Reference](#)[NU_Delete_Semaphore](#)[NU_Established_Semaphores](#)[NU_Semaphore_Information](#)[NU_Create_Semaphore](#)

Semaphore Example Source Code

The following source code demonstrates how to use the basic semaphore function calls. Semaphores are generally used to control access to either a mutually exclusive device, or to a piece of mutually exclusive data, such as a global variable. This example demonstrates both of these uses. The function `init_device()` demonstrates how a semaphore can be used to protect a global variable against being modified by multiple tasks simultaneously. To demonstrate using a semaphore to protect a mutually exclusive device, the function `write_to_device` uses the same semaphore as it's device protection mechanism.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Define a structure typedef called `BUFFER`, and declare an instance of this structure called `my_device`. This example will then protect this global variable by obtaining a semaphore before every modification.

```
typedef struct BUFFER_STRUCT
{
    CHAR buf[128];
    UNSIGNED read;
    UNSIGNED write;
    UNSIGNED num_entries;
} BUFFER;

BUFFER my_device;
```

This program will use three PLUS structures. The variables `task_1` and `task_2` are both task control blocks (`NU_TASK`)_ which will be used to write to the mutually exclusive device. The semaphore control block, `semaphore_device` will be used to control access to the device buffer.

```
NU_TASK task_1;
NU_TASK task_2;
NU_SEMAPHORE semaphore_device;
```

The two void pointers `stack_1`, `stack_2`, will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the tasks stacks will never be deallocated.

```
VOID *stack_1;
VOID *stack_2;
```

Declare the task entry point function for each of the tasks. These will later be passed as a parameter to the `NU_Create_Task` call which will associate these functions with each of their respective tasks.

```
VOID entry_1(UNSIGNED argc, VOID *argv);  
VOID entry_2(UNSIGNED argc, VOID *argv);
```

Two other functions will be used in this demonstration: `init_devices`, and `write_to_device`. The function `init_devices` will be used to initialize the global variable, and will be protected with the previously declared `semaphore_device`. The function `write_to_device` will use this same semaphore to protect the mutually exclusive device.

```
VOID init_device();  
VOID write_to_device(CHAR writechar);
```

`Application_Initialize` is used to create any PLUS structures, allocate any required memory, and to perform any other system initialization that is necessary. Specific to this example, `Application_Initialize` will be used to set up memory for two task stacks: `task_1`, and `task_2`, and also create `semaphore_device`. Lastly, a call to the function `init_devices` is made to initialize the global structure `my_device`.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `mem_pool` dynamic memory pool, which is passed in to `Application_Initialize`. A pointer to the newly allocated memory is assigned to `stack_1`, and `stack_2` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

```
NU_Allocate_Memory(mem_pool, &stack_1, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_1, "TASK1", entry_1, 0, NU_NULL, stack_1, 1024, 10,  
2,  
                NU_PREEMPT, NU_START);  
  
NU_Allocate_Memory(mem_pool, &stack_2, 1024, NU_NO_SUSPEND);  
NU_Create_Task(&task_2, "TASK2", entry_2, 0, NU_NULL, stack_2, 1024, 10,  
2,  
                NU_PREEMPT, NU_START);
```

Create the semaphore that will be used to protect the mutually exclusive structure. The `semaphore_device` semaphore is named "DEVICE", is created with an initial count of 1, and tasks that choose to suspended on this semaphore will be resumed in FIFO order. Nucleus PLUS semaphores can be counting semaphores if the semaphore is created with a count higher than 1. In such a case, the semaphore can be obtained up to the number of times specified.

```
NU_Create_Semaphore(&semaphore_device, "DEVICE", 1, NU_FIFO);
```

Make the function call to `init_devices`. This function will use the above created semaphore to protect the global structure `my_device`.

```
init_device();  
}
```

Both tasks in the system (task_1 and task_2) continuously loop, making a call to write_to_device for each iteration of the loop. In the case of task_1 (which is associated to the entry point entry_1) the task writes a single character, "1", to the device. Accordingly, task_2 (associated to the entry point entry_2) will write a "2" to the device for each iteration of the loop.

```
VOID entry_1(UNSIGNED argc, VOID *argv)
{
    while(1)
    {
        write_to_device('1');
    }
}

VOID entry_2(UNSIGNED argc, VOID *argv)
{
    while(1)
    {
        write_to_device('2');
    }
}
```

The function init_device is used to simulate initializing a device. If using real hardware, this function may set up control registers, clear out data buffers, or any other device dependent initialization. In this example however, we will use a global structure, my_device, to simulate the device. Since this device is mutually exclusive it is protected by using the semaphore_device semaphore. Note that this protection is only necessary if multiple threads of execution could be initializing the device simultaneously.

```
VOID init_device()
{
```

Obtain the semaphore, semaphore_device. Since this semaphore was created with a count of 1, only one thread of execution can have possession of the semaphore at any given time. Therefore, we are guaranteed that only one task at a time can be modifying the my_device structure.

```
    NU_Obtain_Semaphore(&semaphore_device, NU_SUSPEND);
```

Modify the global variable. In the case of real hardware, the following code could be replaced with control register initialization, clearing buffers, or any other device dependent initialization that may be required.

```
    my_device.read = 0;
    my_device.write = 0;
    my_device.num_entries = 0;
```

When finished modifying the mutually exclusive data, release the semaphore so that other threads of execution can then modify the structure.

```
    NU_Release_Semaphore(&semaphore_device);
}
```

Similar to the `init_device` function, the following function, `write_to_device` will use the `semaphore_device` semaphore to protect the mutually exclusive device. In this example, both `task_1`, and `task_2` (see their respective task entry points, `entry_1` and `entry_2`) are using this function to write to the device. Since the semaphore, `semaphore_device` was created as a binary semaphore (count 1), only one of these tasks can be modifying the device at any given time.

```
VOID write_to_device(CHAR writechar)
{
```

Make a call to `NU_Obtain_Semaphore` to obtain the semaphore. If a task already has possession of the semaphore, then the task making the second request will be suspended because suspension was requested by specifying the `NU_SUSPEND` option.

```
    NU_Obtain_Semaphore(&semaphore_device, NU_SUSPEND);
```

Make any necessary modifications to the buffer. If actual hardware were being used, a transmit finished interrupt could be used to read data out of this buffer and place it onto the device. Alternately, one could choose not to use a buffer, and the following code could be replaced with code to place the data onto the physical device.

```
    my_device.buf[my_device.write] = writechar;
    my_device.write++;

    if (my_device.write >= 128)
    {
        my_device.write = 0;
    }

    if (my_device.num_entries < 128)
    {
        my_device.num_entries++;
    }
    else
    {
        my_device.read = my_device.write;
    }
}
```

Release the `semaphore_device` semaphore so that other tasks can modify the device.

```
    NU_Release_Semaphore(&semaphore_device);
}
```

Event Groups

Event groups, much like semaphores, synchronize tasks and provide 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.

Event flags can be set and cleared using logical AND/OR combinations. Event flags can be received in logical AND/OR combinations as well. Additionally, event flags may be reset automatically after they are received.

Event Group Suspension

The receive event flag requests provide options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to receive a combination of event flags that are not present can suspend. Resumption of the task occurs when a set-event-flags operation satisfies the combination of events requested by the task.

Multiple tasks may suspend trying to receive different combinations of event flags from the same event group. All tasks suspended on an event group are checked for resumption when a set-event-flags operation is performed on the event group.

Event Group Dynamic Creation

Nucleus PLUS event groups are created and deleted dynamically. There is no preset limit on the number of event groups an application may have. Each event group requires a control block. The memory for the control block is supplied by the application.

Event Group Determinism

Processing time required for receiving event flags from an event group is constant. However, the processing time required to set event flags in an event group is affected by the number of tasks suspended on the event group.

Event Group Information

Application tasks may obtain a list of active event groups. Detailed information about each event group is also available. This information includes the event group name, current event flags, number of tasks waiting, and the first task waiting.

Event Group Services Function Reference

The following function reference contains all functions related to Nucleus PLUS event groups. The following functions are contained in this reference:

- [NU_Create_Event_Group](#)
- [NU_Delete_Event_Group](#)
- [NU_Established_Event_Groups](#)

- [NU_Event_Group_Information](#)
- [NU_Event_Group_Pointers](#)
- [NU_Retrieve_Events](#)
- [NU_Set_Events](#)

NU_Create_Event_Group

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service creates an event flag group. Each event flag group contains 32 event flags. All event flags are initially set to 0.

Usage

```
STATUS NU_Create_Event_Group (NU_EVENT_GROUP *group,  
                             CHAR             *name);
```

Arguments

- **group**
Pointer to the user-supplied event flag group control block.

Note



All subsequent requests made to the event group require this pointer.

- **name**
Pointer to a seven-character name for the event flag group. The name must be null-terminated.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_GROUP**
The event group control block pointer is NULL or is already in use.

Example

```
/* Assume event group control block "Events" is defined as a global data  
   structure. This is one of several ways to allocate a control block. */  
  
NU_EVENT_GROUP  Events;  
.  
.  
/* Assume status is defined locally. */  
STATUS status; /* Event group creation status */  
  
/* Create an event flag group. */  
status = NU_Create_Event_Group(&Events, "any name");  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[Event Group Services Function Reference](#)

[NU_Event_Group_Information](#)

[Kernel Demo](#)

[NU_Established_Event_Groups](#)

[NU_Event_Group_Pointers](#)

[NU_Delete_Event_Group](#)

NU_Delete_Event_Group

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service deletes a previously created event flag group. The parameter group identifies the event flag group to delete. Tasks suspended on this event group are resumed with the appropriate error status. The application must prevent the use of this event group during and after deletion.

Usage

```
STATUS NU_Delete_Event_Group (NU_EVENT_GROUP *group);
```

Arguments

- group
Pointer to the user-supplied event flag group control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_GROUP
The event flag group pointer is invalid.

Example

```
NU_EVENT_GROUP Group;  
STATUS          status  
.  
.  
/* Delete the event flag group control block "Group". Assume "Group" has  
   previously been created with the Nucleus PLUS NU_Create_Event_Group  
   service call. */  
status = NU_Delete_Event_Group(&Group);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Event Group Services Function Reference](#)

[NU_Established_Event_Groups](#)

[NU_Event_Group_Information](#)

[NU_Event_Group_Information](#)

[NU_Event_Group_Pointers](#)

[NU_Create_Event_Group](#)

NU_Established_Event_Groups

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns the number of established event-flag groups. All created event-flag groups are considered established. Deleted event-flag groups are no longer considered established.

Usage

```
UNSIGNED NU_Established_Event_Groups (VOID);
```

Return Values

- This service call returns the number of created event groups in the system.

Example

```
UNSIGNED total_event_groups;

/* Obtain the total number of event flag groups. */
total_event_groups = NU_Established_Event_Groups( );
```

Related Topics

[Event Group Services Function Reference](#)

[NU_Delete_Event_Group](#)

[NU_Event_Group_Information](#)

[NU_Event_Group_Pointers](#)

[NU_Create_Event_Group](#)

NU_Event_Group_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns various information about the specified event flag group.

Usage

```
STATUS NU_Event_Group_Information (NU_EVENT_GROUP *group,  
                                  CHAR             *name,  
                                  UNSIGNED          *event_flags,  
                                  UNSIGNED          *tasks_waiting,  
                                  NU_TASK           **first_task);
```

Arguments

- **group**
Pointer to the user-supplied event flag group control block.
- **name**
Pointer to an eight-character destination area for the event flag group's name. This includes space for the null terminator.
- **event_flags**
Pointer to a variable to hold the current event flags.
- **tasks_waiting**
Pointer to a variable to hold the number of tasks waiting on the event flag group.
- **first_task**
Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_GROUP**
The event flag group pointer is invalid.

Example

```
NU_EVENT_GROUP Group;  
CHAR            group_name[8];  
UNSIGNED        event_flags;  
UNSIGNED        tasks_suspended;  
NU_TASK         *first_task;  
STATUS          status
```

```
.  
.   
.   
/* Obtain information about the event group control block "Group". Assume  
   "Group" has previously been created with the Nucleus PLUS  
   NU_Create_Event_Group service call.*/  
status =  NU_Event_Group_Information(&Group, group_name,  
                                     &event_flags,  
                                     &tasks_suspended,  
                                     &first_task);  
  
/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

[Event Group Services Function Reference](#)

[NU_Delete_Event_Group](#)

[NU_Established_Event_Groups](#)

[NU_Event_Group_Pointers](#)

[NU_Create_Event_Group](#)

NU_Event_Group_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: No

This service builds a sequential list of pointers to all established event-flag groups in the system.



Note

Event flag-groups that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Event_Group_Pointers (NU_EVENT_GROUP *pointer_list,  
                                UNSIGNED          maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_EVENT_GROUP` pointers. This array will be filled with pointers of established semaphores in the system.
- `maximum_pointers`
The maximum number of `NU_EVENT_GROUP` pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of created event groups in the system.

Example

```
/* Define an array capable of holding 20 event flag group pointers. */  
NU_EVENT_GROUP *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active event flag group pointers (Maximum  
   of 20). */  
number = NU_Event_Group_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, number contains the actual number of pointers in the  
   list. */
```


Related Topics

[Event Group Services Function Reference](#)

[NU_Established_Event_Groups](#)

[NU_Create_Event_Group](#)

[NU_Delete_Event_Group](#)

[NU_Event_Group_Information](#)

NU_Retrieve_Events

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service retrieves the specified event-flag combination from the specified event-flag group. If the combination is present, the service completes immediately.

Usage

```
STATUS NU_Retrieve_Events (NU_EVENT_GROUP *group,  
                           UNSIGNED        requested_events,  
                           OPTION          operation,  
                           UNSIGNED        *retrieved_events,  
                           UNSIGNED        suspend);
```

Arguments

- **group**
Pointer to the user-supplied event flag group control block.
- **requested_events**
Requested event flags. A set bit indicates the corresponding event flag is requested.
- **operation**
There are four operation options available: NU_AND, NU_AND_CONSUME, NU_OR, and NU_OR_CONSUME. NU_AND and NU_AND_CONSUME options indicate that all of the requested event flags are required. NU_OR and NU_OR_CONSUME options indicate that one or more of the requested event flags is sufficient. The CONSUME option automatically clears the event flags present on a successful request.
- **retrieved_events**
Pointer to the event flags actually retrieved.
- **suspend**
Specifies whether to suspend the calling task if the requested event flag combination is not available.

The following is a summary of the possible values for the suspend argument.

NU_NO_SUSPEND

The service returns immediately regardless of whether or not the request can be satisfied. This is the only valid option if the service is called from a non-task thread.

NU_SUSPEND

The calling task is suspended until the event flag combination is available.
timeout value

(1 – 4,294,967,293). The calling task is suspended until the event flag combination is available or until the specified number of ticks has expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_GROUP**
The event flag group pointer is invalid.
- **NU_INVALID_POINTER**
The retrieved event flag pointer is NULL.
- **NU_INVALID_OPERATION**
The operation parameter is invalid.
- **NU_INVALID_SUSPEND**
Indicates that suspend attempted from a non-task thread.
- **NU_NOT_PRESENT**
The requested event flag combination is not currently present.
- **NU_TIMEOUT**
The requested event flag combination is not present even after the specified suspension timeout.
- **NU_GROUP_DELETED**
The event flag group was deleted while the task was suspended.

Example

```

NU_EVENT_GROUP Group;
UNSIGNED      actual_flags;
STATUS        status;
.
.
.
/* Retrieve event flags 7, 2, and 1 from the event group control block
   "Group". Note: all event flags must be present to satisfy the
   request. If they are not, the calling task suspends unconditionally.
   Also, event flags 7, 2, and 1 are consumed when this request is
   satisfied. Assume "Group" has previously been created with the
   Nucleus PLUS NU_Create_Event_Group service call. */

status = NU_Retrieve_Events(&Group, 0x86, NU_AND_CONSUME,
                           &actual_flags, NU_SUSPEND);

```

Related Topics

[Event Group Services Function Reference](#)

[NU_Set_Events](#)

[Kernel Demo](#)

[NU_Event_Group_Information](#)

NU_Set_Events

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service sets the specified event flags in the specified event group. Any task waiting on the event group whose event flag request is satisfied by this service is resumed.

Usage

```
STATUS NU_Set_Events (NU_EVENT_GROUP *group,  
                     UNSIGNED        event_flags,  
                     OPTION          operation);
```

Arguments

- **group**
Pointer to the user-supplied event flag group control block.
- **event_flags**
Event flag values.
- **operation**
There are two operation options available: NU_OR and NU_AND. NU_OR causes the event flags specified to be “OR-ed” with the current event flags in the group. NU_AND causes the event flags specified to be “AND-ed” with the current event flags in the group. Event flags can be cleared with the NU_AND option.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_GROUP**
The event flag group pointer is invalid.
- **NU_INVALID_OPERATION**
The operation parameter is invalid.

Example

```
NU_EVENT_GROUP Group;  
STATUS status;  
.  
.  
.  
/* Set event flags 7, 2, and 1 in the event group control block "Group".  
   Assume "Group" has previously been created with the Nucleus PLUS  
   NU_Create_Event_Group service call.*/
```

```
status = NU_Set_Events(&Group, 0x00000086, NU_OR);

/* If status is NU_SUCCESS the event flags were set. */
```

Related Topics

[Event Group Services Function Reference](#)[NU_Retrieve_Events](#)[Kernel Demo](#)[NU_Event_Group_Information](#)

Event Group Example Source Code

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

There are two possible events, which will be represented by the definitions of EVENT_1 and EVENT_2. The #define of WAIT_EVENTS will be used by the NU_Retrieve_Events function call to suspend on both individual events.

```
#define EVENT_1      0x00000001
#define EVENT_2      0x00000002
#define WAIT_EVENTS  0x00000003
```

Five different Nucleus PLUS structures are included in this example program. All necessary memory will be allocated out of the dynamic memory pool, mem_pool that is passed in to Application_Initialize. There are also three NU_TASK structures which will be used for the three tasks in the system. One task (task_wait) will be of a higher priority, and will suspend on the NU_EVENT_GROUP, eg_wait. The remaining two tasks, task_set1, and task_set2 will set the above defined events, EVENT_1 and EVENT_2. When both of these bits are set task_wait will be resumed.

```
NU_TASK task_wait;
NU_TASK task_set1;
NU_TASK task_set2;
NU_EVENT_GROUP eg_wait;
```

Three void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_wait;
VOID *stack_set1;
VOID *stack_set2;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.

```
VOID wait(UNSIGNED argc, VOID *argv);
```

```
VOID set1(UNSIGNED argc, VOID *argv);  
VOID set2(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to create the dynamic memory pool, out of which memory will be allocated for the three tasks in the system. Application_Initialize will also be used to create the event group, create the three tasks, and associate each of the tasks with a newly created task stack.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the NU_Allocate_Memory call, we are allocating a 1024 byte block of memory out of the mem_pool dynamic memory pool. A pointer to the newly allocated memory is assigned to stack_wait, stack_set1, and stack_set2 respectively. The pointer to this memory allocation is passed to the NU_Create_Task call, which will use this memory as its task stack.

```
    NU_Allocate_Memory(mem_pool, &stack_wait, 1024, NU_NO_SUSPEND);  
    NU_Create_Task(&task_wait, "WAIT", wait, 0, NU_NULL, stack_wait,  
                  1024, 3, 0, NU_PREEMPT, NU_START);  
  
    NU_Allocate_Memory(mem_pool, &stack_set1, 1024, NU_NO_SUSPEND);  
    NU_Create_Task(&task_set1, "SET1", set1, 0, NU_NULL, stack_set1,  
                  1024, 4, 0, NU_PREEMPT, NU_START);  
  
    NU_Allocate_Memory(mem_pool, &stack_set2, 1024, NU_NO_SUSPEND);  
    NU_Create_Task(&task_set2, "SET2", set2, 0, NU_NULL, stack_set2,  
                  1024, 4, 0, NU_PREEMPT, NU_START);
```

Use NU_Create_Event_Group to create an event group with the text name of "WAIT". The tasks task_wait, task_set1, and task_set2 will use this event group to synchronize their activity.

```
    NU_Create_Event_Group(&eg_wait, "WAIT");  
}
```

The function wait is the entry point for the task_wait task. The task_wait task will suspend on the eg_wait event group until both EVENT_1 and EVENT_2 are set by the task_set1 and task_set2 tasks.

```
VOID wait(UNSIGNED argc, VOID *argv)  
{
```

The variable retrieved will be passed as a parameter to NU_Retrieve_Events. Upon successful completion of that service call, it will contain the events that were actually retrieved. The value of this variable can then be used in a construct such as a case statement to perform different actions based upon which signal (represented by distinct bit patterns) was actually sent.

```
    UNSIGNED retrieved;
```

Use the NU_Retrieve_Events service call to suspend until all events are set. Since this is the highest priority task in the system (see the NU_Create_Task service calls in the Application_Initialize function) it will be run first. Therefore, the task_wait task will suspend until the bits specified in WAIT_EVENTS are set.

The `NU_Retrieve_Events` service call will suspend, the result of the `NU_SUSPEND` parameter, on the `eg_wait` event group waiting for all bits in `WAIT_EVENTS` to be set. This behavior could also be modified by changing the `NU_AND` parameter to `NU_OR`, which would cause the `NU_Retrieve_Events` service call to suspend until any of the specified events were set. Consuming, or clearing, of event bits is also available by using the `NU_AND_CONSUME`, and `NU_OR_CONSUME` options.

```
if (NU_Retrieve_Events(&eg_wait, WAIT_EVENTS, NU_AND, &retrieved,
    NU_SUSPEND) == NU_SUCCESS)
{
    /* The requested events were successfully retrieved. */
}
}
```

The `task_set1` and `task_set2` tasks will both set a separate bit in the `eg_wait` event group. When these tasks are run, `task_wait` has already run and has suspended on the `eg_wait` event group. Since these two tasks are the only two remaining tasks in the system, and are of the same priority, they will be run consecutively and will each set their respective bits. After the second `NU_Set_Events` call is executed, `task_wait` will be immediately resumed to continue processing.

```
VOID set1(UNSIGNED argc, VOID *argv)
{
    NU_Set_Events(&eg_wait, EVENT_1, NU_OR);
}

VOID set2(UNSIGNED argc, VOID *argv)
{
    NU_Set_Events(&eg_wait, EVENT_2, NU_OR);
}
```

Signals

Signals are in some ways similar to event flags. However, there are significant differences in operation. Event flag usage is synchronous by nature. The task does not recognize event flags are present until the specific service request is made. Signals operate in an asynchronous manner. When a signal is present, a special signal handling routine, previously designated by the task, is executed when the task is resumed. Each task is capable of handling 32 signals. Each signal is represented by a single bit.

Signal Handling Routine

The task's signal-handling routine must be supplied before any signals are processed. Processing inside a signal-handling routine has virtually the same constraints as a high-level interrupt service routine. Basically, most Nucleus PLUS services are available, provided self-suspension is avoided.

Enable Signal Handling

By default, tasks are created with all signals disabled. Individual signals may be enabled and disabled dynamically by each task.

Clearing Signals

Signals are automatically cleared when signal handling is invoked. Additionally, signals are cleared when a solicited request to receive signals is made.

Note

Tasks cannot suspend on solicited requests to receive signals.

Multiple Signals

Signals for a task are cleared once the signal-handling routine is started. Signal-handling routines are not interrupted by new signals. Processing of any new signals takes place after the current signal-processing completes. Identical signals sent before the first signal is recognized are discarded.

Signal Determinism

Processing time required to send and receive signals is constant, at least in the worst case. Of course, the time required to execute a signal-handling routine is application-specific.

Signal Services Function Reference

The following function reference contains all functions related to Nucleus PLUS signals. The following functions are contained in this reference:

- [NU_Control_Signals](#)
- [NU_Receive_Signals](#)
- [NU_Register_Signal_Handler](#)
- [NU_Send_Signals](#)

NU_Control_Signals

Allowed From: Task

Category: Task Synchronization Services

Tasking Changes: No

This service enables and/or disables signals of the calling task. There are 32 signals available for each task. Each signal is represented by a bit in `signal_enable_mask`. Signal 0 is represented by bit 0 and signal 31 is represented by bit 31. Setting a bit in `signal_enable_mask` enables the corresponding signal, while clearing a bit disables the corresponding signal.

Note



The signal enable mask is cleared during task creation.

Usage

```
UNSIGNED NU_Control_Signals (UNSIGNED enable_signal_mask);
```

Arguments

- `enable_signal_mask`
Bit pattern representing valid signals.

Return Values

- This service returns the previous signal enable/disable mask.

Example

```
UNSIGNED old_signal_mask; /* Previous signal mask */
/* Lockout all of the current task's signals temporarily. */
old_signal_mask = NU_Control_Signals(0);
.
.
.
/* Restore previous signal mask. */
NU_Control_Signals(old_signal_mask);
```

Related Topics

[Signal Services Function Reference](#)

[NU_Register_Signal_Handler](#)

[NU_Send_Signals](#)

[NU_Receive_Signals](#)

NU_Receive_Signals

Allowed From: Task

Category: Task Synchronization Services

Tasking Changes: No

This service returns the current value of each signal associated with the calling task. All signals are automatically cleared as a result of the service call.

Usage

```
UNSIGNED NU_Receive_Signals (VOID);
```

Return Values

- This service call returns the current value of each signal associated with the calling task.

Example

```
UNSIGNED signals;  
/* Receive and clear the signals of the current task. */  
signals = NU_Receive_Signals( );
```

Related Topics

[Signal Services Function Reference](#)

[NU_Register_Signal_Handler](#)

[NU_Send_Signals](#)

[NU_Control_Signals](#)

NU_Register_Signal_Handler

Allowed From: Task

Category: Task Synchronization Services

Tasking Changes: No

This service registers a signal handler pointed to by `signal_handler`, for the calling task. By default, all signals are disabled when the task is created. A signal handler executes on top of the task's context. Most services can be called from a signal handler. However, services called from a signal handler cannot specify suspension.

Usage

```
STATUS NU_Register_Signal_Handler (VOID(*signal_handler) (UNSIGNED));
```

Arguments

- `signal_handler`
Function to call whenever valid signals are received.

Return Values

- `NU_SUCCESS`
Function completed successfully
- `NU_INVALID_TASK`
The supplied task pointer is invalid.
- `NU_INVALID_POINTER`
The signal handler pointer is NULL.

Example

```
STATUS status;

/* Register the function "Signal_Handler" as the task's signal handler. */

VOID Signal_Handler(UNSIGNED signals)
{
    /* Process relative to the signals present. Note that processing has the
       same constraints as HISRs in that self-suspension is not
       permitted. */
}

status = NU_Register_Signal_Handler(Signal_Handler);

/* If status is NU_SUCCESS, Signal_Handler is invoked each time enabled
   signals are sent. */
```

Related Topics

[Signal Services Function Reference](#)

[NU_Send_Signals](#)

[NU_Receive_Signals](#)

[NU_Control_Signals](#)

NU_Send_Signals

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Task Synchronization Services

Tasking Changes: Yes

This service sends the signals indicated by the parameter signals to the task pointed to by the parameter task.

Usage

```
STATUS NU_Send_Signals (NU_TASK *task,  
                        UNSIGNED signals);
```

Arguments

- task
Pointer to the user-supplied task control block.
- signals
Bit pattern representing signals to be sent.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TASK
The task pointer is invalid.

Description

If the receiving task has any of the designated signals enabled, its registered signal handler is executed as soon as the receiving task's priority permits. Each task has 32 available signals that are represented by each bit in signals.

There are several conditions that prevent the receiving task's signal handler from being executed, as follows:

- Receiving task is in a finished or terminated state.
- Receiving task is unconditionally suspended (either it was not started after creation or it was suspended by NU_Suspend_Task). If this is the case, the signal handler does not execute until the task is resumed.
- There is always a task ready at a higher priority than the receiving task.
- The receiving task has not enabled the signals sent.
- The receiving task has not registered a signal handler.

Example

```
NU_TASK Task;
STATUS status;
.
.
.
/* Send signals 1, 7, and 31 to the task control block "Task". Notice that
   the signals correspond to the bit position. Assume "Task" has
   previously
   been created with the Nucleus PLUS NU_Create_Task service call. */
status = NU_Send_Signals(&Task, 0x80000082);
```

Related Topics

[Signal Services Function Reference](#)

[NU_Receive_Signals](#)

[NU_Register_Signal_Handler](#)

[NU_Control_Signals](#)

Signal Example Source Code

In this example, see how Nucleus PLUS signals could be used to implement a control task which will perform various system tasks. Specific to this example, a task will signal that it can now be deleted, and removed from the system.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

In this demonstration, a task, specifically task_1 will send a signal to the control task, task_control, when it has finished processing. The task_control task will then delete the task, and deallocate the memory used for its stack. The #define TASK_1_FINISHED will be used to represent this signal.

```
#define TASK_1_FINISHED    0x00000001
```

Two Nucleus PLUS structures will be used in this example. Two NU_TASK structures are the task control blocks for the two tasks in the system: task_control, and task_1.

```
NU_TASK task_control;
NU_TASK task_1;
```

Two void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. The void pointer, stack_task_1 will be used in this example to deallocate the memory associated with the task_1 stack.

```
VOID *stack_control;
VOID *stack_task_1;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.


```
VOID control(UNSIGNED argc, VOID *argv);  
VOID entry_1(UNSIGNED argc, VOID *argv);  
VOID sh_control(UNSIGNED signals);
```

`Application_Initialize` will be used to set up memory for two task stacks in the system, which will then be created with the `NU_Create_Task` service call. After `Application_Initialize` executes, all tasks will be created, and the system will be ready to begin executing in a multi-tasking environment.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the passed in dynamic memory pool `mem_pool`. A pointer to the newly allocated memory is assigned to `stack_control`, and `stack_task_1` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

```
NU_Allocate_Memory(mem_pool, &stack_control, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_control, "CONTROL", control, 0, NU_NULL,
              stack_control, 1024, 11, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(mem_pool, &stack_task_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_1, "TASK1", entry_1, 0, NU_NULL, stack_task_1,
              1024, 10, 0, NU_PREEMPT, NU_START);
}
```

In order for a Nucleus PLUS task to receive signals, first, a signal handler must be associated with that task. The `task_control` task is used to register and control the `sh_control` signal handler. In a complete system, this task could also be used to run periodic system maintenance that did not depend on a signal being issued.

```
VOID control(UNSIGNED argc, VOID *argv)
{
```

The `NU_Register_Signal_Handler` service call associates a signal handling function with a specific task. After this call, upon a signal being sent to this task, the associated signal handler function will be executed. The associated function will be responsible for determining which signal was sent, and to take the correct action.

```
NU_Register_Signal_Handler(&sh_control);
```

The task also needs to be informed of which signals in the system it should respond to. The `NU_Control_Signals` service call will set the required flags so that the signal handler function is only executed when valid signals are sent.

```
NU_Control_Signals(TASK_1_FINISHED);
```

For this demonstration, if this task is ever executed, sleep for 10 timer ticks. In a real system, code could be inserted to do periodic maintenance regardless of whether a signal was sent to this task.

```
while(1)
{
    NU_Sleep(10);
}
```

The `sh_control` function is the signal handler that was associated with the `task_control` task. It is responsible for examining the current set of signals, evaluating what action to take, and, then executing the correct code to handle that particular signal (or set of signals). Specific to this example, the signal handler will determine if `TASK_1_FINISHED` was sent, and if so, delete the task, and deallocate the memory used for its task stack.

```
VOID sh_control(UNSIGNED signals)
{
```

First, determine if TASK_1_FINISHED was actually sent to the control task. If TASK_1_FINISHED was sent, then delete the task with a call to NU_Delete_Task, and deallocate the memory for the task's stack with a call to NU_Deallocate_Memory.

```
    if (signals & TASK_1_FINISHED)
    {
        NU_Delete_Task(&task_1);
        NU_Deallocate_Memory(&stack_task_1);
    }
    Use NU_Receive_Signals to clear the current set of signals.
    NU_Receive_Signals();
}
```

In this demonstration, task_1 is used to send a signal to the control task indicating that it has completed processing, and can now be removed from the system. Therefore, entry_1, the entry point for task_1, issues a call with NU_Send_Signals to send the TASK_1_FINISHED signal to task_control.

```
VOID entry_1(UNSIGNED argc, VOID *argv)
{
    NU_Send_Signals(&task_control, TASK_1_FINISHED);
}
```

Timers

The timer component provides timer services to Nucleus PLUS tasks and applications.

Most real-time applications require processing on periodic intervals of time. Each Nucleus PLUS task has a built-in timer. This timer is used to provide task sleeping and service call suspension timeouts.

Ticks

A tick is the basic unit of time for all Nucleus PLUS timer facilities. Each tick corresponds to a single hardware timer interrupt. The amount of actual time a tick represents is usually user-programmable.

Margin of Error

A timer request may be satisfied as much as one tick early in actual time. This is because a tick can occur immediately after the timer request. Therefore, the first tick of a timer request represents an actual time ranging from zero to the rate of the hardware timer interrupt. For

example, the amount of actual time expired for a request of n ticks falls between the actual time n and $n-1$ ticks represent.

Hardware Requirement

Nucleus PLUS timer services require a periodic timer interrupt from the hardware. Without such an interrupt, timer facilities will not function. However, other Nucleus PLUS facilities are not affected by the absence of timer facilities.

Continuous Clock

Nucleus PLUS maintains a continuous counting tick clock. The maximum value of this clock is 4,294,967,294. The clock automatically resets on the tick after the maximum value is reached.

This continuous clock is reserved exclusively for application use. It may be read from and written to by the application at any time.

Task Timers

Each task has a built-in timer. This timer is used for task-sleep requests and suspension timeout requests. Additionally, a time-slice timer is available for tasks that require time-slicing.

Application Timers

Nucleus PLUS provides programmable timers for applications. These timers execute a specific user-supplied routine when they expire. The user-supplied expiration routine executes as a high-level interrupt service routine. Therefore, self-suspension requests are not allowed. Additionally, processing should be kept to a minimum.

Re-Scheduling

When a timer expires, the prescribed expiration routine is executed. After execution is complete, the timer is either dormant or rescheduled. If the timer's reschedule value is 0, it is dormant after the initial expiration. However, if the timer's reschedule value is nonzero, it is rescheduled to expire at that interval.

Enable/Disable

Application timers may be automatically enabled during creation. Additionally, timers may be enabled and disabled dynamically.

Reset

The initial ticks, rescheduling rate, and the expiration routine of a timer may be reset dynamically by the application.

Timer Dynamic Creation

Nucleus PLUS application timers are created and deleted dynamically. There is no preset limit on the number of timers an application may have. Each timer requires a control block. The memory for this is supplied by the application.

Timer Determinism

Processing time required to create, enable, disable, and modify application timers is constant. However, processing time required to execute the user-supplied expiration routines depends on the expiration routines themselves and the number of timers that expire simultaneously.

Timer Information

Application tasks may obtain a list of active timers. Detailed information about each timer is also available. This information includes the timer name, status, initial ticks, reschedule value, remaining ticks, and the expiration count.

Timer Services Function Reference

The following function reference contains all functions related to Nucleus PLUS timers. The following functions are contained in this reference:

- [NU_Control_Timer](#)
- [NU_Create_Timer](#)
- [NU_Delete_Timer](#)
- [NU_Established_Timers](#)
- [NU_Get_Remaining_Time](#)
- [NU_Pause_Timer](#)
- [NU_Reset_Timer](#)
- [NU_Resume_Timer](#)
- [NU_Retrieve_Clock](#)
- [NU_Retrieve_Clock64](#)
- [NU_Set_Clock \(Deprecated\)](#)

- [NU_Set_Clock64](#)
- [NU_Ticks_To_Time](#)
- [NU_Time_To_Ticks](#)
- [NU_Get_Time_Stamp](#)
- [NU_Timer_Information](#)
- [NU_Timer_Pointers](#)

NU_Control_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service enables or disables the application timer pointed to by timer. Legal values for the enable parameter are NU_ENABLE_TIMER and NU_DISABLE_TIMER.

Usage

```
STATUS NU_Control_Timer (NU_TIMER *timer,  
                        OPTION  enable);
```

Arguments

- timer
Pointer to the user-supplied timer control block.
- enable
Valid options for this parameter are NU_ENABLE_TIMER and NU_DISABLE_TIMER. NU_ENABLE_TIMER immediately after the function call. NU_DISABLE_TIMER leaves the timer disabled.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TIMER
The timer pointer is invalid.
- NU_INVALID_ENABLE
The enable parameter is invalid.

Example

```
NU_TIMER Timer;  
STATUS status;  
.  
.  
.  
/* Disable the timer control block "Timer". Assume "Timer" has previously  
   been created with the Nucleus PLUS NU_Create_Timer service call. */  
  
status =  NU_Control_Timer(&Timer, NU_DISABLE_TIMER);  
  
/* At this point, status can be examined to determine whether the service  
   request was successful. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Timer_Information](#)

[NU_Reset_Timer](#)

[NU_Create_Timer](#)

NU_Create_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service creates an application timer. The specified expiration routine is executed each time the timer expires. Application expiration routines should avoid task suspension options. Suspension of the expiration routine can cause delays in other application timer requests.

Usage

```
STATUS NU_Create_Timer (NU_TIMER *timer,  
                        CHAR      *name,  
                        VOID      (*expiration_routine) (UNSIGNED),  
                        UNSIGNED id,  
                        UNSIGNED initial_time,  
                        UNSIGNED reschedule_time,  
                        OPTION    enable);
```

Arguments

- **timer**
Pointer to the user-supplied timer control block. All subsequent requests made to the timer require this pointer.
- **name**
Pointer to a seven-character name for the timer. The name must be null-terminated.
- **expiration_routine**
Specifies the application routine to execute when the timer expires.
- **id**
An UNSIGNED data element supplied to the expiration routine. The parameter may be used to help identify timers that use the same expiration routine.
- **initial_time**
Specifies the initial number of timer ticks for timer expiration. A value of 0 will return an error if error checking is enabled.
- **reschedule_time**
Specifies the number of timer ticks for expiration after the first expiration. If this parameter is 0, the timer only expires once.
- **enable**
Valid options for this parameter are NU_ENABLE_TIMER and NU_DISABLE_TIMER. NU_ENABLE_TIMER activates the timer after it is created. NU_DISABLE_TIMER

leaves the timer disabled. Timers created with the `NU_DISABLE_TIMER` must be enabled by a call to `NU_Control_Timer` later.

Return Values

- `NU_SUCCESS`
Function completed successfully
- `NU_INVALID_TIMER`
The timer control block pointer is `NULL` or is already in use.
- `NU_INVALID_FUNCTION`
The expiration function pointer is `NULL`.
- `NU_INVALID_ENABLE`
The enable parameter is invalid.
- `NU_INVALID_OPERATION`
The pause made with an initial time of 0.

Example

```
/* Assume timer control block "Timer" is defined as a global data
   structure. This is one of several ways to allocate a control block. */

NU_TIMER  Timer;
.
.
/* Assume status is defined locally.  */

STATUS status; /* Timer creation status  */

/* Create a timer that has an expiration function "timer_expire", an ID of
   0, an initial expiration of 23 timer ticks. After the initial
   expiration, the timer expires every 5 timer ticks. Note that the timer
   is enabled during creation.  */
status =  NU_Create_Timer(&Timer,"any name", timer_expire, 0, 23, 5,
                        NU_ENABLE_TIMER);

/* At this point status indicates if the service was successful.  */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Reset_Timer](#)

[NU_Timer_Pointers](#)

[NU_Established_Timers](#)

[NU_Timer_Information](#)

[NU_Delete_Timer](#)

NU_Delete_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service deletes a previously created application timer. The parameter timer identifies the timer to delete.

Note

The specified timer must be disabled prior to this service request. The application must prevent the use of this timer during and after deletion.

Usage

```
STATUS NU_Delete_Timer (NU_TIMER *timer);
```

Arguments

- timer
Pointer to the user-supplied timer control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TIMER
The timer pointer is invalid.
- NU_NOT_DISABLED
The specified timer is not disabled.

Example

```
NU_TIMER Timer;  
STATUS status;  
.  
.  
.  
/* Delete the timer control block "Timer". Assume "Timer" has previously  
   been created with the Nucleus PLUS NU_Create_Timer service call. */  
status = NU_Delete_Timer(&Timer);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Reset_Timer](#)

[NU_Timer_Pointers](#)

[NU_Established_Timers](#)

[NU_Timer_Information](#)

[NU_Create_Timer](#)

NU_Established_Timers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service returns the number of established timers. All created timers are considered established. Deleted timers are no longer considered established.

Usage

```
UNSIGNED NU_Established_Timers (VOID);
```

Return Values

- This service returns the number of established timers.

Example

```
UNSIGNED  total_timers;

/* Obtain the total number of timers. */
total_timers =  NU_Established_Timers();
```

Related Topics

[Timer Services Function Reference](#)

[NU_Reset_Timer](#)

[NU_Timer_Pointers](#)

[NU_Delete_Timer](#)

[NU_Timer_Information](#)

[NU_Create_Timer](#)

NU_Get_Remaining_Time

Allowed From: HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

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

Usage

```
STATUS NU_Get_Remaining_Time (NU_TIMER *timer,  
                             UNSIGNED *remaining_time);
```

Arguments

- timer
Pointer to the user-supplied timer control block.
- remaining_time
Pointer to the number of clock ticks until the timer expires.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TIMER
The timer pointer is invalid.

Example

```
NU_TIMER Timer;  
UNSIGNED time_left;  
STATUS status;  
.  
.  
.  
/* Assume "Timer" has previously been created with the Nucleus PLUS  
   NU_Create_Timer service call. */  
  
status = NU_Get_Remaining_Time(&Timer, &time_left);  
  
/* At this point, status can be examined to determine whether the service  
   request was successful. If so, time_left holds the tick value until  
   Timer expires. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Delete_Timer](#)

[NU_Control_Timer](#)

[NU_Create_Timer](#)

[NU_Timer_Information](#)

NU_Pause_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service pauses an application timer.

Usage

```
STATUS NU_Pause_Timer(NU_TIMER *timer);
```

Arguments

- timer
Pointer to the timer to be paused.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_TIMER
The timer control block pointer is invalid.
- NU_INVALID_PAUSE_TIMER
The passed in timer is already paused.

Example

```
STATUS status;  
/* Pause the application timer. */  
status = NU_Pause_Timer(my_timer);
```

Related Topics

[Timer Services Function Reference](#)

[NU_Resume_Timer](#)

NU_Reset_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service resets the specified timer with new operating parameters.

Note



The timer must be disabled before this service is called.

Usage

```
STATUS NU_Reset_Timer (NU_TIMER *timer,  
                      VOID      (*expiration_routine) (UNSIGNED) ,  
                      UNSIGNED  initial_time,  
                      UNSIGNED  reschedule_time,  
                      OPTION    enable)
```

Arguments

- **timer**
Pointer to the timer.
- **expiration_routine**
Specifies the application routine to execute when the timer expires.
- **initial_time**
Specifies the initial number of timer ticks for timer expiration. A value of 0 will return an error if error checking is enabled.
- **reschedule_time**
Specifies the number of timer ticks for expiration after the first expiration. If this parameter is 0, the timer only expires once.
- **enable**
Valid options for this parameter are `NU_ENABLE_TIMER` and `NU_DISABLE_TIMER`. `NU_ENABLE_TIMER` activates the timer immediately after it is reset. `NU_DISABLE_TIMER` leaves the timer disabled. Timers reset with `NU_DISABLE_TIMER` must be enabled by a call to `NU_Control_Timer` at a later time.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_TIMER**
The timer control block pointer is invalid.

- **NU_INVALID_FUNCTION**
The expiration function pointer is NULL.
- **NU_INVALID_ENABLE**
The enable parameter is invalid.
- **NU_NOT_DISABLED**
The timer is currently enabled. It must be disabled before it can be reset.

Example

```
NU_TIMER Timer;
STATUS status
.
.
.
/* Reset the timer control block "Timer" to expire initially after 3 timer
   ticks and then expire every 30 timer ticks. Also, the new expiration
   routine is "new_expire". Automatically enable the timer after it is
   reset. Assume "Timer" has previously been created with the Nucleus
   PLUS NU_Create_Timer service call. */
status = NU_Reset_Timer(&Timer, new_expire, 3, 30, NU_ENABLE_TIMER);

/* Contents of status indicates whether or not the service was
   successful. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Create_Timer](#)

[NU_Delete_Timer](#)

[NU_Timer_Information](#)

[NU_Control_Timer](#)

NU_Resume_Timer

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service resumes a paused application timer.

Usage

```
STATUS NU_Resume_Timer(NU_TIMER *timer);
```

Arguments

- timer
Pointer to the paused timer.

Return Values

- NU_SUCCESS
Function completed successfully
- NU_INVALID_TIMER
The timer control block pointer is invalid.
- NU_INVALID_RESUME_TIMER
The passed in timer was not previously paused

Example

```
STATUS status;  
/* Resume the paused timer. */  
status = NU_Resume_Timer(my_timer);
```

Related Topics

[Timer Services Function Reference](#)

[NU_Pause_Timer](#)

NU_Retrieve_Clock

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service returns the current value of the continuously incrementing timer tick counter. The counter increments once for every timer interrupt.

Usage

```
UNSIGNED NU_Retrieve_Clock (VOID);
```

Return Values

- This service call returns the current value of the system clock.

Example

```
UNSIGNED clock_value;

/* Read the current value of the system tick clock. */
clock_value = NU_Retrieve_Clock( );
```

Related Topics

[Timer Services Function Reference](#)

[Kernel Demo](#)

[NU_Retrieve_Clock64](#)

NU_Retrieve_Clock64

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service returns the current value of the continuously incrementing timer tick counter plus the offset. The counter increments once for every timer interrupt.

Usage

```
UINT64 NU_Retrieve_Clock64 (VOID);
```

Return Values

- This service call returns the current value of the system clock plus the offset.

Example

```
UINT64 clock_value;

/* Read the current value of the system tick clock plus the offset. */
clock_value = NU_Retrieve_Clock64( );
```

Related Topics

[Timer Services Function Reference](#)

[NU_Set_Clock64](#)

NU_Set_Clock (*Deprecated*)

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service sets the continuously counting system clock to the value specified by new_value.

Usage

```
VOID NU_Set_Clock (UNSIGNED new_value);
```

Arguments

- new_value
The new value for the system clock.

Example

```
/* Set the system clock to 0. */  
NU_Set_Clock(0);
```



Note

Use this API with caution. The value of the clock returned is a shared system resource and the correct operation of existing algorithms can depend on its value or expected future value. Nucleus Middleware makes use of this clock value for timing that is critical to operation of the middleware components.

This API has been deprecated. Using this API with NU_Retrieve_Clock64/NU_Set_Clock64 will cause unpredictable results with the values returned by these APIs.

Related Topics

[Timer Services Function Reference](#)

[NU_Retrieve_Clock](#)

NU_Set_Clock64

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service sets the system clock offset calculated from the specified value.

Usage

```
VOID NU_Set_Clock64 (UINT64 new_value);
```

Arguments

- new_value

The new value for the system clock plus offset.

Example

```
/* Set the system clock offset to 500 minus the current system clock . */  
NU_Set_Clock64(500);
```

Related Topics

[Timer Services Function Reference](#)

[NU_Retrieve_Clock64](#)

NU_Ticks_To_Time

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service converts a Nucleus 64-bit tick count to a time_t "calendar time" value.

Usage

```
time_t NU_Ticks_To_Time (UINT64 ticks);
```

Arguments

- ticks
The 64-bit count value.

Return Values

- cal_time
The calendar time calculated from the ticks input.

Note



Type time_t is "long int", so it can hold only 32-bits. With larger than 32-bit value of ticks, the result will be truncated.

Example

```
time_t new_time;  
  
/* Get the calendar time of some tick value. */  
new_time = NU_Ticks_To_Time(500000);
```

Related Topics

[Timer Services Function Reference](#)

[NU_Time_To_Ticks](#)

NU_Time_To_Ticks

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service converts a calendar time to a 64-bit tick value.

Usage

```
UINT64 NU_Time_To_Ticks (time_t cal_time);
```

Arguments

- `cal_time`
The calendar time that needs to be converted to ticks.

Return Values

- `ticks`
The 64-bit count value calculated from the calendar time input.

Example

```
UINT64 ticks;  
  
/* Get the tick value of a given calendar time. */  
ticks = NU_Time_To_Ticks(12500);
```

Related Topics

[Timer Services Function Reference](#)

[NU_Set_Clock64](#)

NU_Get_Time_Stamp

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service returns a 64-bit UINT time stamp in clock ticks.

Usage

```
UINT64 NU_Get_Time_Stamp (VOID);
```

Arguments

- None

Return Values

- Returns a 64-bit UINT time stamp in clock ticks.

Examples

```
UINT64 ticks;

/* Get the tick value at the current time. */
ticks = NU_Get_Time_Stamp();
```

Related Topics

[Timer Services Function Reference](#)

[NU_Time_To_Ticks](#)

NU_Timer_Information

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service returns various information about the specified application timer.

Usage

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

Arguments

- **timer**
 Pointer to the application timer.
- **name**
 Pointer to an eight-character destination area for the timer's name. This includes space for the null terminator.
- **enable**
 Pointer to a variable to hold the timer's current enable state, either `NU_ENABLE_TIMER` or `NU_DISABLE_TIMER`.
- **expirations**
 Pointer to a variable to hold the number of times the timer has expired.
- **id**
 Pointer to a variable to hold the user-supplied ID.
- **initial_time**
 Pointer to a variable to hold the initial timer expiration value.
- **reschedule_time**
 Pointer to a variable to hold the timer's reschedule value.

Return Values

- **NU_SUCCESS**
 Function completed successfully.

- **NU_INVALID_TIMER**
The timer pointer is invalid.

Example

```
NU_TIMER Timer;
CHAR      timer_name[8];
OPTION    enable;
UNSIGNED  expirations;
UNSIGNED  id;
UNSIGNED  initial_time;
UNSIGNED  reschedule_time;
STATUS    status;
.
.
.
/* Obtain information about the timer control block "Timer". Assume
   "Timer" has previously been created with the Nucleus PLUS
   NU_Create_Timer service call. */
status = NU_Timer_Information(&Timer, timer_name, &enable, &expiration,
                             &id, &initial_time, &reschedule_time);

/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Established_Timers](#)

[NU_Timer_Pointers](#)

[NU_Delete_Timer](#)

[NU_Reset_Timer](#)

[NU_Create_Timer](#)

NU_Timer_Pointers

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Timer Services

Tasking Changes: No

This service builds a sequential list of pointers to all established application timers in the system.

Note

Timers that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_Timer_Pointers (NU_TIMER **pointer_list,  
                           UNSIGNED maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of `NU_TIMER` pointers. This array will be filled with pointers of established timers in the system.
- `maximum_pointers`
The maximum number of `NU_TIMER` pointers to place into the array. Typically this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of timers that are active in the system.

Example

```
/* Define an array capable of holding 20 timer pointers. */  
NU_TIMER *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active timer pointers (Maximum of 20). */  
number = NU_Timer_Pointers(&Pointer_Array[0], 20);  
  
/* At this point, number contains the actual number of pointers in the  
list. */
```

Related Topics

[Timer Services Function Reference](#)

[NU_Delete_Timer](#)

[NU_Established_Timers](#)

[NU_Reset_Timer](#)

[NU_Timer_Information](#)

[NU_Create_Timer](#)

Timer Example Source Code

The following example program demonstrates how a Nucleus PLUS timer could be used to execute code on a periodic basis. The following Nucleus PLUS program contains a single timer that expires every five timer ticks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single Nucleus PLUS structure is required for this demonstration. The timer control block, NU_TIMER, will be associated with a timer expiration routine using the NU_Create_Timer service call.

```
NU_TIMER timer_demo;
```

The function expiration_routine will serve as the timer expiration routine for the timer_demo timer. The only parameter necessary for a timer expiration routine is a single UNSIGNED which will contain the timer ID for which this timer was associated with in the NU_Create_Timer service call. As an example, this ID could be used to allow the same expiration routine to be used for multiple timers.

```
VOID expiration_routine(UNSIGNED id);
```

In this demonstration, the Application_Initialize function will be used to create the single Nucleus PLUS timer. After Application_Initialize executes, all tasks will be created, and the system will be ready to begin executing in a multi-tasking environment.

```
VOID Application_Initialize(NU_MEMORY_POOL* mem_pool,  
                           NU_MEMORY_POOL* uncached_mem_pool)  
{
```

Create the Nucleus PLUS timer with the NU_Create_Timer service call. The timer, timer_demo will be named "TIMER", and will be associated with the timer expiration routine, expiration_routine. The timer will be given the ID of 1, will expire five timer ticks after processing begins, and will expire every five timer ticks thereafter. The NU_ENABLE_TIMER parameter specifies that this timer should be immediately enabled. The parameter NU_DISABLE_TIMER could also be used, which would require that the function NU_Control_Timer was issued later to begin timer processing. The use of this method would allow for timers to be enabled and disabled based upon the current status of the system. Similarly, NU_Reset_Timer could also be used to later modify the functionality of the timer.

```
    NU_Create_Timer(&timer_demo, "TIMER", expiration_routine, 1, 5, 5,  
                   NU_ENABLE_TIMER);  
}
```

The function `expiration_routine` is the function that will be executed whenever the `timer_demo` expires. This function will be associated with `timer_demo` using the `NU_Create_Timer` service call.

```
VOID expiration_routine(UNSIGNED id)
{
}
```

Interrupts

Interrupts provide Nucleus PLUS with a capability to respond to asynchronous, non-periodic events within the minimum specified time.

An interrupt is a mechanism for providing immediate response to an external or internal event. When an interrupt occurs, the processor suspends the current path of execution and transfers control to the appropriate Interrupt Service Routine (ISR). The exact operation of an interrupt is inherently processor-specific.

Nucleus PLUS supports managed ISRs. A managed ISR is one that does not need to save and restore context. Managed ISRs may be written in C or assembly language.

Interrupt Protection

Interrupts pose interesting problems for all real-time kernels. Nucleus PLUS is no exception. The main problem stems from the fact that ISRs need to have access to Nucleus PLUS services. On the surface this may not seem like a problem; however, it requires protection of data structures manipulated during a service call from simultaneous access by an ISR. The simplest method of protection is to lock out interrupts for the duration of the service.

Responding to interrupts quickly is a cornerstone of real-time systems. Therefore, locking out interrupts to protect internal data structures is not desirable. Nucleus PLUS handles this protection problem by dividing application ISRs into low-level and high-level components.

Low-Level ISR

The Low-Level Interrupt Service Routine (LISR) executes as a normal ISR, which includes using the current stack. Nucleus PLUS saves context before calling a LISR and restores context after the LISR returns. Therefore, LISRs may be written in C and may call other C routines. However, there are only a few Nucleus PLUS services available to a LISR. If the interrupt processing requires additional Nucleus PLUS services, a High-Level Interrupt Service Routine (HISR) must be activated. Nucleus PLUS supports nesting of multiple LISRs. The following services are available from LISRs:

- [NU_Activate_HISR](#)


- [NU_Current_HISR_Pointer](#)
- [NU_Current_Task_Pointer](#)
- [NU_Local_Control_Interrupts](#)
- [NU_Retrieve_Clock](#)

High-Level ISR

The High-Level Interrupt Service Routine (HISR)s are created and deleted dynamically. Each HISR has its own stack space and its own control block. The memory for each is supplied by the application. Of course, the HISR must be created before it is activated by a LISR.

Since a HISR has its own stack and control block, it can be temporarily blocked if it tries to access a Nucleus PLUS data structure that is already being accessed.

HISRs are allowed access to most Nucleus PLUS services, with the exception of self-suspension services. Additionally, since a HISR cannot suspend on a Nucleus PLUS service, the “suspend” parameter must always be set to `NU_NO_SUSPEND`.

 **Note** “printf” calls are not allowed from LISR or HISR context.

There are three priority levels available to HISRs. If a higher priority HISR is activated during processing of a lower priority HISR, the lower priority HISR is preempted in much the same manner as a task gets preempted. HISRs of the same priority are executed in the order in which they were originally activated. All activated HISRs are processed before normal task scheduling is resumed.

An activation counter is maintained for each HISR. This counter is used to insure that each HISR is executed once for each activation. Note that each additional activation of an already active HISR is processed by successive calls to that HISR.

HISR Information

Application tasks may obtain a list of active HISRs. Detailed information about each HISR is also available. This information includes the HISR name, total scheduled count, priority, and stack parameters.

Interrupt Latency

Interrupt latency is a term that describes the amount of time for which interrupts are locked out. Since Nucleus PLUS does not rely on locking out interrupts to protect against simultaneous ISR access, interrupt latency is small and constant. In fact, interrupts are only locked out over several instructions in some Nucleus PLUS ports.

Application Interrupt Lockout

Applications are provided with the ability to disable and enable interrupts. An interrupt locked out by the application remains locked out until the application unlocks it.

Direct Vector Access

Nucleus PLUS provides the ability to directly set up interrupt vectors. ISRs loaded directly into the vector table are required to save and restore registers used. Therefore, ISRs entered directly into the vector table are often written in assembly language. Such ISRs, providing certain conventions are followed, may activate a HISR.

Managed ISRs

Managed ISRs are referred to in this document as LISR. LISRs execute in the same fashion as a traditional ISR, except all context saving and restoring is taken care of by Nucleus PLUS.

The following is an example segment of code that defines a LISR function and registers it with vector 10:

```
VOID (*old_lisr)(INT);
VOID Example_LISR(INT vector);
INT  Interrupt_Count = 0;
.
.
/* Register the LISR with vector 10. The previously registered LISR is
   returned in old_lisr. */
NU_Register_LISR(10, Example_LISR, &old_lisr);
.
.
.

/* Actual definition of the LISR associated with vector 10. */
VOID Example_LISR (INT vector)
{
    /* Increment the global interrupt counter. */
    Interrupt_Count++;
}
```

When interrupt 10 occurs, Example_LISR is called with the vector parameter set to 10. Interrupt processing consists of incrementing a global variable, which is completed when Example_LISR returns. It is important to note that LISRs have extremely limited access to Nucleus PLUS services. For example, if a task must be resumed as a result of interrupt 10, a High-Level Interrupt Service Routine (HISR) must be activated from within the LISR.

The following example resumes the task pointed to by Task_0_Ptr when interrupt 10 occurs:

```
extern NU_TASK *Task_0_Ptr;
NU_HISR      HISR_Control;
CHAR        HISR_Stack[500];
```

```

VOID          (*old_lisr) (INT);
VOID          Example_LISR(INT vector);
VOID          Example_HISR(VOID);
.
.
.

/* Create a HISR. This HISR is activated by the LISR associated with
   vector 10. */
NU_Create_HISR(&HISR_Control, "EXMPHISR", Example_HISR, 2, HISR_Stack,
500);

/* Register the LISR with vector 10. The previously registered LISR is
   returned in old_lisr. */

NU_Register_LISR(10, Example_LISR, &old_lisr);
.
.
.
/* Actual definition of the LISR associated with vector 10. */
VOID Example_LISR(INT vector)
{

    /* Activate Example_HISR to resume the task pointed to by
       "Task_0_Ptr."
       Not allowed to call most Nucleus PLUS services from LISR. */

    NU_Activate_HISR(&HISR_Control);

}

/* Actual definition of the HISR associated with the Example_LISR
   function. */

VOID Example_HISR(VOID)
{
    /* Resume the task pointed to by "Task_0_Ptr" */

    NU_Resume_Task(Task_0_Ptr);

}

```

Interrupt Services Function Reference

The following function reference contains all functions related to Nucleus PLUS interrupts. The following functions are contained in this reference:

- [NU_Activate_HISR](#)
- [NU_Control_Interrupts](#)
- [NU_Create_HISR](#)
- [NU_Current_HISR_Pointer](#)

- [NU_Delete_HISR](#)
- [NU_Established_HISRs](#)
- [NU_HISR_Information](#)
- [NU_HISR_Pointers](#)
- [NU_Local_Control_Interrupts](#)
- [NU_Register_LISR](#)

NU_Activate_HISR

Allowed From: LISR, HISR, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service activates the HISR pointed to by `histr`. If the specified HISR is currently executing, this activation request is not processed until the current execution is complete. A HISR is executed once for each activation request.

Usage

```
STATUS NU_Activate_HISR (NU_HISR *histr);
```

Note



Activating a HISR from within a task or signal handler will not cause the HISR to be immediately scheduled. The HISR will not be scheduled until execution is returned to the scheduler. This will occur after the current task or signal handler has relinquished control of the processor or after the next managed interrupt (this includes the Nucleus PLUS timer interrupt).

Note



Activating a higher priority HISR from within the context of a lower priority HISR will not cause the higher priority HISR to be immediately scheduled. The higher priority HISR will be scheduled after the completion of the lower priority HISR or after the next managed interrupt (this includes the Nucleus PLUS timer interrupt).

Arguments

- `histr`
Pointer to the user-supplied HISR control block.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_HISR`
The HISR control block pointer is not valid.

Example

```
NU_HISR Operator_Input;  
STATUS status;  
  
/* Activate the previously created operator input HISR for which the  
   control block is Operator_Input. */  
status = NU_Activate_HISR(&Operator_Input);
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_HISR_Information](#)

[NU_Delete_HISR](#)

[NU_Create_HISR](#)

NU_Control_Interrupts

Allowed From: HISR, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service enables or disables interrupts according to the value specified in `new_level`.

Usage

```
INT NU_Control_Interrupts (INT new_level);
```

Arguments

- `new_level`

New interrupt level for the system. The options `NU_DISABLE_INTERRUPTS` (disable all interrupts) and `NU_ENABLE_INTERRUPTS` (enable all interrupts) are always available. Other options may be available depending upon architecture.

Return Values

- This service returns the previous level of enabled interrupts.

Description

Interrupts are disabled and enabled in a task-independent manner. Therefore, an interrupt disabled by this service remains disabled until enabled by a subsequent call to this service. Values of `new_level` are processor dependent. However, the values `NU_DISABLE_INTERRUPTS` and `NU_ENABLE_INTERRUPTS` may be used to disable all interrupts and enable all interrupts, respectively.

Note



The operation of this service is based on the setting in *plus_cfg.h* for global interrupt locking. If this configuration setting is set to `NU_FALSE`, this service performs the same as `NU_Local_Control_Interrupts` (defined as follows). If set to `NU_TRUE`, interrupts will be disabled in a task-independent manner.

Example

```
INT old_level; /* Old interrupt level. */

/* Lockout all interrupts temporarily. */
old_level = NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);
.
.
.
/* Restore previous interrupt lockout level. */
NU_Control_Interrupts(old_level);
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Register_LISR](#)

[NU_Delete_HISR](#)

[NU_Create_HISR](#)

NU_Create_HISR

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service creates a HISR. HISRs are allowed to call most Nucleus PLUS services, unlike LISRs.

Usage

```
STATUS NU_Create_HISR (NU_HISR *hisr,  
                      CHAR      *name,  
                      VOID      (*hisr_entry) (VOID) ,  
                      OPTION    priority,  
                      VOID      *stack_pointer,  
                      UNSIGNED   stack_size);
```

Arguments

- **hisr**
Pointer to the user-supplied HISR control block. All subsequent requests made to this HISR require this pointer.
- **name**
Pointer to a seven-character name for the HISR. The name must be null-terminated.
- **hisr_entry**
Specifies the function entry point of the HISR.
- **priority**
There are three HISR priorities (0-2). Priority 0 is the highest.
- **stack_pointer**
Pointer to the HISR's stack area. Each HISR has its own stack area. Note that the HISR stack is pre-allocated by the caller.
- **stack_size**
Number of bytes in the HISR stack.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_HISR**
The HISR control block pointer is NULL or is already in use.

- **NU_INVALID_ENTRY**
The HISR entry pointer is NULL.
- **NU_INVALID_PRIORITY**
The HISR priority is invalid.
- **NU_INVALID_MEMORY**
The stack pointer is NULL.
- **NU_INVALID_SIZE**
The stack size is too small.

Example

```
/* Assume HISR control block "HISR" is defined as a global data structure.
   This is one of several ways to allocate a control block. */

NU_HISR HISR
.
.
/* Assume status is defined locally. */

STATUS status; /* HISR creation status */

/* Create an HISR. Note that the HISR entry function is "HISR_Entry" and
   the "stack_pointer" points to a previously allocated block of memory
   that contains 400 - bytes. */
status = NU_Create_HISR(&HISR, "any name", HISR_Entry,
                        2, stack_pointer, 400);

/* status indicates if the service was successful. */
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Established_HISRs](#)

[NU_HISR_Information](#)

[NU_HISR_Pointers](#)

[NU_Delete_HISR](#)

NU_Current_HISR_Pointer

Allowed From: LISR, HISR

Category: Interrupt Services

Tasking Changes: No

This service returns the currently executing HISR's pointer. If the caller is not an HISR, the value returned is NU_NULL.

Usage

```
NU_HISR *NU_Current_HISR_Pointer (VOID);
```

Return Values

- This service call returns a pointer the currently executing HISR's control block.

Example

```
NU_HISR *HISR_ptr;  
  
/* Get the currently running HISR pointer. */  
HISR_ptr = NU_Current_HISR_Pointer( );
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_HISR_Information](#)

[NU_HISR_Pointers](#)

[NU_Established_HISRs](#)

NU_Delete_HISR

Allowed From: Application _Initialize, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service deletes a previously created HISR. The parameter HISR identifies the HISR to delete. The application must prevent the use of this HISR during and after deletion.

Usage

```
STATUS NU_Delete_HISR (NU_HISR *hisr);
```

Arguments

- hisr
Pointer to the user-supplied HISR control block.

Return Values

- NU_SUCCESS
Function completed successfully.
- NU_INVALID_HISR
The HISR pointer is invalid.

Example

```
NU_HISR Hisr;  
STATUS status;  
.  
.  
.  
/* Delete the HISR control block "Hisr". Assume "Hisr" has previously  
   been created with the Nucleus PLUS NU_Create_HISR service call. */  
  
status = NU_Delete_HISR(&Hisr);  
  
/* At this point, status indicates whether the service request was  
   successful. */
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Established_HISRs](#)

[NU_HISR_Information](#)

[NU_HISR_Pointers](#)

[NU_Create_HISR](#)

NU_Established_HISRs

Allowed From: Application_Initialize, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service returns the number of established HISRs. All created HISRs are considered established. Deleted HISRs are no longer considered established.

Usage

```
UNSIGNED NU_Established_HISRs (VOID);
```

Return Values

- This service call returns the number of established HISRs in the system.

Example

```
UNSIGNED total_hisrs;  
  
/* Obtain the total number of HISRs. */  
total_hisrs = NU_Established_HISRs( );
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Delete_HISR](#)

[NU_HISR_Information](#)

[NU_HISR_Pointers](#)

[NU_Create_HISR](#)

NU_HISR_Information

Allowed From: Application_Initialize, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service returns various information about the specified HISR.

Usage

```
STATUS NU_HISR_Information (NU_HISR      *hisr,  
                           CHAR          *name,  
                           UNSIGNED      *scheduled_count,  
                           DATA_ELEMENT *priority,  
                           VOID          **stack_base,  
                           UNSIGNED      *stack_size,  
                           UNSIGNED      *minimum_stack);
```

Arguments

- **hisr**
Pointer to the HISR.
- **name**
Pointer to an eight-character destination area for the HISR's name. This includes space for the null terminator.
- **scheduled_count**
Pointer to a variable for holding the total number of times this HISR has been scheduled.
- **priority**
Pointer to a variable for holding the HISR's priority.
- **stack_base**
Pointer to a pointer for holding the original stack pointer. This is the same pointer supplied during creation of the HISR.
- **stack_size**
Pointer to a variable for holding the total size of the HISR's stack.
- **minimum_stack**
Pointer to a variable for holding the minimum amount of available stack space detected during HISR execution.

Return Values

- **NU_SUCCESS**
Function completed successfully.

- **NU_INVALID_HISR**
 The HISR pointer is invalid.

Example

```

NU_HISR      Hisr;
CHAR         hisr_name[8];
UNSIGNED     activations;
DATA_ELEMENT priority;
VOID         *stack_base;
UNSIGNED     stack_size;
UNSIGNED     minimum_stack;
STATUS       status
.
.
.
/* Obtain information about the HISR control block "Hisr". Assume "Hisr"
   has previously been created with the Nucleus PLUS NU_Create_HISR
   service call. */
status =  NU_HISR_Information(&Hisr, hisr_name, &activations,
                             &priority, &stack_base, &stack_size,
                             &minimum_stack);

/* If status is NU_SUCCESS, the other information is accurate. */

```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Delete_HISR](#)

[NU_Established_HISRs](#)

[NU_HISR_Pointers](#)

[NU_Create_HISR](#)

NU_HISR_Pointers

Allowed From: Application_Initialize, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service builds a sequential list of pointers to all established HISRs in the system.

Note



HISRs that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Usage

```
UNSIGNED NU_HISR_Pointers (NU_HISR **pointer_list,  
                           UNSIGNED maximum_pointers);
```

Arguments

- `pointer_list`
Pointer to an array of NU_HISR pointers. This array will be filled with pointers of established HISRs in the system.
- `maximum_pointers`
The maximum number of NU_HISR pointers to place into the array. Typically, this will be the size of the `pointer_list` array.

Return Values

- This service call returns the number of HISRS that are active in the system.

Example

```
/* Define an array capable of holding 20 HISR pointers. */  
NU_HISR *Pointer_Array[20];  
UNSIGNED number;  
  
/* Obtain a list of currently active HISR pointers (Maximum of 20). */  
number = NU_HISR_Pointers(&Pointer_Array[0],20);  
  
/* At this point, number contains the actual number of pointers in the  
list. */
```


Related Topics

[Interrupt Services Function Reference](#)

[NU_Established_HISRs](#)

[NU_Create_HISR](#)

[NU_Delete_HISR](#)

[NU_HISR_Information](#)

NU_Local_Control_Interrupts

Allowed From: LISR, HISR, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service enables or disables interrupts according to the value specified in `new_level`.

Usage

```
INT NU_Local_Control_Interrupts (INT new_level);
```

Arguments

- `new_level`

New interrupt level for the current subroutine. The options `NU_DISABLE_INTERRUPTS` (disable all interrupts) and `NU_ENABLE_INTERRUPTS` (enable all interrupts) are always available. Other options may be available depending upon architecture.

Return Values

- This service returns the previous level of enabled interrupts.

Description

Interrupts are disabled and enabled in a subroutine-dependent manner. This service changes the Status Register to the value specified. The Status Register will be set back to value set by the last call to `NU_Control_Interrupts` on the next context switch. Values of `new_level` are processor dependent. However, the values `NU_DISABLE_INTERRUPTS` and `NU_ENABLE_INTERRUPTS` may be used to disable all interrupts and enable all interrupts, respectively.

Example

```
INT old_level; /* Old interrupt level. */

/* Lockout all interrupts temporarily. */
old_level=NU_Local_Control_Interrupts(NU_DISABLE_INTERRUPTS);
.
.
.
return; /* Or interrupt return. */
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Delete_HISR](#)

[NU_Register_LISR](#)

[NU_Create_HISR](#)

NU_Register_LISR

Allowed From: Application_Initialize, HISR, Signal Handler, Task

Category: Interrupt Services

Tasking Changes: No

This service associates the LISR function pointed to by `lISR_entry` with the interrupt vector specified by `vector`.

Usage

```
STATUS NU_Register_LISR (INT      vector,
                        VOID      (*lISR_entry) (INT) ,
                        VOID      (**old_lISR) (INT) );
```

Arguments

- `vector`

The interrupt vector at which to register the interrupt. The following files will contain the various vector IDs for the target hardware:

bsp\<bsp_name>\include\bsp\arch\<bsp_name>\<bsp_name>_defs.h.

- `lISR_entry`

The subroutine to register at the vector.

- `old_lISR`

The subroutine previously registered at the specified vector.

Return Values

- `NU_SUCCESS`

Function completed successfully.

- `NU_INVALID_VECTOR`

The specified vector is invalid.

- `NU_NOT_REGISTERED`

The vector is not currently registered and deregistration was specified by `lISR_entry`.

Description

System context is automatically saved before calling the specified LISR and is restored after the LISR returns. Therefore, LISR functions may be written in C. However, LISRs are permitted access to only a few of Nucleus PLUS services. If interaction with other Nucleus PLUS services is required, a HISR must be activated by the LISR. The following services are available from LISRs:

- [NU_Activate_HISR](#)

- [NU_Local_Control_Interrupts](#)
- [NU_Current_HISR_Pointer](#)
- [NU_Current_Task_Pointer](#)
- [NU_Retrieve_Clock](#)

If the `lISR_entry` parameter is `NU_NULL`, the registration of the specified vector is cleared.

Note

If an LISR is written in assembly language, it must follow the C compiler's conventions regarding register usage and the return mechanism. See your compiler documentation for specific requirements of C-assembly language interaction.

Note

Floating point registers are not saved prior to executing LISRs. LISRs requiring floating point register usage must save and restore these registers as necessary.

Example

```
STATUS status;
VOID (*old_lISR)(INT);

/* Associate vector 10 with the LISR function "LISR_example".

VOID LISR_example(INT vector_number);
.
.
.
/* vector_number contains the actual interrupt vector number. */

/* Nucleus PLUS service calls, with the exception of NU_Activate_HISR and
several others, are not allowed in this function. */

status = NU_Register_LISR(10, LISR_example, &old_lISR);

/* If status is NU_SUCCESS, LISR_example is executed when interrupt vector
10 occurs. Note: "old_lISR" contains the previously registered LISR. */
```

Related Topics

[Interrupt Services Function Reference](#)

[NU_Control_Interrupts](#)

[NU_Create_HISR](#)

[NU_Delete_HISR](#)

[NU_Activate_HISR](#)

System Diagnostics

Nucleus PLUS system diagnostics services provide several facilities that enable you to perform system analysis following a failure.

Error Management

If a fatal system error occurs, processing is transferred to a common error handling routine. By default, this routine prepares an ASCII error message and halts the system. However, you may add additional error processing.

- [NU_ASSERT](#)
- [NU_CHECK](#)

NU_ASSERT

Allowed From: Application_Initialize, HISR, LISR, Signal Handler, Task

Category: Error Management Services

Tasking Changes: No

Implemented By: ERC_Assert and a macro definition in *er_extr.h*

This macro tests the provided assertion, and if that assertion is not true, calls the ERC_Assert internal routine. By default, ERC_Assert simply increments ERD_Assert_Count. Depending on the specific target, ERC_Assert may perform other actions such as triggering a breakpoint in a debugger.

NU_ASSERT is enabled by defining NU_DEBUG during compilation of the target application.

Usage

```
NU_ASSERT (assertion);
```

Arguments

- `assertion`
Assertion to test

Example

```
UNSIGNED size;  
/* ...size is set to some value... */  
/* Calls ERC_Assert if this assertion is not true (size <= 0) */  
NU_ASSERT(size > 0);
```

Related Topics

[System Diagnostics](#)

[NU_CHECK](#)

NU_CHECK

Allowed From: Application_Initialize, HISR, LISR, Signal Handler, Task

Category: Error Management Services

Tasking Changes: No

Implemented By: ERC_Assert and a macro definition in *er_extr.h*, and possibly NU_ASSERT

This macro tests the provided assertion, and if that assertion is not true, executes the statement specified by the action parameter.

NU_CHECK is disabled by defining NU_NO_ERROR_CHECKING during compilation of the Nucleus PLUS library and the target application. If NU_DEBUG is defined during compilation of a target application, each NU_CHECK will also contain an NU_ASSERT (assertion) macro.

Usage

```
NU_CHECK (assertion,  
         action);
```

Arguments

- **assertion**
Assertion to test
- **action**
Action to perform if assertion is not true

Example

```
UNSIGNED size;  
  
/* ...size is set to some value... */  
  
/* Sets status to NU_INVALID_SIZE if this assertion is not true  
   (size <= 0) */  
NU_CHECK(size > 0, status = NU_INVALID_SIZE);
```

Related Topics

[System Diagnostics](#)

[NU_ASSERT](#)

Run Time Library

This section describes library (RTL) functions that are tested and approved for use within Nucleus PLUS.

Enabling RTL Support

RTL support is enabled by default. To disable RTL support, disable RTL under *nu.os.kern.rtl*.

Memory Allocation

Memory allocation functions are utilized in several other RTL functions. For this reason functions such as malloc have been tested. It is however, highly recommended to use functions such as NU_Allocate_Memory to do all memory allocation within Nucleus PLUS.

Reentrancy

Not all RTL functions are reentrant. To make some RTL functions thread safe semaphores may be needed. Please see the appropriate toolset document for proper usage.

RTL Functions

Note



Some of the following functions are only supported on specific targets. See the Release Notes for your port to determine what functions are supported.

<ctype.h>

Table 5-2. Supported Functions in ctype.h

Functions	
isalnum	ispunct
isalpha	isupper
isctrl	isxdigit
isdigit	isprint
isgraph	isspace
islower	tolower
toupper	

<limits.h>

Table 5-3. Supported Constants/Macros/Types in limits.h

Constants/Macros/Types	
CHAR_BIT	UINT_MAX
SCHAR_MIN	LONG_MIN
SCHAR_MAX	LONG_MAX
UCHAR_MAX	ULONG_MAX
CHAR_MIN	LLONG_MIN
CHAR_MAX	LLONG_MAX
MB_LEN_MAX	ULLONG_MAX
SHRT_MIN	USHRT_MAX
SHRT_MAX	UINT_MAX
INT_MIN	INT_MAX

<math.h>

Table 5-4. Supported Functions in math.h

Functions	
acos	sqrt
asin	ceil
atan	floor
cos	fmod
sin	atan2
tan	cosh
exp	sinh
log	tanh
log10	frexp
fabs	ldexp
pow	modf

<stdarg.h>

Table 5-5. Supported Constants/Macros/Types in stdarg.h

Constants/Macros/Types
va_list

Table 5-6. Supported Functions in stdarg.h

Functions
va_arg
va_end
va_start

<stddef.h>

Table 5-7. Supported Constants/Macros/Types in stddef.h

Constants/Macros/Types
ptrdiff_t
NULL

Table 5-8. Supported Functions in stddef.h

Functions
offsetof

<stdio.h>

Table 5-9. Supported Constants/Macros/Types in stdio.h

Constants/Macros/Types	
EOF	stderr
stdin	size_t
stdout	

Table 5-10. Supported Functions in stdio.h

Functions	
getc	vsprintf
getchar	vsnprintf
gets	putc
printf	putchar
vprintf	scanf
sprintf	puts
snprintf	perror

<stdlib.h>

Table 5-11. Supported Constants/Macros/Types in stdlib.h

Constants/Macros/Types	
RAND_MAX	MB_CUR_MAX

Table 5-12. Supported Functions in stdlib.h

Functions	
atof	malloc
atoi	realloc
atol	abs
strtod	labs
strtol	mblen
strtoul	mbtowc
rand	wctomb
srand	mbstowcs

Table 5-12. Supported Functions in stdlib.h (cont.)

Functions	
calloc	wcstombs
free	qsort

<string.h>

Table 5-13. Supported Functions in string.h

Functions	
memcpy	strcoll
memmove	strcpy
memchr	strncpy
memcmp	strerror
memset	strlen
strcat	strspn
strncat	strcspn
strchr	strpbrk
strrchr	strstr
strcmp	strtok
strxfrm	

<wchar.h>

Table 5-14. Supported Constants/Macros/Types in wchar.h

Constants/Macros/Types
wchar_t

Table 5-15. Supported Functions in wchar.h

Functions	
wscmp	wcsncpy
wcstok	wmemset
wcslen	wcschr
wcscpy	

<time.h>

Table 5-16. Supported Functions in time.h

Functions	
asctime	localtime
clock	mktime
ctime	strftime
difftime	time
gmtime	tzset

Note

The supported functions in <time.h> work only with the CSGNU tools.

Ncleus Plus Examples

The following example demonstrate the services provided by the Nucleus Plus component of the Nucleus kernel:

- [Kernel Demo](#)

Related Topics

[Nucleus Plus Overview](#)

Kernel Demo

This example demonstrates the “Nucleus Kernel Demo” sample project.

Purpose & Goals

This sample application illustrates the use of Plus Kernel APIs to show basic use of task synchronization and communication primitives. The simple application does the following:

- Task_0: Creates a timer task that just increments a count every 1 second and sets an event
- Task_1: Creates a consumer and producer task that sends messages using a queue
- Task_2: Creates a consumer and producer task that receives messages using a queue
- Task_3_and_4: Creates two tasks that compete for a semaphore and share the same task entry function
- Task_5: Creates a task that waits on the event set by the timer task and increments a count
- Task_6: Creates a task that outputs the state of the system to standard out and refreshes this information whenever a key is pressed

Setup

To get started with the kernel demo project, create a new C project, select the Nucleus Kernel Demo project, and read the README file included with the demo that is placed in your workspace.

Components Used

This application uses the following component:

- nu.os.kern.plus

API Reference

This example uses the following services:

Table 5-17. Kernel Demo Services

Service	Function Using Service
NU_Allocate_Memory	Application_Initialize
NU_Create_Event_Group	Application_Initialize
NU_Create_Queue	Application_Initialize
NU_Create_Semaphore	Application_Initialize

Table 5-17. Kernel Demo Services (cont.)

Service	Function Using Service
NU_Create_Task	Application_Initialize
NU_Sleep	Task_0_Entry, Task_3_and_4_Entry
NU_Set_Events	Task_0_Entry
NU_Send_To_Queue	Task_1_Entry
NU_Receive_From_Queue	Task_2_Entry
NU_Obtain_Semaphore	Task_3_and_4_Entry
NU_Current_Task_Pointer	Task_3_and_4_Entry
NU_Release_Semaphore	Task_3_and_4_Entry
NU_Retrieve_Events	Task_5_Entry
NU_Retrieve_Clock	Task_6_Entry

Results

Figure 5-1 shows the output of this example viewed in a serial terminal (like TeraTerm) while running on a hardware platform. Note that after running tasks 0 through 5, task 6 outputs the state of the system.

Figure 5-1. Kernel Demo Hardware Platform Output

```

Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window Help

*****
Nucleus PLUS 2.3 Demonstration
Build Timestamp - Jan 27 2012/08:38:58
*****

Task 0 Time:                0
Timer Interrupts:           126

Task 1 Messages Sent:       100

Task 2 Messages Received:    101
Task 2 Invalid Messages:     0

Task 3/4 Resource Owner:    Task 3

Task 5 Event Detections:     0

-----PRESS ANY KEY TO UPDATE-----

```


Chapter 6

Nucleus Kernel User's Manual

The intention of this manual is to give you an understanding of the Nucleus kernel operation, including configuration and initialization, and an insight into the core and supplemental Nucleus kernel components and their usage.

Kernel User Overview	441
Nucleus Kernel Operation.....	443
Nucleus System Configuration	443
Nucleus System Initialization	446
Nucleus Kernel Internals.....	453
Dynamic Memory	453
Partitioned Memory	454
Partitioned Memory	454
Interrupts	455
Tasks	458
Timers	462
Semaphores	464
Event Groups	467
Queues	468
Event Queue Manager	469
Mailboxes	471
Mailboxes	471
Signals	472

The intention of this manual is to give you an understanding of the Nucleus kernel operation, including configuration and initialization. It also offers an insight into the core and supplemental Nucleus kernel components and their usage.

Kernel User Overview

The Nucleus Kernel is the heart of the Nucleus Real-Time Operating System (RTOS). It consists of the core scheduling framework that enables multitasking and provides various OS primitives that enable applications to implement the following:

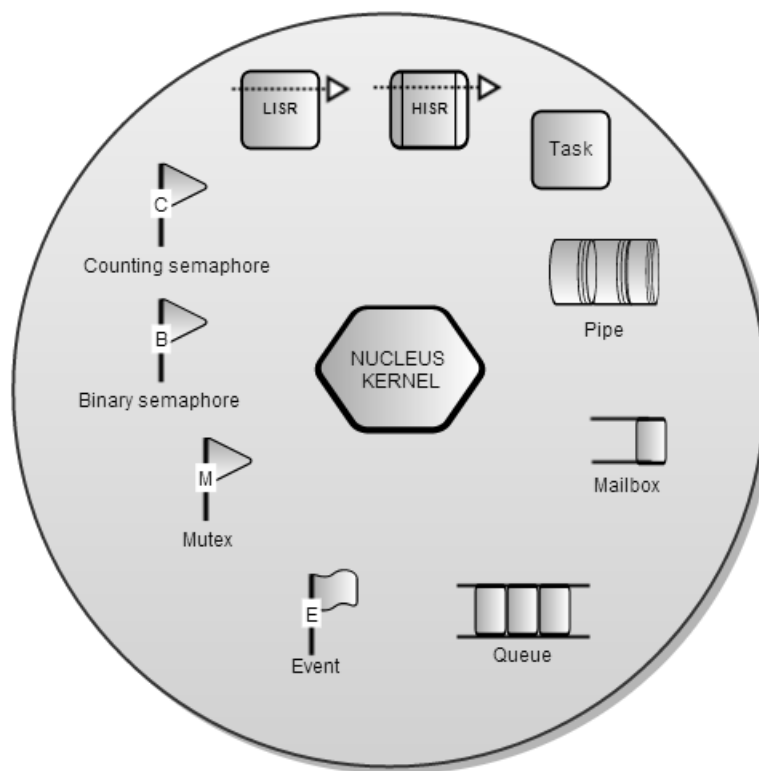
- inter-task communications and notifications
- synchronization
- protection by mutual exclusion

- system memory management
- interrupt management and service
- timing control

The Nucleus kernel scheduler is a priority-based, preemptive scheduler that provides a framework for real-time embedded applications to achieve their hard and soft deadlines. Approximately 99% of Nucleus PLUS is written in ANSI C, with the rest in assembly. Because of this, Nucleus PLUS is extremely portable and is currently available for use with most microprocessor families. [Figure 6-1](#) shows a top view of the Nucleus Kernel Components.

This document describes configuration, initialization, typical usage models, and insight into Nucleus Kernel internals.

Figure 6-1. Nucleus Kernel Components



Nucleus Kernel Operation

This section contains details about Nucleus system configuration and initialization.

Nucleus System Configuration

This section explains all levels of configuration needed for your application: kernel configuration (as well as the run-time library configuration), run-level configuration, and device manager configuration.

For more information about creating a custom configuration, see “Creating a Custom Configuration”, in the chapter “Getting Started with Nucleus ReadyStart Using Sourcery Code Bench”, in the *Nucleus ReadyStart Guide*.

Kernel Configuration

The Nucleus Kernel is partitioned into two components based on Kernel services:

- Core
- Supplementary

Most commonly used Kernel services are in the Core component, which includes:

- Tasks
- Timer
- LISR
- HISR
- Dynamic Memory
- Events
- Queues

The Supplementary Kernel services include:

- Mailbox
- Pipe
- Partition Memory

These are configured in on need basis. In addition to these two broad categories, the Kernel contains several configuration options to design suitable systems. These configurations allow Nucleus Kernel to be suitable for memory constraint processors or micro-controllers.

The Nucleus Kernel Core component is enabled with the following metadata option.

```
nu.os.kern.plus.core.enable = true;
```

The Nucleus Kernel Supplementary component is enabled with the following metadata option.

```
nu.os.kern.plus.supplement.enable = true;
```

A few key Nucleus Kernel Core options are shown in [Table 6-1](#).

Table 6-1. Nucleus Kernel Core Options

Option Name	Default Value	Description
ticks_per_sec	100	Number of OS ticks per second (one OS tick is 10ms)
num_taks_priorities	256	Number of task priorities allowed. Maximum is 256 and minimum is 8.
time_hisr_stack_size	512	Stack size for the Timer HISR.
stack_checking	false	Enable/Disable stack checking.
error_checking	true	Enable/Disable error checking for Nucleus PLUS API parameters.
rom_to_ram_copy	false	Enable/Disable toolset specific data copy from ROM to RAM.

Related Topics

[Nucleus System Configuration](#)

[Run-Time Library Configuration](#)

Run-Time Library Configuration

The Run-Time Library allocator functions allocate memory from the heap space, if the RTL component is enabled in the Nucleus Kernel. The heap space is dynamically allocated from Nucleus System_Memory pool at startup and is utilized for all RTL memory allocations. The RTL component is enabled with the following metadata option.

```
nu.kern.rtl.enable = true;
```

In case of Code Sourcery GNU toolset, the RTL allocator functions are mapped to Nucleus RTL allocator functions using the *--wrap* option. The Nucleus RTL allocator functions allocate memory from the metadata configurable memory pool and do not use common heap space. The [Table 6-2](#) shows the RTL memory allocation function mapping:.

Table 6-2. Nucleus RTL Allocation Functions

Toolset RTL Function	Nucleus RTL Function
malloc	RTL_malloc
calloc	RTL_calloc
realloc	RTL_realloc

Table 6-2. Nucleus RTL Allocation Functions (cont.)

Toolset RTL Function	Nucleus RTL Function
free	RTL_free

The RTL component metadata options are shown in [Table 6-3](#):

Table 6-3. RTL Component Options

Option Name	Default Value	Description
heap-size	512	Heap size in bytes. Allocated from Nucleus System_Memory pool.
malloc_pool	0	Nucleus memory pool used for malloc: <ul style="list-style-type: none">• 0 = System_Memory• 1 = Uncached_System_Memory• 2 = NU_Malloc_User_Mem_Pool (provided by user)

Related Topics

[Kernel Configuration](#)

Run-Level Configuration

There are no configuration options in Run-Level. The Nucleus system components (middleware and devices) within run-level define their run-level as a metadata option either in the *.metadata* file or in the BSP *.platform* file.

Related Topics

[Nucleus System Configuration](#)

[Device Manager Configuration](#)

Device Manager Configuration

The Nucleus system initialization is influenced by the Device Manager configuration. The Device Discovery thread can be configured -in or -out using the Device Manager metadata option. The Device Manager provides options to configure the Nucleus system components (middleware and devices) attribute.

The Device Manager is enabled with the following metadata option:

```
nu.os.kern.devmgr.enable = true;
```

A few key Device Manager options are shown in [Table 6-4](#).

Table 6-4. Device Manager Options

Option Name	Default Value	Description
discovery_task_enable	1	Enable/Disable device discovery task.
discovery_task_max_id_cnt	30	Maximum number of device IDs that can be returned by the Device Discovery Task.
discovery_task_stack_size	10240	Stack size of the Device Discovery Task.
max_dev_id_cnt	30	Maximum number of device IDs allowed in the system.
max_dev_label_cnt	5	Maximum number of labels allowed for each device.
max_dev_session_cnt	3	Maximum number of sessions allowed for each device.

Related Topics

[Nucleus System Configuration](#)

[Run-Level Configuration](#)

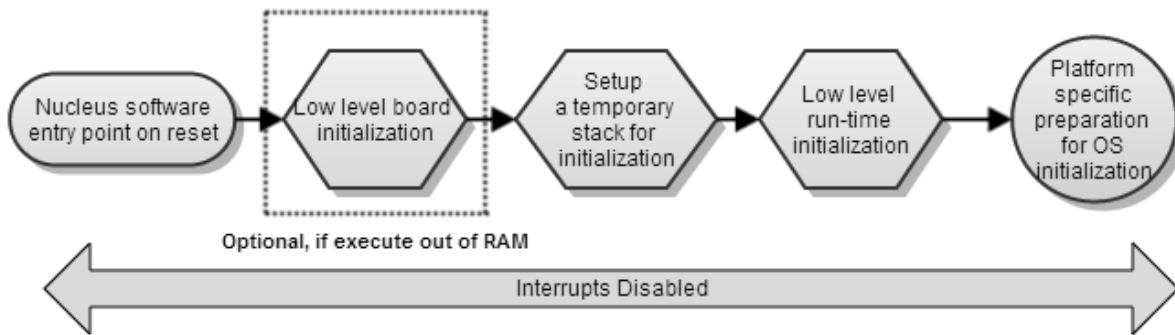
Nucleus System Initialization

This section explains all levels of initialization needed for your application: kernel initialization, run-level initialization, device discovery mechanism (as well as the relation between run-level and device discovery), and application initialization entry.

Kernel Initialization

The Nucleus Kernel is bare-metal and performs low level platform specific initialization at power reset. The Nucleus Kernel entry point is `ESAL_Entry`. In `ESAL_Entry`, low level board initialization is performed by setting up platform Clocks, PLLs, Memories, and so on. The low level board initialization is followed by setting up a temporary stack for initialization and a toolset-specific low level run-time environment. Most of these operations are written in assembly. The Nucleus Kernel also supports execution out of RAM in debug-mode. In that case, low level board initialization is performed through initialization scripts of the JTAG debug probe. [Figure 6-2](#) depicts the Nucleus Kernel initialization at power reset.

Figure 6-2. Nucleus Kernel Initialization at Power Reset

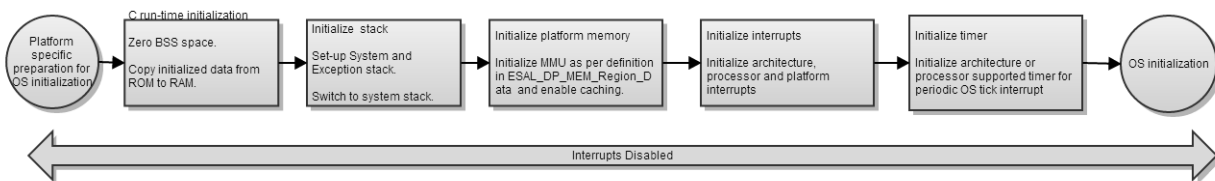


The Nucleus Kernel prepares the platform for OS initialization in `OS_Init_Entry`. The function performs several platform specific operations:

- clear BSS
- ROM-to-RAM data copy
- system/exception stacks setup
- initialize System memory
- enable caches/MMU
- initialize interrupts and timer for OS tick

The ROM-to-RAM data copy operation is executed if the Nucleus Kernel execution is out of ROM. In case of out-of-RAM execution (debug-mode), the data is loaded in RAM by the JTAG debug probe. These operations are written in ANSI C. [Figure 6-3](#) depicts the platform specific Nucleus Kernel initialization.

Figure 6-3. Platform Specific Nucleus Kernel Initialization



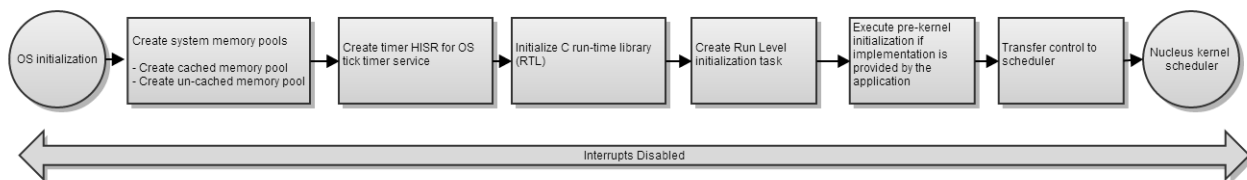
Once the platform specific initialization is complete, OS specific initialization is carried out in `INC_Initialize`, as shown in [Figure 6-4](#). The function performs several OS specific operations:

- create system memory pools (cached/un-cached) as defined in `ESAL_DP_MEM_Region_Data` table in the Board Support Package (BSP)

- create timer HISR to process OS timer service
- initialize toolset specific C run-time library
- create Run-Level initialization task
- execute pre-kernel initialization of application-supplied initialization function
- kick off the Nucleus Kernel scheduler

Until this point the system-wide interrupts are disabled. The interrupts are enabled by the Nucleus Kernel scheduler.

Figure 6-4. OS Specific Initialization



Related Topics

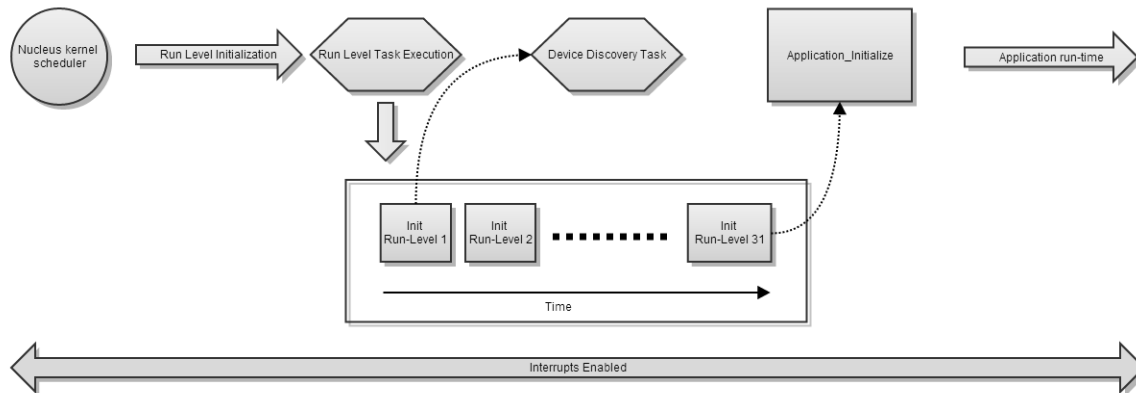
[Nucleus System Initialization](#)

[Run-Level Initialization](#)

Run-Level Initialization

The Nucleus system components are initialized within run-levels. There are 32 run-levels in Nucleus. A single run-level is a collection of one or more components that are initialized in no pre-determined order. The Nucleus Kernel scheduler initiates a Run-Level thread that enables all the run-level components, as shown in [Figure 6-5](#). This thread initializes all components at a given run-level before initializing components at the next run-level. The process starts at run-level 1 and continues through run-level 31. If a run-level has no components to initialize, processing immediately moves to the next run-level. The run-level 0 is not initialized by this thread. This run-level requires manual initialization from the application using the [NU_RunLevel_0_Init](#) API. The Run-Level thread is created with a auto-clean attribute and is removed from the system by the Nucleus Kernel once run-level initialization is complete.

Figure 6-5. Nucleus Scheduler Initialization



The [Table 6-5](#) shows the default settings of all 32 run-levels.

Table 6-5. Default Run-Level Settings

Run-Level	Devices	Description
0	None	Components in this run-level are not automatically initialized. They require manual initialization by applications using NU_RunLevel_0_Init API.
1	Device Manager	Initialize the device manager early because many other components have dependencies on it.
2-3	CPU Driver Reserved for BSP devices	Initialize platform devices if device discovery is disabled.
4	Power Management Syslogger	Power Management must have a task running to detect any drivers initialized in later run-levels
5	Buses (USB, SPI, I2C, SDIO, CAN, Serial)	Bus protocols must be initialized in an earlier run-level because they are used by many stacks.
6	File System	The file system is utilized by many other stacks and services.
7	Net Stack and USB class drivers	IP stack (IPv4 and IPv6) and base USB class drivers (function and host - hid, comm, ms).
8	USB drivers	USB user drivers like comm-ethernet, comm-modem, hid-keyboard, and hid-mouse.
9	Networking Protocols and Servers	Networking protocols (IPSec, SNMP, etc) and servers (webserv, telnet/FTP servers, etc) are initialized after the networking stack.

Table 6-5. Default Run-Level Settings (cont.)

Run-Level	Devices	Description
10-11	Reserved for BSP drivers	Initialize platform devices when using device discovery task.
12	C++ services	C++ initialization
13-15	UI, Debug Agent, Trace Service, Profiler, Shell	User Interface (Inflexion, Grafix RS, Input Management, etc) and OS services such as debug agent, POSIX core, profiler (14-15), and shell.
16-30	Application reserved	
31	Application initialize	Application initialization entry.

Note



Although the order of components within these run-levels is configurable, there are risks associated with changing the order of kernel components. These risks include the possibility of OS components not getting initialized or system failure and/or system crash.

The following rules are associated with the run-level initialization. You must understand and follow them for correct kernel initialization.

- Component initialization routines **cannot** block (sleep, suspend, and so on).
- Component initialization routines **cannot** have ordering dependencies on another component at the same run-level. There is no guaranteed order of initialization within a run-level.
- Component initialization routines can create threads that require blocking services (threads to wait for device registration, and so on).
- Run-level 0 components are **not** auto-initialized as part of the system initialization sequence.
- Components with no run-level designation are not placed in any run-level (that is, no *default* run-level for these components). This means no initialization is required or initialization occurs as part of the application's use of the component.
- Full system (kernel + application) functionality is only guaranteed when all run-levels have completed (that is all kernel services available). This excludes hot-swappable devices that are not present when run-level initialization completes.

Related Topics

[Nucleus System Initialization](#)

[Device Discovery Mechanism](#)

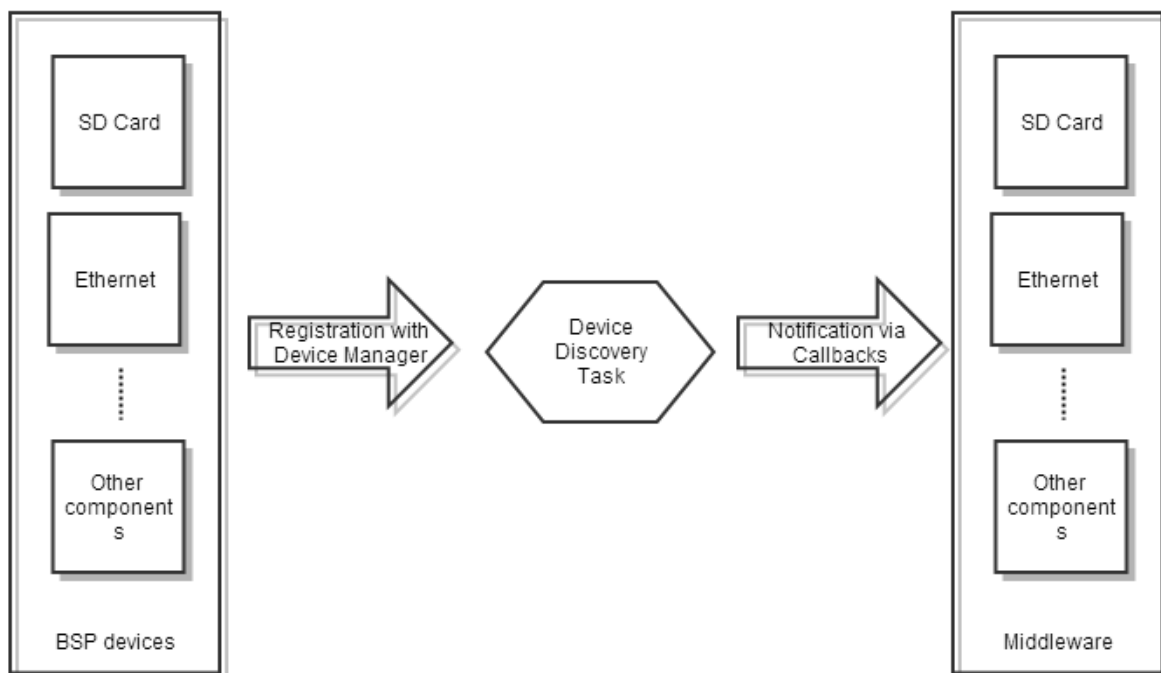
Device Discovery Mechanism

The Run-Level allows components to be inserted in the system, the Device Manager gets these components to connect with each other. The Device Manager is also one of the run-level components initialized at run-level 1. The Device Manager creates a Device Discovery thread during initialization. The components are either registering themselves ([DVC_Dev_Register](#)) or registering callbacks ([DVC_Reg_Change_Notify](#)) with the Device Manager. Usually, BSP devices are the components registering with the Device Manager, and middleware components are the components registering callbacks. The Device Discovery thread keeps waiting for BSP devices to register and invokes callback functions of the middleware components.

The Nucleus system allows a configuration option in the Device Manager to disable the Device Discovery thread. Once disabled, the Device Discovery thread never gets created and, instead of middleware components registering callbacks, [DVC_Reg_Change_Notify](#) probes the Device Manager for available BSP devices immediately and calls middleware component callbacks. In this mechanism, the callbacks are called from Run-Level thread context instead of Device Discovery thread context. Also, in this configuration BSP devices must be initialized at run-levels lower than their respective middleware components.

Figure 6-6 depicts the device discovery mechanism.

Figure 6-6. Device Discovery Mechanism



Run-Level and Device Discovery Relation

The Run-Level Initialization and the Device Discovery thread have some important properties:

- The Device Discovery thread is running at a much higher priority than the Run-Level Initialization thread and is non-preemptable.
- The Run-Level thread is preemptable. This implies that any device registering in the system via the Run-Level thread will be discovered by the Device Manager instantly (and notified to the client via a callback mechanism), by preempting the Run-Level thread.

Therefore, by the time the Run-Level thread completes, all the devices in the system are initialized and ready to use (except hot-swappable devices).

Completion of Run-Level initialization does not indicate initialization of all the components within the system. There may be threads created by the RunLevel initialization being suspended on events, and so on, due to hot-swappable devices (USB, SD Card, Ethernet, and so on) in the system which were not present during Run-Level initialization. The middleware components provide appropriate APIs for the availability of devices. For example, the FAT file system provides the `NU_Storage_Device_Wait` API to ensure that storage media has been inserted/mounted and ready to use. Similarly, the Networking stack provides the `NETBOOT_Wait_For_Network_Up` API for a network up event.

Caution



The only issue in the Run-Level / Device Discovery method is: if any device initialization ends up with failure status during Run-Level Initialization, it cannot be discovered by the Device Manager.

Related Topics

[Nucleus System Initialization](#)

[Application Initialization Entry](#)

Application Initialization Entry

The Nucleus application initialization function `Application_Initialize` is invoked at run-level 31. A typical Nucleus application allocates memory and creates kernel objects required by the application like Tasks, Semaphores, Queues, etc from within the `Application_Initialize` function.

The `Application_Initialize` function provides two parameters as input arguments. A pointer to cached memory and a pointer to un-cached system memory. These pointers may be used within the application initialize function to create application memory pools or allocate memory for kernel objects required by the application.

Related Topics

[Nucleus System Initialization](#)

[Device Discovery Mechanism](#)

Nucleus Kernel Internals

This section describes the Nucleus core and supplemental kernel components.

Dynamic Memory

A dynamic memory pool contains a user-specified number of bytes. The memory location of the pool is determined by the application. Variable-length allocation and deallocation services are provided for the dynamic memory pool. Allocations are performed in a first-fit manner. For example, the first available memory that satisfies the request is allocated. If the allocated block is significantly larger than the request, the unused memory is returned to the dynamic memory pool.

Each allocation from a memory pool requires some additional overhead to allow for its pointer structure. This overhead is consumed out of the memory pool from which the allocation is requested.

Dynamic Memory Suspension

The allocate dynamic memory service provides options for unconditional suspension, suspension with a time-out, and no suspension.

A task attempting to allocate dynamic memory from a pool that does not currently have enough available memory may suspend. Resumption of the task is possible when enough previously allocated memory is returned to the pool.

Multiple tasks may suspend on a single dynamic memory pool. Tasks are suspended in either FIFO or priority order, depending on how the dynamic memory pool was created. If the dynamic memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the dynamic memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Dynamic Memory Component Dynamic Creation

Nucleus PLUS dynamic memory pools are created and deleted dynamically. There is no preset limit on the number of dynamic memory pools an application may have. Each dynamic memory pool requires a control block and a pointer to the actual dynamic memory area. The memory for both the control block and the memory area is supplied by the application.

Dynamic Memory Determinism

Allocating memory from a dynamic memory pool is inherently undeterministic. This is largely due to possible memory fragmentation within the pool. The first-fit algorithm is basically a linear search and, as a result, the worst-case performance depends on the amount of fragmentation.

However, memory deallocation is constant. Processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the dynamic memory pool.

Dynamic Memory Pool Information

Application tasks may obtain a list of active dynamic memory pools. Detailed information about each dynamic memory pool is also available. This information includes the dynamic memory pool name, starting pool address, total size, free bytes, number of tasks suspended, and the identity of the first suspended task.

Related Topics

[Nucleus Kernel Internals](#)

[Partitioned Memory](#)

Partitioned Memory

Partition memory is ideal for tasks that require deterministic handling.

A partition memory pool contains a specific number of fixed size memory partitions. The memory location of the pool, the number of bytes in the pool, and the number of bytes in each partition are determined by the application. Individual partitions are allocated and deallocated from the partition memory pool.

Allocation from a memory pool requires some additional overhead to allow for its pointer structure.

Partition Memory Suspension

The allocate partition service provides options for unconditional suspension, suspension with a time-out, and no suspension.

A task attempting to allocate a partition from an empty pool can suspend. Resumption of that task is possible when a partition is returned to the pool.

Multiple tasks may suspend on a single partition memory pool. Tasks are suspended in either FIFO or priority order, depending on how the partition memory pool was created. If the partition memory pool supports FIFO suspension, tasks are resumed in the order in which they

were suspended. Otherwise, if the partition memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Partition Memory Dynamic Creation

Nucleus PLUS partition memory pools are created and deleted dynamically. There is no preset limit on the number of partition memory pools an application may have. Each partition memory pool requires a control block and a pointer to the memory area for the partition. The memory for both the control block and the partition area is supplied by the application.

Partition Determinism

Since searching is completely avoided, processing required for allocating and deallocating partitions is fast and constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the partition memory pool.

Partition Information

Application tasks may obtain a list of active partition memory pools. Detailed information about each partition memory pool is also available. This information includes:

- the partition memory pool name
- the starting pool address
- the total partitions
- the partition size
- the remaining partitions
- the number of tasks suspended
- the identity of the first suspended task

Related Topics

[Nucleus Kernel Internals](#)

[Dynamic Memory](#)

Interrupts

Interrupts provide Nucleus PLUS with a capability to respond to asynchronous, non-periodic events within the minimum specified time.

An interrupt is a mechanism for providing immediate response to an external or internal event. When an interrupt occurs, the processor suspends the current path of execution and transfers control to the appropriate Interrupt Service Routine (ISR). The exact operation of an interrupt is inherently processor-specific.

Nucleus PLUS supports managed ISRs. A managed ISR is one that does not need to save and restore context. Managed ISRs may be written in C or assembly language.

Interrupt Protection

Interrupts pose interesting problems for all real-time kernels. Nucleus PLUS is no exception. The main problem stems from the fact that ISRs need to have access to Nucleus PLUS services. On the surface this may not seem like a problem; however, it requires protection of data structures manipulated during a service call from simultaneous access by an ISR. The simplest method of protection is to lock out interrupts for the duration of the service. Responding to interrupts quickly is a cornerstone of real-time systems. Therefore, locking out interrupts to protect internal data structures is not desirable. Nucleus PLUS handles this protection problem by dividing application ISRs into low-level and high-level components.

Low-Level ISR

The Low-Level Interrupt Service Routine (LISR) executes as a normal ISR, which includes using the current stack. Nucleus PLUS saves context before calling a LISR and restores context after the LISR returns. Therefore, LISRs may be written in C and may call other C routines. However, there are only a few Nucleus PLUS services available to a LISR. If the interrupt processing requires additional Nucleus PLUS services, a High-Level Interrupt Service Routine (HISR) must be activated. Nucleus PLUS supports nesting of multiple LISRs. The following services are available from LISRs:

- NU_Activate_HISR
- NU_Current_HISR_Pointer
- NU_Current_Task_Pointer
- NU_Local_Control_Interrupts
- NU_Retrieve_Clock

High-Level ISR

The High-Level Interrupt Service Routine (HISR)s are created and deleted dynamically. Each HISR has its own stack space and its own control block. The memory for each is supplied by the application. Of course, the HISR must be created before it is activated by a LISR.

Since a HISR has its own stack and control block, it can be temporarily blocked if it tries to access a Nucleus PLUS data structure that is already being accessed.

HISRs are allowed access to most Nucleus PLUS services, with the exception of self-suspension services. Additionally, since a HISR cannot suspend on a Nucleus PLUS service, the “suspend” parameter must always be set to NU_NO_SUSPEND.

Note



“printf” calls are not allowed from LISR or HISR context.

There are three priority levels available to HISRs. If a higher priority HISR is activated during processing of a lower priority HISR, the lower priority HISR is preempted in much the same manner as a task gets preempted. HISRs of the same priority are executed in the order in which they were originally activated. All activated HISRs are processed before normal task scheduling is resumed.

An activation counter is maintained for each HISR. This counter is used to insure that each HISR is executed once for each activation.

Note



Each additional activation of an already active HISR is processed by successive calls to that HISR.

HISR Information

Application tasks may obtain a list of active HISRs. Detailed information about each HISR is also available. This information includes the HISR name, total scheduled count, priority, and stack parameters.

Interrupt Latency

Interrupt latency is a term that describes the amount of time for which interrupts are locked out. Since Nucleus PLUS does not rely on locking out interrupts to protect against simultaneous ISR access, interrupt latency is small and constant. In fact, interrupts are only locked out over several instructions in some Nucleus PLUS ports

Application Interrupt Lockout

Applications are provided with the ability to disable and enable interrupts. An interrupt locked out by the application remains locked out until the application unlocks it.

Direct Vector Access

Nucleus PLUS provides the ability to directly set up interrupt vectors. ISRs loaded directly into the vector table are required to save and restore registers used. Therefore, ISRs entered directly into the vector table are often written in assembly language. Such ISRs, providing certain conventions are followed, may activate a HISR.

Related Topics

[Nucleus Kernel Internals](#)

[Tasks](#)

Tasks

The Task Control services manage the execution of multiple competing tasks in a real-time application. A task is a semi-independent program segment with a dedicated purpose. Most modern real-time applications require multiple tasks. Additionally, these tasks often have varying degrees of importance.

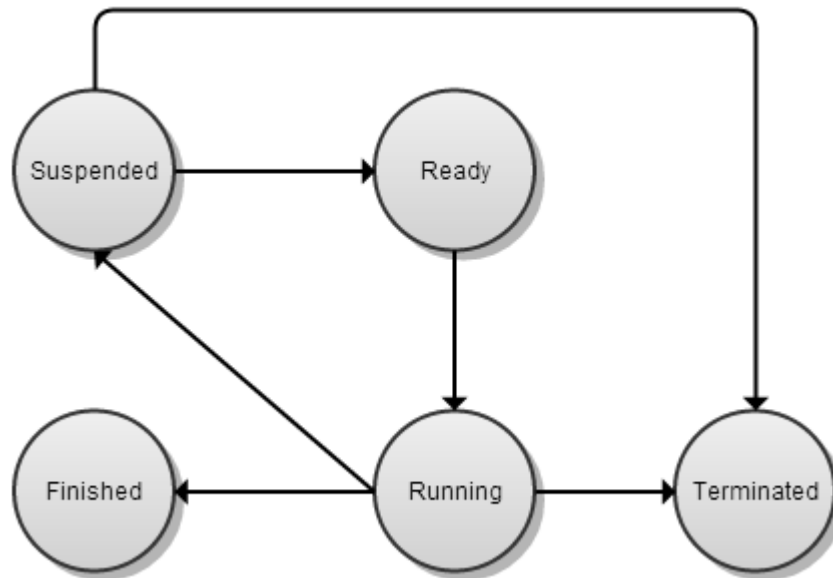
Tasks States

Each task is always in one of five states: running, ready, suspended, terminated, or finished. [Table 6-6](#) describes each of the task states.

Table 6-6. Task States

State	Description
Running	The task is currently running.
Ready	The task is ready, but another task is currently running.
Suspended	The task is dormant while waiting for the completion of a service request. When the request is complete, the task is moved to the ready state.
Terminated	The task was killed. Once in this state, the task cannot execute again until it is reset.
Finished	The task finished its processing and returned from initial entry routine. Once in this state, the tasks cannot execute again until it is reset.

The states and the transitions between states are shown in [Figure 6-7](#);

Figure 6-7. Task States and Transitions

Preemption

Preemption is the act of suspending a lower priority task when a higher priority task becomes ready. For example, suppose a task with a priority of 100 is executing. If an interrupt occurs that readies a task with a priority of 20, the task with priority 20 is executed before the interrupted task is resumed. Preemption also occurs when a lower priority task calls a Nucleus PLUS service that makes a higher priority task ready.

Preemption may be disabled on an individual task basis. When preemption is disabled, no other task is allowed to run until the executing task suspends, relinquishes control, or enables preemption. A task that suspends or relinquishes control with preemption disabled has preemption disabled when it is resumed.

A task is created with preemption either enabled or disabled. Preemption may also be enabled and disabled during task execution.

Relinquish

A mechanism is provided to share the processor with other ready tasks at the same priority level in a round-robin fashion. When a task requests this service, all other ready tasks at the same priority are executed before the originally executing task is resumed.

Time Slicing

Time slicing provides another mechanism to share the processor with tasks having the same priority. A time slice corresponds to the maximum number of timer ticks (timer interrupts) that can occur before all other ready tasks at the same priority level are given a chance to execute. A time slice behaves like an unsolicited task relinquish.

Note



Disabling preemption also disables time slicing.

Timer Interrupt

Nucleus PLUS requires a periodic interrupt in order to provide time-oriented services such as time-slicing, service call time-outs, and application timers. TMCT_Timer_Interrupt is the generic timer interrupt handler. Initialization and interrupt vector assignments are target specific.

Dynamic Task Creation

Nucleus PLUS tasks are created and deleted dynamically. There is no preset limit on the number of tasks an application may have. Each task requires a control block and a stack. The memory for each element is supplied by the application.

Task Determinism

Processing time associated with suspending and resuming tasks is a constant. It is not affected by the number of application tasks. Additionally, the method in which tasks execute is not only predictable, but also guaranteed. Higher priority, ready tasks execute before lower priority, ready tasks. Ready tasks of the same priority execute in the order they became ready.

Stack Checking

Application tasks may check the amount of memory left on the current stack. This function also keeps track of maximum stack usage. Stack checking may also be enabled inside Nucleus PLUS services through a conditional compilation option.

Task Information

Application tasks may obtain a list of active tasks. Detailed information about each task can also be obtained. This information includes the task name, current state, scheduled count, priority, and stack parameters.

Priority

A task's priority is defined during task creation. Additionally, dynamic modification of a task's priority is supported. A task that has a numerical priority of 0 has a higher priority than a task with a numerical priority of 255. Nucleus PLUS executes higher priority tasks before lower priority tasks. Tasks having the same priority are executed in the order in which they became ready for execution.

Caution



Care must be taken when assigning priorities to application tasks. If care is not taken, the priorities can cause task starvation and excessive system overhead.

A task may only execute if it is the highest priority, ready task. Therefore, if a task or tasks at a certain priority are always ready, all tasks of a lower priority never execute. This situation is called starvation. There are several cures for this. First, higher priority tasks should suspend to allow lower priority tasks to execute.

Tasks that run at or near continuously should have a relatively low priority. Another technique to combat starvation is to gradually raise the priority of the starving task.

A substantial amount of additional overhead may be incurred if task priorities are used improperly. Consider a system of three tasks named A, B, and C. Each task has similar processing that consists of waiting for a message and/or sending a message in an infinite loop.

Task A waits for a message from an Interrupt Service Routine (ISR) and, then, sends a message to task B. Task B waits for a message from task A and, then, sends a message to task C. Task C waits for a message from task B and, then, increments a counter. After this simple system starts (regardless of priority), all tasks execute briefly, and then suspend waiting for a message.

If all of the tasks have the same priority, the following set of events take place after the ISR sends a message to task A:

- a. Task A is resumed
- b. Task A sends a message to task B, making task B ready
- c. Task A suspends waiting for another message
- d. Task B is resumed
- e. Task B sends a message to task C, making task C ready
- f. Task B suspends waiting for another message
- g. Task C is resumed
- h. Task C increments a counter
- i. Task C suspends waiting for another message

Now assume that task A is lower priority than task B and task B is lower priority than task C.

The following events take place after the ISR sends a message to task A:

- a. Task A is resumed
- b. Task A sends a message to task B, making task B ready
- c. Task A relinquishes to higher priority task B
- d. Task B is resumed
- e. Task B sends a message to task C, making task C ready
- f. Task B relinquishes to higher priority task C
- g. Task C is resumed
- h. Task C increments a counter
- i. Task C suspends waiting for another message
- j. Task B is resumed again
- k. Task B suspends waiting for another message
- l. Task A is resumed again
- m. Task A suspends waiting for another message

The application work performed in both of the previous examples is the same, that is, two tasks sent messages and three tasks received messages. However, the amount of system overhead in resuming and suspending tasks doubled. Also notice the delay incurred in task A between sending a message and waiting for another message in the last example.

Obviously the previous example systems are useful only to show how priorities can affect system overhead. Different priorities are necessary for real-time applications to respond to external events and to allocate processing time to relatively more important tasks. However, in order to reduce unnecessary system overhead, the number of different priorities in an application should be minimized.

Related Topics

[Nucleus Kernel Internals](#)

[Interrupts](#)

Timers

The timer component provides timer services to Nucleus PLUS tasks and applications.

Most real-time applications require processing on periodic intervals of time. Each Nucleus PLUS task has a built-in timer. This timer is used to provide task sleeping and service call suspension time-outs.

Ticks

A tick is the basic unit of time for all Nucleus PLUS timer facilities. Each tick corresponds to a single hardware timer interrupt. The amount of actual time a tick represents is usually user-programmable.

Margin of Error

A timer request may be satisfied as much as one tick early in actual time. This is because a tick can occur immediately after the timer request. Therefore, the first tick of a timer request represents an actual time ranging from zero to the rate of the hardware timer interrupt. For example, the amount of actual time expired for a request of n ticks falls between the actual time n and $n-1$ ticks represent.

Hardware Requirement

Nucleus PLUS timer services require a periodic timer interrupt from the hardware. The hardware timer interrupt is required for the timer facilities to function. However, other Nucleus PLUS facilities are not affected by the absence of timer facilities.

Continuous Clock

Nucleus PLUS maintains a continuous counting tick clock. The maximum value of this clock is 4,294,967,294. The clock automatically resets on the tick after the maximum value is reached. This continuous clock is reserved exclusively for application use. It may be read from and written to by the application at any time.

Application Timers

Nucleus PLUS provides programmable timers for applications. These timers execute a specific user-supplied routine when they expire. The user-supplied expiration routine executes as a high-level interrupt service routine. Therefore, self-suspension requests are not allowed. Additionally, processing should be kept to a minimum.

Re-Scheduling

When a timer expires, the prescribed expiration routine is executed. After execution is complete, the timer is either dormant or rescheduled. If the timer's reschedule value is 0, it is dormant after the initial expiration. However, if the timer's reschedule value is non-zero, it is rescheduled to expire at that interval.

Enable/Disable

Application timers may be automatically enabled during creation. Additionally, timers may be enabled and disabled dynamically.

Reset

The initial ticks, rescheduling rate, and the expiration routine of a timer may be reset dynamically by the application.

Timer Dynamic Creation

Nucleus PLUS application timers are created and deleted dynamically. There is no preset limit on the number of timers an application may have. Each timer requires a control block. The memory for this is supplied by the application.

Timer Determinism

Processing time required to create, enable, disable, and modify application timers is constant. However, processing time required to execute the user-supplied expiration routines depends on the expiration routines themselves and the number of timers that expire simultaneously.

Timer Information

Application tasks may obtain a list of active timers. Detailed information about each timer is also available. This information includes:

- the timer name
- the timer status
- the initial ticks
- the reschedule value
- the remaining ticks
- the expiration count

Related Topics

[Nucleus Kernel Internals](#)

[Tasks](#)

Semaphores

Semaphores synchronize tasks and provide a mechanism to control execution of critical sections of an application. Nucleus PLUS provides counting semaphores that range in value from 0 to 4,294,967,294. The two basic operations on a semaphore are obtain and release. Obtain-

semaphore requests decrement the semaphore, while release-semaphore requests increment the semaphore.

Resource allocation is the most common application of a semaphore. Semaphores created with an initial value can be used to indicate an event.

In addition to counting semaphores, Nucleus also provides semaphores with Priority Inheritance features, or Mutexes. These behave in a binary-semaphore like manner with possible counts of 0 or 1 only.

Mutexes use the same set of semaphore APIs extended to support Priority Inheritance and can be used to prevent priority inversion and avoid deadlock situations.

Semaphore Suspension

The obtain-semaphore service provides options for unconditional suspension, suspension with a time-out, and no suspension.

A task attempting to obtain a semaphore whose count is currently zero can suspend. Resumption of the task is possible when a release-semaphore request is made.

Multiple tasks may suspend trying to obtain a single semaphore. Tasks are suspended in either FIFO or priority order, depending on how the semaphore was created. If the semaphore supports FIFO suspension, tasks are resumed in the order in which they tried to obtain the semaphore. Otherwise, if the semaphore supports priority suspension, tasks are resumed from high priority to low priority.

Deadlock

A deadlock refers to a situation where two or more tasks are forever suspended attempting to obtain two or more semaphores. The simplest example of this situation is a system with two tasks and two semaphores. Suppose the first task has the second semaphore and the second task has the first semaphore. Now, suppose that the second task attempts to obtain the second semaphore and the first task attempts to obtain the first semaphore. Since each task has the semaphore that the other needs, the tasks could suspend on the semaphores forever.

Prevention is the preferred way to deal with deadlocks. This technique imposes rules on how semaphores are used by the application. For example, if tasks are not allowed to possess more than one semaphore at a time, deadlocks are prevented. Alternatively, deadlocks may be prevented if tasks obtain multiple semaphores in the same order.

Semaphores that support Priority Inheritance features/Mutexes can be used to prevent deadlock situations. The optional time-out on obtain-semaphore suspension can also be used to recover from a deadlock situation.

Priority Inversion

Priority inversion occurs when a higher priority task is suspended on a semaphore that a lower priority task has. This situation is unavoidable if different priority tasks share the same protected resources. In such situations, a limited and predictable amount of time in priority inversion is acceptable.

However, if the low priority task is preempted by a middle priority task during a priority inversion situation, the amount of time in priority inversion is no longer deterministic. Such a situation can be avoided by insuring that all tasks using the same counting semaphore have the same priority, at least while they own the semaphore.

Alternatively, Priority Inheritance semaphores/Mutexes can be used to prevent priority inversion situations. These allow a lower priority task to inherit the priority of a higher priority task blocked on a priority inheritance mutex owned by the low priority task.

Semaphore Dynamic Creation

Nucleus PLUS semaphores are created and deleted dynamically. There is no preset limit on the number of semaphores an application may have. Each semaphore requires a control block. The memory for the control block is supplied by the application. Counting semaphores can be created with any initial count. In contrast, Priority Inheritance semaphores must be created with an initial count of 1.

Semaphore Determinism

Processing time required for obtaining and releasing semaphores is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the semaphore.

Semaphore Information

Application tasks can obtain a list of active semaphores. Detailed information about each semaphore is also available. This information includes the semaphore name, current count, suspension type, number of tasks waiting, and the first task waiting.

In the case of Priority Inheritance semaphores/mutexes, information about the owner can be obtained by using the [NU_Get_Semaphore_Owner](#) API.

Related Topics

[Nucleus Kernel Internals](#)

[Queues](#)

Event Groups

Event groups, much like semaphores, synchronize tasks and provide 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. Event flags can be set and cleared using logical AND/OR combinations. Event flags can be received in logical AND/OR combinations as well. Additionally, event flags may be reset automatically after they are received.

Event Group Suspension

The receive event flag requests provide options for unconditional suspension, suspension with a time-out, and no suspension.

A task attempting to receive a combination of event flags that are not present can suspend. Resumption of the task occurs when a set-event-flags operation satisfies the combination of events requested by the task.

Multiple tasks may suspend trying to receive different combinations of event flags from the same event group. All tasks suspended on an event group are checked for resumption when a set-event-flags operation is performed on the event group.

Event Group Dynamic Creation

Nucleus PLUS event groups are created and deleted dynamically. There is no preset limit on the number of event groups an application may have. Each event group requires a control block. The memory for the control block is supplied by the application.

Event Group Determinism

Processing time required for receiving event flags from an event group is constant. However, the processing time required to set event flags in an event group is affected by the number of tasks suspended on the event group.

Event Group Information

Application tasks may obtain a list of active event groups. Detailed information about each event group is also available. This information includes the event group name, current event flags, number of tasks waiting, and the first task waiting.

Related Topics

[Nucleus Kernel Internals](#)

[Event Queue Manager](#)

Queues

Queueing is a method of communication between tasks intended for multiple messages of fixed or variable length.

Queues provide a mechanism to transmit multiple messages. Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue. Messages may be placed at the front of the queue or at the back of the queue.

Queue Message Size

A queue message consists of one or more 32-bit words. Both fixed and variable-length messages are supported. The type of message format is defined when the queue is created. Variable-length message queues require an additional 32-bit word of overhead for each message in the queue. Additionally, receive message requests on variable-length message queues specify the maximum message size. The same requests on fixed-length message queues specify the exact message size.

Queue Suspension

Send and receive queue services provide options for unconditional suspension, suspension with a time-out, and no suspension.

Tasks may suspend on a queue for several reasons. A task attempting to receive a message from an empty queue can suspend. Additionally, a task attempting to send a message to a full queue can suspend. A suspended task is resumed when the queue is able to satisfy that task's request. For example, suppose a task is suspended on a queue waiting to receive a message. When a message is sent to the queue, the suspended task is resumed.

Multiple tasks may suspend on a single queue. Tasks are suspended in either FIFO or priority order, depending on how the queue was created. If the queue supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the queue supports priority suspension, tasks are resumed from high priority to low priority.

Queue Broadcast

A queue message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the queue are given the broadcast message.

Queue Dynamic Creation

Nucleus PLUS queues are created and deleted dynamically. There is no preset limit on the number of queues an application may have. Each queue requires a control block and a queue data area. The memory for each is supplied by the application.

Queue Determinism

Basic processing time required for sending and receiving queue messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the queue.

Queue Information

Application tasks may obtain a list of active queues. Detailed information about each queue can also be obtained. This information includes the queue name, message format, suspension type, number of messages present, and the first task waiting.

Related Topics

[Nucleus Kernel Internals](#)

[Event Queue Manager](#)

Event Queue Manager

The Event Queue Manager (EQM) facilitates communications between components via events that are dispatched globally and can also carry data.

EQM shares data through a global array of event data. This array is operated in circular fashion. Events are sent to the waiting components. When an event is received it is stored in the EQM circular buffer and a notification is sent to all the waiting components. The event will be sent to only those components that are waiting for it. This architecture also handles the case when events will not be sent to components that are currently not in a waiting state and are busy processing any received event. This is handled with the provision of receiving the relevant event that has occurred after the last processed event.

Related Topics

[Nucleus Kernel Internals](#)

[Event Groups](#)

Pipes

Pipes is a method of communication between tasks similar to the queues.

Pipes provide a mechanism for transmitting multiple messages. Messages are sent and received by value. A send-message request copies the message into the pipe, while a receive-message request copies the message out of the pipe. Messages may be placed at the front of the pipe or at the back of the pipe.

Pipe Message Size

A pipe message consists of one or more bytes. Both fixed-length and variable-length messages are supported. The type of message format is defined when the pipe is created. Variable-length message pipes require an additional 32-bit word of overhead for each message in the pipe. Additionally, receive-message requests on variable-length message pipes specify the maximum message size, while the same request on fixed-length message pipes specify the exact message size.

Pipe Suspension

Send and receive pipe services provide options for unconditional suspension, suspension with a time-out, and no suspension.

Tasks may suspend on a pipe for several reasons. Tasks attempting to receive a message from an empty pipe can suspend. Also, a task attempting to send a message to a full pipe can suspend. A suspended task is resumed when the pipe is able to satisfy that task's request. For example, suppose a task is suspended on a pipe waiting to receive a message. When a message is sent to the pipe, the suspended task is resumed.

Multiple tasks may suspend on a single pipe. Tasks are suspended in either FIFO or priority order, depending on how the pipe was created. If the pipe supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the pipe supports priority suspension, tasks are resumed from high priority to low priority.

Pipe Broadcast

A pipe message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the pipe are given the broadcast message.

Pipe Dynamic Creation

Nucleus PLUS pipes are created and deleted dynamically. There is no preset limit on the number of pipes an application may have. Each pipe requires a control block and a pipe data area. The memory for each is supplied by the application.

Pipe Determinism

Basic processing time required for sending and receiving pipe messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the pipe.

Pipe Information

Application tasks may obtain a list of active pipes. Detailed information about each pipe can also be obtained. This information includes:

- the pipe name,
- the message format,
- the suspension type,
- the number of messages present
- the first task waiting

Related Topics

[Nucleus Kernel Internals](#)

[Mailboxes](#)

Mailboxes

Mailboxes are a method of communication between tasks, intended for short messages.

Mailboxes provide a low-overhead mechanism to transmit simple messages. Each mailbox is capable of holding a single message the size of four 32-bit words. Messages are sent and received by value. A send message request copies the message into the mailbox, while a receive message request copies the message out of the mailbox.

Mailbox Suspension

Send and receive mailbox services provide options for unconditional suspension, suspension with a time-out, and no suspension.

Tasks can suspend on a mailbox for several reasons. A task attempting to receive a message from an empty mailbox can suspend. Also, a task attempting to send a message to a non-empty mailbox can suspend. A suspended task is resumed when the mailbox is able to satisfy that task's request. For example, suppose a task is suspended on a mailbox waiting to receive a message. When a message is sent to the mailbox, the suspended task is resumed.

Multiple tasks can suspend on a single mailbox. Tasks are suspended in either FIFO or priority order, depending on how the mailbox was created. If the mailbox supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the mailbox supports priority suspension, tasks are resumed from high priority to low priority.

Mailbox Broadcast

A mailbox message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the mailbox are given the broadcast message.

Mailbox Dynamic Creation

Nucleus PLUS mailboxes are created and deleted dynamically. There is no preset limit on the number of mailboxes an application may have. Each mailbox requires a control block. The memory for the control block is supplied by the application.

Mailbox Determinism

Processing time required for sending and receiving mailbox messages is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the mailbox.

Mailbox Information

Application tasks may obtain a list of active mailboxes. Detailed information about each mailbox can also be obtained. This information includes the mailbox name, suspension type, whether a message is present, and the first task waiting.

Related Topics

[Nucleus Kernel Internals](#)

[Signals](#)

Signals

Signals are in some ways similar to event flags. However, there are significant differences in operation. Event flag usage is synchronous by nature. The task does not recognize event flags are present until the specific service request is made. Signals operate in an asynchronous manner. When a signal is present, a special signal handling routine, previously designated by the task, is executed when the task is resumed. Each task is capable of handling 32 signals. Each signal is represented by a single bit.

Signal Handling Routine

The task's signal-handling routine must be supplied before any signals are processed. Processing inside a signal-handling routine has virtually the same constraints as a high-level interrupt service routine. Basically, most Nucleus PLUS services are available, provided self-suspension is avoided.

Enable Signal Handling

By default, tasks are created with all signals disabled. Individual signals may be enabled and disabled dynamically by each task.

Clearing Signals

Signals are automatically cleared when signal handling is invoked. Additionally, signals are cleared when a solicited request to receive signals is made.

Multiple Signals

Signals for a task are cleared once the signal handling routine is started. Signal handling routines are not interrupted by new signals. Processing of any new signals takes place after the current signal processing completes. Identical signals sent before the first signal is recognized are discarded.

Signal Determinism

Processing time required to send and receive signals is constant, at least in the worst case. Of course, the time required to execute a signal-handling routine is application-specific.

Related Topics

[Nucleus Kernel Internals](#)

[Queues](#)

Nucleus Processes Overview

The Nucleus Process model design incorporates dynamic loading of binary images into a running Nucleus OS system with optional memory management to provide a light-weight process model.

Features of the Nucleus Processes model include:

- Processes can be isolated from each other and the OS for improved system reliability. The level of isolation depends on configuration settings and availability of supporting hardware, such as a MMU.
- Processes can export symbols (functions) for use by other processes or application code. This allows a process to act as a loadable library, dynamically adding functionality to a running system.
- Simple and light-weight design provides efficient use of target resources and high performance

Limitations of the Nucleus Processes model include:

- Nucleus Processes must be loaded from file system on the target.
- Nucleus Processes cannot be loaded to user-supplied addresses. All processes are currently loaded into system-specified, dynamically allocated memory.
- Processes, by default, have access to a limited set of Nucleus OS APIs (for example, only those exported by the Nucleus OS).

Process IDs and Information

Every process in the system has a unique ID by which it is referenced. The process ID is assigned when a process is loaded, with the exception of the Kernel Process, which is always present and always has an ID of zero. In addition to an ID, processes in the system have various characteristics based on how they were loaded. Some important characteristics include the name and load address (base address where process is loaded in target memory). An information API is provided to get the current number of processes in the system as well as information about them, including the process ID. This information may be used in conjunction with other APIs to control processes in the system, for example, getting a list of all processes and stopping them during an emergency stop sequence.

Related Topics

[Nucleus Processes Overview](#)

[Process Initialization and Termination](#)

Process Initialization and Termination

When a process is loaded, memory resources for the process are allocated and the process root thread is created. The loading process transitions the process into a stopped state. Initialization of the process occurs when the process is started. All initialization actions occurs in the context of the process root thread. The initialization activities include the invocation of C++ language constructors for static objects in the process and the execution of the process entry function, [main](#). The process entry function provides a place for initialization code that is guaranteed to be executed first when a process is started.

When [main](#) returns, the returned value determines the behavior of the process. [Table 7-1](#) shows the exit codes that can be returned by a process via [main](#) exiting, by a call to `exit()` at process run-time, or by other activity in the system.

Table 7-1. Process Exit Codes

Exit Code Name	Description
EXIT_SUCCESS	When returned by main , this exit code indicates successful exit and the process is placed into a stopped state.
EXIT_CONTINUE	When returned by main , this exit code indicates that the process will continue to run after exiting main (unlike <code>EXIT_SUCCESS</code>). This exit code can be used to allow threaded processes to continue running after exiting main or libraries to continue running.
EXIT_FAILURE	When returned by main , this exit code indicates that an error of some type occurred during execution of main and the process is placed into a stopped state.
EXIT_ABORT	Process exit code set when abort is called within a process context. The process is placed in a stopped state.
EXIT_FPU_ERROR	Process exit code set when an FPU error occurs in the context of a process. The process is placed in a stopped state.
EXIT_ILLEGAL_INST	Process exit code set when an illegal instruction is executed in the context of a process. The process is placed in a stopped state.
EXIT_KILL	Process exit code set when a process is killed. The process is placed in a stopped state and then unloaded.
EXIT_BUS_ERROR	Process exit code set when a bus error occurs in the context of a process. The process is placed in a stopped state.

Table 7-1. Process Exit Codes (cont.)

Exit Code Name	Description
EXIT_INVALID_MEM	Process exit code set when invalid memory access occurs within the context of a process. The process is placed in a stopped state.
EXIT_STOP	Process exit code set when a process is stopped by another process. The affected process is placed in a stopped state.

Note

The [main](#) function is optional and, when not present in the user process code, results in an internal default version to be used which always returns the EXIT_CONTINUE value. This default [main](#) facilitates the creation of library processes which simply contains code and does not necessarily need a [main](#) function to initialize the library.

When a process is stopped the C++ destructors are called for any static objects in the process, as well as any de-initialization functions registered through the [atexit](#) API function. The ability to register de-initialization functions provides an optional way to perform clean-up of any resources allocated by the process. If the process is stopped by use of the [abort](#) API, the de-initialization functions will not be called.

Related Topics

[Nucleus Processes Overview](#)[Process IDs and Information](#)

Processes Data Structures

This section describes the following data structures used by Nucleus processes:

- [NU_LOAD_EXTENSION](#)
- [NU_PROCESS_INFO](#)
- [NU_PROCESS_EXCEPTION](#)

NU_LOAD_EXTENSION

NU_LOAD_EXTENSION is an optional parameter that may be provided to the [NU_Load](#) function to set specific characteristics of the loaded process. The structure is defined as follows:

```
typedef struct
{
    UNSIGNED    heap_size;
    UNSIGNED    stack_size;
    BOOLEAN     kernel_mode;
} NU_LOAD_EXTENSION;
```

The members of the structure are defined in [Table 7-2](#):

Table 7-2. NU_LOAD_EXTENSION

Member	Description
heap_size	Size (in bytes) of the total memory resource available to the process (the memory heap).
stack_size	Size (in bytes), of the process root thread's stack.
kernel_mode	Indicates if the process is loaded in kernel-mode or user-mode.

Note



Only loading to user-mode is currently supported.

Related Topics

[Processes Data Structures](#)

[NU_Load](#)

NU_PROCESS_INFO

NU_PROCESS_INFO is a structure used to return information about a process. The structure is defined as follows:

```
typedef struct
{
    INT            id;
    CHAR           name[PROC_NAME_LENGTH];
    INT            state;
    VOID           *load_addr;
    VOID           (*entry_addr)(INT, VOID *);
    INT            exit_code;
    BOOLEAN        kernel_mode;
} NU_PROCESS_INFO;
```

The members of this structure are defined as in [Table 7-3](#):

Table 7-3. NU_PROCESS_INFO

Member	Description
id	The process ID.
name	The name of the process.
state	The current state value of the process.
load_addr	The load address of the process.
entry_addr	The address of the process entry (main) function.

Table 7-3. NU_PROCESS_INFO

Member	Description
exit_code	The exit code of the process entry (main), function once the process main function has exited.
kernel_mode	Indicates if the process is running in user-mode or kernel-mode.

Related Topics

[Processes Data Structures](#)[NU_Load](#)

NU_PROCESS_EXCEPTION

NU_PROCESS_EXCEPTION is a structure passed to an exception handler containing details about the exception which occurred during process execution. The structure is defined as follows:

```
typedef struct
{
    INT          pid;
    NU_TASK      *task;
    VOID         *address;
    VOID         *return_address;
    UNSIGNED     type;
    BOOLEAN      interrupt_context;
    BOOLEAN      kernel_process;
    VOID         *exception_information;
} NU_PROCESS_EXCEPTION;
```

The members of this structure are defined as in [Table 7-4](#).

Table 7-4. NU_PROCESS_EXCEPTION

Member	Description
pid	ID of the process on which the exception occurred.
task	Pointer to the offending thread.
address	Address where exception occurred.
return_address	Point of exception and return address.
type	Type of error that occurred. See Table 7-1 for information on possible values.
interrupt_context	Boolean indicating if exception occurred in an interrupt context.
kernel_process	Boolean indicates if exception occurred in a kernel-mode process.

Table 7-4. NU_PROCESS_EXCEPTION

Member	Description
exception_information	Pointer to additional, architecture-specific information such as a stack frame, exception specific bit masks, etc.

Related Topics

[Processes Data Structures](#)

[Process_Exception_Handler](#)

Processes APIs

This section describes the following APIs:

- [NU_Load](#)
- [NU_Start](#)
- [NU_Stop](#)
- [NU_Unload](#)
- [NU_Kill](#)
- [NU_EXPORT_SYMBOL](#)
- [NU_Symbol](#)
- [NU_Symbol_Close](#)
- [NU_Getpid](#)
- [NU_Get_Exit_Code](#)
- [NU_Established_Processes](#)
- [NU_Processes_Information](#)
- [NU_Process_Information](#)
- [NU_Memory_Map](#)
- [NU_Memory_Unmap](#)
- [NU_Memory_Get_ID](#)
- [NU_Memory_Share](#)
- [Process_Exception_Handler](#)
- [main](#)
- [atexit](#)

- [exit](#)
- [abort](#)

NU_Load

Load an ELF image from the target file system to a dynamically allocated memory location.

Note



The name of the process will be the name of the ELF file, derived from the specified path.

Usage

```
STATUS NU_Load (CHAR      *name,  
                INT       *pid,  
                VOID      *load_addr,  
                VOID      *extension,  
                UNSIGNED   suspend);
```

Arguments

- **name**
Pointer to the name of the image including path information used to determine where to load the image (ELF format) from in the target file system.
- **pid**
On return, it points to the process ID of the loaded image.
- **load_addr**
Pointer to memory where the process should be loaded or a predefined value. Valid value for this parameter is:

 NU_LOAD_DYNAMIC

Note



Currently only loading to dynamically allocated memory is supported. The load_addr value must be NU_LOAD_DYNAMIC.

- **extension**
A pointer to a structure of type [NU_LOAD_EXTENSION](#) which provides a means to set specific process parameters for the loaded processes. If left as NULL, default values are used for the loaded process.
- **suspend**
Allow for a suspension:

 NU_SUSPEND waits for the operation to complete. Time-out is 1-0xFFFFFFFF.
 NU_NO_SUSPEND returns immediately while the operation completes from the context of another thread.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_UNAVAILABLE**
The specified base address is invalid.
- **NU_NOT_PRESENT**
Indicates the image contains a reference to a symbol not present in the system.
- **NU_INVALID_POINTER**
The (file) name or return pointer for the ID specified is invalid.
- **NU_INVALID_STATE**
The process is not in the correct state to be loaded.
- **NU_INVALID_OPERATION**
Indicates an attempt to load a process in kernel-mode from a user-mode process (not allowed).

Examples

```
STATUS status;
INT     pid;
.
.
.
/* Load the process from the process image file (.load) found on the
target file system to dynamically allocated memory with no optional
parameters, suspending until the operation completes. The process ID will
be returned in the pid parameter. */
status = NU_Load("A:\\process.load", &pid, NU_LOAD_DYNAMIC, NU_NULL,
                NU_SUSPEND);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Processes APIs](#)[NU_Unload](#)

NU_Start

Completes the initialization of a process and starts execution of the root thread. This includes finishing all memory setup (clearing BSS, copying initialized data, and so on) and running the root thread to call *main()*.

Usage

```
STATUS NU_Start (INT      pid,  
                 VOID     *args,  
                 UNSIGNED suspend);
```

Arguments

- **pid**
ID of the process to start.
- **args**
Pointer to arguments that are passed to the process. (Note: The args parameter is not currently used.)
- **suspend**
Allow for a suspension:

NU_SUSPEND waits for the operation to complete. Time-out is 1-0xFFFFFFFFE.
NU_NO_SUSPEND returns immediately while the operation completes from the context of another thread.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_PROCESS**
The process ID specified is not valid.
- **NU_INVALID_STATE**
The process is in an invalid state (for example, it is not in a stopped state).

Examples

```
STATUS status;  
INT      pid;  
.  
.  
.  
/* Starts the loaded process with process ID of pid, waiting for the  
operation to complete. */  
status = NU_Start(pid, NU_NULL, NU_SUSPEND);  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Stop](#)

NU_Stop

Stops a running process, performing any termination activities needed by the process (for example terminate and delete all tasks within the process, run C++ static object destructors, call de-init function, etc). Essentially, the process is set into the same state it was in immediately after loading ([NU_Load](#)).

Usage

```
STATUS NU_Stop (INT      pid,  
                INT      exit_code,  
                UNSIGNED suspend);
```

Arguments

- **pid**
ID of the process to stop.
- **exit_code**
Value set as the exit code for the process.
- **suspend**
Allow for a suspension:

NU_SUSPEND waits for the operation to complete. Time-out is 1-0xFFFFFFFF.
NU_NO_SUSPEND returns immediately while the operation completes from the context of another thread.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_PROCESS**
The process ID specified is not valid.
- **NU_INVALID_STATE**
The specified process is in an invalid state to stop (for example, it is not in a started state).
- **NU_SYMBOLS_IN_USE**
The specified process cannot be stopped since symbols it has exported are currently in use by other processes.
- **NU_MEMORY_IS_SHARED**
The specified process cannot be stopped since it currently has shared memory regions with other processes.

Examples

```
STATUS status;
INT      pid;
.
.
.
/* Stops the running process with process ID of pid, waiting for the
operation to complete. */
status = NU_Stop(pid, 0, NU_SUSPEND);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Start](#)

NU_Unload

Unloads a stopped process and frees any resources allocated during the loading process.

Usage

```
STATUS NU_Unload(INT      pid,  
                  UNSIGNED suspend);
```

Arguments

- **pid**
ID of the process to stop.
- **suspend**
Allow for a suspension:

NU_SUSPEND waits for the operation to complete. Time-out is 1-0xFFFFFFFFE.
NU_NO_SUSPEND returns immediately while the operation completes from the context of another thread.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_PROCESS**
The process ID specified is not valid.
- **NU_INVALID_STATE**
The specified process is in an invalid state to unload (for example, it is running).

Examples

```
STATUS status;  
INT      pid;  
.  
.  
.  
/* Unloads the stopped process with process ID of pid, waiting for the  
operation to complete. */  
status = NU_Unload(pid, NU_SUSPEND);  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Load](#)

NU_Kill

Stops and unloads a loaded and running process. If the process is stopped, it will simply be unloaded. This essentially performs both [NU_Stop](#) and [NU_Unload](#) actions on the specified process.

Usage

```
STATUS NU_Kill (INT      pid,  
                UNSIGNED suspend);
```

Arguments

- **pid**
ID of the process to stop.
- **suspend**
Allow for a suspension:

NU_SUSPEND waits for the operation to complete. Time-out is 1-0xFFFFFFFF.
NU_NO_SUSPEND returns immediately while the operation completes from the context of another thread.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_PROCESS**
The process ID specified is not valid.
- **NU_INVALID_STATE**
The specified process is in an invalid state to be killed.
- **NU_SYMBOLS_IN_USE**
The specified process cannot be killed since symbols it has exported are currently in use by other processes.
- **NU_MEMORY_IS_SHARED**
The specified process cannot be killed since it currently has shared memory regions with other processes.

Examples

```
STATUS status;
INT      pid;
.
.
.
/* Kills the process with process ID of pid, waiting for the operation to
complete. */
status = NU_Kill(pid, NU_SUSPEND);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Start](#)

NU_EXPORT_SYMBOL

Export a symbol for use by other processes. This allows a process or a statically linked application to provide access to its functions from other processes. For example, a library process might export all of the functions it contains for use by application processes.

Usage

```
NU_EXPORT_SYMBOL (symbol) ;
```

Arguments

- symbol
Symbol to be exported

Return Values

- None

Examples

```
#include "nucleus.h"
#include "kernel/nu_kernel.h"

/* Library Function */
INT dyn_lib_func(INT val)
{
    /* Return the value passed in. */
    return (val);
}

/* Export library symbols. */
NU_EXPORT_SYMBOL (dyn_lib_func);
```


Related Topics

[Processes APIs](#)

[NU_Symbol](#)

NU_Symbol

Gets the address of a symbol from the specified process. This allows a process to load other processes and use interfaces exported by these processes.

 **Note** When process memory management is enabled, symbol use automatically shares memory between two processes involved in symbol sharing. There is no need to manually share memory between the two processes involved.

Usage

```
STATUS NU_Symbol (INT      pid,  
                  const CHAR *sym_name,  
                  VOID      **sym_addr);
```

Arguments

- **pid**
ID of the process from which to get the symbol.
- **sym_name**
Pointer to the string name of the process.
- **sym_addr**
Double pointer to the address used to store the symbol (if found).

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_NOT_PRESENT**
Indicates the image contains a reference to a symbol not present in the system.
- **NU_INVALID_ID**
The process ID is not valid.
- **NU_INVALID_OPERATION**
Indicates an unspecified error occurred during the operation.

Examples

```
STATUS status;
INT      lib_pid;
INT      (*dyn_lib_func)(INT);
.
.
.
/* Returns the address of the "dyn_lib_func" exported by the process
   identified by lib_pid. */
status = NU_Symbol(lib_pid, "dyn_lib_func", (VOID **)&dyn_lib_func);

/* At this point, status indicates whether the service request was
   successful. */
```

Related Topics

[Processes APIs](#)

[NU_Getpid](#)

NU_Symbol_Close

Releases all symbols used by the current process and provided by the specified process. If the specified process is no longer providing any symbols (to any process) it may be stopped.

Usage

```
STATUS NU_Symbol_Close(INT      pid,  
                       BOOLEAN  stop,  
                       BOOLEAN  *stopped)
```

Arguments

- **pid**
ID of the process that is providing the symbols (symbol owner).
- **stop**
Indicates if the process will be halted if all symbols are free.
 NU_TRUE - If all symbols are free the process will be stopped.
 NU_FALSE - The process state will not be changed.
- **stopped**
On return, it points to a parameter that indicates whether the process was stopped as a result of the call or not. If a NULL is provided for this parameter then the resulting state of the process is not be returned.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_PROCESS**
The process ID specified is not valid.

Description

This function provides a means to release use of symbols so that a process may be unloaded.

Examples

```
STATUS  status;  
INT     lib_pid;  
BOOLEAN lib_stopped;  
.  
.  
.  
/* Releases symbols used from lib_pid by the current process.  If the  
lib_pid process no longer has any symbols in use by other processes, it  
will be stopped.  The resulting state of the lib_pid process is indicated  
by the lib_stopped parameter. */  
status = NU_Symbol_Close(lib_pid, NU_TRUE, &lib_stopped);
```

```
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Symbol](#)

NU_Getpid

Gets the process ID of the currently running process.

Usage

```
INT NU_Getpid(VOID);
```

Arguments

- None

Return Values

- Process ID of the currently running process.

Description

This function is needed for a process to know what its ID is and allows for different functionality to exist within a process that otherwise could not exist (for example, kill self, etc).

Examples

```
INT    current_pid;  
.  
.  
.  
/* Get the process ID of the current process. */  
current_pid = NU_Getpid();
```

Related Topics

[Processes APIs](#)

[NU_Symbol](#)

NU_Get_Exit_Code

Provides the exit code for a process which has stopped. The exit code is as described in [Table 7-1](#). The exit code can be used to determine why the process is no longer running and possibly to take action based on the reason.

Note

Processes that are killed or unloaded are no longer in the system and their exit codes are unavailable.

Usage

```
STATUS NU_Get_Exit_Code(INT  pid,  
                        INT  *exit_code);
```

Arguments

- `pid`
The ID of the process to get the exit code for.
- `exit_code`
The exit code for the process.

Return Values

- `NU_SUCCESS`
The function completed successfully.
- `NU_INVALID_PROCESS`
The process ID specified is not valid.
- `NU_INVALID_STATE`
The specified process is in an invalid state (for example, not stopped).

Examples

```
STATUS status;  
INT pid;  
INT exit_code;  
.  
.  
.  
/* Get the exit code for a stopped process specified by pid. */  
status = NU_Get_Exit_Code(pid, &exit_code);  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)[NU_Stop](#)

NU_Established_Processes

This function provides a count of the currently loaded processes at the time of the API call.

Usage

```
UNSIGNED NU_Established_Processes (VOID);
```

Arguments

- None

Return Values

- Number of currently loaded processes

Description

This function provides a means to get the number of processes in the system for use with the information functions.

Examples

```
UNSIGNED proc_count;  
.  
.  
.  
/* Get the number of processes in the system. */  
proc_count = NU_Established_Processes();
```

Related Topics

[Processes APIs](#)

[NU_Processes_Information](#)

[NU_Process_Information](#)

NU_Processes_Information

Returns information on processes, up to the maximum number specified. This function provides a means to get information about many (or all) of the processes in the system. It is designed to be used in conjunction with the [NU_Established_Processes](#) function to retrieve information about multiple processes currently in the system.

Usage

```
STATUS NU_Processes_Information(UNSIGNED      max_processes,
                               NU_PROCESS_INFO info_array[]);
```

Arguments

- **max_processes**
 The maximum number of processes to get information for. The actual number of processes whose information is returned in info_array is returned in this pointer.
- **info_array**
 Array of [NU_PROCESS_INFO](#) data structures which will be filled with process information.

Return Values

- **NU_SUCCESS**
 The function completed successfully.
- **NU_INVALID_POINTER**
 The pointer to [NU_PROCESS_INFO](#) data structure is invalid.

Examples

```
STATUS      status;
UNSIGNED    proc_count;
NU_PROCESS_INFO *info_array;
.
.
.
/* Get the number of processes in the system. */
proc_count = NU_Established_Processes();

/* Get memory for process information array. */
.
.
.

/* Get information about all processes in the system. */
status = NU_Process_Information(pid, info_array);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Established_Processes](#)

[NU_Process_Information](#)

NU_Process_Information

This function returns information for a single process.

Usage

```
STATUS NU_Process_Information(INT          pid,  
                             NU_PROCESS_INFO *info);
```

Arguments

- **pid**
The ID of the process to get information for.
- **info**
Pointer to a [NU_PROCESS_INFO](#) data structure which will be filled with information for the specified process if the operation is successful.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_POINTER**
The pointer to [NU_PROCESS_INFO](#) data structure is invalid.
- **NU_INVALID_INVALID**
The process is invalid (for example, no longer in the system).

Examples

```
STATUS          status;  
INT             pid;  
NU_PROCESS_INFO info;  
.  
.  
.  
/* Get information about the process specified by pid. */  
status = NU_Process_Information(pid, &info);  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Established_Processes](#)

[NU_Processes_Information](#)

NU_Memory_Map

Allows a process to map physical memory not currently mapped into the address space of the process.

Note



This function is only available if the Process Memory Management component is enabled.

Usage

```
STATUS NU_Memory_Map(INT          *mem_id,  
                     CHAR         *name,  
                     VOID         *phys_addr,  
                     VOID         **actual_addr,  
                     UNSIGNED size,  
                     UNSIGNED options);
```

Arguments

- **mem_id**
A pointer for the return identifier. The identifier will be used for other functions including [NU_Memory_Unmap](#).
- **name**
This parameter is optional, it helps keep up with the general needs of a memory region and may also be useful for searching for specific regions and debugging.
- **phys_addr**
This is the address to be mapped. If this value is past a region boundary the boundary would be moved back to the aligned region containing the memory, and the size will be adjusted to compensate. If an address is used that is already mapped, an error is returned. If the value is `NU_MEMORY_UNDEFINED`, a region will be obtained of the appropriate size.
- **actual_addr**
This is the return value for the region. If the `phys_addr` is not aligned the value returned in `actual_addr` will differ. If `phys_addr` is `NU_MEMORY_UNDEFINED` and a region of appropriate size is available this will hold the starting address.
- **size**
Size requested. This value may be smaller than the smallest region size, but it will be adjusted to match the region sizes. If the size causes overlap with a mapped region, an error is returned.
- **options**
Use the standard MMU region values, which are:

`NU_MEM_READ`

The region is read only.

NU_MEM_WRITE

The region is writable (in most architectures this implies read access).

NU_MEM_EXEC

The region is executable.

NU_SHARE_READ

The region is available to be shared as read only.

NU_SHARE_WRITE

The region is available to be shared as writable.

NU_SHARE_EXEC

The region is available to be shared as executable.

NU_CASH_INHIBIT

The region disallows all cache features.

NU_CASH_WRITE_THROUGH

The region uses write through cache.

NU_CASH_NO_COHERENT

The region does not enable cache coherency.

Note

All cache attributes are based on hardware availability.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_REGION_OVERLAPS**
The region requested overlaps with an existing region within the process, or overlaps with a region in another process that does not allow sharing.
- **NU_INVALID_SIZE**
The requested memory region size is invalid.
- **NU_INVALID_POINTER**
The pointer for the return address is NULL.

Examples

```
STATUS status;  
INT     mem_id;  
VOID    *mem_addr;  
.
```

```
.
.
/* Obtain 1000 bytes of new memory. */
status = NU_Memory_Map(&mem_id,
                      "new_mem",
                      NU_MEMORY_UNDEFINED,
                      &mem_addr,
                      1000,
                      NU_MEM_WRITE | NU_MEM_READ);

/* At this point, status indicates whether the service request was
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Memory_Unmap](#)

NU_Memory_Unmap

This function removes any region associated with the unique identifier.

Note

This function is only available if the Process Memory Management component is enabled.

Usage

```
STATUS NU_Memory_Unmap(INT mem_id);
```

Arguments

- **mem_id**
Identifier of the region to be unmapped.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_MEMORY_REGION**
The identifier does not reference a valid memory region.
- **NU_MEMORY_IS_SHARED**
The memory region is shared by other memory regions and cannot be deleted.

Examples

```
STATUS status;  
INT      mem_id;  
.  
.  
.  
/* Release memory specified by mem_id. */  
status = NU_Memory_Unmap(mem_id);  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Memory_Map](#)

NU_Memory_Get_ID

This function obtains a memory identifier or a specified address.

Note



This function is only available if the Process Memory Management component is enabled.

Usage

```
STATUS NU_Memory_Get_ID(INT    *mem_id,  
                        VOID    *phys_addr,  
                        CHAR    *name);
```

Arguments

- **mem_id**
Return pointer for the requested identifier.
- **phys_addr**
Pointer to memory within the address space that an ID is requested. This value can take `NU_MEMORY_UNDEFINED` to only search by name.
- **name**
String value that represents a memory region. If this is a non null value it is used with the `phys_addr` to find a specific region.

Return Values

- **NU_SUCCESS**
The function completed successfully.
- **NU_INVALID_MEMORY_REGION**
The address space is not mapped.
- **NU_INVALID_POINTER**
The pointer for the return address is NULL.
- **NU_INVALID_OPERATION**
Both search parameters are invalid.

Examples

```
STATUS status;  
INT    mem_id;  
.  
.  
.  
/* Find the memory region ID for the memory region named "xtr_mem". */  
status = NU_Memory_Get_ID(&mem_id, NU_MEMORY_UNDEFINED, "xtr_mem");
```

```
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Memory_Share](#)

NU_Memory_Share

This function facilitates the sharing of memory between two processes. It creates a new memory region in a second process based on an existing memory (source) region in the first process which has the sharing attribute. For example, if the first process has created a memory region for sharing with a specific name, the second process may discover this region by name, and then use the `NU_Memory_Share` function to gain access to the memory region, effectively allowing the memory region to be shared between the two processes. Only memory regions with the sharing attributes set may be shared.

Note



This function is only available if the Process Memory Management component is enabled.

Usage

```
STATUS NU_Memory_Share(INT  *mem_id,  
                      INT   source_id,  
                      CHAR  *name);
```

Arguments

- `mem_id`
This parameter is a pointer for the return identifier. The identifier will be used for other functions including [NU_Memory_Unmap](#).
- `source_id`
Identifier of the region to be shared.
- `name`
This parameter is optional, it helps keep up with the general needs of a memory region and may also be useful for searching for specific regions and debugging.

Return Values

- `NU_SUCCESS`
The function completed successfully.
- `NU_INVALID_MEMORY_REGION`
The identifier does not reference a valid memory region.

Examples

```
STATUS status;  
INT     new_mem_id;  
INT     orig_mem_id;  
.  
.  
.  
/* Create new memory region called "shr_mem" with same  
   characteristics as original memory region. */
```

```
status = NU_Memory_Share(&new_mem_id, orig_mem_id, "shr_mem");  
  
/* At this point, status indicates whether the service request was  
successful. */
```

Related Topics

[Processes APIs](#)

[NU_Memory_Get_ID](#)

Process_Exception_Handler

An exception handler which will be called for all exceptions. A default, system-wide implementation is provided which will stop execution of a process in which an exception occurs. By providing a function with an identical name and signature the default implementation may be overridden to provide custom exception handling. When a custom exception handler is used, the return value determines how the system operates after the exception handler has completed.

Usage

```
UNSIGNED Process_Exception_Handler(NU_PROCESS_EXCEPTION *exception);
```

Arguments

- exception

This is a pointer to a [NU_PROCESS_EXCEPTION](#) structure which will contain details about the exception. The [NU_PROCESS_EXCEPTION](#) structure is described in the [Processes Data Structures](#) section.

Return Values

- NU_PROC_SCHEDULE

Upon return execution will be returned to the OS scheduler. This return value may be used if the exception has been handled sufficiently to allow the system to resume execution, but the task on which the exception occurred is no longer available, requiring the return to the OS scheduler to select another task to run.

- NU_PROC_RESUME_TASK

Upon return execution will resume in the task on which the exception occurred. This return value may be used if the exception has been handled sufficiently to allow the system to resume execution and the task on which the exception occurred is still available and no longer in danger of causing an exception.

- NU_PROC_UNRECOVERABLE

Upon return the system is no longer considered viable and the system error handler will be called.

Examples

```
/* Fatalistic process exception handler */
UNSIGNED Process_Exception_Handler(NU_PROCESS_EXCEPTION *exception)
{
    /* Always go to system error handler. */
    return(NU_PROC_UNRECOVERABLE);
}
```

Related Topics

[Processes APIs](#)

main

Process entry function which will be executed when a process is started. A default implementation is provided which will return immediately but continue execution of the process after the entry function exits. *main()* provides a known location where execution will begin when a process is started. Typically, *main()* will contain any initialization of resources needed by the process and start the process running, similar to the way the *Application_Initialize()* function is used in a statically linked Nucleus application. When an entry function exits, the value returned is examined. If the returned value (called an exit code) is *EXIT_CONTINUE*, then the process will continue to execute, otherwise the exit code will be stored by the system for provision through the process APIs and the process will stop executing.

Usage

```
int main(int argc,  
         char *argv[]);
```

Arguments

- *argc*

The number of arguments passed to the entry function.

Note

The *argc* parameter is not used.

- *argv*

The vector containing the arguments passed to the entry function.

Note

The *argv* parameter is not used.

Return Values

- See the Exit Codes in [Table 7-1](#).

Examples

```
/* Determine if process stops after entry point returns. */
BOOLEAN    proc_stop = NU_TRUE;
INT main()
{
    INT      exit_code;
    /* Determine exit code based process stop value. */
    if (proc_stop == NU_TRUE)
    {
        exit_code = EXIT_SUCCESS;
    }
    else
    {
        exit_code = EXIT_CONTINUE;
    }
    return(exit_code);
}
```

Related Topics

[Processes APIs](#)

[NU_Start](#)

atexit

This function provides a way to register de-initialization (call-back) functions which will be executed when a process stops. Registered functions will be called in reverse registration order, from most recently registered to the oldest registered function.

Note



Processes which stop in a failed state do not invoke registered de-initialization functions as the process is considered corrupt and unstable. For example, if a process stops through a call to the exit function passing EXIT_ABORT as the exit code value or through a call to the abort function, then registered de-initialization functions will not be called.

Usage

```
int atexit(void (*func)(void));
```

Arguments

- **func**
Pointer to de-initialization function to be registered.

Return Values

- 0
The operation completed successfully. The specified de-initialization function is registered.
- 1
The operation failed as there is no more room to add further de-initialization functions.

Examples

```
void deinit_func(void)
{
    /* De-allocate and clean up process resources */
}

INT main()
{
    /* Allocate process resources */

    /* Register de-initialization function to perform clean-up of process
       resources when process stops. */
    atexit(deinit_func);

    /* Return from process entry function causing process to stop and de-
       initialization function to be called. */
    return(0);
}
```

Related Topics

[Processes APIs](#)
[main](#)

[NU_Stop](#)
[exit](#)

exit

This function provides a way to stop execution of the current process passing an exit code.

Usage

```
void exit(int exit_code);
```

Arguments

- `exit_code`

The exit code to be set for the stopped process. Some exit codes invoke special behavior as the process is stopped. See the Exit Code [Table 7-1](#) for more information.

Note

The `EXIT_CONTINUE` does not denote any special behavior when provided as an `exit_code` value to the `exit` function.

Return Values

- None

Examples

```
INT main()
{
    INT i;
    for (i = 0; i < 100; i++)
    {
        /* exit on 50th iteration */
        if (i == 50)
        {
            /* Exit with exit code as loop iteration */
            exit(i);
        }
        /* Do process iteration work... */
    }
    return(0);
}
```

Related Topics

[Processes APIs](#)

[atexit](#)

abort

This function provides a way to stop execution of a process indicating a failure occurred. In the case of a failure, the process is stopped, but no de-initialization occurs since the process is considered to be in a corrupted (failed) state. The exit code for the stopped process is `EXIT_ABORT`.

Usage

```
void abort(void);
```

Arguments

- None

Return Values

- None

Examples

```
INT main()
{
    while(1)
    {
        /* Do process iteration work... */
        if ( /* error occurred */ )
        {
            /* Abort execution of the process */
            abort();
        }
    }
    return(0);
}
```

Related Topics

[Processes APIs](#)

[exit](#)

Shell Commands

The following commands can be issued from a Nucleus command shell terminal (for example a serial terminal, telnet, and so on) and provide much of the same functionality as available through the run-time APIs described above.

- [load](#)
- [tryload](#)
- [start](#)
- [stop](#)

- [unload](#)
- [kill](#)
- [proclist](#)

load

Loads an ELF image of a process.

Usage

```
load <name> --address=<load address>  
            --heap=<heap size>  
            --stack=<stack size>  
            --km=<T|F>
```

Arguments

- **Mandatory**
name
Name of the image, including the path information.
- **Optional**
address
Specify load address for the image as opposed to being loaded to dynamically allocated memory.
heap
Specify the heap size allocated for the process when created.
stack
Specify the stack size allocated for the root task of the process when created.
km
Specify whether the process will be loaded in kernel-mode or user-mode.

Note



The `--km` option is reserved for future development and should not be used.

Return Values

- If operation is successful:
PID
Load address
Entry point address (if a process/hybrid, NULL otherwise).
- Kernel-mode status.
- If operation fails:
Applicable error string (for example File not found, Not enough Memory, etc).

Examples

```
> load C:\my_app.load
PID: 2
Load Address: 0x00200108
Entry Point: 0x00200208
Kernel-Mode: FALSE

> load static:\my_app.load --heap=0x10000
PID: 4
Load Address: 0x00204090
Entry Point: 0x00204190
Kernel-Mode: FALSE
```

Related Topics

[Shell Commands](#)

[tryload](#)

tryload

Tries to load an ELF image of a process. This command is the same as the [load](#) command except that it will print a list of missing symbols to the command shell if symbols cannot be resolved.

Usage

```
tryload <name>
      --address=<load address>
      --heap=<heap size> -
      --stack=<stack size>
      --km=<T|F>
```

Arguments

- Mandatory

<name>

Name of the image, including path information.

- Optional

address = <load address>

Reserved for future development.

heap = <heap size>

Specify the heap size allocated for the process when created.

stack = <stack size>

Specify the stack size allocated for the root task of the process when created.

km = <T|F>

Specify whether the process will be loaded in kernel-mode or user-mode.

Note



The --km option is reserved for future development and should not be used.

Return Values

- If operation is successful:

PID

Load address

Entry point address (if a process/hybrid, NULL otherwise).

- Kernel-mode status.

- If operation fails:

List of missing symbols.

Applicable error string (for example File not found, Not enough Memory, etc).

Examples

```
> load A:\my_app.load
PID: 1
Load Address: 0x00100108
Entry Point: 0x00100208
Kernel-Mode: FALSE

> tryload A:\my_app.load
PID: 2
Load Address: 0x00200108
Entry Point: 0x00200208
Kernel-Mode: FALSE

> load A:\my_app2.load
ERROR: Image load error (error = -51)

> tryload A:\my_app2.load
Symbol(s) not found:
    global_var
    foo
    testit
ERROR: Image load error (error = -51)
```

Related Topics

[Shell Commands](#)[load](#)

start

Starts the specified process. Use [start](#) in conjunction with the [load](#) command to start/run the specified process.

Usage

```
start <pid>
```

Arguments

- `<pid>`
ID of a loaded process.

Return Values

- If operation is successful:
Message stating that the process has been started.
- If operation fails:
Error message.

Examples

```
> load a:\test.load
PID: 1
Load Address: 0x00204090
Entry Point: 0x00204190
Kernel-Mode: FALSE

> start 1
test.load started

> start 2
ERROR: Process start error (error = -84)
```

Related Topics

[Shell Commands](#)

[stop](#)

stop

Stops the specified process.

Usage

```
stop <pid>
```

Arguments

- <pid>
ID of the process to stop.

Return Values

- If operation is successful
Message stating that the process has been stopped.
- If operation fails
Error message.

Examples

```
> load a:\test.load
PID: 1
Load Address: 0x00204090
Entry Point: 0x00204190
Kernel-Mode: FALSE

> start 1
test.load started

> stop 1
test.load stopped

> stop 2
ERROR: Process stop error (error = -84)
```

Related Topics

[Shell Commands](#)

[start](#)

unload

This commands unloads the specified process.

Usage

```
unload <pid>
```

Arguments

- `<pid>`
ID of the process to unload.

Return Values

- If operation is successful
Message stating that the process has been unloaded.
- If operation fails
Error message.

Examples

```
> load a:\test.load
PID: 1
Load Address: 0x00204090
Entry Point: 0x00204190
Kernel-Mode: FALSE

> start 1
test.load started

> stop 1
test.load stopped

> unload 1
test.load unloaded

> unload 2
ERROR: Process unload error (error = -84)
```

Related Topics

[Shell Commands](#)

[load](#)

kill

Kills the specified process.

Usage

```
kill <pid>
```

Arguments

- `<pid>`
ID of the process to kill.

Return Values

- If operation is successful:
Message stating that the process has been killed.
- If operation fails:
Error message.

Examples

```
> load a:\test.load
PID: 1
Load Address: 0x00204090
Entry Point: 0x00204190
Kernel-Mode: FALSE

> start 1
test.load started

> kill 1
test.load killed

> kill 2
ERROR: Process kill error (error = -84)
```

Related Topics

[Shell Commands](#)

[unload](#)

proclist

Provides a list of processes currently in the system with information about each process.

Usage

```
proclist --delimited
```

Arguments

- - delimited (optional)

Outputs the data using `<>` as delimiters instead of spaces. This provides an easier way to parse format for the host.

Return Values

- If operation is successful:
List of processes/libraries and information for each item.
- If operation fails:
Error message.

Description

This is useful for development purposes.

Examples

```
> proclist
PID  Name      State   Load Address Entry Point Exit Code  Kernel-Mode
===  ===      =====
0    kernel    Started 0x00000000 0x00000000 0          TRUE
1    test.load Stopped 0x00100000 0x00100040 145        FALSE
2    mylib.load Started 0x00200304 0x00200344 0          FALSE
3    myapp.load Started 0x80004000 0x80004040 0          FALSE
```

```
> proclist --delimited
<PID(integer)><Name(string)><Status(string)><Load Address(integer)>
<Entry Point(integer)><Kernel-Mode(string)>
<0><kernel><Started><0x00000000><0x00000000><0><TRUE>
<1><test.load><Stopped><0x00100000><0x00100040><145><FALSE>
<2><mylib.load><Started><0x00200304><0x00200344><0><FALSE>
<3><myapp.load><Started><0x80004000><0x80004040><0><FALSE>
```

```
> proclist
PID  Name      State   Load Address Entry Point Exit Code  Kernel-Mode
===  ===      =====
0    kernel    Started 0x00000000 0x00000000 0          TRUE
```

```
> proclist --delimited
<PID(integer)><Name(string)><Status(string)><Load Address(integer)>
<Kernel-Mode(string)>
<0><kernel><Started><0x00000000><0x00000000><0><TRUE>
```

Note



The kernel process (process 0) will always show up in the process list.

Related Topics

[Shell Commands](#)

Chapter 8

Nucleus Processes User Manual

Nucleus Processes User Overview	527
Operation	528
Initialization	528
Nucleus Processes Configuration	529
Use Cases	532
Dynamic Linking Techniques	533
Leveraging Open Source Libraries	533
Exporting Symbols	534
Symbol Exports Validation	534
Obtaining and Sharing Memory Resources	535
Internals	535
Processes States	536
Resource Management	536
Dynamic Memory Allocation	537
Process Memory Layout	538

Nucleus Processes User Overview

Nucleus Processes provide a lightweight process model environment in Nucleus RTOS. Nucleus Processes are built as position-independent relocatable ELF executables and have the following key features:

- Can be dynamically loaded and unloaded.
- Enable dynamic linking - to the root kernel image (statically deployed Nucleus Application with Nucleus Processes runtime support enabled) and other Nucleus Processes.
- Can be developed and deployed independently from the rest of the system.
- Provide namespace isolation without introducing a separate address space (for example, virtual addressing is not used).
- Enable isolating the user space from the kernel space (optional and requires hardware support).
- Enable isolating memory regions of each process from the other processes (optional and requires MMU support in hardware).

- There are no separate forms for “application” and “library” processes. Each process can import or export symbols and hence, depending upon the program design, can act like a pure application, library or a hybrid module (having both application part and library functionality).

Note

While the Nucleus Processes can be used similarly to a shared library, they do not behave identically to typical shared libraries in the traditional sense. The primary difference is that each process is loaded at the system level context whereas the traditional shared libraries are loaded in the context of an application process. As a result Nucleus Processes used as libraries (library processes) do not maintain a per process state for the processes using their exported symbols.

Related Topics

[Nucleus Processes User Overview](#)[Operation](#)

Operation

Initialization

Nucleus Processes run-time depends on the following components:

- Kernel (Nucleus PLUS)
- The Registry Service

If development workflow is needed, the following additional components are also required:

- Device Manager
- Networking Stack (Core)
- Telnet Server
- TFTP Server
- Storage
- RAM Disk
- Shell Service

The Nucleus Process run-time itself, does not require any explicit initialization and only needs the initialization of the shell interface extensions it adds for the development workflow. The shell extension component *nu.os.kern.process.shell*, if enabled, is initialized automatically at run-level 15.

Related Topics

[Operation](#)[Nucleus Processes Configuration](#)

Nucleus Processes Configuration

Nucleus Processes run-time has two parts: one part is linked with the root kernel image and the other is linked with Nucleus Process executable images.

Core Runtime (Linked with the Root Kernel Image)

The Core Runtime part of the Nucleus Processes, provides the run-time linking and loading/unloading support for the processes. It consists of the following components:

```
nu.os.kern.process.core
nu.os.kern.process.linkload
nu.os.kern.process.mem_mgmt
```

The last component (`nu.os.kern.process.mem_mgmt`) provides support for memory regions isolation and requires MMU support in the hardware. If memory region isolation is not needed, this component can be disabled (default). The other two components are mandatory and must be enabled to use Nucleus Processes in the system.

Configuration Options

Notable configuration options that are available for `nu.os.kern.process.core` are shown in [Table 8-1](#).

Table 8-1. Core Configuration Options

Option Name	Default Value	Description
max_processes	64	The maximum number of processes that are allowed in the system. Updating this value has implications on the amount of memory required for translation tables (if memory regions isolation support is enabled).
heap_size	65536	Size of the heap created for processes. This value is used unless a separate value is passed via NU_Load API.
stack_size	4096	Size of the stack utilized by the process root thread.
min_user_task_priority	16	Minimum task priority value (lower means higher priority) allowed for user process tasks.
sup_user_mode	true	Determines if supervisor/user mode switching is enabled within the system.

Table 8-1. Core Configuration Options (cont.)

Option Name	Default Value	Description
dev_support	true	<ul style="list-style-type: none">• true Enables integration with the IDE to facilitate process development. This includes enabling code to notify the IDE of process loading and unloading.• false Removes the IDE integration code and gives better performance.

The configuration options in [Table 8-2](#) are available for *nu.os.kern.process.linkload*.

Table 8-2. Linkload Configuration Options

Option Name	Default Value	Description
dup_symbol_check	true	<ul style="list-style-type: none">• true Enables duplicate symbol detection during the loading process. This means that a process exporting a symbol that has already been exported by another loaded process will fail to load.• false Disables this check and improves load time.

User Runtime (Linked with Nucleus Processes Executable Images)

User Runtime is linked statically with each Nucleus Process executable image. It provides the pre-main initialization (such as global/static objects initialization in case of C++) and post-main de-initialization (such as [atexit](#) and C++ exception handling) that needs to be run in the process context. It consists of the following component:

```
nu.os.kern.process.user
```

This component is built into a separate static library *nucleus_user.lib* (whereas all other components in the Nucleus OS source tree are built into *nucleus.lib*). This library must be linked with each Nucleus Process otherwise the resulting ELF will not be a valid Nucleus Process executable.

Table 8-3 shows user run-time configuration options.

Table 8-3. User Run-tim Configuration Options

Option Name	Default Value	Description
atexit_max_funcs	32	The maximum number of functions that a process can register via atexit() at runtime.
cxx_support	true	<ul style="list-style-type: none">• true Enables C++ language support in the run-time user library linked with processes.• false C++ language support is removed to improve performance (for example, no attempt to call C++ static object constructors, and so on), and reduce library size.

Other OS Components required for using Nucleus Processes

Nucleus Processes run-time does not depend directly on any other OS component. However currently Nucleus Processes can only be deployed via the file system. This means that you cannot embed the process ELF image in ROM and load directly by address. Therefore, some type of a disk device and a suitable file system support must also be available.

There are two distinct usage scenarios:

- Temporary deployment (during development)
- Final deployment

Requirements for both are discussed in the following sections.

Development Workflow

Deployment via the file system requires that the process images be transferred to the disk device during each debug cycle, if the process images have been updated or the disk device is not persistent (such as a RAM disk). To address this issue, Nucleus ReadyStart provides a development workflow which enables you to debug the processes during development seamlessly right after the build.

The development workflow uses:

- a RAM disk
- the TFTP protocol to transfer the process images to the RAM disk
- finally uses Nucleus Shell over Telnet to load the processes from the RAM disk.

All these steps are automated in the IDE and work transparently without user intervention. For details on the development workflow in the IDE, refer to the “Using Nucleus Processes” chapter in the *Nucleus ReadyStart Guide*.

Enabling the following component in the Nucleus Configuration Editor enables the support for the development workflow (all the required components, such as the file system, TFTP and so on, are automatically enabled):

```
nu.os.kern.process.shell.enable = true
```

The RAM disk device needs to be enabled separately. To enable it in the Nucleus Configuration Editor, enable *ramdisk0* under the *<platform>* node.

Note

To be able to load larger process images, you may need to increase the RAM disk size. You can control the RAM disk size by changing the number of pages, where each page is 4096 bytes. For example, the following setting would set the RAM disk size to 24 MBytes: `nu.os.drvr.fat_rd.num_ramdisk_pages = 6144`

Note

For more information about creating a custom configuration, see “Creating a Custom Configuration”, in the chapter “Getting Started with Nucleus ReadyStart Using Sourcery Code Bench”, in the *Nucleus ReadyStart Guide*.

Deployment Scenario

For the final deployment most of the requirements for the development workflow are not needed. You still need the file system and a physical disk device (such as an SD card).

Related Topics

[Other OS Components required for using Nucleus Processes](#)

[Core Runtime \(Linked with the Root Kernel Image\)](#)

Use Cases

This section will discuss some common operations and scenarios when working with the Nucleus Process model. The use cases will be presented at a very high level with references to the Nucleus Process APIs which support the solutions. Please see the Nucleus Process API Reference in the [Nucleus Processes](#) chapter for more detail about the API functions found in the discussion.

Dynamic Linking Techniques

A Nucleus Process can dynamically link to the symbols exported by other processes or the root kernel image. There are two ways a process can use the dynamic linking:

- [Implicit Linking](#)
- [Explicit Linking](#)

Implicit Linking

This is the default approach. To use any external symbol, you only need the signature of that symbol. Building the process will automatically produce proper information in the ELF image for the run-time linker to resolve those symbols. Any symbols that are not found in the objects and libraries statically linked with the process itself at the build time, will be treated as external references to be resolved dynamically at run-time.

A process with any symbol imports requires those symbols to be present in the system before it can be successfully loaded. Nucleus Processes run-time has no provision to track these dependencies (unlike the shared libraries on Linux have) and does not automatically load the dependencies. It is the responsibility of the system integrator to make sure that the process dependencies are loaded first.

Explicit Linking

In addition to the implicit linking explained above, Nucleus Processes also support explicit linking. In this technique, a process uses the [NU_Symbol](#) API to get the address of the required symbol by name from a specified process. If that process is present in the system (it must have already been loaded in the system by the root kernel image, some other process or the current process itself) and it exports the specified symbol, its address will be returned. The calling process can now use that symbol similarly to how function pointers are used.

Since with this approach the symbol address is explicitly obtained, the calling process must release the symbol explicitly (by using the [NU_Symbol_Close](#) API) before the other process whose symbol it is using, can be unloaded from the system. If the calling process does not release the symbols, the other process will be barred from being unloaded until the calling process itself is stopped.

Related Topics

[Leveraging Open Source Libraries](#)

[Exporting Symbols](#)

Leveraging Open Source Libraries

Some open source libraries provide valuable functionality that is desirable in the proprietary applications. At times the library license makes it unfeasible, by disallowing static linking, such as the LGPL license. Nucleus Processes enable taking advantage of such libraries by deploying

them as Nucleus Processes and linking to them dynamically. The dynamic linking is compatible with licenses like LGPL.

Related Topics

[Use Cases](#)

[Dynamic Linking Techniques](#)

Exporting Symbols

The root kernel image or any process can export any number of symbols to make them available to other processes in the system.

Exporting a symbol is very simple and is accomplished using the following macro:

```
NU_EXPORT_SYMBOL(symbol_name)
```

If a global data symbol needs to be shared with other processes, a getter/setter-style set of function wrappers should be provided. These functions should be exported instead of trying to export the data symbol directly.

It is your responsibility to ensure that the source files containing the exports are not optimized by the linker. Generally, if exports are placed in a separate source file (which does not contain any other symbol that is being referenced anywhere in the code), the exports file will get optimized by the linker. To avoid this you can place the exports in the source file which is guaranteed to get linked depending on your specific implementation. For example, if you provide a *main()* implementation, the file containing the *main()* is guaranteed to be linked and exports can be placed or included in this file.

Related Topics

[Dynamic Linking Techniques](#)

[Symbol Exports Validation](#)

Symbol Exports Validation

A symbol with a given name can be exported only by one process in the system at a time. If more than one process export symbols with the same name, those cannot be loaded at the same time.

Nucleus Processes run-time supports validating the symbol exports in the system as a configurable option. Enabling this is useful during development as accidental export of same-named symbols in multiple processes may result in unexpected bugs. The symbol lookup mechanism in the Nucleus Processes run-time uses the first instance of the symbol export it encounters which belongs to the process that is loaded first. Symbol validation will prevent such scenarios. It aborts the load of the processes which export any symbol with the same name as the name of another symbol exported by any of the currently loaded processes or the root kernel image.

This option should be disabled before the final deployment as it may considerably increase the process loading time.

Related Topics

[Exporting Symbols](#)

[Dynamic Linking Techniques](#)

Obtaining and Sharing Memory Resources

A process is created with a fixed amount of memory available to it, called the memory heap. The size of a process memory heap may be set when the process is loaded, otherwise a default value is used. If the process memory heap is insufficient the memory management APIs provide a means to allocate additional memory resources for a process. Further, the memory management APIs may be used to share allocated memory with other processes to facilitate inter-process communication.

Obtaining Additional memory

In order to obtain additional memory, the [NU_Memory_Map](#) function should be called to allocate the desired amount of memory with a specified name and attributes. If the memory is to be shared with another process, the memory attributes should include sharing permissions. At this point the process has access to a new memory region in addition to the original process memory heap.

Sharing Memory

Any memory resource with sharing permissions may be shared with other processes. One approach to sharing memory between two processes is for one process to create a new memory region with a specific name using the memory management APIs. Next, a second process will obtain the memory region ID by searching for the memory region by name using the [NU_Memory_Get_ID](#) API. Finally, the second process will gain access to the memory region using the [NU_Memory_Share](#) API function, passing the memory region ID. The result is two processes with access to a shared memory region.

Related Topics

[Dynamic Memory Allocation](#)

[Symbol Exports Validation](#)

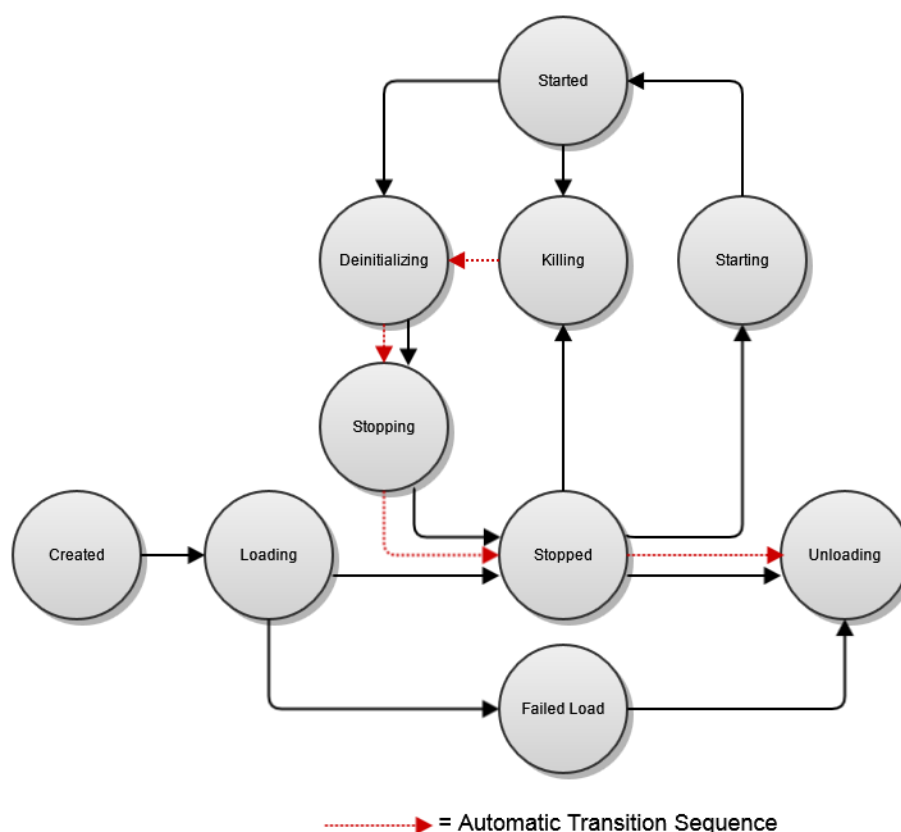
Internals

The following sections provide details about the internal interactions, sequences and concepts which underly the Nucleus Process model.

Processes States

Each Nucleus Process moves through a number of states during its life-cycle. Many of the APIs provided for working with Nucleus Processes revolve around changing the state of a process. [Figure 8-1](#) shows the states of a process and the transition paths between states. A typical scenario involves loading a process, starting and stopping the process one or more times, then unloading the process. The kill operation is provided for both convenience and stability as it allows a process to be stopped and unloaded from almost any other state in a single API call.

Figure 8-1. Process State Transitions



Related Topics

Internals

Resource Management

Resource Management

Conceptually, there are a set of resources associated with each process. Generally these resources are OS objects created during the execution of the process. These include the memory heap, the root thread, any user process tasks, semaphores, pipes, queues, and so on. In the Nucleus Processes model, some of these resources are managed by the system while the rest are expected to be managed by the process itself. This design allows the Nucleus Processes model

to be simple and lightweight through limited resource tracking done by the system while an API provides support for heavier resource tracking, if needed. The Nucleus Process model system manages the following resources:

- Process memory heap
- Process threads (including the root thread)
- C++ language static object construction and destruction

All other resources are considered to be managed by the process, which means that you are responsible for performing cleanup of any other resources created during operation of the process. For example, if the process creates a queue for communications between two process threads, the process should also register a de-initialization function to clean up the queue (the threads will be handled by the system). If a registered de-initialization function is not provided to clean up the created queue, the queue will remain in the system after the process is stopped, potentially corrupting the OS queue services by having a queue pointed to de-allocated memory.

Note

The use of registered de-initialization functions is not required, but it is highly recommended as the correct way to perform resource clean up since any registered de-initialization functions are called when the process is stopped, regardless of how it was stopped. The only exception to this is if [abort](#) is called from within a process.

Related Topics

[Internals](#)[Dynamic Memory Allocation](#)

Dynamic Memory Allocation

Each process has a configurable amount of memory available to it called the process memory heap. The memory heap is allocated when the process is loaded. Memory allocations within the process will draw from this pool of memory (for example, use of `malloc()` or Nucleus related APIs). If a process requires additional memory resources after it has been loaded, or if the process wishes to share memory with another process, the memory mapping API is provided.

If memory management support is enabled and the appropriate hardware is present (for example an MMU) then the process memory will be isolated from memory owned by the system and other processes. This isolation provides a measure of security and stability by preventing a process from disrupting the operation of the system or other processes, either accidentally or maliciously. Attempts to access memory that is off-limits to a process results in a memory exception. A default, system-wide default exception handler is provided within the Nucleus Processes model and a custom exception handler may be provided by the user to perform steps specific to the device requirements.

Related Topics

[Internals](#)

[Process Memory Layout](#)

Process Memory Layout

Each process (except the root kernel image process) has the following standard memory regions, in the specified order:

- a. text
- b. read-only data (rodata)
- c. read/write data (wrdata)
- d. root task stack
- e. the process memory heap

Note that not all these sections need to exist. For example, if no read-only data is present in a process, the read-only region will have a size of zero and it will not be defined.

If MMU is available and memory regions support is enabled, each of the regions is set up as a separate memory region with appropriate attributes. Moreover each memory region will be aligned on the page boundaries and the regions will be padded, if necessary, to a size that is a multiple of the page size.

The following figures ([Figure 8-2](#) and [Figure 8-3](#)) show the layout of an example process as it may appear with and without memory regions isolation support (MMU). The black strips in the memory layout with memory regions support enabled show the padding to align the memory region sizes to the page boundaries. It is evident from the comparison that the processes run-time memory requirement is less when memory regions support is not enabled (as page alignment is not needed).

Figure 8-2. Memory Layout without Memory Regions Isolation Support

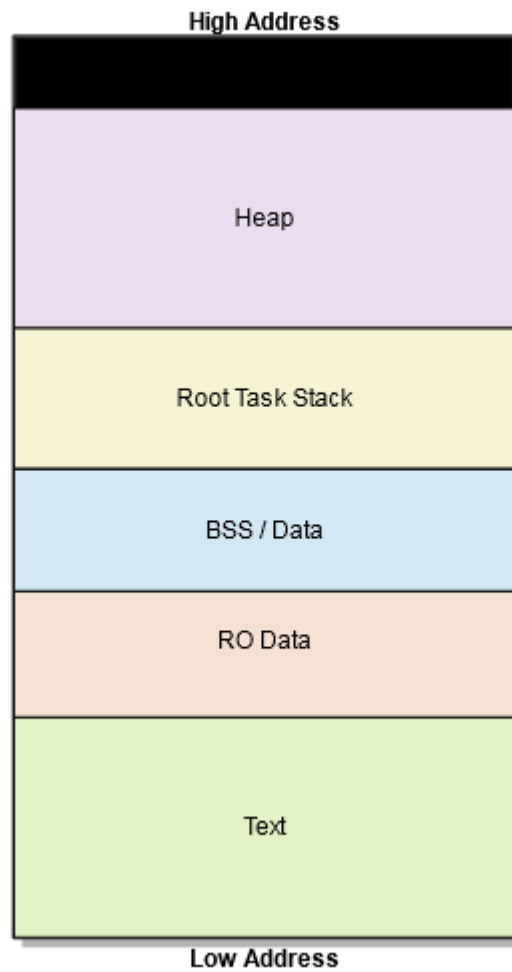
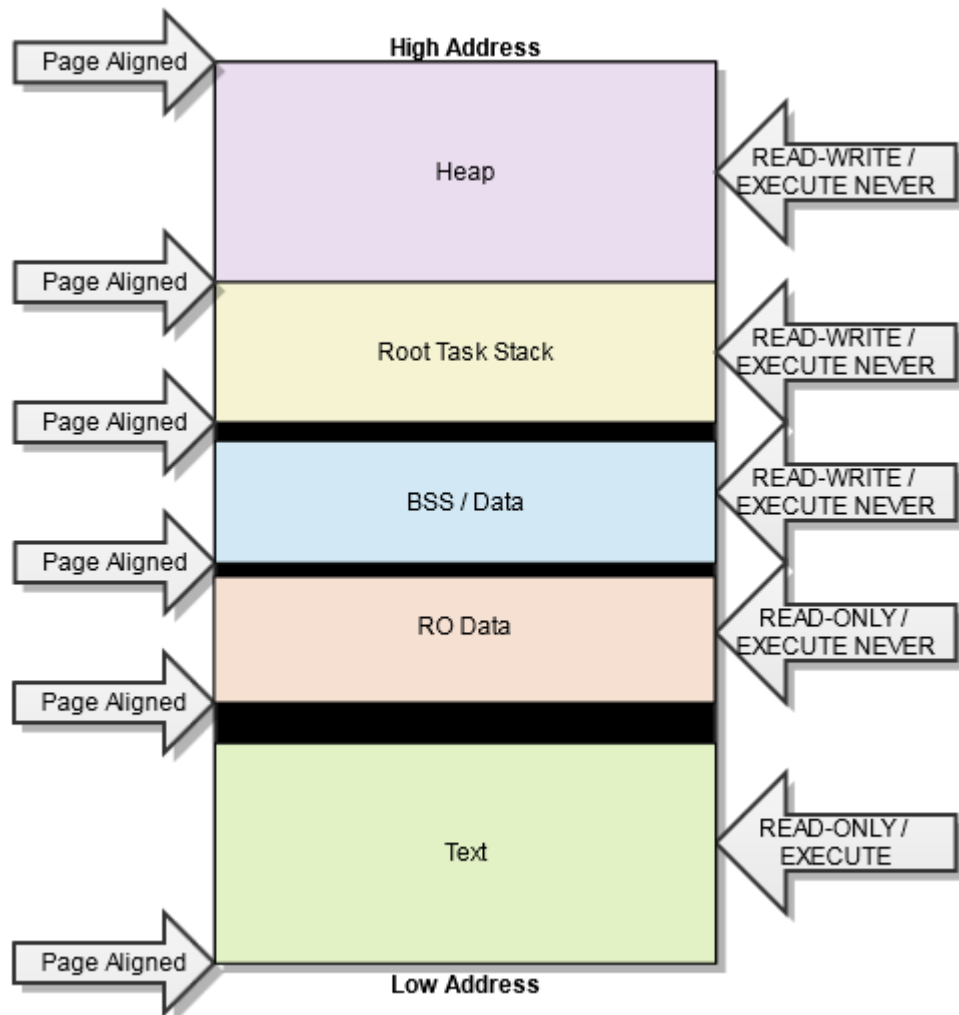


Figure 8-3. Memory Layout with Memory Regions Isolation Support Enabled



When memory regions isolation support is enabled, a process can define/create further memory regions with custom attributes. For example, a process "A" can create a memory region to share data only with another process "B" but not any other process in the system. For details on the related API, refer to the [Nucleus Processes](#) chapter.

Related Topics

[Internals](#)

[Dynamic Memory Allocation](#)

Nucleus C++ service supports C++ object oriented programming in development of embedded applications. This chapter provides information on using the C++ services.

C++ Overview

Nucleus C++ is an embedded software service that directly supports the object-oriented method for developing applications.

Nucleus C++ is used to create high quality embedded applications at a significantly faster rate. The main advantage of using Nucleus C++ is that software development teams can build an arsenal of reusable software components that are optimized for use in their particular application domain.

There is an easy migration path to Nucleus C++ since your legacy C/C++ application source code can be reused in a new Nucleus C++ application. This feature allows you to reuse existing Nucleus 'C' applications or any other existing portable 'C' or C++ source code.

C++ Portability

A major feature of Nucleus C++ is portability for your application code.

Nucleus C++ is a pre-requisite for using the C++ programming language in any Nucleus application that incorporates C++, even if the Nucleus 'C' API is used. This is because required support components and initialization details must be present for seamless C/C++ interoperability.

C++ Programming Language Support

When the ANSI C++ committee released the first draft of the C++ standard, support for it across different compiler vendors was scattered and varied. Even today, there are many compiler vendors that do not have complete support for the ISO C++ 2.0 specification.

Most of the ISO C++ 2.0 specification language features have little to do with the fact that you are programming for a multithreaded embedded environment. For the most part, if your compiler supports the language feature, you can use it.

This depends very much, however, on the compiler's implementation of the feature. For example, many compilers produce C++ exception handling code that is not reentrant and therefore not safe to use in a multithreaded environment.

You should always verify that the implementation of a given C++ language feature is thread safe before using it in a multithreaded application.

C++ Limitations

There following are limitations to the Nucleus C++ service:

- C++ Language support is largely provided by and limited by the toolset's C++ support implementation.
- No support/handling for the C++ "bad_alloc" exception.

The "bad_alloc" exception occurs when a C++ new operation fails. There is no internal support for the "bad_alloc" exception handling and the results are undefined.

Note



There are no longer any user-facing APIs in the C++ service, now that MMU and C++ support is integrated into the process model.

Nucleus Shell Overview

The Nucleus Shell service provides a simple console interface for executing registered commands. The service consists of a user interface that runs over a serial port, providing a console-like prompt, and an API to dynamically register and un-register commands. The user interface supports listing registered commands and invoking those commands through the console. The API provides a means to dynamically register commands. The commands themselves may include parameters that will be passed from the console to the command in the form of an array of string arguments (for instance, `argv`).

The shell is configured by default with serial I/O enabled, provided the `nu.os.svcs.shl.serial_enabled` option is true. In this case you only need to call [NU_Get_Shell_Serial_Session_ID](#) in order to get the “*p_shell*” handle for the automatically created shell. Then you can register and unregister commands with [NU_Register_Command](#) and [NU_Unregister_Command](#).

For an advanced case (or, if the `nu.os.svcs.shl.serial_enabled` option is false), to create your own Shell session, start with [NU_Create_Shell](#), and then you can register and unregister commands. To delete your session, use [NU_Delete_Shell](#).

Nucleus Shell Function Reference

This section lists the available shell APIs:

- [NU_Register_Command](#)
- [NU_Unregister_Command](#)
- [NU_Create_Shell](#)
- [NU_Delete_Shell](#)
- [NU_Get_Shell_Serial_Session_ID](#)

NU_Register_Command

This function is used to register external commands with Nucleus Shell.

Usage

```
STATUS NU_Register_Command(NU_SHELL *p_shell  
                           char      *cmd,  
                           STATUS    (*cmd_fcn) (int, char **));
```

Arguments

- `p_shell`
Shell session handle.
- `cmd`
Command name string pointer.
- `cmd_fcn`
Command function pointer.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_DUPLICATE_CMD`
Duplicate command exists.
- `NU_INVALID_OPERATION`
Indicates service not ready.

Example

```
/* Assume variables are defined locally. */  
STATUS status; /* Register Command status */  
  
/* Command Function to be registered */  
STATUS my_cmd_function (INT argc, CHAR ** argv)  
{  
    /* Do my function's work here...*/  
    return(NU_SUCCESS);  
}  
  
/* Register command. */  
status = NU_Register_Command(p_shell, "my_cmd", &my_cmd_function);  
  
/* At this point status indicates if the service was successful and the  
command was registered. */
```

Related Topics

[NU_Unregister_Command](#)

[Nucleus Shell Function Reference](#)

NU_Unregister_Command

This function is used to remove a command from Nucleus shell.

Usage

```
STATUS NU_Unregister_Command (NU_SHELL *p_shell,  
                             char      *cmd);
```

Arguments

- `p_shell`
Shell session handle.
- `cmd`
Command name string pointer.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_UNAVAILABLE`
Command was not found.
- `NU_INVALID_OPERATION`
Indicates service not ready.

Example

```
/* Assume variables are defined locally. */  
STATUS status; /* Unregister Command status */  
  
/* Unregister command. */  
status = NU_Unregister_Command(p_shell, "my_cmd");  
  
/* At this point status indicates if the service was successful and the  
command named "my_cmd" was found and unregistered. */
```

Note



Once a command has been un-registered, it is removed from the list and cannot be used again. It will also not display in the current list of commands available for use.

Related Topics

[NU_Register_Command](#)

[Nucleus Shell Function Reference](#)

NU_Create_Shell

This function is used to create a Shell session.

Usage

```
STATUS NU_Create_Shell (NU_SHELL      **p_shell_return,
                       STATUS          (*io_init) (NU_SHELL *),
                       STATUS          (*io_deinit) (NU_SHELL *),
                       VOID            (*io_puts) (NU_SHELL *, const CHAR *),
                       CHAR            (*io_getch) (NU_SHELL *),
                       VOID            (*io_special) (NU_SHELL *, CHAR),
                       INT             io_echo_on);
```

Arguments

- **p_shell_return**
Pointer to where the Shell session handle will be returned.
- **io_init**
Pointer to I/O init routine.
- **io_deinit**
Pointer to I/O de init routine.
- **io_puts**
Pointer to the I/O put string routine.
- **io_getch**
Pointer to I/O get char routine.
- **io_special**
Pointer to I/O specific handler for “special” characters.
- **io_echo_on**
NU_TRUE means that characters received will be echoed.

Return Values

- **NU_SUCCESS**
Function completed successfully: Session created.
- Operating-system specific error code
Otherwise.

Example

```
/* Assume variables are defined locally. */
STATUS status; /* Command status */

/* Parameters 2-6 are functions that the user defines and they are
registered as part of creating the Shell session */
```

```
NU_SHELL *shl_session;

/* Create Shell session */
status = NU_Create_Shell(&shl_session,
                        shl_init,
                        shl_deinit,
                        shl_puts,
                        shl_getch,
                        shl_special,
                        NU_TRUE);
```

Related Topics

[NU_Delete_Shell](#)

[Nucleus Shell Function Reference](#)

NU_Delete_Shell

This function is used to delete a Shell session.

Usage

```
STATUS NU_Delete_Shell (NU_SHELL *p_shell);
```

Arguments

- `p_shell`
Shell session handle.

Return Values

- `NU_SUCCESS`
Function completed successfully: Session deleted.
- Operating-system specific error code
Otherwise.

Example

```
/* Assume variables are defined locally. */
STATUS status; /* Command status */

/* The parameter is the Shell session that was returned in
   NU_Create_Shell */
NU_SHELL *shl_session;

/* Delete Shell session */
status = NU_Delete_Shell(shl_session);
```

Related Topics

[NU_Create_Shell](#)

[Nucleus Shell Function Reference](#)

NU_Get_Shell_Serial_Session_ID

This function returns the serial Shell session handle.

Note



If you use the default settings that create a Shell session connected to serial I/O, then you need to call this function to get the Shell session handle so that you can register/unregister commands. If you build Shell with serial I/O turned off, this function will return NU_NULL since no Shell session was automatically created.

Usage

```
STATUS NU_Get_Shell_Serial_Session_ID (NU_SHELL **p_shell_return);
```

Arguments

- p_shell_return
Pointer to where the Shell session handle is saved.

Return Values

- NU_SUCCESS
Function completed successfully: Session handle returned.
- NU_NO_SERIAL_SHELL
Only for the case with serial I/O turned off.

Example

```
/* Assume variables are defined locally. */  
STATUS status; /* Command status */  
  
NU_SHELL *shl_session;  
  
/* Get Shell session handle */  
status = NU_Get_Shell_Serial_Session_ID(&shl_session);
```

Related Topics

[NU_Register_Command](#)

[Nucleus Shell Function Reference](#)

Chapter 11

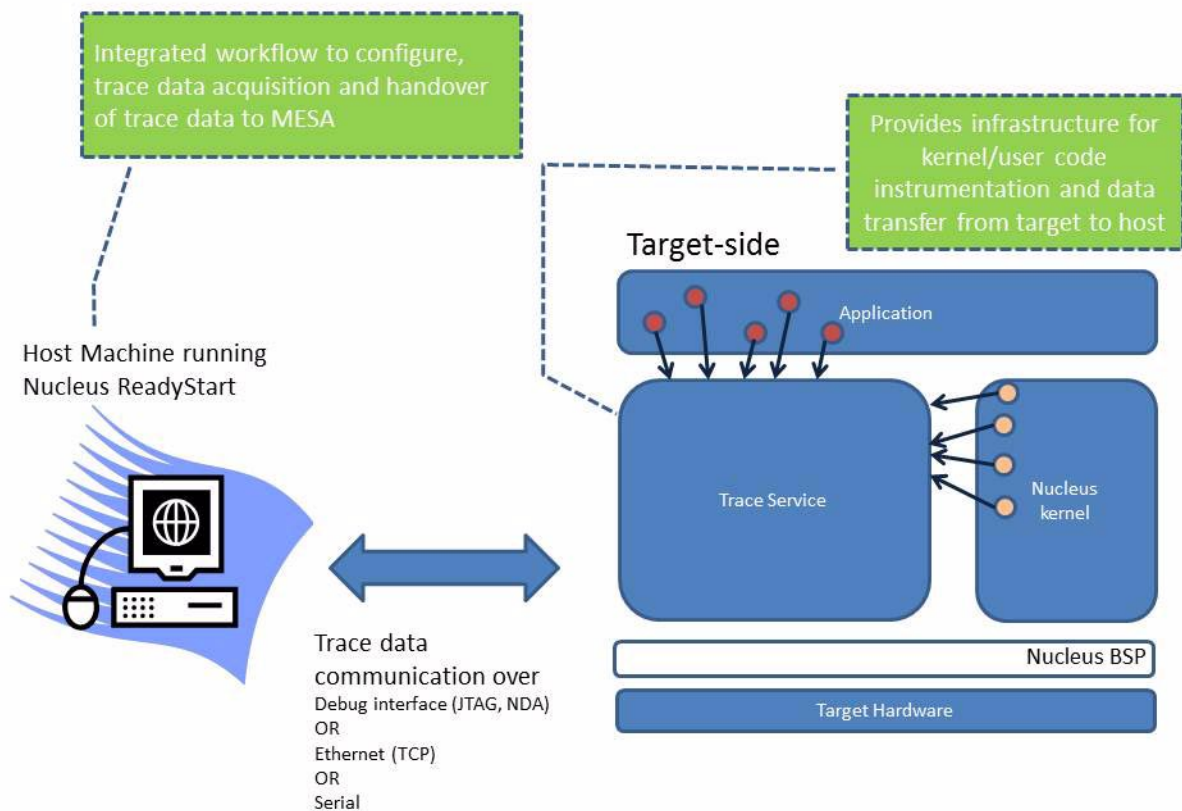
Nucleus Trace Service

Nucleus Trace Overview

Nucleus Trace is an OS service that enables software tracing of Nucleus kernel and user applications. Built-in code instrumentation of the Nucleus Kernel enables tracing of kernel software run-time. The Nucleus Trace API provides you with a flexible framework for instrumentation and tracing application software. The trace data generated at run-time is collected in the memory of the embedded target and can be transmitted to the host using one of many communication methods supported by the trace service, for example Debug Interface (JTAG or Nucleus debug agent), Serial, and Ethernet (TCP).

On the host side, Nucleus ReadyStart Sourcery CodeBench IDE provides an integrated workflow for configuration, acquisition and visualization of trace data generated from the target. [Figure 11-1](#) showcases a high level view of the Nucleus software trace solution.

Figure 11-1. Nucleus Software Trace High Level View



Nucleus Trace Configuration and Initialization

This section defines the various trace related build configurations and describes the decisions you need to make at build time to configure the trace component to suit the needs of the

application. Table 11-1 shows these configuration options, which can be found in the Nucleus UI configurator under *nu.os.svcs.trace.core* and *nu.os.svcs.trace.comms*.

Table 11-1. Nucleus Trace configuration options

Option Name	Default Value	Description
trace_buffer_size in <i>nu.os.svcs.trace.core</i>	0x00400000	This size is be used for allocation of RAM buffer space for run-time trace data collection. The buffering system supports “stop when full” and “overwrite old data” buffering modes. The size of the trace buffer along with the buffering mode choice will determine if the full trace data set or the last N trace logs are available to you at any point in time.
max_trace_pkt_size in <i>nu.os.svcs.trace.core</i>	512	Defines the maximum size a single trace packet can be. This is determined by the maximum payload size used in the application markers.
trace_support in <i>nu.os.svcs.trace.core</i>	true	Enables or disables kernel instrumentation for tracing kernel and middleware run-time.
kernel_default_trace_mask in <i>nu.os.svcs.trace.core</i>	0xFFFFFFFF	A 32-bit mask that enables/disables the pre-instrumented trace markers present in the Nucleus kernel. The granularity of control is at the level of “kernel object type”, for example Tasks, LISRs, HISRs, Queues, Pipes etc.
net_trace_support in <i>nu.os.svcs.trace.core</i>	true	Enables or disables instrumentation for tracing networking run-time.
storage_trace_support in <i>nu.os.svcs.trace.core</i>	true	Enables or disables instrumentation for tracing storage run-time.
pms_trace_support in <i>nu.os.svcs.trace.core</i>	true	Enables or disables instrumentation for tracing power management services run-time.
pc_hotspot_support in <i>nu.os.svcs.trace.core</i>	false	Enables or disables run-time PC sampling hot spot analysis.

Table 11-1. Nucleus Trace configuration options (cont.)

Option Name	Default Value	Description
trace_kernel_boot_time in <i>nu.os.svcs.trace.core</i>	true	Enables or disables tracing of the Nucleus kernel run-time during the bring-up (boot) process.
comms_task_priority in <i>nu.os.svcs.trace.core</i>	255	The Nucleus Trace component consists of a communications task that periodically transmits trace data to the host. This option determines the priority of the trace communications task.
data_tx_period in <i>nu.os.svcs.trace.core</i>	10	The Nucleus Trace component consists of a communications task that periodically transmits trace data to the host. This option determines the timer period for the trace data transmission rate.
comms_flush_default_priority in <i>nu.os.svcs.trace.core</i>	0	This option determines the priority the calling application task will be elevated to when a NU_Trace_Comms_Flush call is made from the application.
overwrite_old_data in <i>nu.os.svcs.trace.core</i>	true	This setting determines the buffering mode of the trace buffering system. "true" configures the system for buffer overwrite mode which causes overwrite of old trace data when the buffer gets full. "false" configures the system for "stop logging when full" mode in which trace logging stops when the buffer is full.
track_trace_overhead in <i>nu.os.svcs.trace.core</i>	false	Enables or disables trace latency timing test.
channel in <i>nu.os.svcs.trace.comms</i>	0	Communication channel used for target to host trace data transfer. 0 = Debug Interface (JTAG or Nucleus Debug Agent) 1 = Serial 2 = Ethernet (TCP) Note: When Serial communication channel is used ensure <platform>.<serial_port>.stdio=0 3 = File

Trace Initialization

The Nucleus Trace service is initialized automatically at system start-up by run-level initialization at run-level 15. When execution control reaches the application, the trace APIs for code instrumentation, arming/disarming of trace markers, and trace communications can be used. There is no need to call [NU_Trace_Initialize](#) from the application to bring up the trace service. During application run-time if a call to [NU_Trace_Deinitialize](#) is made by the application, all trace markers are disabled and trace resources are reclaimed by the system. A subsequent call the [NU_Trace_Initialize](#) will re-initialize the trace service for run-time usage.

Trace Configuration

Nucleus Trace service can be setup to log kernel run-time, application software, middleware (for example, networking and/or storage), and services (for example, power management services), as shown in the following sections:

- [Setup for Kernel Tracing](#)
- [Setup for Application Tracing](#)
- [Setup for Networking Tracing](#)
- [Setup for Storage Tracing](#)
- [Setup for Power Management Services Tracing](#)
- [Setup for Hot-Spot Tracing](#)

Setup for Kernel Tracing

Tracing of Nucleus kernel run-time is accomplished by built-in trace instrumentation present in the Nucleus kernel. This built-in instrumentation is present in kernel code at all time and only takes effect if *nu.os.svcs.trace.core* and *nu.os.svcs.trace.core.trace_support* are enabled.

These can be set and configured using the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#), as in [Figure 11-2](#):

Figure 11-2. Nucleus Configuration Tree for Kernel Tracing

None Manage Filters Template: None Number Format: Hexadecimal

is Configuration Tree

- nu
 - bsp
 - os
 - arch
 - conn
 - drv
 - kern
 - net
 - stor
 - svcs
 - appinit
 - cox
 - dbg
 - dbg_adv
 - init
 - posix
 - pwr
 - registry
 - shell
 - syslog
 - trace
 - comms
 - core
 - ui
 - toolset_settings
 - realview_eb_ct926ejs

Component: core

trace_buffer_size:	0x400000
trace_support:	true
kernel_default_trace_mask:	0xffffffff
trace_kernel_boot_time:	true
net_trace_support:	true
storage_trace_support:	true
pms_trace_support:	true
pc_hotspot_support:	false
max_trace_pkt_size:	0x200
overwrite_old_data:	true
comms_task_priority:	0xff
data_tx_period:	0xa
comms_flush_default_priority:	0x0
track_trace_overhead:	false
runlevel:	15

In order to enable tracing of Nucleus kernel run-time you need to configure the following build time configuration parameters:

- trace_support
- kernel_default_trace_mask
- trace_kernel_boot_time

Related Topics

[Trace Configuration](#)

[Setup for Application Tracing](#)

Setup for Application Tracing

The Nucleus trace component provides you with a flexible set of APIs that enable tracing of application software. Application variables or any significant application event can be traced and visualized using the application trace marker APIs.

NU_Trace_Mark_xx APIs provide a low-overhead means to trace application variables of various data types. The following sample code snippet shows the usage of trace markers to trace application variables. Usage of both NU_Trace_Mark_xx APIs and [NU_Trace_Mark](#) API is shown in the sample code snippet provided.

You can use application trace markers APIs with the ‘.’ notation to group trace data. In the following example the ‘.’ notation adds loop information as follows:

```
INT32  i = 0;
UINT32 count = 0;
float  x = 0;
float  float_val = 0;

for (i = 0; i < 1000; i++)
{
    /* Log Loop Entry */
    NU_Trace_Mark_String("Sine Wave Generator.loop event", "Entering
                        Loop");

    /* Log loop iterator */
    NU_Trace_Mark_I32("Sine Wave Generator.loop iterator", i);

    /* Log count variable */
    NU_Trace_Mark_U32("Sine Wave Generator.count", count);

    /* Compute sine of variable */
    float_val = sin(x);

    /* log float value */
    NU_Trace_Mark_Float("Sine Wave Generator.sin(x)", float_val);

    x +=0.1;
    count +=100;

    if (x == 360){
        x = 0;
    }
}
```

Alternatively, [NU_Trace_Mark](#) API can accept a variable number of arguments and can be used for open-ended trace data logging. Note there is more overhead involved when using this API due to format parsing requirement.

```
INT32  i = 0;
UINT32 count = 0;
float  x = 0;
float  float_val = 0;
```

```
for (i = 0; i < 1000; i++)
{
    /* Compute sine of variable x */
    float_val = sin(x);

    /* log loop iterator */
    NU_Trace_Mark("Sine Wave Generator.sine wave", "%s%f%%s%f%s%i%s%u",
        "x", x, "sin(x)", float_val, "iterator", i, "count", count);

    x +=0.1;
    count +=100;

    if (x == 360){
        x = 0;
    }
}
```

Any significant application event can be traced and visualized. A good example would be to capture entry and exit events around a section of code to identify the time taken to execute the section of code.

Related Topics

[Trace Configuration](#)

[Setup for Kernel Tracing](#)

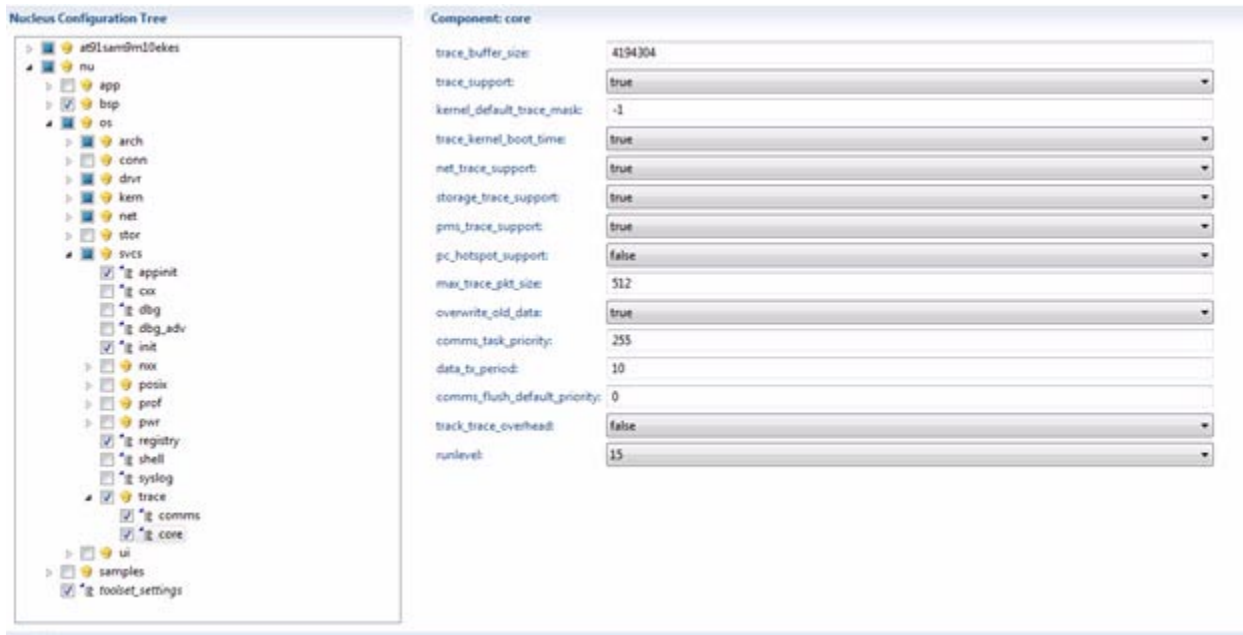
Setup for Networking Tracing

Tracing of Nucleus networking (IPv4/IPv6) run-time is accomplished by built-in trace instrumentation present in Nucleus Networking. This built-in instrumentation is present in the

kernel code at all time and only takes effect if *nu.os.svcs.trace.core*, and *nu.os.svcs.trace.core.net_trace_support* are enabled.

These can be set and configured using the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#), as in [Figure 11-3](#)

Figure 11-3. Nucleus Configuration Tree for Middleware and Services Tracing



In order to enable tracing of Nucleus networking run-time you need to configure the following:

- *trace_support*
- *kernel_default_trace_mask*
- *net_trace_support*

Related Topics

[Trace Configuration](#)

[Setup for Storage Tracing](#)

Setup for Storage Tracing

Tracing of Nucleus storage run-time is accomplished by built-in trace instrumentation present in Nucleus storage. This built-in instrumentation is present in storage code at all time and only takes effect if *nu.os.svcs.trace.core* and *nu.os.svcs.trace.core.storage_trace_support* are enabled.

These can be set and configured using the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#), as in [Figure 11-3](#).

In order to enable tracing of storage run-time you need to configure the following:

- `trace_support`
- `kernel_default_trace_mask`
- `storage_trace_support`

Related Topics

[Trace Configuration](#)

[Setup for Networking Tracing](#)

Setup for Power Management Services Tracing

Tracing of Nucleus power management services (PMS) run-time is accomplished by built-in trace instrumentation present in Nucleus Services. This built-in instrumentation is present in the power services core code at all time and only takes effect if *nu.os.svcs.trace.core* and *nu.os.svcs.trace.core.pms_trace_support* are enabled.

These can be set and configured using the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#), as in [Figure 11-3](#).

In order to enable tracing of Nucleus Power Management Services run-time you need to configure the following:

- `trace_support`
- `kernel_default_trace_mask`
- `pms_trace_support`

Related Topics

[Trace Configuration](#)

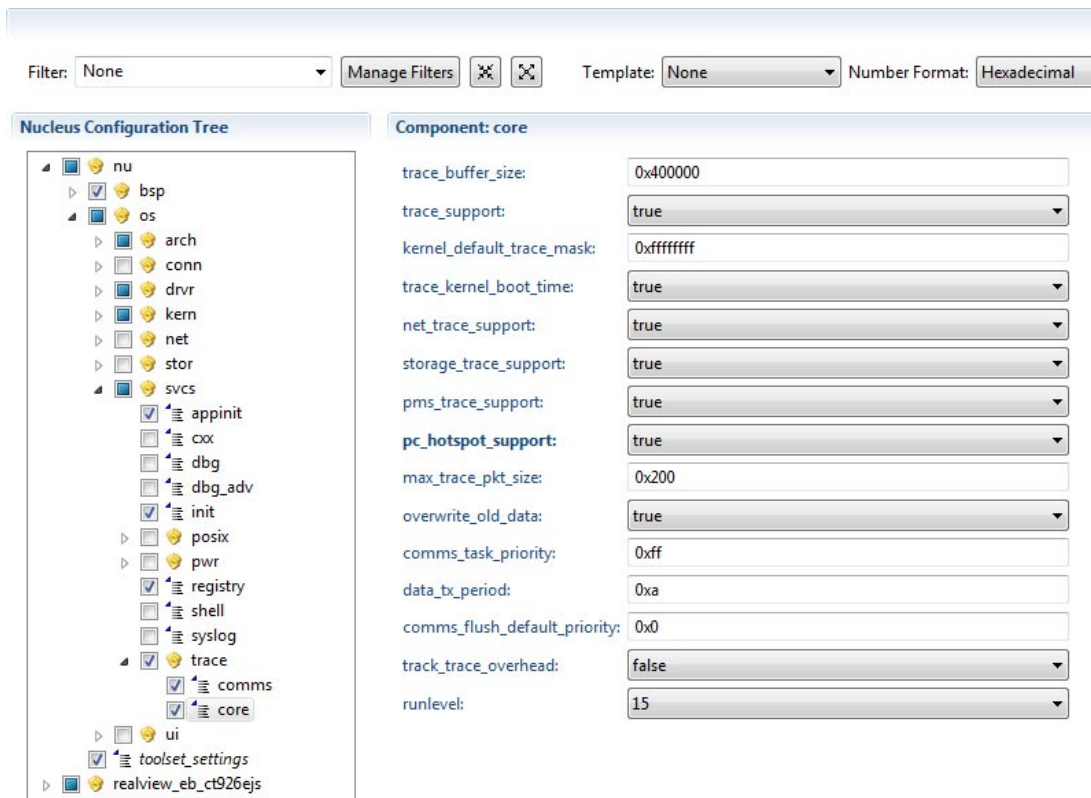
[Setup for Storage Tracing](#)

Setup for Hot-Spot Tracing

Tracing of Nucleus hot spot run-time is accomplished by built-in trace instrumentation present in Nucleus Services. This built-in instrumentation is present in the kernel code at all time and only takes effect if *nu.os.svcs.trace.core* and *nu.os.svcs.trace.core.pc_hotspot_support* are enabled.

These can be set and configured using the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#), as in [Figure 11-4](#).

Figure 11-4. Nucleus Configuration Tree for Hot-Spot Tracing



In order to enable tracing of hot-spot run-time you need to configure the following:

- trace_support
- kernel_default_trace_mask
- pc_hotspot_support

Related Topics

[Trace Configuration](#)

[Setup for Storage Tracing](#)

Controlling the Amount of Trace Data

You can arm or disarm the Nucleus kernel and application trace markers at any point during run-time in order to control the amount of trace data that will be logged. The arming/disarming of trace markers is controlled by the 32-bit mask. The default value of this mask is a build time configured parameter.

In order to control the kernel objects that will be enabled for trace logging during system start up, you should modify the configuration option bellow. Please refer to the [NU_Trace_Arm](#) API reference for mask-bit to kernel object mappings.

- `default_trace_mask`

Once the execution control is in the application, you can use [NU_Trace_Arm](#) or [NU_Trace_Disarm](#) to control logging the data. For example, logging LISRS/HISRS activity could generate a large amount of data which is unnecessary if you are debugging problems outside the context of LISRS/HISRS processing. Therefore you could either choose to disable logging of LISRS/HISRS activity at build time by using an appropriate value for `default_trace_mask` or disable LISRS/HISRS activity from the application using the trace APIs as shown below.

Note

Doing the latter would still cause interrupt activity to be logged during boot time if the default build configuration is used for the trace mask.

```
/* Arm tracing for all objects */
NU_Trace_Arm(NU_TRACE_ALL);

/* Disarm logging of LISRS/HISRS activity */
NU_Trace_Disarm (NU_TRACE_HISRS | NU_TRACE_LISRS);
```

Trace Data Communication

Software based tracing is intrusive on the system run-time. The intrusiveness falls under two categories:

1. The CPU bandwidth consumption by the trace marker APIs.
2. The CPU bandwidth consumption by trace communications.

While [1](#) cannot be avoided, the application developer can minimize the overhead caused by [2](#) by proper usage of the trace communication APIs.

The following sections describe the various options available for transmitting trace data collected on the target to the host:

- [Debug Interface for Trace Data Communication](#)
- [Serial or Ethernet \(TCP\) for Trace Data Communication](#)

Debug Interface for Trace Data Communication

Using this trace communication method you can avoid all overhead associated with trace communications. This is possible because the host asynchronously obtains all of trace data available on the target's trace buffers when the system is halted over the debug channel (the debug channel could be JTAG or Nucleus Debug Agent). An important constraint to take into account here is the trace buffer size, since it determines the amount of trace data that can be

collected and visualized at any point in time. A small buffer can cause buffer over-run (in “overwrite” buffering mode) which can cause loss of old trace data since the buffer would always maintain the last N trace logs. If the buffering system is configured for “non-overwrite” mode the buffer size limits the amount of trace data that can be collected (if the buffer is full any new trace data coming in will not be logged). The limitation with this approach is that the system needs to be halted to fetch trace data from the target.

However, using this approach you will have an out-of-box SW trace experience without having to consider target side configurations for trace communications and physical connections involved.

Serial or Ethernet (TCP) for Trace Data Communication

Using Serial or Ethernet based trace communication is an alternative approach that obviates some of the limitations of Debug Interface based trace communications. In this approach you can make use of trace communications APIs to either enable built-in trace communications infrastructure present within the trace component to periodically transmit trace data to the host or choose the point during application run-time when the trace data available on the target will be transmitted/flushed to the host. This approach makes an effort to make memory available for tracing by either periodically or user controlled transmission of trace data from target to the host during run-time.

FTP Comms Interface for Trace Data Communication

Using FTP Comms based trace communication interface is an ideal solution for profiling a Nucleus based deployed system. This method does not rely on an active debug connection for parsing and/or retrieving trace data from the target. Trace data can be imported from the remote system and local machine over FTP using the Nucleus Trace Data Import wizard based on the built in Eclipse FTP client.

Using the built-in trace data communication task

You can use the [NU_Trace_Comms_Start](#) and [NU_Trace_Comms_Stop](#) APIs to enable the communications task present within the trace component to periodically transmit trace data to the host over the underlying communications medium (Serial or Ethernet TCP). The [NU_Trace_Comms_Start](#) API starts the trace communications task at the priority and trace data transmission time period requested by you. The [NU_Trace_Comms_Stop](#) API stops trace communications. Using these APIs you can start and stop trace communications at any point in the application.

Note



When using this approach, trace data available for visualization may not be the complete trace data set collected, since there is a good chance that the SW run-time produced more trace data than what was consumed by the trace communications task. This approach is suitable when the application requires only light-weight application data tracing.

```
/* Start transmitting trace data */
status = NU_Trace_Comms_Start(0,1);
...
/* Application Code */
...
/* Stop transmitting trace data */
status = NU_Trace_Comms_Stop();
```

Using the trace buffer flush API

The [NU_Trace_Comms_Flush](#) API elevates the priority of the calling application task to a build time configured (*nu.os.svcs.trace.core.comms_flush_default_priority*) high priority and transmits all of trace data available in the trace buffers to the host over the underlying communications medium (Serial or Ethernet TCP). You can either choose to periodically call this API during application run-time or call it at a point in time during application run-time with a break-point after the call to halt the target. With the latter mentioned approach, all of trace data available on the target will be available for visualization when the target halts at the break-point. A sample code snippet is shown below:

```
/* Application code that will be traced */
...

/* This will flush all trace data from the buffer */
status = NU_Trace_Comms_Flush();
```

Using application controlled methods to transmit trace data

The [NU_Trace_Comms_Transmit_N_Packets](#) API allows you to fully control trace data transmission from target to host. You can call this API from the application in a fashion that best fits your application's needs. The priority of the calling task and the frequency at which this API is called will determine the rate at which trace data will be transmitted from target to host.

Configuration for Serial Trace Communications

In order to use serial communications to transmit trace data from target to host, you should ensure the physical connection is made between the target and the host, and the following components are enabled during configuration. To enable and configure components, use the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#).

- *`${platform}.serial0.enable = true`*
- *`${platform}.serial0.stdio = 0`*
- *`nu.os.drvr.serial.enable = true`*
- *`nu.os.svcs.trace.comms.enable = true`*
- *`nu.os.svcs.trace.comms.channel = 1`*

Configuration for Ethernet TCP Based Communications

In order to use Ethernet TCP communications to transmit trace data from target to host, you should ensure that the physical connection is made between target and host.

Note



The preferred method to use Ethernet TCP communication for trace is by assigning a static IP address to the target and using the user assigned static IP address on the host to connect to the target.

The following static IP configuration and components required should be enabled. Use the Nucleus Configuration Editor described in the [Nucleus Readystart Guide](#) to enable/disable and set options as shown below:

- *`${platform}.ethernet0.enable=true`*
- *`${platform}.ethernet0.mw_settings.eth_enable_ipv4_dhcp = 0`*
- *`${platform}.ethernet0.mw_settings.eth_ipv4_address = xxx.xxx.xxx.xxx`*
- *`${platform}.ethernet0.mw_settings.eth_ipv4_netmask = xxx.xxx.xxx.xxx`*
- *`nu.os.net.stack.enable=true`*
- *`nu.os.drvr.eth.enable=true`*
- *`nu.os.svcs.trace.comms.enable = true`*
- *`nu.os.svcs.trace.comms.channel = 2`*

Configuration for FTP Comms Trace Communications

In order to use FTP communications to transmit trace data from target to host, you should ensure the physical connection is made between the target and the host, and the following components are enabled during configuration. To enable and configure components, use the Nucleus Configuration Editor, described in the [Nucleus Readystart Guide](#).

- *`nu.os.svcs.trace.comms.channel = 3`*
- *`nu.os.drvr.eth.enable = true`*

- *nu.os.net.stack.enable = true*
- *nu.os.net.prot.ftp.enable = true*
- *nu.os.drivr.fat_rd.enable = true*
- *nu.os.stor.file.vfs.enable = true*
- *nu.os.stor.file.vfs.mw_mnt_info.enable = true*
- *nu.os.stor.file.fs.fat.enable = true*
- *\${platform}.ethernet0.enable = true*
- *\${platform}.ethernet0.mw_settings.eth_enable_ipv4_dhcp = false*
- *\${platform}.ethernet0.mw_settings.eth_ipv4_address = xxx.xxx.xxx.xxx*
- *\${platform}.ramdisk0.enable = true*

Trace Function APIs

This section lists APIs for tracing application variables and events, arming and disarming markers, and trace data communication. The following is a list of these APIs:

- [NU_Trace_Initialize](#)
- [NU_Trace_Deinitialize](#)
- [NU_Trace_Arm](#)
- [NU_Trace_Disarm](#)
- [NU_Trace_Mark_I32](#)
- [NU_Trace_Mark_U32](#)
- [NU_Trace_Mark_Float](#)
- [NU_Trace_Mark](#)
- [NU_Trace_Mark_String](#)
- [NU_Trace_Comms_Start](#)
- [NU_Trace_Comms_Stop](#)
- [NU_Trace_Comms_Flush](#)
- [NU_Trace_Comms_Transmit_N_Packets](#)

NU_Trace_Initialize

This function initializes the trace service. It also allocates memory for trace buffer and scratch buffer, and it initializes the buffering system.

There is no need to call [NU_Trace_Initialize](#) from the application to bring up the trace service. Trace is automatically initialized during run-level initialization. During application run-time if a call to [NU_Trace_Deinitialize](#) is made by the application, a subsequent call to [NU_Trace_Initialize](#) will re-initialize the trace service for run-time usage.

Usage

```
STATUS NU_Trace_Initialize (VOID);
```

Arguments

- None

Return Values

- `NU_TRUE`
Function completed successfully, trace service initialized.
- Error codes
Function did not complete. Error codes are defined in the file:
`<install_root>\os\include\services\nu_trace.h`.

Example

```
{  
/* Application code */  
  
..  
  
/* De-initialize trace service */  
NU_Trace_Deinitialize();  
  
....  
  
/* Re-initialize trace service */  
NU_Trace_Initialize()  
}
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Deinitialize](#)

NU_Trace_Deinitialize

This function de-initializes the trace service. This includes disabling all trace instrumentation, and de-allocating memory that was allocated for trace and trace communications.

Usage

```
STATUS NU_Trace_Deinitialize (VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Function completed successfully, the trace component was finalized.
- Error codes
Function did not complete. Error codes are defined in the file:
`<install_root>\os\include\services\nu_trace.h`.

Example

```
{  
/* Application code */  
  
..  
  
/* De-initialize trace service */  
NU_Trace_Deinitialize();  
  
....  
  
/* Re-initialize trace service */  
NU_Trace_Initialize()  
}
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Initialize](#)

NU_Trace_Arm

This function arms or enables kernel and/or application trace instrumentation based on the mask field provided.

Usage

```
STATUS NU_Trace_Arm (UINT32 mask);
```

Arguments

- **mask**

Indicates the component to be armed. The values are defined in the file:

`<install_root>\os\include\services\nu_trace.h`

```
/* Mask definitions for each kernel object */
#define NU_TRACE_ALL          0xFFFFFFFF
#define NU_TRACE_APP          (1<<0x0)
#define NU_TRACE_KERN          (~NU_TRACE_APP) /* Enable all kernel hooks */
#define NU_TRACE_LISRS         (1<<0x2)
#define NU_TRACE_HISRS         (1<<0x3)
#define NU_TRACE_TASKS         (1<<0x4)
#define NU_TRACE_DMEM          (1<<0x5)
#define NU_TRACE_PMEM          (1<<0x6)
#define NU_TRACE_MAILBOX       (1<<0x7)
#define NU_TRACE_QUEUE         (1<<0x8)
#define NU_TRACE_PIPE          (1<<0x9)
#define NU_TRACE_SEMAPHORE     (1<<0xA)
#define NU_TRACE_EVENT         (1<<0xB)
#define NU_TRACE_SIGNAL        (1<<0xC)
#define NU_TRACE_TIMER         (1<<0xD)
#define NU_TRACE_CPU_STATE     (1<<0xE)
```

Return Values

- **NU_SUCCESS**

Function completed successfully, the intended components of kernel were armed.

- **Error codes**

Function did not complete. Error codes are defined in the file:

`<install_root>\os\include\services\nu_trace.h`.

Example

```
/* Arm tracing for LISRS and HISRS objects */
NU_Trace_Arm((NU_TRACE_HISRS | NU_TRACE_LISRS));

/* Arm tracing for all objects */
NU_Trace_Arm(NU_TRACE_ALL);

/* Arm kernel and Application tracing (Arm ALL) */
NU_Trace_Arm (NU_TRACE_KERN | NU_TRACE_APP);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Disarm](#)

NU_Trace_Disarm

This function disarms or disables the kernel and/or application trace instrumentation based on the mask field provided.

Usage

```
STATUS NU_Trace_Disarm (UINT32 mask);
```

Arguments

- **mask**
Indicates the component to be disarmed. Values are defined in the file:
<install_root>\os\include\services\nu_trace.h

Return Values

- **NU_SUCCESS**
Function completed successfully, trace component was finalized.
- **Error codes**
Function did not complete. Error codes are defined in the file:
<install_root>\os\include\services\nu_trace.h.

Example

```
/* Disable Partition Memory profiling */  
NU_Trace_Disarm(NU_TRACE_PMEM);  
  
/* Disarm all object profiling */  
NU_Trace_Disarm(NU_TRACE_ALL);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Arm](#)

NU_Trace_Mark_I32

This function logs an event <string, value> to trace a signed integer.

Usage

```
VOID NU_Trace_Mark_I32 (char          *evt_str,  
                       signed int    i32_val);
```

Arguments

- `event_str`
A unique string identifier for an event type.
- `i32_val`
Value of the point traced.

Return Values

- None

Example

```
/* Log application integer data */  
NU_Trace_Mark_I32("Application Integer Data", i32_val);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Mark_U32](#)

[NU_Trace_Mark_Float](#)

[NU_Trace_Mark](#)

NU_Trace_Mark_U32

This function logs an event <string, value> to trace an unsigned integer.

Usage

```
VOID NU_Trace_Mark_U32 (char          *evt_str,  
                       unsigned long  u32_val);
```

Arguments

- `evt_str`
A unique string identifier for an event type.
- `u32_val`
Value of the point traced.

Return Values

- None

Example

```
/* Log application unsigned integer data */  
NU_Trace_Mark_U32("Application Unsigned Integer Data",  
                 u32_val);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Mark_I32](#)

[NU_Trace_Mark_Float](#)

[NU_Trace_Mark](#)

NU_Trace_Mark_Float

This function logs an event <string, value> to trace a float value.

Usage

```
VOID NU_Trace_Mark_Float (char          *evt_str,  
                        float          float_val);
```

Arguments

- `evt_str`
A unique string identifier for an event type.
- `float_val`
Value of the point traced.

Return Values

- None

Example

```
/* Log application float data */  
NU_Trace_Mark_Float("Application floating point data",  
                  float_val);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Mark](#)

[NU_Trace_Mark_I32](#)

[NU_Trace_Mark_U32](#)

NU_Trace_Mark_String

This function logs an event <string, value> to trace a string value.

Usage

```
VOID NU_Trace_Mark_String (char          *evt_str,  
                           char          str_val);
```

Arguments

- `evt_str`
A unique string identifier for an event type.
- `str_val`
Value of the point traced.

Return Values

- None

Examples

```
/* Log application string data */  
NU_Trace_Mark_Float("Application string data",  
                   str_val);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Mark_Float](#)

NU_Trace_Mark

This function logs a <format_specifier, values...> n-tuple to trace a variable number of arguments.

Usage

```
VOID NU_Trace_Mark (char *event_type,  
                   char *format,  
                   ...);
```

Arguments

- event_type
A unique string identifier for an event type.
- format
Format of the event. The following formats are supported:
 - %s - string
 - %d - integer
 - %i - integer
 - %u - unsigned integer
 - %x - unsigned integer
 - %X - unsigned integer
 - %f - float
- ...
Other arguments to be traced.

Return Values

- None

Description

The [NU_Trace_Mark](#) API is a variable parameter function that requires at least three parameters. The first three parameters are the event id number, event string, and the third parameter is a string format similar to printf.

This function can be used as follows:

```
NU_Trace_Mark("USER VAR TRACE DATA" ,"%s%f%i%u",  
             "STR Example", user_float, user_int, user_unsigned_int)
```

Alternatively this function can be used for “printf” like trace logging.

Example

```
/* Trace vehicle dynamics- variables of interest */
```

```
NU_Trace_Mark("vehicle_dynamics", "%s%f%s%f%s%f", "vehicle_speed",  
             vehicle_speed,"actuation", throttle, "noise", noise);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Mark_Float](#)

NU_Trace_Comms_Start

This function starts the communications task present within trace service at the priority and trace data transmission time period specified by the arguments of this API. The trace communications task periodically transmits trace data to the host over the underlying communications medium (Serial or Ethernet TCP).

Usage

```
STATUS NU_Trace_Comms_Start (UINT8  comms_tsk_priority,  
                             UINT32  comms_tx_period);
```

Arguments

- **comms_tsk_priority**
Priority of the trace communication task. The default value is a build time configuration *nu.os.svcs.trace.core.comms_task_priority*.
- **comms_tx_period**
Time period for the data transmission rate of the trace communications task. The default value is a build time configuration *nu.os.svcs.trace.core.data_tx_period*.

Return Values

- **NU_SUCCESS**
Function completed successfully, the trace component was started.
- **Error codes**
Function did not complete. Error codes are defined in the file:
<install_root>\os\include\services\nu_trace.h.

Example

```
STATUS status;  
  
/* Start trace data communications */  
status = NU_Trace_Comms_Start(0,10);  
  
/* Application code */  
....  
  
/* Stop trace data communications */  
status = NU_Trace_Comms_Stop();
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Comms_Stop](#)

NU_Trace_Comms_Stop

This function stops the communications task present within the trace service. The trace communications task periodically transmits trace data to the host over the underlying communications medium (Serial or Ethernet TCP).

Usage

```
STATUS NU_Trace_Comms_Stop (VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Function completed successfully, trace communication stopped.
- **Error codes**
Function did not complete. Error codes are defined in the file:
<install_root>\os\include\services\nu_trace.h.

Example

```
STATUS status;  
  
/* Start trace data communications */  
status = NU_Trace_Comms_Start(0,10);  
  
/* Application code */  
....  
  
/* Stop trace data communications */  
status = NU_Trace_Comms_Stop();
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Comms_Start](#)

[NU_Trace_Comms_Flush](#)

NU_Trace_Comms_Flush

This function elevates the priority of the calling application task to a build time configured (*nu.os.svcs.trace.core.comms_flush_default_priority*) high priority and transmits all of trace data available in the trace buffers to the host over the underlying communications medium (Serial or Ethernet TCP). Control returns to the caller only when the trace buffer is empty or on an error condition.

Usage

```
STATUS NU_Trace_Comms_Flush (VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Function completed successfully, all trace data available was flushed.
- **Error codes**
Function did not complete. Error codes are defined in the file:
<install_root>\os\include\services\nu_trace.h.

Example

```
STATUS status;  
  
/* Application code */  
....  
  
/* This will flush all trace data from the buffer */  
status = NU_Trace_Comms_Flush();
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Comms_Start](#)

[NU_Trace_Comms_Stop](#)

NU_Trace_Comms_Transmit_N_Packets

This function transmits N trace packets.

Note



This call will block the caller until N trace data packets are transmitted to the host. If the buffer is empty the call will return control to the caller.

Usage

```
UINT32 NU_Trace_Comms_Transmit_N_Packets (UINT32 num_packets);
```

Arguments

- `num_packets`
Number of packets to transmit.

Return Values

- Number of data packets transmitted.
- Error codes

Function did not complete. Error codes are defined in the file:

`<install_root>\os\include\services\nu_trace.h`.

Example

```
STATUS status;  
  
/* Application code */  
....  
  
/* Trace 10 communication packets */  
UINT32 NU_Trace_Comms_Transmit_N_Packets (10);
```

Related Topics

[Trace Function APIs](#)

[NU_Trace_Comms_Flush](#)

Chapter 12

Dynamic Download

Dynamic Download Overview

Dynamic Download is a runtime service to enable dynamic loading, execution and unloading of applications to Nucleus.

Dynamic Download consists of two major components: the Loader and the Runtime. The Loader component, when enabled, is initialized at runlevel 15 and resides with the kernel. The runtime component facilitates the development of a dynamically loadable application.

Dynamic Download Limitations

- The DDL service only allows C applications and does not support downloading applications written in C++.
- Loading of a dynamic application using debug agent is not supported. When an application is loaded from the file system, all the dynamic loaded application structures will be available through the debug agent and newly created tasks can be debugged.

Dynamic Download Terminology

Table 12-1. Nucleus Dynamic Download Terminology

Term	Description
Position Independent Code (PIC)	Code that is generated to execute based on a relative offset from any address. The code base address is typically stored in a register that is used to calculate all jumps and branches.
Position Independent Data (PID)	Data that is referenced based on an offset from a base address. When the base address is supplied, it is used to calculate the physical address of data being manipulated.
Dynamic Application	An executable image that resides on some storage media that is downloaded to the target and executed on the fly.
Service Export Table	A table or structure of function pointers that contain the address of their respective API service calls.
Application Module ID	A unique identity given to an application module by the kernel executive when that module is loaded and executed.

Dynamic Download Application Development

Developing Dynamic Applications is very similar to developing standard applications. The basic differences are that in a standard application, there is code to initialize the hardware, create the various Nucleus components that the application will use, and during the link phase of the build process, linking to the Nucleus library. In a Dynamic Application, because there is no startup code to initialize the hardware, and there is no Nucleus library to link to in the build process, the applications must be linked with the Runtime library from the Dynamic Download service. All references to Nucleus services are resolved by linking to the Service Export Table.

No public data is visible to any dynamic application. If an application must gain access to or use global data from the Kernel, it can make a Nucleus service call to gain access. Public data in an application is also limited in scope to being visible only in that application.

Depending on your application, you must consider the following options when developing dynamically loadable applications and modules:

- Link to *runtime.lib* instead of *nucleus.lib*.
- Specify the toolset to generate position independent code.
- Specify `NU_Main` as the entry function for the application.
- Use *rvct_ddl_app.ld* or *csgnu_arm_ddl_app.ld* (depending on your toolset) as the linker command file for your application.

- Include the following lines in your downloadable application files:

```
#define LOADABLE_APPLICATION
#include "services/nu_services.h"
```

- Include an `NU_Exit()` function. Your application must provide an `NU_Exit()` function. This function is called when the kill API is called for the module. This function is responsible for cleaning up the resources this module used during its life.

The following example file section is for both `csgnu_arm` and `rvct` toolsets. The metadata file section addresses the caveats for the linker command file and toolset command line switches:

```
ldflags "csgnu_arm" => "-Wl,--entry=NU_Main"
ldscript "csgnu_arm" => "../../../../../toolset/csgnu_arm_ddl_app.ld"

ldflags "rvct" => "--entry NU_Main"
ldscript "rvct" => "../../../../../toolset/rvct_ddl_app.ld"

sources { cflags "csgnu_arm" => "-fPIC"
          cflags "rvct" => "--apcs=fpic"
  Dir.glob("*.c") }
libraries { ["runtime.lib"] }
```

Dynamic Download Function Reference

Nucleus Dynamic Download contains three APIs that load, resume, or exit application modules.

- [NU_Load_Module_From_File](#)
- [NU_Get_Load_Address](#)
- [NU_Kill_Module](#)
- [NU_Resume_Module](#)

NU_Load_Module_From_File

This function loads a module into the memory from the target file system. It creates the main thread for Dynamic Application and depending upon the options parameter it starts it running or leaves in suspended state.

Usage


```
STATUS NU_Load_Module_From_File (CHAR    *path,  
                                INT      argc,  
                                CHAR    **argv,  
                                UINT32   *module_id,  
                                UNSIGNED options);
```

Arguments

- **path**
Path of the module file (ELF format only) to be loaded in the target file system.
- **argc**
User specified argument vector element count to be passed to application
- **argv**
User specified argument vector passed to application
- **module_id**
Return parameter that will be updated to contain the loaded module ID value if the operation is successful.
- **options**
Operation options.
 - 0 - Indicates no options applied; default behavior.
 - NU_LOAD_SUSPEND - Indicates module should be loaded and then placed in suspended state (main task).
 - NU_LOAD_RUN - Indicates module should be loaded and then placed in ready to run state.

Return Value

- **NU_SUCCESS**
Function completed successfully.
- **<Nucleus FILE error codes>**
If a problem is encountered accessing the file then the Nucleus FILE error codes will be returned.
- **<Loader error codes>**
If a problem is encountered in the loader component, then a Loader error code will be returned.

 **Note**
Currently, this release supports loading a module from a file only. NU_SUCCESS or an error code indicating the reason for failure.

Example

```
STATUS      status;  
CHAR        root_string[20]="x:\\App_Image.out";  
UINT32      module_id;  
  
status = NU_Load_Module_From_File(root_string, NU_NULL, NU_NULL,  
                                &module_id, NU_LOAD_RUN);
```

Related Topics

[Dynamic Download Function Reference](#)

[NU_Kill_Module](#)

[NU_Resume_Module](#)

NU_Get_Load_Address

This function returns the load address of the executable identified by `module_id`.

Usage

```
STATUS NU_Get_Load_Address(UINT32 module_id,  
                           VOID    **load_addr);
```

Arguments

- `module_id`
Module ID of the loaded executable image.
- `load_addr`
On return, if the STATUS is `NU_SUCCESS`, it returns the address where the image was loaded.

Return Values

- `NU_SUCCESS`
Operation completed successfully.
- `UNKNOWN_MODULE_ID`
Did not find a matching module ID.

Example

```
STATUS    status;  
CHAR      root_string[20]="x:\\App_Image.out";  
UINT32    module_id;  
VOID      **load_addr;  
  
status = NU_Load_Module_From_File(root_string, NU_NULL, NU_NULL,  
                                  &module_id, NU_LOAD_RUN);  
  
/* Get the load address */  
status = NU_Get_Load_Address(module_id, &load_addr);
```

Related Topics

[Dynamic Download Function Reference](#)

[NU_Load_Module_From_File](#)

NU_Kill_Module

Unloads a module from the system. The default behavior will be to call the application's registered terminate function and then terminate the module's main thread.

Usage

```
STATUS NU_Kill_Module(UINT  module_id,  
                      INT32 options);
```

Arguments

- `module_id`
ID of previously loaded module to be killed.
- `options`
Operation options.
0 - Indicates no options applied; default behavior.

Return Value

- `NU_SUCCESS`
Function completed successfully.
- `<Loader error codes>`
If a problem is encountered in the loader component, then a loader error code will be returned.

Example

```
STATUS      status;  
CHAR        root_string[20]="x:\\App_Image.out";  
UINT32      module_id;  
  
status = NU_Load_Module_From_File(root_string, NU_NULL, NU_NULL,  
                                &module_id, NU_LOAD_RUN);  
  
status = NU_Kill_Module(module_id, NU_NULL);
```

Related Topics

[Dynamic Download Function Reference](#)

[NU_Load_Module_From_File](#)

[NU_Resume_Module](#)

NU_Resume_Module

Resumes a loaded module if it is not already running. The default behavior will be to resume execution of the main task for the module.

Usage

```
STATUS NU_Resume_Module(UINT    module_id,  
                        UINT32  options);
```

Arguments

- `module_id`
ID of previously loaded and suspended module to resume.
- `options`
Operation options.
0 - Indicates no options applied; default behavior.

Return Value

- `NU_SUCCESS`
Function completed successfully.
- `<Loader error codes>`
If a problem is encountered in the loader component, then a loader error code will be returned.

Example

```
STATUS      status;  
CHAR        root_string[20]="x:\\App_Image.out";  
UINT32      module_id;  
  
status = NU_Load_Module_From_File(root_string, NU_NULL, NU_NULL,  
                                &module_id, NU_LOAD_SUSPEND);  
  
status = NU_Resume_Module(module_id, NU_NULL);
```

Related Topics

[Dynamic Download Function Reference](#)

[NU_Kill_Module](#)

[NU_Load_Module_From_File](#)

Chapter 13

Power Management Services (PMS)

PMS Overview

PMS are divided into the following subsections:

- **Idle Scheduler** – Hardware APIs related to CPU scheduling functionality. Some functionality, such as the provision of CPU_Idle function is provided by the CPU Driver (instead of the API). See [“Saving Memory During Hibernate”](#) on page 593.
- **DVFS Service** – Hardware APIs to manage and transition between supported Voltage-Frequency operating points of the CPU. Hardware-specific information, such as available DVFS operating points, is exposed by the hardware agnostic DVFS Service APIs. [“DVFS API Functions”](#) on page 597.
- **Peripheral State Service** – Hardware APIs to manage and transition between supported Peripheral Device Driver power states. [“Peripheral State API Functions”](#) on page 615.
- **System State Service** – APIs to manage and transition between different system power states. A system power state is defined as a set of peripheral power states. [“System State Service APIs”](#) on page 622.
- **Watchdog Service** – APIs available to the drivers and applications to set up watchdog timers. Watchdog timers can be used by drivers when hardware timers are not readily available. [“Watchdog API Functions”](#) on page 633.
- **Hibernate** - Is a method to save power when the system is in long periods of non-use. The CPU and all peripherals are put into a low power mode or turned off to minimize power usage. When an event occurs, the system is awoken and Nucleus fully restores the state of the target. Currently Nucleus supports two hibernate levels (availability based on BSP): dormant and standby. [“Hibernate APIs”](#) on page 640.
- **Non-Volatile Memory** - NVM is a memory resource provided by a target that is guaranteed to be persistent across power cycles. NVM is used to store the state of the system during a transition to the dormant hibernate level, allowing the system to be shut down and later return to that same state during a subsequent power-up. There are various technologies that might provide NVM resources on a target, for example NAND/NOR flash memory or an SD card. Targets may have one or more hardware resources suitable for providing NVM resources to the hibernate system, or a target may have none, in which case dormant level hibernation is not possible on that target. [“NVM APIs”](#) on page 646.

Note

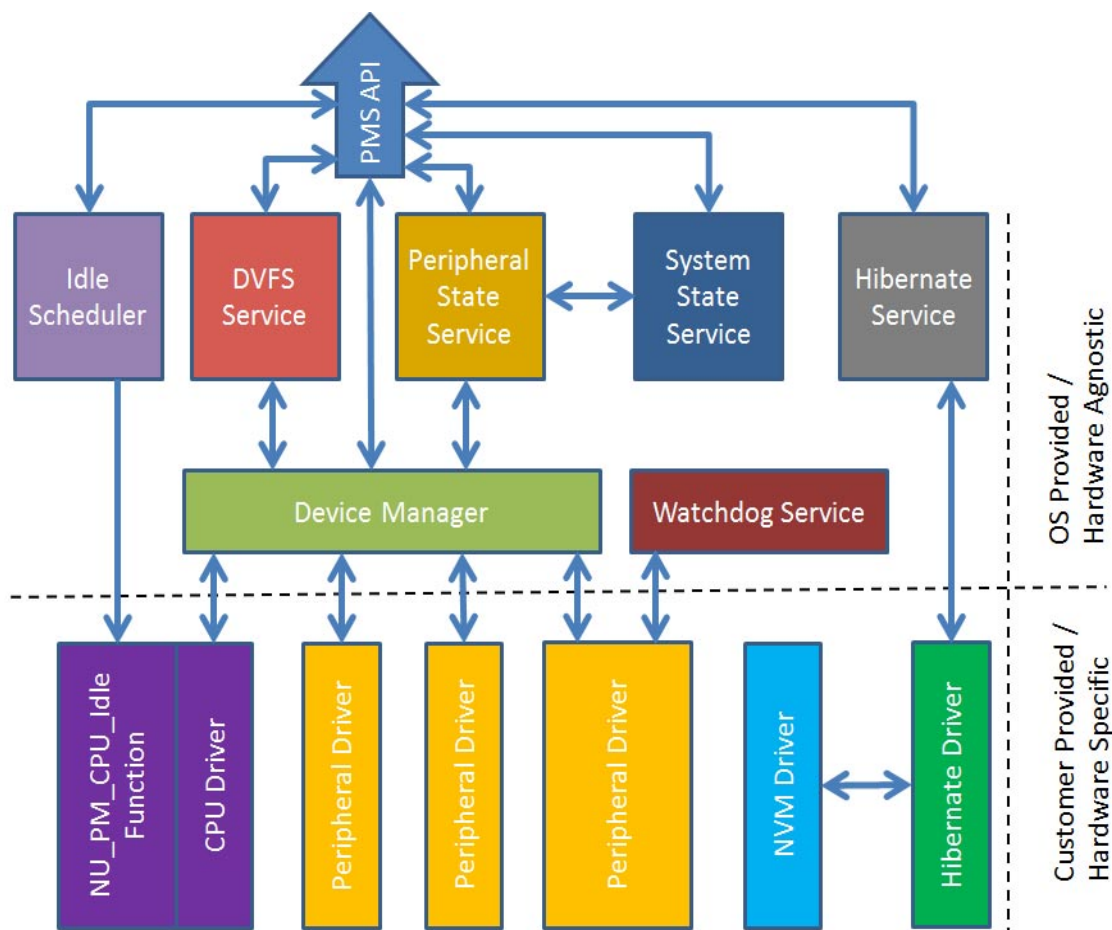
The Power Interface Driver must be enabled (*nu.os.drvr.pwr_intf.enable=true*) when Power Services are enabled (*nu.os.svcs.pwr.enable=true*). This is the default configuration in the source tree and in the *power.config* file at *<install_root>\nucleus\config*.

Figure 13-1 shows a functional block diagram of the PMS system including each API subsection.

Caution

In your applications, use only interfaces, structures, macros, and so on, that are documented within this and other Nucleus reference guides. There is no guarantee of future support or compatibility for any interface that is not documented.

Figure 13-1. PMS Components



Terminology

Table 13-1 describes terminology used throughout this guide.

Table 13-1. Terminology

Term/Abbreviation	Description
PMS	Power Management Services
Peripheral	This term refers to a hardware component connected to the CPU, for example, UART, USB controller, Camera, Keypad, and Display Module.
System	This term refers to the entire system product, including the CPU and all of its peripherals.
Application layer	The software layers above the PMS, including any user applications.
Device ID	The unique ID for every device driver instance (one for each hardware device present, plus any virtual devices) assigned by and obtained from the Device Manager.
Park	The process of ceasing use of the bus.
Hibernate	A system that allows extensive power savings on multiple levels. Hibernate levels are different from operating points (OPs) in that OPs are frequency and voltage pairings while hibernate levels are defined per BSP to maximize power savings.
Dormant	A hibernate level where the CPU is placed in a very low power or turned off, devices such as serial ports, Ethernet ports, LCD devices, etc. are turned off, and RAM is turned off. Before turning the RAM off, a record of all the CPU registers and state information, as well as the contents of the RAM are saved to non-volatile memory. Entry into and exit from dormant mode is sometimes lengthy based on the speed of saving RAM to NVM.
Standby	A hibernate level where the CPU is placed in a very low power state or turned off and RAM is placed in a self refresh state. Entry into and exit from standby mode is much faster than entry into and exit from dormant mode as nothing in RAM is saved to an external location. Standby hibernate mode is possible without NVM, since the contents of the RAM do not need to be saved to NVM.
NVM	The NVM system provides a means for the Nucleus OS to save off and recover state information across a power cycle of the target. This allows the Nucleus OS to save power by hibernating when services are not needed for long periods of time.

Saving Memory During Hibernate

The hibernate system saves and restores all information necessary to transition a Nucleus system to a hibernate level and back. This includes various parts of memory, such as the *.data*

and *.bss* sections of the image and any memory allocated from Nucleus memory pools. The static memory regions (for example the *.data* and *.bss* sections of the image) that are saved and restored are kept in a table that is used during the hibernate level transition process. If there are additional, custom memory sections that should be saved and restored in order to have a Nucleus application function correctly, these sections may be added to the table. That table can be located in the hibernate driver with the BSP.

Hibernate Run-level Usage

All devices and middle-ware are notified about a pending transition to hibernate via the run-level interface. A value of `RUNLEVEL_HIBERNATE` will be sent on transition to hibernate and a value of `RUNLEVEL_RESUME` will be sent when exiting a hibernate state. It should be noted that unlike normal run-level these states are not run with the scheduler active.

Idle Scheduler API Functions

This section describes the following Idle Scheduler API functions:

- [NU_PM_Get_CPU_Counters](#)
- [NU_PM_Start_Tick_Suppress](#)
- [NU_PM_Stop_Tick_Suppress](#)

NU_PM_Get_CPU_Counters

This function retrieves the free running CPU Utilization Counters. Applications can poll these at desired intervals and calculate CPU utilization since the last poll:

```
CPU_Idle_Percentage = 100 * (idle_time - last_idle_time) /  
    (total_time - last_total_time)  
CPU_Utilization_Percentage = 100 - CPU_Idle_Percentage;
```

where `total_time = *total_time_ptr`, `idle_time = *idle_time_ptr`, `last_idle_time` and `last_total_time` are values obtained on the previous poll.

Usage

```
STATUS NU_PM_Get_CPU_Counters (UINT32 *total_time_ptr,  
                               UINT32 *idle_time_ptr);
```

Arguments

- `total_time_ptr`
This is a pointer where the total time counter value is to be retrieved to.
- `idle_time_ptr`
This is a pointer where the idle time counter value is to be retrieved to.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This indicates that one of the return pointers is null.

Description

Although the units of these counters are arbitrary, they are the same for both counters on any system. This means that although the units can change from system to system, because they are the same for any given system, the busy and idle percentage calculations are always valid.

Both counters are retrieved as an atomic operation to prevent synchronization issues.

The counters never reset and they overflow periodically for no less than 60 seconds each time.

Related Topics

[Saving Memory During Hibernate](#)

[NU_PM_Start_Tick_Suppress](#)

[NU_PM_Stop_Tick_Suppress](#)

NU_PM_Start_Tick_Suppress

This function activates the OS timer tick suppression

Usage

```
STATUS NU_PM_Start_Tick_Suppress (VOID);
```

Return Values

- **NU_SUCCESS**
Function completed successfully.

Related Topics

[Saving Memory During Hibernate](#)

[NU_PM_Get_CPU_Counters](#)

[NU_PM_Stop_Tick_Suppress](#)

NU_PM_Stop_Tick_Suppress

This function deactivates the OS timer tick suppression.

Usage

```
STATUS NU_PM_Stop_Tick_Suppress (VOID);
```

Return Values

- **NU_SUCCESS**
Function completed successfully.

Related Topics

[Saving Memory During Hibernate](#)

[NU_PM_Start_Tick_Suppress](#)

[NU_PM_Get_CPU_Counters](#)

DVFS API Functions

This section describes the following DVFS Service API functions:

- [NU_PM_DVFS_Control_Transition](#)
- [NU_PM_DVFS_Register](#)
- [NU_PM_DVFS_Unregister](#)
- [NU_PM_DVFS_Update_MPL](#)
- [NU_PM_Get_Current_OP](#)
- [NU_PM_Get_Freq_Count](#)
- [NU_PM_Get_Freq_Info](#)
- [NU_PM_Get_OP_Additional_Info](#)
- [NU_PM_Get_OP_Specific_Info](#)
- [NU_PM_Get_OP_Count](#)
- [NU_PM_Get_OP_VF](#)
- [NU_PM_Get_Voltage_Count](#)
- [NU_PM_Get_Voltage_Info](#)
- [NU_PM_Release_Min_OP](#)
- [NU_PM_Request_Min_OP](#)
- [NU_PM_Set_Current_OP](#)

NU_PM_DVFS_Control_Transition

This function is called to control DVFS transitions.

NU_TRUE allows DVFS transitions to occur and *NU_FALSE* cancels any in progress and prevents any from occurring in the future.

Usage

```
STATUS NU_PM_DVFS_Control_Transition(BOOLEAN new_value,  
                                     BOOLEAN *previous_value);
```

Arguments

- `new_value`
Enable/Disable DVFS transition value.
- `previous_value`
Pointer to the old value will be written.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This indicates the provided `previous_value` pointer is invalid.
- `PM_INVALID_PARAMETER`
This indicates `new_value` has unknown value.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_DVFS_Update_MPL](#)

NU_PM_DVFS_Register

This function is called by the driver to register for DVFS notification/synchronization.

Usage

```
STATUS NU_PM_DVFS_Register(PM_DVFS_HANDLE *dvfs_handle_ptr,  
                           VOID           *instance_handle_ptr,  
                           PM_NOTIFY_FUNC dvfs_notify_cb);
```

Arguments

- `dvfs_handle_ptr`
This is the pointer to the location where the handle will be written.
- `instance_handle_ptr`
This is the pointer to the device instance handle.
- `dvfs_notify_cb`
This is the function pointer that will be called for park and resume notification calls.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_ALREADY_REGISTERED`
This indicates that the driver is already registered.
- `PM_INVALID_POINTER`
This indicates one of the provided pointers, *dvfs_handle_ptr* or *dvfs_notify_cb* is null.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Control_Transition](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_DVFS_Update_MPL](#)

NU_PM_DVFS_Unregister

This function is called by the driver to un-register for DVFS notification/synchronization.

Usage

```
STATUS NU_PM_DVFS_Unregister(PM_DVFS_HANDLE dvfs_handle );
```

Arguments

- `dvfs_handle`

This is the handle returned by the `NU_PM_DVFS_Register` when registering.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_NOT_REGISTERED`
This indicates that the driver is not registered.
- `PM_INVALID_REG_HANDLE`
This indicates the provided handle is not found in the system.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Control_Transition](#)

[NU_PM_DVFS_Update_MPL](#)

NU_PM_DVFS_Update_MPL

This function is called by the driver to update the Maximum Parking Latencies.

Usage

```
STATUS NU_PM_DVFS_Update_MPL (PM_DVFS_HANDLE dvfs_handle,  
                              PM_MPL         *mpl_ptr,  
                              PM_DVFS_NOTIFY dvfs_notify );
```

Arguments

- **dvfs_handle**
This is the handle of the DVFS-aware device that is updating.
- **mpl_ptr**
This is a pointer to the structures containing the MPL parameters.
- **dvfs_notify**
This is an enumeration that toggles notification from the DVFS system.

Return Values

- **NU_SUCCESS**
Function completed successfully
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.
- **PM_INVALID_POINTER**
This indicates the `mpl_ptr` is null.
- **PM_INVALID_MPL**
This indicates the time the duration is shorter than the combination of the park and resume times.
- **PM_INVALID_REG_HANDLE**
This indicates the `dvfs_handle` is invalid.

Example

```
typedef struct PM_MPL_CB  
{  
    UINT32 duration; /* Maximum time driver can be parked */  
    UINT32 park_time; /* Maximum time to park driver in */  
    UINT32 resume_time; /* Maximum time needed to resume */  
} PM_MPL;
```

All times are in nanoseconds.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Control_Transition](#)

NU_PM_Get_Current_OP

This function retrieves the current Operating Point ID.

Usage

```
STATUS NU_PM_Get_Current_OP (UINT8 *op_id_ptr );
```

Arguments

- `op_id_ptr`
This points to the location the OP ID will be retrieved to.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This error indicates the provided pointer is invalid.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_DVFS_Update_MPL](#)

NU_PM_Get_Freq_Count

This function returns the total numbers of frequency IDs (max_id+1).

Usage

```
STATUS NU_PM_Get_Freq_Count (UINT8 *freq_count_ptr );
```

Arguments

- `freq_count_ptr`
This points to the location the frequency ID's will be retrieved into.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This error indicates the provided pointer is invalid.
- `PM_NOT_INITIALIZED`
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_Get_Current_OP](#)

NU_PM_Get_Freq_Info

This function retrieves the frequency (in Hz) associated with the freq_id.

Usage

```
STATUS NU_PM_Get_Freq_Info (UINT8    freq_id,  
                           UINT32 *freq_ptr );
```

Arguments

- **freq_id**
This is the freq_id point in question.
- **freq_ptr**
This is the pointer where the frequency (in Hz) will be retrieved to.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_FREQ_ID**
This error code indicates the provided frequency ID does not exist.
- **PM_INVALID_POINTER**
This error indicates the provided pointer is invalid.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_Get_Freq_Count](#)

NU_PM_Get_OP_Additional_Info

This function retrieves any additional information available on the specified OP.

The information returned is specific to the CPU driver.

Usage

```
STATUS NU_PM_Get_OP_Additional_Info(UINT8 op_id,  
                                   VOID *info_ptr,  
                                   UINT32 size);
```

Arguments

- **op_id**
ID of the operating point of interest
- **info_ptr**
This points to the location where the retrieved information will be placed.
- **size**
Size of the information pointer

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_OP_ID**
This indicates the provided OP is invalid.
- **PM_INVALID_POINTER**
This error indicates the provided pointer is invalid.
- **PM_DVFS_INFO_FAILURE**
This error indicates the IOCTL function is not implemented in the CPU driver.
- **PM_NOT_INITIALIZED**
This error indicates the power services are not fully initialized.
- **PM_INVALID_SIZE**
This error indicates the size passed to the function is 0.

Related Topics

[DVFS API Functions](#)

[NU_PM_DVFS_Register](#)

[NU_PM_DVFS_Unregister](#)

[NU_PM_Get_Current_OP](#)

NU_PM_Get_OP_Specific_Info

This function retrieves any additional information available on the specified operating points.

Usage

```
STATUS NU_PM_Get_OP_Specific_Info(UINT8 op_id,  
                                  CHAR  *identifier,  
                                  VOID  *info_ptr,  
                                  UINT32 size);
```

Arguments

- **op_id**
ID of the operating point of interest.
- **identifier**
String that identifies what is being requested. You can obtain valid values for *identifier* by either looking at your hardware manual or by looking at `cpu.h` for your bsp. The `cpu.h` file resides in the `<nucleus_install_dir>/bsp/$TARGET/include/bsp/drivers/cpu/$TARGET` directory.
- **info_ptr**
Points to where the retrieved information will be placed.
- **size**
Size of the information pointer.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_POINTER**
This error indicates one or both provided pointers are invalid.
- **PM_INVALID_OP_ID**
This error indicates the provided operating point is invalid.
- **PM_INVALID_SIZE**
This error indicates the size passed to the function is 0. Size cannot be zero.
- **PM_NOT_INITIALIZED**
This error indicates the power services are not fully initialized.
- **PM_DVFS_INFO_FAILURE**
This error indicates the IOCTL function is not implemented in the CPU driver.

Related Topics

[DVFS API Functions](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Additional_Info](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Get_OP_Count

This function retrieves the total number of operating points available (max op_id+1).

Usage

```
STATUS NU_PM_Get_OP_Count (UINT8 *op_count_ptr );
```

Arguments

- `op_count_ptr`
Pointer to the number of operating points available.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_OP_ID`
This error indicates the provided operating point is invalid.
- `PM_INVALID_POINTER`
This error indicates one or both provided pointers are invalid.
- `PM_NOT_INITIALIZED`
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_Get_OP_Additional_Info](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Specific_Info](#)

NU_PM_Get_OP_VF

This function retrieves the Voltage and Frequency ID's associated with a particular operating point (not the current op).

Usage

```
STATUS NU_PM_Get_OP_VF (UINT8 op_id,  
                        UINT8 *freq_id_ptr,  
                        UINT8 *voltage_id_ptr);
```

Arguments

- **op_id**
This is the operating point ID of interest.
- **freq_id_ptr**
This points to the location the frequency ID will be retrieved into.
- **voltage_id_ptr**
This points to the location the voltage ID information will be retrieved into.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_OP_ID**
This error indicates the provided operating point is invalid.
- **PM_NOT_INITIALIZED**
This error indicates the power services are not fully initialized.
- **PM_INVALID_POINTER**
This error indicates voltage or frequency pointers are null.

Related Topics

[DVFS API Functions](#)

[NU_PM_Get_OP_Additional_Info](#)

[NU_PM_Get_OP_Specific_Info](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Get_Voltage_Count

This function returns the total number of voltage IDs (max_id+1).

Usage

```
STATUS NU_PM_Get_Voltage_Count (UINT8 *voltage_count_ptr );
```

Arguments

- `voltage_count_ptr`
This points to the location the number of voltage IDs will be retrieved into.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This error indicates the provided pointer is invalid.
- `PM_NOT_INITIALIZED`
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_Get_Voltage_Info](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Get_Voltage_Info

This function retrieves a voltage value (in mV) associated with the `voltage_point_id`.

Usage

```
STATUS NU_PM_Get_Voltage_Info(UINT8 voltage_id,  
                              INT16 *voltage_ptr );
```

Arguments

- `voltage_id`
This is the specified voltage ID.
- `voltage_ptr`
This points to the location where the retrieved voltage value will be placed.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_VOLTAGE_ID`
This error code indicates that the provided frequency ID does not exist.
- `PM_INVALID_POINTER`
This error indicates the provided pointer is invalid.
- `PM_NOT_INITIALIZED`
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_Voltage_Count](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Release_Min_OP

This function releases a minimum operating point request by way of the NU_PM_Request_Min_OP call.

Usage

```
STATUS NU_PM_Release_Min_OP(PM_MIN_REQ_HANDLE handle );
```

Arguments

- **handle**
This is the request handle returned from NU_PM_Request_Min_OP.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_REQ_HANDLE**
This error indicates the provided handle is invalid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.
- **PM_NOT_INITIALIZED**
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_Request_Min_OP](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Request_Min_OP

This function requests a minimum operating point below which the DVFS Service will not switch to until the request is released by way of the NU_PM_Release_Min_OP call.

Usage

```
STATUS NU_PM_Request_Min_OP(UINT8          op_id,  
                             PM_MIN_REQ_HANDLE *handle_ptr );
```

Arguments

- **op_id**
This is the minimum operating point requested.
- **handle_ptr**
This is the pointer location where the request handle will be written.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_TRANSITION_FAILED**
This indicates that an attempted transition to the requested OP failed. The request is canceled in this case.
- **PM_INVALID_OP_ID**
This error indicates that the supplied OP is invalid.
- **PM_INVALID_POINTER**
This error indicates the provided pointer is invalid.
- **PM_NOT_INITIALIZED**
This error indicates the power services are not fully initialized.

Related Topics

[DVFS API Functions](#)

[NU_PM_Release_Min_OP](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Count](#)

NU_PM_Set_Current_OP

This function attempts to transition the CPU into a specified operating point.

Note



This API gives up the current thread to run a separate thread that handles OP changes.

Usage

```
STATUS NU_PM_Set_Current_OP (UINT8 op_id );
```

Arguments

- `op_id`
This is the new operating point ID.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_TRANSITION_FAILED`
This return code indicates that the DVFS service was unable to transition due to current peripheral activity.
- `PM_INVALID_OP_ID`
This error indicates the provided pointer is invalid.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.
- `PM_NOT_INITIALIZED`
This error indicates the power services are not fully initialized.
- `PM_REQUEST_BELOW_MIN`
This error indicates the requested OP is below the established minimum OP.

Related Topics

[DVFS API Functions](#)

[NU_PM_Request_Min_OP](#)

[NU_PM_Get_OP_VF](#)

[NU_PM_Get_OP_Count](#)

Peripheral State API Functions

This section describes the following Peripheral State Service API functions:

- [NU_PM_Get_Power_State](#)
- [NU_PM_Get_Power_State_Count](#)

- [NU_PM_Min_Power_State_Release](#)
- [NU_PM_Min_Power_State_Request](#)
- [NU_PM_Set_Power_State](#)

NU_PM_Get_Power_State

This function returns the current power state of the peripheral.

Usage

```
STATUS NU_PM_Get_Power_State(DM_DEV_ID dev_id,  
                             PM_STATE_ID *state_ptr );
```

Arguments

- **dev_id**
This is the device ID of the peripheral.
- **state_ptr**
This points to the location to place the retrieved state ID.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_POINTER**
This error indicates the provided pointer is invalid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Peripheral State API Functions](#)

[NU_PM_Get_Power_State_Count](#)

[NU_PM_Min_Power_State_Release](#)

[NU_PM_Min_Power_State_Request](#)

NU_PM_Get_Power_State_Count

This function returns the total number of power states supported by the specified device (max_state_id + 1).

Usage

```
STATUS NU_PM_Get_Power_State_Count(DM_DEV_ID dev_id,  
                                   PM_STATE_ID *state_count_ptr);
```

Arguments

- dev_id
This is the device ID of the peripheral.
- state_count_ptr
This points to the location to place the retrieved state count.

Return Values

- NU_SUCCESS
Function completed successfully.
- PM_INVALID_POINTER
This error indicates the provided pointer is invalid.
- PM_UNEXPECTED_ERROR
This indicates an unexpected error has occurred.

Related Topics

[Peripheral State API Functions](#)

[NU_PM_Get_Power_State](#)

[NU_PM_Min_Power_State_Release](#)

[NU_PM_Min_Power_State_Request](#)

NU_PM_Min_Power_State_Release

This function releases a previously placed request for a minimum peripheral power state.

Usage

```
STATUS NU_PM_Min_Power_State_Release (DM_DEV_ID      dev_id,  
                                     PM_MIN_REQ_HANDLE handle );
```

Arguments

- **dev_id**
This is the device ID of the peripheral.
- **handle**
This is the request handle originally returned by `NU_PM_Min_Power_State_Request`.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_REQ_HANDLE**
This error indicates that the provided request handle is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Peripheral State API Functions](#)

[NU_PM_Get_Power_State_Count](#)

[NU_PM_Set_Power_State](#)

[NU_PM_Min_Power_State_Request](#)

NU_PM_Min_Power_State_Request

This function requests that a device not transition below a specified power state. That minimum value is held until the caller releases the request using the `NU_PM_Min_Power_State_Release` handle retrieved from this call.

Usage

```
STATUS NU_PM_Min_Power_State_Request (DM_DEV_ID      dev_id,  
                                     PM_STATE_ID    state,  
                                     PM_MIN_REQ_HANDLE *handle_ptr );
```

Arguments

- `dev_id`
This is the device ID of the peripheral.
- `state`
This is the desired minimum state.
- `handle_ptr`
This points to the location to place the request handle.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POWER_STATE`
This indicates that the power state requested is not valid for the device.
- `PM_INVALID_POINTER`
This error indicates the provided pointer is invalid.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[Peripheral State API Functions](#)

[NU_PM_Get_Power_State_Count](#)

[NU_PM_Min_Power_State_Release](#)

[NU_PM_Set_Power_State](#)

NU_PM_Set_Power_State

This function invokes the transition of the specified device to the specified power state.

Usage

```
STATUS NU_PM_Set_Power_State (DM_DEV_ID    dev_id,  
                             PM_STATE_ID  state );
```

Arguments

- **dev_id**
This is the device ID of the peripheral.
- **state**
This is the desired new state.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_DEFERRED**
This indicates that the peripheral noted the requested state change, however, at this time there is a Minimum State Request holding the state at a higher value.**PM_INVALID_POWER_STATE**
This indicates that the power state requested is not valid for the device.

Description

- **FIXED POWER STATE VALUES**
0 is always OFF, 255 is always substituted by this function to the maximum power state available for a particular device (up to and including a value of 255).

Related Topics

[Peripheral State API Functions](#)

[NU_PM_Get_Power_State_Count](#)

[NU_PM_Min_Power_State_Release](#)

[NU_PM_Min_Power_State_Request](#)

System State Service APIs

This section describes the following System State Service API functions:

- [NU_PM_Emergency_State_Release](#)
- [NU_PM_Emergency_State_Request](#)
- [NU_PM_Get_System_State](#)
- [NU_PM_Get_System_State_Count](#)
- [NU_PM_Get_System_State_Map](#)
- [NU_PM_Map_System_Power_State](#)
- [NU_PM_Unmap_System_Power_State](#)
- [NU_PM_System_Min_State_Release](#)
- [NU_PM_System_Min_State_Request](#)
- [NU_PM_Set_System_State](#)
- [NU_PM_System_State_Init](#)

NU_PM_Emergency_State_Release

This function releases a previously placed request for an emergency system state.

Usage

```
STATUS NU_PM_Emergency_State_Release(PM_MIN_REQ_HANDLE handle);
```

Arguments

- **handle**
Request handle to be released.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_REQ_HANDLE**
This indicates that the requested handle is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Get_System_State](#)

[NU_PM_Emergency_State_Request](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_Emergency_State_Request

This function places a request for an emergency system state. The system state is not allowed to rise above this requested state until the request is released.

Usage

```
STATUS NU_PM_Emergency_State_Request(UINT8 state,  
                                     PM_MIN_REQ_HANDLE *handle_ptr);
```

Arguments

- **state**
Desired system state
- **handle_ptr**
Pointer to location of request handle

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_POINTER**
This error indicates the provided pointer is invalid
- **PM_INVALID_SYSTEM_STATE**
This indicates the system state requested is not valid for the device
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred

Related Topics

[System State Service APIs](#)

[NU_PM_Get_System_State](#)

[NU_PM_Emergency_State_Release](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_Get_System_State

This function returns the current system state.

Usage

```
STATUS NU_PM_Get_System_State (UINT8 *system_state_ptr );
```

Arguments

- `system_state_ptr`
This points to the location to place the retrieved system state.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This error code indicates that the supplied pointer is not valid.
- `PM_SYSTEM_STATE_NEED_INIT`
This indicates that the system states service has not been initialized.

Related Topics

[System State Service APIs](#)

[NU_PM_Emergency_State_Request](#)

[NU_PM_Get_System_State_Count](#)

[NU_PM_Get_System_State_Map](#)

NU_PM_Get_System_State_Count

This function retrieves the total number of system states available (max state_id+1).

Usage

```
STATUS NU_PM_Get_System_State_Count (UINT8 *state_count_ptr);
```

Arguments

- `state_count_ptr`
This is the total number of system states.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_POINTER`
This indicates that the provided pointer is invalid.
- `PM_SYSTEM_STATE_NEED_INIT`
This indicates that system state service is not initialized.

Related Topics

[System State Service APIs](#)

[NU_PM_Get_System_State](#)

[NU_PM_Emergency_State_Request](#)

[NU_PM_Get_System_State_Map](#)

NU_PM_Get_System_State_Map

This function retrieves a set peripheral state defined for a specified system state.

Usage

```
STATUS NU_PM_Get_System_Map (UNIT8      system_state,  
                             DM_DEV_ID  dev_id,  
                             PM_STATE_ID *state_ptr );
```

Arguments

- **system_state**
This specifies the system power state to map to.
- **dev_id**
This is the device ID of the peripheral to map to the system state.
- **state_ptr**
This is the pointer to the location where the defined peripheral state is retrieved to.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_SYSTEM_STATE**
This indicates that an invalid system state is specified.
- **PM_INVALID_STATE**
This indicates an invalid state for the device ID.

Related Topics

[System State Service APIs](#)

[NU_PM_Set_System_State](#)

[NU_PM_Get_System_State](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_Map_System_Power_State

This function maps a system power state into a specific peripheral power state.

Usage

```
STATUS NU_PM_Map_System_Power_State(UNIT8      system_state,  
                                     DM_DEV_ID  dev_id,  
                                     PM_STATE_ID state);
```

Arguments

- **system_state**
This specifies the system power state to map to.
- **dev_id**
This is the device ID of the peripheral to map to the system state.
- **state**
This is the power state of the peripheral to map to the system state.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_SYSTEM_STATE**
This indicates that an invalid system state is specified.
- **PM_INVALID_STATE**
This indicates an invalid state for the device ID.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Set_System_State](#)

[NU_PM_Get_System_State](#)

[NU_PM_Unmap_System_Power_State](#)

NU_PM_Unmap_System_Power_State

This function unmaps a system power state from a specific peripheral power state.

Usage

```
STATUS NU_PM_Unmap_System_Power_State(DV_DEV_ID dev_id);
```

Arguments

- `dev_id`
Device id of the peripheral to unmap to the system state.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_DEVICE_ID`
This indicates that the supplied `dev_id` is not valid or not mapped.
- `PM_SYSTEM_STATE_NEED_INIT`
This indicates that the system state services are not initialized.
- `PM_UNEXPECTED_ERROR`
This indicates that an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Map_System_Power_State](#)

NU_PM_System_Min_State_Release

This function releases the previously placed request for the minimum system power state.

Usage

```
STATUS NU_PM_System_Min_State_Release (PM_MIN_REQ_HANDLE handle);
```

Arguments

- **handle**
This is the request handle to be released.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_REQ_HANDLE**
This error indicates that the provided request handle is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Get_System_State](#)

[NU_PM_System_Min_State_Request](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_System_Min_State_Request

This function places a request for a minimum system state.

The system state will not be allowed to be drop below this requested state until the request is released by way of the `NU_PM_Release_Min_System_State` call.

Usage

```
STATUS NU_PM_System_Min_State_Request (UINT8          system_state,  
                                       PM_MIN_REQ_HANDLE *handle_ptr );
```

Arguments

- `system_state`
This is the minimum state requested.
- `handle_ptr`
This is the pointer used to request the handle that is used to release the request.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_SYSTEM_STATE`
This indicates that the power state requested is not valid for the device.
- `PM_INVALID_POINTER`
This error indicates that the provided pointer is invalid.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_System_Min_State_Release](#)

[NU_PM_Get_System_State](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_Set_System_State

This function executes a transition to the specified system power state.

Usage

```
STATUS NU_PM_Set_System_State (UINT8 system_state);
```

Arguments

- `system_state`
This is the system state ID to transition to.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_DEFERRED`
This indicates that the peripheral noted the requested state change, however, at this time there is a Minimum State Request holding the state at a higher value.
- `PM_INVALID_STATE`
This error code indicates that the supplied `state_id` is not valid.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Get_System_State_Map](#)

[NU_PM_Get_System_State](#)

[NU_PM_Get_System_State_Count](#)

NU_PM_System_State_Init

This function initializes the System State Service with a specified number of states.

Note



This function can only be run once.

Usage

```
STATUS NU_PM_System_State_Init (UINT8 system_state_count );
```

Arguments

- `system_state_count`
This is the total number of system states to be initialized.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_DUPLICATE_INIT`
This indicates that the system states have already been initialized.
- `PM_UNEXPECTED_ERROR`
This indicates an unexpected error has occurred.

Related Topics

[System State Service APIs](#)

[NU_PM_Set_System_State](#)

[NU_PM_Get_System_State](#)

[NU_PM_Get_System_State_Count](#)

Watchdog API Functions

This section describes the following Watchdog Service API functions:

- [NU_PM_Create_Watchdog](#)
- [NU_PM_Delete_Watchdog](#)
- [NU_PM_Set_Watchdog_Notification](#)
- [NU_PM_Reset_Watchdog](#)
- [NU_PM_Set_Watchdog_Notification](#)
- [NU_PM_Is_Watchdog_Expired](#)

NU_PM_Create_Watchdog

This function creates a watchdog timer with a specified timeout duration.

Usage

```
STATUS NU_PM_Create_Watchdog (UINT16      timeout_value,  
                             PM_WD_HANDLE *handle_ptr );
```

Arguments

- **timeout_value**
This is the desired timeout value in 100's of ms (value of 1 = 100ms timeout).
- **handle_ptr**
This points to the location where the watchdog handle will be returned.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_POINTER**
This error indicates that the provided pointer is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Delete_Watchdog](#)

[NU_PM_Is_Watchdog_Active](#)

[NU_PM_Reset_Watchdog](#)

NU_PM_Delete_Watchdog

This function deletes the created watchdog. It also releases any blocked NU_PM_Is_Watchdog_Expired calls for the deleted watchdog.

Usage

```
STATUS NU_PM_Delete_Watchdog (PM_WD_HANDLE handle );
```

Arguments

- **handle**
This is the handle of the watchdog to be deleted.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_INVALID_WD_HANDLE**
This error indicates that the provided watchdog handle is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Create_Watchdog](#)

[NU_PM_Is_Watchdog_Active](#)

[NU_PM_Reset_Watchdog](#)

NU_PM_Is_Watchdog_Active

This function checks whether a specified watchdog is not expired.

Usage

```
STATUS NU_PM_Is_Watchdog_Active(PM_WD_HANDLE handle,  
                                UINT8          blocking_flag );
```

Arguments

- **handle**
This is the handle of the watchdog to be checked.
- **blocking_flag**
If this is non-zero, the call blocks while the watchdog is expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_WD_EXPIRED**
This indicates that the watchdog is expired.
- **PM_WD_NOT_EXPIRED**
This indicates that the watchdog is not expired.
- **PM_WD_DELETED**
This indicates that the watchdog has been deleted.
- **PM_INVALID_WD_HANDLE**
This error indicates that the provided watchdog is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Delete_Watchdog](#)

[NU_PM_Set_Watchdog_Notification](#)

[NU_PM_Reset_Watchdog](#)

NU_PM_Reset_Watchdog

This function resets the watchdog counter.

Note



This function must be compiled inline. It is a fast, fixed time function.

Usage

```
STATUS NU_PM_Reset_Watchdog (PM_WD_HANDLE handle );
```

Arguments

- handle

This is the handle of the watchdog to be reset.

Return Values

- NU_SUCCESS

Function completed successfully.

- PM_INVALID_WD_HANDLE

This error indicates that the provided watchdog handle is not valid.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Create_Watchdog](#)

[NU_PM_Is_Watchdog_Active](#)

[NU_PM_Is_Watchdog_Expired](#)

NU_PM_Set_Watchdog_Notification

This function sets a notification to be sent on behalf of the driver when the watchdog transitions into a specified `wd_status` state.

The sent notification always has a NUL msg.

Usage

```
STATUS NU_PM_Set_Watchdog_Notify(PM_WD_HANDLE handle,  
                                STATUS          wd_status,  
                                DM_DEV_ID      sender_dev_id);
```

Arguments

- `handle`
This is the handle of the watchdog.
- `wd_status`
This is the status that triggers the notification send. The only valid values are:
 - `PM_WD_EXPIRED` – notification to be sent when watchdog expires
 - `PM_WD_NOT_EXPIRED` – notification to be sent when watchdog becomes active
 - `PM_WD_DELETED` – notification to be sent when watchdog is deleted
- `sender_dev_id`
This is the device ID to use as the sender of the notification.

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `PM_INVALID_WD_HANDLE`
This error indicates that the provided watchdog is not valid.
- `PM_INVALID_WD_STATUS`
This error indicates that an unsupported `wd_status` was specified.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Delete_Watchdog](#)

[NU_PM_Is_Watchdog_Active](#)

[NU_PM_Is_Watchdog_Expired](#)

NU_PM_Is_Watchdog_Expired

This function checks whether a specified watchdog is expired.

Usage

```
STATUS NU_PM_Is_Watchdog_Expired(PM_WD_HANDLE handle,  
                                UINT8          blocking_flag );
```

Arguments

- **handle**
This is the handle of the watchdog to be checked.
- **blocking_flag**
If this is non-zero, the call blocks while the watchdog is not expired.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **PM_WD_EXPIRED**
This indicates that the watchdog is expired.
- **PM_WD_NOT_EXPIRED**
This indicates that the watchdog is not expired.
- **PM_WD_DELETED**
This indicates that the watchdog has been deleted.
- **PM_INVALID_WD_HANDLE**
This error indicates that the provided watchdog is not valid.
- **PM_UNEXPECTED_ERROR**
This indicates an unexpected error has occurred.

Related Topics

[Watchdog API Functions](#)

[NU_PM_Is_Watchdog_Active](#)

[NU_PM_Set_Watchdog_Notification](#)

[NU_PM_Reset_Watchdog](#)

Hibernate APIs

This section contains descriptions of the following APIs:

- [NU_PM_Set_Hibernate_Level](#)
- [NU_PM_Hibernate_Boot](#)
- [NU_PM_Get_Hibernate_Exit_OP](#)
- [NU_PM_Set_Hibernate_Exit_OP](#)
- [NU_PM_Hibernate_Level_Count](#)

NU_PM_Set_Hibernate_Level

This function transitions the CPU into a specified hibernate level.

Note



This API gives up the current thread to run a separate thread that handles the CPU configuration.

```
STATUS NU_PM_Set_Hibernate_Level (UINT8 level);
```

Arguments

- level

This is the new hibernate level:

NU_STANDBY

CPU in the lowest power mode, RAM in lower power mode

NU_DORMANT

CPU in lowest power mode, RAM off

Return Values

- NU_SUCCESS
Indicate successful operation.
- PM_UNEXPECTED_ERROR
Indicates an unexpected error has occurred.
- PM_NOT_INITIALIZED
Indicates the power services are not fully initialized.

Example

```
/* Assume variables are defined locally. */
STATUS status;          /* Set hibernate level status */

/* Request that the system transition to the dormant hibernate level.*/
status = NU_PM_Set_Hibernate_Level(NU_DORMANT);

/* If status is NU_SUCCESS, the system has transitioned to the dormant
hibernate level.*/
```

Related Topics

[NU_PM_Hibernate_Boot](#)

[Hibernate APIs](#)

NU_PM_Hibernate_Boot

This function retrieves the previous boot state for the system.

Usage

```
BOOLEAN NU_PM_Hibernate_Boot (VOID);
```

Arguments

- None

Return Values

- **NU_TRUE**
Previous boot state was hibernate.
- **NU_FALSE**
Previous boot state was normal boot.

Example

```
/* Assume variables are defined locally. */  
BOOLEAN is_hibernate_boot; /* Is the previous boot state hibernate? */  
  
/* Retrieve the previous boot state. */  
is_hibernate_boot = NU_PM_Hibernate_Boot();  
  
/* The is_hibernate_boot variable will indicate if the previous boot state  
is hibernate or normal. */
```

Related Topics

[NU_PM_Set_Hibernate_Level](#)

[Hibernate APIs](#)

NU_PM_Get_Hibernate_Exit_OP

This function gets the current value of the hibernate system internal OP variable that will be used to set the OP of the system when exiting a hibernate state.

Usage

```
STATUS NU_PM_Get_Hibernate_Exit_OP (UINT8 *op_id_ptr);
```

Arguments

- `op_id_ptr`
Pointer to the location of the retrieved OP id.

Return Values

- `NU_SUCCESS`
Successful transition.
- `NU_INVALID_POINTER`
Invalid return pointer.

Example

```
/* Assume variables are defined locally. */
STATUS status;          /* Get hibernate exit OP status */
UINT8 op_id;           /* ID of the OP that will be set on hibernate exit. */

/* Retrieve the current hibernate exit OP. */
status = NU_PM_Get_Hibernate_Exit_OP(&op_id);

/* If status is NU_SUCCESS, then op_id will contain the current OP that
will be set on exit from a hibernate level. */
```

Related Topics

[NU_PM_Set_Hibernate_Exit_OP](#)

[Hibernate APIs](#)

NU_PM_Set_Hibernate_Exit_OP

This function sets the desired OP that will transition the CPU on exit from hibernate.

Note



Note that setting the OP to a value below the minimum will result in the minimum OP being set on hibernate exit.

Usage

```
STATUS NU_PM_Set_Hibernate_Exit_OP (UINT8 op_id);
```

Arguments

- `op_id`
This is the desired new OP id.

Return Values

- `NU_SUCCESS`
Indicates successful operation.
- `PM_INVALID_OP_ID`
Invalid OP.

Example

```
/* Assume variables are defined locally. */  
STATUS status;          /* Get hibernate exit OP status */  
  
/* Set the current hibernate exit OP. */  
status = NU_PM_Get_Hibernate_Exit_OP(3);  
  
/* If status is NU_SUCCESS, then 3 will be set as the current OP that will  
be set as the system OP on exit from a hibernate level. */
```

Related Topics

[NU_PM_Get_Hibernate_Exit_OP](#)

[Hibernate APIs](#)

NU_PM_Hibernate_Level_Count

This function returns the number of hibernate levels in the system.

Usage

```
UINT8 NU_PM_Hibernate_Level_Count (VOID);
```

Arguments

- None

Return Values

- Number of hibernate levels in the system.

Example

```
/* Assume variables are defined locally. */
UINT8 hibernate_level_count;          /* Number of hibernate levels */

/* Retrieve the number of hibernate levels available in the system */
hibernate_level_count = NU_PM_Hibernate_Level_Count();

/* The hibernate_level_count variable will contain the number of hibernate
levels available in the system. */
```

Related Topics

[NU_PM_Set_Hibernate_Level](#)

[Hibernate APIs](#)

NVM APIs

The NVM API is made up of a number of predefined structures and data types which support a set of functions. The basic high-level operations supported by the API are:

- **Information**
Provides a way for the Nucleus OS to determine size of the hardware NVM resources available on the target.
- **Writing**
Provides a way for the Nucleus OS to store system data to the NVM resource.
- **Reading**
Provides a way for the Nucleus OS to retrieve system data from the NVM resource.

The API presents the NVM resource as a sequential series of items of variable size, like an array with variable-sized elements. The total size of the NVM resource and the size of the largest single item currently stored in the NVM resource may be retrieved through the API. All read and write operations are performed within a session context, with the NVM resource being opened at the start of the session and closed at the end. Sessions for writing and reading should be separate, meaning that all writing should be done in one single session and reading done in the subsequent session. The writing process is sequential and must begin with a "reset" to prepare for a new series of writes. Each successive write operation copies a variable-sized item (range of memory) into the NVM resource and returns an index for the item. The first written item will receive an index of 0, the second an index of 1, etc. The read process allows random-access of items stored in the NVM resource based on the item index provided.

This section describes the following Non-Volatile Memory Service API functions:

- [NVM_Tgt_Open](#)
- [NVM_Tgt_Close](#)
- [NVM_Tgt_Read](#)
- [NVM_Tgt_Write](#)
- [NVM_Tgt_Reset](#)
- [NVM_Tgt_Info](#)

NVM_Tgt_Open

This function opens the NVM resource and session.

Note



Note that this function must be called before any other calls. This function initializes the supporting target hardware.

Usage

```
Status NVM_Tgt_Open (VOID);
```

Arguments

- None

Return Values:

- NU_SUCCESS
Indicates successful operation.
- Other internal error values.

Example

```
/* Assume variables are defined locally. */  
STATUS status;  
/* Non-Volatile Memory open status */  
  
/* Open NVM storage. */  
status = NVM_Tgt_Open();  
/* At this point status indicates if the service was successful. */
```

Related Topics

[NVM_Tgt_Close](#)

[NVM APIs](#)

NVM_Tgt_Close

This function closes the NVM resource and session.

Note



Note that this function must be called after all calls are complete. Termination activities may include the flushing of write buffers to target hardware.

Usage

```
STATUS NVM_Tgt_Close(VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Indicates successful operation.
- Other internal error values.

Example

```
/* Assume variables are defined locally. */  
STATUS status;  
/* Non-Volatile Memory close status */  
  
/* Close NVM storage. */  
status = NVM_Tgt_Close();  
/* At this point status indicates if the service was successful. */
```

Related Topics

[NVM_Tgt_Open](#)

[NVM APIs](#)

NVM_Tgt_Read

This function reads the specified item from the NVM resource.

Usage

```
STATUS NVM_Tgt_Read (VOID *p_item,  
                    UINT *p_item_size,  
                    UINT index);
```

Arguments

- **p_item**
Pointer to item to be read.
- **p_item_size**
Points to the return value that contains the size of the item read (in bytes), if successful. If NULL is passed, this value is not returned.
- **index**
Index to read item from.

Return Values

- **NU_SUCCESS**
Indicates successful operation.
- **NU_INVALID_FUNCTION**
Indicates invalid parameters.

Example

```
/* Assume variables are defined locally. */  
STATUS status;      /* Non-Volatile Memory write status */  
CHAR data[10];      /* Location where read data is to be stored */  
UINT data_size;     /* Size of the read data */  
UINT index;         /* Index of data in NVM */  
  
/* Read the data at the specified index from NVM into the data storage  
location returning the size of the read data. */  
status = NVM_Tgt_Read(&data[0], &data_size, index);  
  
/* If status is NU_SUCCESS, the data has been read from NVM and the  
data_size variable will contain the size of the read data. */
```

Related Topics

[NVM_Tgt_Write](#)

[NVM APIs](#)

NVM_Tgt_Write

This function writes an item to the NVM resource.

Usage

```
STATUS NVM_Tgt_Write(VOID      *p_item,  
                    UINT      item_size,  
                    UINT      *p_index,  
                    BOOLEAN verify);
```

Arguments

- **p_item**
Pointer to the item to be written.
- **item_size**
Size of item (in bytes).
- **p_index**
Points to the return value containing the index of the written data (if the operation is successful).
- **verify**
Indicates if the write operation should be verified (using [NVM_Tgt_Read](#)) or not.

Return Values

- **NU_SUCCESS**
Indicates successful operation.
- **NU_INVALID_FUNCTION**
Indicates invalid parameter.
- **NU_INVALID_OPERATION**
Indicates no more items may be written.
- **NU_INVALID_MEMORY**
Indicates insufficient memory available.

Example

```
/* Assume variables are defined locally. */  
STATUS status; /* Non-Volatile Memory write status */  
CHAR data[] = "hello"; /* Data to be written to NVM */  
UINT index; /* Index of written data in NVM */  
  
/* Write the data into NVM passing a pointer to the data, the data size, a  
place to return the index of the written data and indicate that the write  
operation should not be verified. */  
status = NVM_Tgt_Write(&data[0], sizeof(data), &index, NU_FALSE);
```

```
/* If status is NU_SUCCESS, the data has been written to NVM and the  
index variable will contain the index of the data in NVM. */
```

Related Topics

[NVM_Tgt_Read](#)

[NVM APIs](#)

NVM_Tgt_Reset

This function resets the NVM write process.

Note



This function should be called at the beginning of each write session.

Usage

```
STATUS NVM_Tgt_Reset (VOID);
```

Arguments

- None

Return Values

- NU_SUCCESS

Indicates successful operation.

Example

```
/* Assume status is defined locally. */  
STATUS status; /* Non-Volatile Memory reset status */  
  
/* Reset NVM storage. */  
status = NVM_Tgt_Reset();  
  
/* At this point status indicates if the service was successful */
```

Related Topics

[NVM_Tgt_Write](#)

[NVM APIs](#)

NVM_Tgt_Info

This function provides information about the NVM resource.

Usage

```
STATUS NVM_Tgt_Info(UINT *p_nvm_size,  
                   UINT *p_item_size_max);
```

Arguments

- `p_nvm_size`
Pointer to the size (in bytes) of the NVM resource.
- `p_item_size_max`
Pointer to the largest item (in bytes) currently in the NVM resource.

Return Values

- `NU_SUCCESS`
Indicates successful operation.
- `NU_INVALID_FUNCTION`
Indicates invalid parameters.

Example

```
/* Assume variables are defined locally. */  
STATUS status;           /* Non-Volatile Memory info status */  
UINT  nvm_size;          /* Total size of NVM in bytes */  
UINT  item_size_max;     /* Size (in bytes) of the largest item currently in  
                           NVM */  
  
status = NVM_Tgt_Info(&nvm_size, &item_size_max);  
/* If status is NU_SUCCESS, the other information is accurate. */
```

Related Topics

[NVM_Tgt_Reset](#)

[NVM APIs](#)

Chapter 14

Nucleus Power Services User Manual

Power Services User Overview

Software driven power management is crucial for battery operated or low power budget embedded systems. Embedded developers can now take advantage of the latest power saving features in today's processors with the built-in Power Management Framework in the Nucleus RTOS. Developers specify application requirements with high-level hardware agnostic APIs, and Nucleus automatically discovers power-aware components to help simplify the design process, increase code reuse, and speed time to market.

Operation

This section details the [Configuration](#) of Power Services and the [Power Services Initialization](#), and shows a few relevant [Use Cases](#).

Configuration

The following components must be enabled to use Power Services:

```
nu.os.svcs.pwr = true
```

Enables Nucleus Power Management Services.

```
nu.os.svcs.pwr.core = true
```

Enables Nucleus Power Management Core Services.

```
nu.os.kern.eqm = true
```

Enables Nucleus Event Queue Manager.

```
nu.os.drvr.pwr_intrf = true
```

Enables Nucleus Power Interface for Power-Aware device drivers.

Note



When you enable **nu.os.svcs.pwr**, it will enable all the other power management required components.

The configuration options shown in [Table 14-1](#) are available under **nu.os.svcs.pwr.core** in the Nucleus UI configurator:

Table 14-1. Core Power Services Configuration Options

Configuration Option	Description	Notes on Dependencies	Default
enable_dvfs	Enable / Disable Dynamic Voltage and Frequency Scaling (DVFS)	Requires CPU driver	True
enable_peripheral	Enable / Disable peripheral state services to control peripheral power states		True
enable_idle	Enable / Disable CPU idle scheduling.	Requires CPU driver	True
enable_cpu_usage	Enable / Disable CPU usage and utilization services	Requires CPU driver	False
enable_selfrefresh	Enable / Disable SDRAM self refresh mode	Requires CPU driver	False
enable_system	Enable / Disable system state services	Requires Peripheral State Services	True
enable_watchdog	Enable / Disable watchdog component		True
enable_hibernate	Enable / Disable hibernate component	Requires DVFS Services	False
initial_op	This is the OP that will be set once DVFS services has completed initialization, a value of 255 will use the highest available OP		255
set_def_power_state_off	When set to true each device's default power state is turned off during configuration overriding BSP settings		False
nvm_available	This indicates that a non-volatile memory driver is available		False

For more information about creating a custom configuration, see “Creating a Custom Configuration”, in the chapter “Getting Started with Nucleus ReadyStart Using Sourcery Code Bench”, in the *Nucleus ReadyStart Guide*.

Power Services Initialization

Nucleus Power Services get initialized automatically (at Run-level 4) during the initialization sequence of the operating system. Please refer to the [Initialization](#) chapter for more information on Run-level initialization sequence. The following components of the OS must successfully initialize first in order for Nucleus Power Services initialization to be successful:

- The Kernel (Nucleus PLUS)
- The Device Manager
- The Registry Service
- A target specific CPU driver

Use Cases

This section describes the following use cases:

- [Using the Peripheral State \(PS\)](#)
 - [Setting/Clearing minimal PS](#)
 - [PS behavior when minimal PS requests are active](#)
- [Creating and Using the System State \(SS\)](#)
 - [Setting/Clearing minimal SS](#)
 - [SS behavior when minimal SS requests are active](#)
- [Using the Operating Point \(OP\)](#)
 - [Setting/Clearing Minimal OP](#)
 - [OP behavior when Minimal OP Requests are Active](#)

Using the Peripheral State (PS)

Power aware devices are controlled by their Peripheral State and the application can control this state using the following commands.

Prerequisites

- This use case assumes that there are no active minimal Peripheral State requests.

Procedure

1. Get the number of supported Peripheral States of the device using [NU_PM_Get_Power_State_Count](#). All devices have at least 2 states represented by OFF

(state 0) and ON (state 1), but they are not limited to just these. Therefore knowing the state count can be useful.

2. Get the current Peripheral State of the device using [NU_PM_Get_Power_State](#).
3. Set the Peripheral State of the device using [NU_PM_Set_Power_State](#). Regardless of the number of states supported by the device, you can always use the predefined constants `POWER_ON_STATE` and `POWER_OFF_STATE` to place a device into either the ON or OFF state. If the device has more than two states, you can also use any value between 0 (OFF) and "number of states - 1" (ON) to set the device state.

Related Topics

[PS behavior when minimal PS requests are active](#)

[Setting/Clearing minimal PS](#)

Setting/Clearing minimal PS

Setting a minimal Peripheral State tells the system to not drop a device below a certain power level or state. This prevents an application from accidentally changing the Peripheral State to a level that will make the device unusable.

Prerequisites

- Assume, for example, a device that has 3 Peripheral States: OFF(0), IDLE(1), ON(2).

Procedure

1. Set the Peripheral State of the device to ON.
2. Request a minimum Peripheral State of IDLE using [NU_PM_Min_Power_State_Request](#). This does not change the current Peripheral State, but it prevents the application from changing the Peripheral State to anything lower than the IDLE state.
3. Use [NU_PM_Min_Power_State_Release](#) to release a minimum Peripheral State.

Note



You can have multiple minimum Peripheral State requests active and the system will make sure that it does not drop the device below the highest set minimal request.

Related Topics

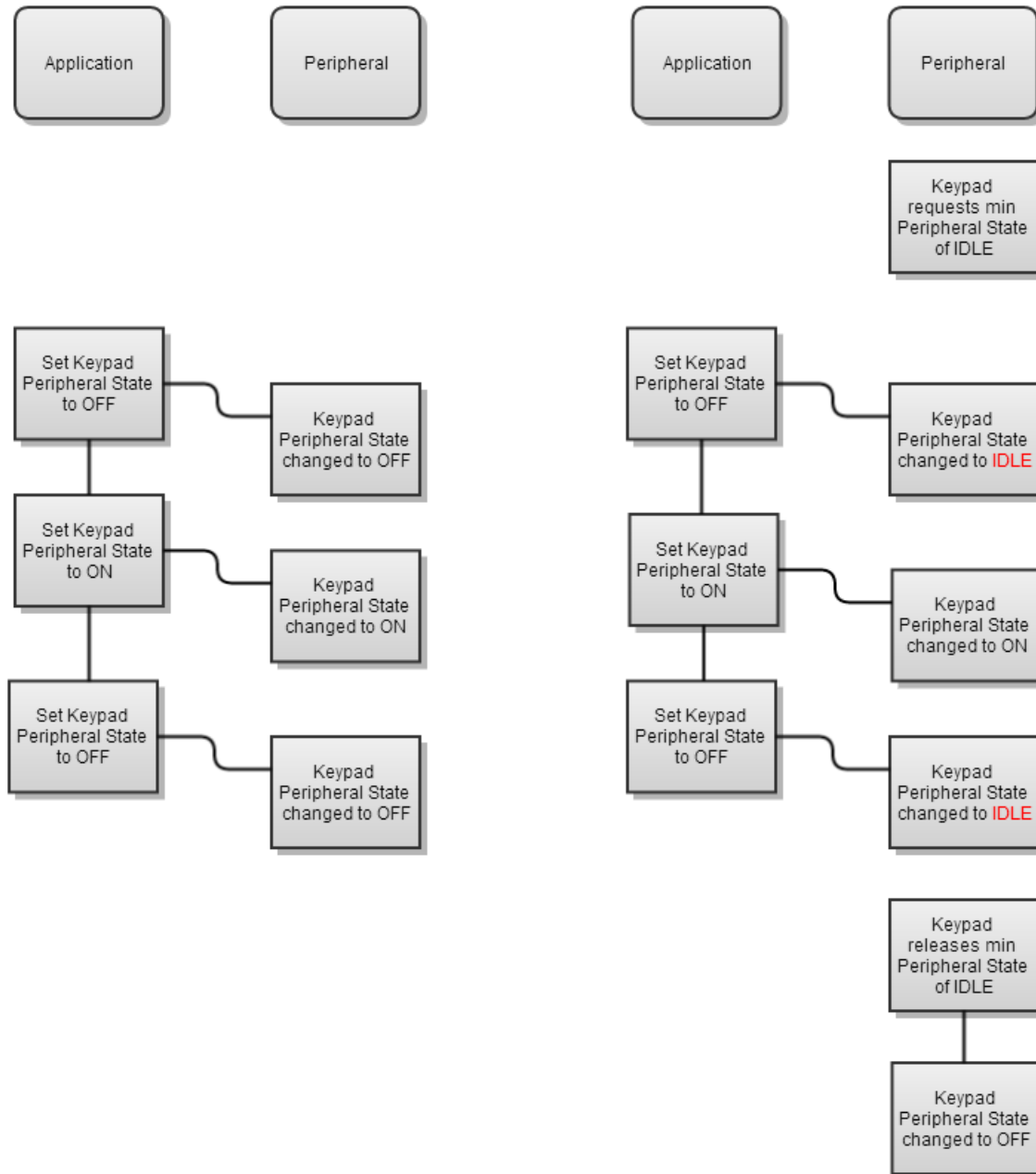
[PS behavior when minimal PS requests are active](#)

[Using the Peripheral State \(PS\)](#)

PS behavior when minimal PS requests are active

When minimal Peripheral State requests are active, the Peripheral State API's are limited as to how low they can set a peripheral's power state. As an example, say we have a keypad and it has 3 power states; ON(2), IDLE(1) and OFF(0). In [Figure 14-1](#), the left side shows the behavior of the Peripheral State APIs when there are no minimal requests and the right side shows the behavior when there is a minimal request.

Figure 14-1. PS APIs With and Without Minimal Requests Active



Related Topics

[Using the Peripheral State \(PS\)](#)

[Setting/Clearing minimal PS](#)

Creating and Using the System State (SS)

Creating a System State allows you to control multiple devices with just one command. In this simple example, we define a System State with just ON and OFF, however you can define one that contains many system states where some of the devices are ON and others are OFF or in a low power state.

Prerequisites

- This use case assumes that there are no active minimal System State requests.

Procedure

1. Initialize System State services to 3 states using [NU_PM_System_State_Init](#). The three states represent OFF (state 0), IDLE (state 1) and ON (state 2).
2. Map peripheral power states to System States using [NU_PM_Map_System_Power_State](#). For the device's states you can use the predefined constants `POWER_ON_STATE` and `POWER_OFF_STATE` as every device must understand these values:
 - System State 0 (OFF): Keypad State OFF, LCD State OFF.
 - System State 1 (IDLE): Keypad State ON, LCD State OFF.
 - System State 2 (ON): Keypad State ON, LCD State ON.
3. Set the System State to IDLE (state 1) using [NU_PM_Set_System_State](#). By setting the System State to IDLE, the Keypad is set to the ON state and the LCD is set to the OFF state.
4. To determine the current System State use [NU_PM_Get_System_State](#).
5. To un-map a device from System State use [NU_PM_Unmap_System_Power_State](#).

Related Topics

[Setting/Clearing minimal SS](#)

[SS behavior when minimal SS requests are active](#)

Setting/Clearing minimal SS

Setting a minimal System State tells the system to not drop below a certain power level or state. This prevents an application from accidentally changing the System State to a level that will make some of the devices unusable.

Procedure

1. Create a System State with 3 states: OFF(0), IDLE(1), ON(2).
2. Map devices to these states as shown below:
 - System State 0 (OFF): Keypad State OFF, LCD State OFF.
 - System State 1 (IDLE): Keypad State ON, LCD State OFF.
 - System State 2 (ON): Keypad State ON, LCD State ON.
3. Set the System State to ON.
4. Request a minimum System State of IDLE using [NU_PM_System_Min_State_Request](#). This does not change the current System State, but it prevents the application from changing the System State to anything lower than the IDLE state.
5. Use [NU_PM_System_Min_State_Release](#) to release a minimum System State.

Note



You can have multiple minimum System State requests active and the system will make sure that it does not drop below the highest set minimal request.

Related Topics

[Creating and Using the System State \(SS\)](#)

[SS behavior when minimal SS requests are active](#)

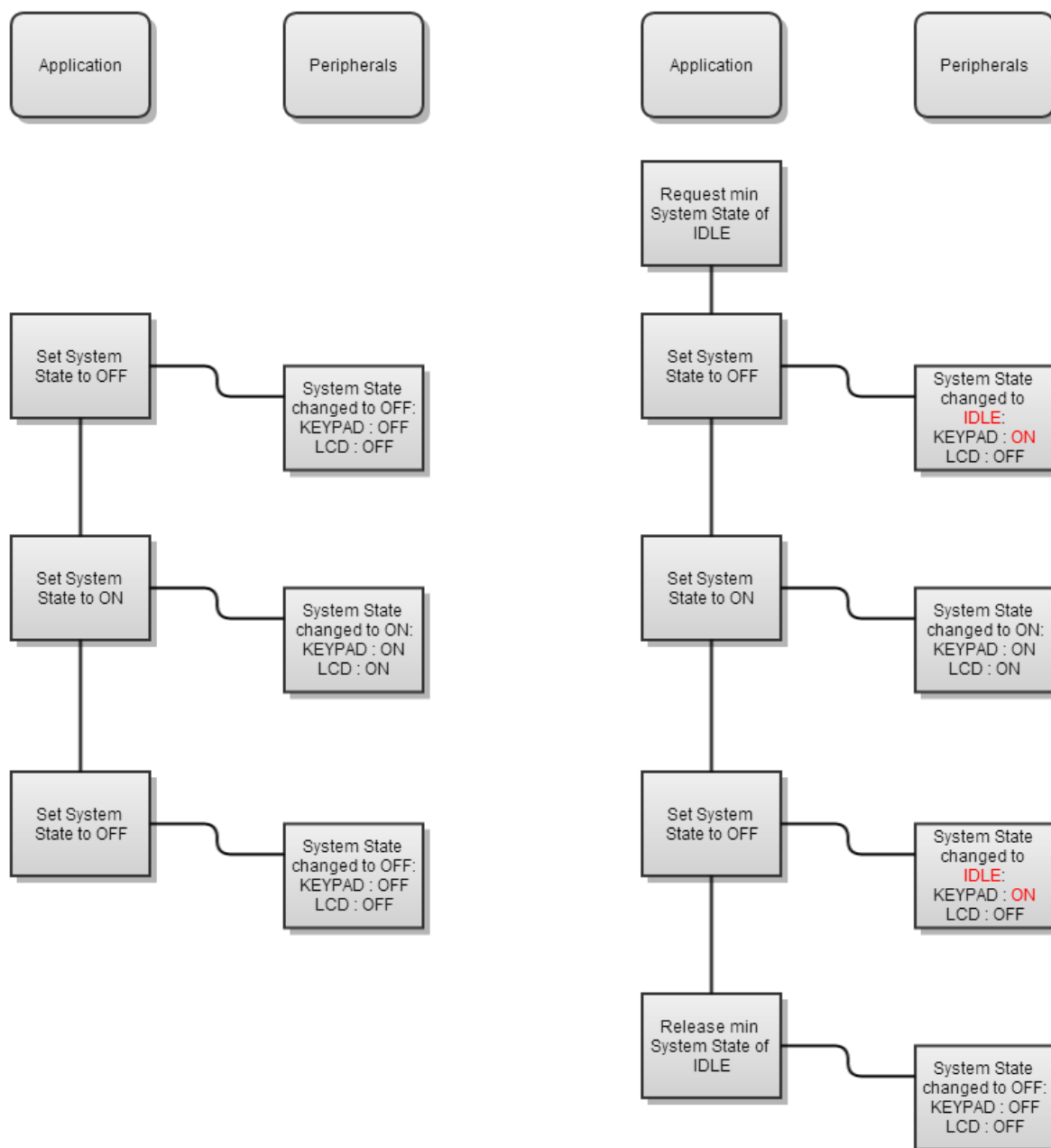
SS behavior when minimal SS requests are active

When minimal System State requests are active, the System State API's are limited as to how low they can set the system's power state. As an example, say we have 2 devices, a keypad and a lcd, and each has 2 power states; ON (1) and OFF (0) and we have mapped the System States like this:

- System State 0 (OFF): Keypad State OFF, LCD State OFF.
- System State 1 (IDLE): Keypad State ON, LCD State OFF.
- System State 2 (ON): Keypad State ON, LCD State ON.

In [Figure 14-2](#) the left side shows the behavior of the System State APIs when there are no minimal requests and the right side shows the behavior when there is a minimal request.

Figure 14-2. SS APIs With and Without Minimal Requests Active



Related Topics

[Creating and Using the System State \(SS\)](#)

[Setting/Clearing minimal SS](#)

Using the Operating Point (OP)

Operating Points (OP) are a Clock Frequency / Voltage pair that is defined by the Nucleus CPU driver. The OP allows the application to adjust power consumption of the CPU by using one command. The trade off for power consumption is performance so setting the system to a lower Operating Point may degrade performance.

Prerequisites

- This use case assumes that there are no active minimal Operating Point requests.

Procedure

1. Get the number of Operating Points using [NU_PM_Get_OP_Count](#). This value is determined by the target specific CPU driver and therefore is different for each target.
2. Set the system to an Operating Point using [NU_PM_Set_Current_OP](#). The value can be anything from 0 (lowest CPU setting) to "operating count - 1" (highest CPU setting).

Note



Operating Point set commands replace any currently active Operating Point settings.

Related Topics

[Setting/Clearing Minimal OP](#)

[OP behavior when Minimal OP Requests are Active](#)

Setting/Clearing Minimal OP

Setting a minimal Operating Point tells the system to not drop below a certain CPU power level or state. This prevents an application from accidentally changing the Operating Point to a level that is not acceptable.

Prerequisites

- Assume a system that has 3 Operating Points of 0, 1, 2 where 0 is the lowest power (worse performance) and 2 is highest power (best performance) with 1 being somewhere in between.

Procedure

1. Set the system to Operating Point 2.
2. Request a minimum Operating Point of 1 using [NU_PM_Request_Min_OP](#). This does not change the current Operating Point, but it prevents the application from changing the current Operating Point to anything lower than 1.
3. Use [NU_PM_Release_Min_OP](#) to release a minimum Operating Point.

 **Note** You can have multiple minimum Operating Point requests active and the system will make sure that it does not drop the current Operating Point below the highest set minimal request.

Related Topics

[Using the Operating Point \(OP\)](#)

[OP behavior when Minimal OP Requests are Active](#)

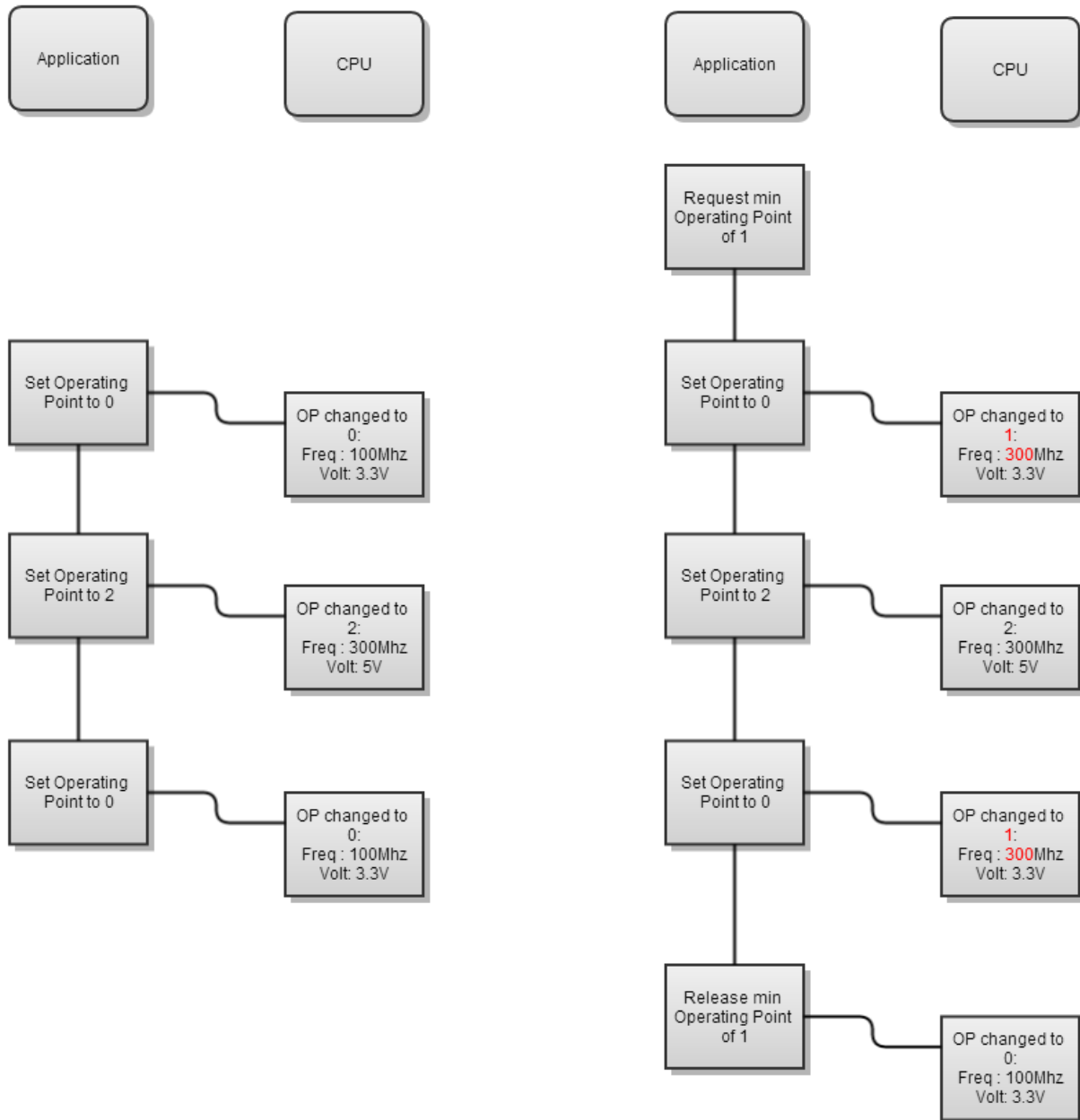
OP behavior when Minimal OP Requests are Active

When minimal Operating Point requests are active, the Operating Point API's are limited as to how low they can set the CPU's power state. As an example, say we have 3 Operating Points:

- OP 0: CPU Frequency 100Mhz, Voltage 3.3V
- OP 1: CPU Frequency 300Mhz, Voltage 3.3V
- OP 2: CPU Frequency 300Mhz, Voltage 5V

In [Figure 14-3](#), the left side shows the behavior of the Operating Point APIs when there are no minimal requests and the right side shows the behavior when there is a minimal request.

Figure 14-3. OP APIs With and Without Minimal Requests Active



Related Topics

[Using the Operating Point \(OP\)](#)

[Setting/Clearing Minimal OP](#)

Nucleus Power Services Internals

This section offers details about:

- [Tasks](#)
- [Queues](#)
- [Semaphores](#)
- [Event Groups](#)
- [Timers](#)

Tasks

[Table 14-2](#) summarizes all the OS Tasks that are created as part of Nucleus Power Services initialization:

Table 14-2. OS Tasks Created at Power Services Initialization

Name	Entry Function	Prior ity	Time Slice	Stack Size
PMCclean	PMS_Cleanup_Task_Entry	0	10	CFG_NU_OS_SVCS_PWR_CORE_INIT_TASK_STACK_SIZE (nu.os.svcs.pwr.core.init_task_stack_size)
PMErr	PMS_Init_Entry	0	10	CFG_NU_OS_SVCS_PWR_CORE_INIT_TASK_STACK_SIZE (nu.os.svcs.pwr.core.init_task_stack_size)
DVFS OP	PMS_DVFS_Set_OP_Task_Entry	0	0	CFG_NU_OS_SVCS_PWR_CORE_SET_OP_STACK_SIZE (nu.os.svcs.pwr.core.set_op_stack_size)
WATCHD	PMS_Init_Task_Entry	0	10	CFG_NU_OS_SVCS_PWR_CORE_INIT_TASK_STACK_SIZE (nu.os.svcs.pwr.core.init_task_stack_size)
WDTask	PMS_Watchdog_Task_Entry	10	5	(NU_MIN_STACK_SIZE * 8)
SELFR	PMS_Init_Task_Entry	0	10	CFG_NU_OS_SVCS_PWR_CORE_INIT_TASK_STACK_SIZE (nu.os.svcs.pwr.core.init_task_stack_size)

PMCclean

This task waits on notification from initialization tasks and frees any allocated memory and deletes the tasks.

PMError

This task serves as the entry point for Power Services error handling.

Note



PMS_Init_Task_Entry is a common entry point that is passed in variable **argc**, **argv** parameters. For this case, the function being called is **PMS_Error_Entry**.

DVFS OP

This task only runs when a call to set OP is made. This task is non-preemptive and may only be called one at a time. It will do the set OP operation and return the result through a queue and run to completion. The task will need to be reset with new arguments from the set current OP API.

WATCHD

This task initializes the Watchdog Component by creating the main watchdog task. This task is a temporary task and once it completes, it is terminated and deleted.

Note



PMS_Init_Task_Entry is a common entry point that is passed in variable **argc**, **argv** parameters. For this case, the function being called is **PMS_Watchdog_Initialize**.

WDTask

This task will suspend until the expiration of the lowest `next_inactivity_check` value.

SELFR

This task initializes the SDRAM self-refresh mode. This task is a temporary task and once it completes, it is terminated and deleted.

Note



PMS_Init_Task_Entry is a common entry point that is passed in variable **argc**, **argv** parameters. For this case, the function being called is **PMS_SelfRefresh_Initialize**.

Queues

Table 14-3 summarizes all the OS Queues that are created as part of Nucleus Power Services initialization:

Table 14-3. Queues created at Power Services Initialization

Name	Control Block	Type	Size
PMClean	PMS_Cleanup_Queue	NU_FIFO	(sizeof(PM_INIT) / sizeof(UNSIGNED))
PMERR	PMS_Error_Queue	NU_FIFO	PM_ERROR_QUEUE_AREA_SIZE
DVFS OP	PMS_Set_OP_Queue	NU_FIFO	PM_SET_OP_QUEUE_SIZE

PMClean

The non-preemptive task **PMS_DVFS_Set_OP_Task_Entry** places the result of the OP change in this queue and the result is retrieved by [NU_PM_Set_Current_OP](#).

PMERR

Queue to be used by the error system to pass messages.

DVFS OP

As init tasks complete, a message is added to this queue to inform **PMS_Cleanup_Task_Entry** that the init task can be deleted.

Semaphores

Table 14-4 summarizes all the OS semaphores that are created as part of Nucleus Power Services Initialization.

Table 14-4. Semaphores Created at Power Services Initialization

Name	Control Block	Type
DVFS OP	PMS_Set_OP_Semaphore	NU_FIFO
WDSem	Dynamically allocated as needed	NU_FIFO

DVFS OP

Semaphore to guarantee that only one caller may utilize the set OP thread at a time.

WDSem

WDSem is created when a calling task needs to suspend on a watchdog until it becomes active or expires.

Event Groups

Table 14-5 summarizes all the OS Event Groups that are created as part of Nucleus Power Services initialization:

Table 14-5. Event Groups Created at Power Services Initialization

Name	Control Block
PMEDEVT	PM_WD_Process_Event

PMEDEVT

Used by **PMS_Watchdog_Task_Entry** to determine when a new watchdog event has occurred.

Timers

Table 14-6 summarizes all the OS Timers that are created as part of Nucleus Power Services initialization:

Table 14-6. Timers Created at Power Services Initialization

Name	Control Block
WSTimer	PMS_Watchdog_Timer

WSTimer

Resumes the main watchdog task on watchdog expiration and resets itself to expire at the lowest 'next_inactivity_time_check'.

Nucleus Power Services Hibernate Internals

The Hibernate functionality provided by Nucleus Power Services allows a target to enter a special mode designed to reduce power consumption by suspending execution and turning off devices. There are two hibernation states:

- standby
In standby mode RAM is placed into a self-refresh mode and devices are turned off.
- dormant
In dormant mode the state of the system is saved to non-volatile memory and all devices (including RAM) are turned off.

The primary differences between the two hibernation states is the time it takes to enter/exit the states and the power savings. Standby mode is the quickest to enter/exit while dormant mode has the most power savings. Choosing which state is appropriate for an application involves weighing the responsiveness of the system vs. the power savings.

Target Specific Hibernate Driver Functions

The generic portion of the Nucleus Hibernate Core code calls target specific functions in order to control the hardware. Below is the list of those functions and their definitions:

- [Hibernate_Tgt_Initialize](#)
- [Hibernate_Tgt_Shutdown](#)
- [Hibernate_Tgt_Standby_Enter](#)
- [Hibernate_Tgt_Stanby_Exit](#)
- [Hibernate_Tgt_Exit](#)
- [Hibernate_Tgt_Get_Region_Count](#)
- [Hibernate_Tgt_Pre_Save](#)
- [Hibernate_Tgt_Get_Region_Info](#)
- [nu_bsp_drvr_hibernate_<platform>_init](#)

Hibernate_Tgt_Initialize

This function does any target specific initialization for hibernate including memory region setup and interrupt handler registration.

Usage

```
VOID Hibernate_Tgt_Initialize(VOID);
```

Arguments

- None

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Shutdown](#)

Hibernate_Tgt_Shutdown

This function executes all code required to shutdown the CPU.

Usage

```
Status Hibernate_Tgt_Shutdown (VOID) ;
```

Arguments

- None

Return Values

- NU_SUCCESS
Indicates function completed successfully.
- Other
Indicates an error has occurred.

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Initialize](#)

Hibernate_Tgt_Standby_Enter

This function enables self-refresh on RAM and places the CPU in a low power state.

Usage

```
VOID Hibernate_Tgt_Standby_Enter (VOID) ;
```

Arguments

- None

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#) [Hibernate_Tgt_Stanby_Exit](#)

Hibernate_Tgt_Stanby_Exit

This function disables self-refresh on RAM.

Usage

```
VOID Hibernate_Tgt_Standby_Exit (VOID) ;
```

Arguments

- None

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Standby_Enter](#)

Hibernate_Tgt_Exit

This function performs target specific operations when system returns from hibernate.

Usage

```
VOID Hibernate_Tgt_Exit (VOID) ;
```

Arguments

- None

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Pre_Save](#)

Hibernate_Tgt_Get_Region_Count

This function returns the number of regions that have been setup for saving during hibernation.

Usage

```
UINT32 Hibernate_Tgt_Get_Region_Count (VOID);
```

Arguments

- None

Return Values

- Number of target specific regions to be saved

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Get_Region_Info](#)

Hibernate_Tgt_Pre_Save

This function executes anything that must be completed prior to saving data during hibernation.

Usage

```
VOID Hibernate_Tgt_Pre_Save(VOID) ;
```

Arguments

- None

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Get_Region_Info](#)

Hibernate_Tgt_Get_Region_Info

This function returns the start address and size of regions defined to be saved/restored during hibernation.

Usage

```
VOID Hibernate_Tgt_Get_Region_Info(UINT32 region_index,  
                                   VOID    **start,  
                                   UINT32  *size);
```

Arguments

- **region_index**
Index into the memory region table.
- **start**
Pointer to where the start address for this region will be returned.
- **size**
Pointer to where the size of this region will be returned.

Return Values

- None

Related Topics

[Target Specific Hibernate Driver Functions](#)

[Hibernate_Tgt_Get_Region_Count](#)

nu_bsp_drvr_hibernate_<platform>_init

This function is the Hibernate driver init function that is called by the runlevel system. The *<platform>* string is defined by the BSP name. As an example, for the i.MX28 target platform, the function name will be **nu_bsp_drvr_hibernate_imx28_init**:

Usage

```
STATUS nu_bsp_drvr_hibernate_imx28_init(const CHAR *key,
                                       INT          startstop);
```

Arguments

- **key**
Pointer to the registry path associated with this driver init function.
- **startstop**
Option to Register/Unregister this device.

Return Values

- **NU_SUCCESS**
Function completed successfully.

Related Topics

[Target Specific Hibernate Driver Functions](#) [Hibernate_Tgt_Initialize](#)

Target specific Non-Volatile Memory (NVM) Driver Functions

The generic portion of the Nucleus Hibernate NVM code calls target specific functions in order to control the hardware. Below is the list of those functions and their definitions:

- [NVM_Tgt_Open](#)
- [NVM_Tgt_Close](#)
- [NVM_Tgt_Read](#)
- [NVM_Tgt_Write](#)
- [NVM_Tgt_Reset](#)
- [NVM_Tgt_Info](#)

NVM_Tgt_Open

This function opens the driver.

Usage

```
STATUS NVM_Tgt_Open (VOID) ;
```

Arguments

- None

Return Values

- NU_SUCCESS
Function completed successfully.
- Other
Indicates an error has occurred.

Related Topics

[Target specific Non-Volatile Memory \(NVM\) Driver Functions](#) [NVM_Tgt_Close](#)

NVM_Tgt_Close

This function closes the driver.

Usage

```
STATUS NVM_Tgt_Close(VOID);
```

Arguments

- None

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **Other**
Indicates an error has occurred.

Related Topics

[Target specific Non-Volatile Memory \(NVM\) Driver Functions](#) [NVM_Tgt_Open](#)

NVM_Tgt_Read

This function reads the specific item from NVM.

Usage

```
STATUS NVM_Tgt_Read(VOID *p_item,  
                    UINT *p_item_size,  
                    UINT index);
```

Arguments

- **p_intem**
Pointer to the item to be read.
- **p_item_size**
Pointer to the return parameter that will contain the size of the item read (in bytes) if the operation is successful. If NULL is passed in then the value is not returned.
- **index**
Index to read item from.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_FUNCTION**
Indicates invalid parameters.

Related Topics

[Target specific Non-Volatile Memory \(NVM\)](#) [NVM_Tgt_Write](#)
[Driver Functions](#)

NVM_Tgt_Write

This function reads the specific item from NVM.

Usage

```
STATUS NVM_Tgt_Write(VOID      *p_item,  
                    UINT      item_size,  
                    UINT      *p_index  
                    BOOLEAN verify);
```

Arguments

- **p_intem**
Pointer to the item to be written.
- **item_size**
Size of the item (in bytes).
- **p_index**
Pointer to the return parameter that will be updated to indicate the index of the data written if the operation is successful.
- **verify**
Indicates if the write operation should be verified (using [NVM_Tgt_Read](#)) or not.

Return Values

- **NU_SUCCESS**
Function completed successfully.
- **NU_INVALID_FUNCTION**
Indicates invalid parameters.
- **NU_INVALID_OPERATION**
Indicates no more items may be written.
- **NU_INVALID_MEMORY**
Indicates insufficient memory available.

Related Topics

[Target specific Non-Volatile Memory \(NVM\)](#) [NVM_Tgt_Read](#)
[Driver Functions](#)

NVM_Tgt_Reset

This function resets the NVM write process.

Usage

```
STATUS NVM_Tgt_Reset (VOID) ;
```

Arguments

- None

Return Values

- NU_SUCCESS
Indicates function completed successfully.

Related Topics

[Target specific Non-Volatile Memory \(NVM\)](#) [NVM_Tgt_Info](#)
[Driver Functions](#)

NVM_Tgt_Info

This function provides information about NVM.

Usage

```
STATUS NVM_Tgt_Info(UINT *p_nvm_size,  
                   UINT *p_item_size_max);
```

Arguments

- `p_nvm_size`
Pointer to size of NVM (in bytes).
- `p_item_size_max`
Pointer to size of largest item (in bytes)

Return Values

- `NU_SUCCESS`
Function completed successfully.
- `NU_INVALID_FUNCTION`
Indicates invalid parameters.

Related Topics

[Target specific Non-Volatile Memory \(NVM\)](#) [NVM_Tgt_Reset Driver Functions](#)

Modifications Required in the *.platform* file

The platform file of your BSP needs to be modified to add devices and options to support Nucleus Hibernate. This file is located at:

`<System_Project\<bsp>\<platform>\<platform>.platform`.

The examples below show entries for the i.MX28 target board:

- Add an entry for the hibernate driver:

```
device("hibernate0"){  
    description    "Hibernate device"  
    enable         false  
    driver         "nu.bsp.drivr.hibernate.imx28"  
    runlevel       2  
    setup_entry    false  
  
    option("hibernate_dev") {  
        enregister true  
        default     1  
        description "This marks the device to be a Hibernate-Aware  
                    Device"    }}
```

```
        hidden      true
    }
}
```

- Add an entry for the NVM driver:

```
device("nvm0") {
    description      "Non-Volatile Memory device 0"
    enable           false
    driver           "nu.bsp.drvr.nvm.imx28"
    setup_entry      false
}
```

- Update the CPU driver entry to mark it as hibernate aware by adding the “hibernate_dev” option:

```
device("cpu0") {
    description      "CPU device 0"
    enable           true
    driver           "nu.bsp.drvr.cpu.imx28"
    runlevel         2
    setup_entry      true

    option("hibernate_dev") {
        enregister    true
        default       1
        description   "This marks the device to be a Hibernate-Aware
                        Device"
        hidden        true
    }
}
```

- Add the “hibernate_dev” option to any other driver that will be hibernate aware:

```
option("hibernate_dev") {
    enregister    true
    default       1
    description   "This marks the component to be hibernate aware"
    hidden        true
}
```

Nucleus Power Services Hibernate Low-level Driver Changes (BSP)

This section describes:

- [Run-level Init Function Changes](#)

Run-level Init Function Changes

The following changes need to be made to the target-specific “run-level init” function located in the `<device>_tgt.c` file.

Instance Handles

Save off every instance handle created during RUNLEVEL_START in a global array. Instance handles will be passed to the hibernate enter and resume functions.

Caution



This code may already exist in the `<device>_tgt.c` file.

```

/*****
 * GLOBAL VARIABLES
 *****/
SERIAL_INSTANCE_HANDLE*Serial_Tgt_Inst_Handle_Array[SERIAL_MAX_INSTANCES;
INT Serial_Tgt_Instance_Index = 0;

/*****
 * Run-level Init code
 *****/

/* Save the instance handle in a global array*/
Serial_Tgt_Inst_Handle_Array[Serial_Tgt_Instance_Index] = inst_handle;

/* Ensure index has not overrun */
if (Serial_Tgt_Instance_Index < SERIAL_MAX_INSTANCES)
{
    /* Increment the instance index */
    Serial_Tgt_Instance_Index++;
}

```

Included Function Prototypes

Add the following function prototypes to the `<device>_tgt.c` file:

```

#if (defined(CFG_NU_OS_SVCS_PWR_ENABLE) &&
      (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE))
extern VOID Serial_Tgt_Pwr_Hibernate_Resume (SERIAL_INSTANCE_HANDLE
                                             *inst_handle);
#endif /* (defined(CFG_NU_OS_SVCS_PWR_ENABLE) &&
          (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE ==
           NU_TRUE)) */

```

Component Control Checks

Add checks for component control options. Devices enter and exit a Hibernate mode in the same order they were initialized and are controlled by Nucleus Run Level Component Control code. Target run level initialization code in `<device>_tgt.c` must be updated to include options for Hibernate. These are RUNLEVEL_HIBERNATE, used when a device enters a hibernate mode and RUNLEVEL_RESUME, used when a device resumes from a hibernate mode.

```

    #if (defined(CFG_NU_OS_SVCS_PWR_ENABLE) &&
        (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE))
    /* Local variable for tracking device instance */
    UINT8    instance_index = 0;
    #endif

    if (compctrl == RUNLEVEL_START)
    {
        /* Perform RUNLEVEL START operations here */
    }
    #if (defined(CFG_NU_OS_SVCS_PWR_ENABLE) &&
        (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE))
    else if (compctrl == RUNLEVEL_RESUME)
    {
        /* Search every node until the reg path entry is found */
        for (instance_index = 0; (instance_index < Serial_Tgt_Instance_Index);
            instance_index++)
        {
            /* Verify this is a valid instance handle slot */
            if (Serial_Tgt_Inst_Handle_Array[instance_index] != NU_NULL)
            {
                {
                    if (strcmp(key,
                        Serial_Tgt_Inst_Handle_Array[instance_index]->reg_path) == 0)
                    {
                        /* Call the Hibernate function for the device */
                        Serial_Tgt_Pwr_Hibernate_Resume(Serial_Tgt_Inst_Handle_Array
                            [instance_index]);
                    }
                }
            }
        }
    }
    else if (compctrl == RUNLEVEL_HIBERNATE)
    {
        /* Search every node until the reg path entry is found */
        for (instance_index = 0; (instance_index < Serial_Tgt_Instance_Index);
            instance_index++)
        {
            /* Verify this is a valid instance handle slot */
            if (Serial_Tgt_Inst_Handle_Array[instance_index] != NU_NULL)
            {
                {
                    if (strcmp(key,
                        Serial_Tgt_Inst_Handle_Array[instance_index]->reg_path) == 0)
                    {
                        /* Call the Hibernate function for the device*/
                        Serial_Tgt_Disable(Serial_Tgt_Inst_Handle_Array
                            [instance_index]);
                    }
                }
            }
        }
    }
    #endif else
    {
        /* Perform other RUNLEVEL operations here */
    }

```

Hibernate Resume and Hibernate Restore Functions

The following changes need to be made in `<device>_tgt_power.h` and `<device>_tgt_power.c` files.

Function Declaration

The following “hibernate resume” and “hibernate restore” functions must be declared in the `<device>_tgt_power.h` header file:

```
#if (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE)
VOID      Serial_Tgt_Pwr_Hibernate_Resume (SERIAL_INSTANCE_HANDLE
                                           *inst_handle);
STATUS     Serial_Tgt_Pwr_Hibernate_Restore (SERIAL_SESSION_HANDLE
                                           *session_handle);
#endif
```

Function Definition

In the `<device>_tgt_power.c` source file, define the above functions. These functions contain code necessary to resume or restore a device from a Hibernate mode.

```
#if (CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE)
/*****
 *
 *
 *   FUNCTION
 *
 *       Serial_Tgt_Pwr_Hibernate_Resume
 *
 *   DESCRIPTION
 *
 *       This function resumes the device from Hibernate
 *
 *   INPUTS
 *
 *       SERIAL_INSTANCE_HANDLE *inst_handle - Device instance handle
 *
 *   OUTPUTS
 *
 *       None
 *****/
VOID Serial_Tgt_Pwr_Hibernate_Resume (SERIAL_INSTANCE_HANDLE
*inst_handle)
{
    /* Restore the device from a hibernate state.*/
    (VOID)NU_PM_Restore_Hibernate_Device (inst_handle->dev_id,
                                           (IOCTL_SERIAL_BASE + SERIAL_PWR_HIB_RESTORE));
}
#endif /* CFG_NU_OS_SVCS_PWR_CORE_ENABLE_HIBERNATE == NU_TRUE */
```

```

/*****
*
*
*   FUNCTION
*
*       Serial_Tgt_Pwr_Hibernate_Restore
*
*   DESCRIPTION
*
*       This function restores a serial device after hibernate.
*
*   INPUTS
*
*       session_handle - Handle to a session that is to be restored.
*
*   OUTPUTS
*
*       NU_SUCCESS - Indicates successful operation.
*
*       <other> - Indicates (other) internal error occurred.
*****/
STATUS Serial_Tgt_Pwr_Hibernate_Restore (SERIAL_SESSION_HANDLE *
session_handle)
{
    STATUS      status = NU_SUCCESS;
    /* Restore state of the session. */
    Serial_Tgt_Setup(session_handle->instance_ptr,
                      &(session_handle->instance_ptr->attrs));
    Serial_Tgt_Enable(session_handle->instance_ptr);
    status = Serial_PR_Int_Enable(session_handle);
    return(status);
}

.

```


Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:
 - 1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;
 - 1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;
 - 1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;
 - 1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;
 - 1.5. "Embedded Software" means Software that is embeddable;
 - 1.6. "End-User" means Customer's customer;
 - 1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;
 - 1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;
 - 1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;
 - 1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;
 - 1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;
 - 1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;
 - 1.13. "Products" means Software, Term-Licensed Products and/or Hardware;
 - 1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

- 1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;
- 1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;
- 1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;
- 1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;
- 1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;
- 1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;
- 1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;
- 1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and
- 1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. Orders, Fees and Payment.

- 2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. Grant of License.

- 3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**
- 3.2. License Type. The license type shall be identified in the applicable Addenda.
- 3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

- 3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.
- 3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.
- 3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. Beta Code.

- 4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. Restrictions on Use.

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.
- 5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.
- 5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. Support Services.

- 6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at <http://supportnet.mentor.com/about/legal/>.
- 6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at <http://www.mentor.com/codebench-support-legal>.
- 6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at <http://www.mentor.com/vsipl-support-legal>.
- 6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased

solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at <http://www.mentor.com/mel-support-legal>.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.
8. **Limited Warranty.**
 - 8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.
 - 8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR \$500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
10. **Hazardous Applications.**
 - 10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").
 - 10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.
 - 10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.
 - 10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. Infringement.

- 11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.
 - 11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.
 - 11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.
12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.
- 12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.
 - 12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.
13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.
14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.